October 23–24, 2023
Cascais, Portugal

Association for Computing Machinery

Advancing Computing as a Science & Profession

SLE '23

Proceedings of the 16th ACM SIGPLAN International Conference on

Software Language Engineering

Edited by:

João Saraiva, Thomas Degueule, and Elizabeth Scott

Sponsored by:

ACM SIGPLAN

Co-located with:

SPLASH '23

Association for Computing Machinery, Inc.
1601 Broadway, 10th Floor
New York, NY 10019-7434
USA

ACM ISBN: 979-8-4007-0396-6

Cover photo:
Title: “Cascais resort in Portugal”
Photographer: RossHelen
Licensed to SPLASH 2023
Cropped from original:
https://elements.envato.com/cascais-resort-in-portugal-KCGY987

Production: Conference Publishing Consulting
D-94034 Passau, Germany, info@conference-publishing.com

Welcome from the Chairs

Welcome to the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE) held in October 2023 as part of SPLASH 2023. Software Language Engineering (SLE) is a thriving research discipline targeted at establishing an engineering approach to the development, use, and maintenance of software languages, that is, of languages for the specification, modeling and tooling of software. Key topics of interest for SLE include approaches, methodologies and tools for language design and implementation with a focus on techniques for static and behavioral semantics, generative or interpretative approaches (including transformation languages and code generation) as well as meta-languages and tools (including language workbenches). Techniques enabling the testing, simulation or formal verification for language validation purposes are also of particular interest. SLE also accommodates empirical evaluation and experience reports of language engineering tools, such as user studies evaluating usability, performance benchmarks or industrial applications.

In 2023, SLE called for submission in four categories:
(1) Research Papers, detailing research contributions to SLE,
(2) New Ideas and Vision Papers, describing new, unconventional SLE research approaches that depart from standard practice,
(3) SLE Body of Knowledge Papers: surveys, essays, open challenges, empirical observations and case study papers which provide a comprehensive description of the concepts and best practices developed by the SLE community, and
(4) Tool Papers, focusing on tooling aspects and insights that are likely to be useful to other tool implementers or users in the future.

As in 2022, there was a two-phase submission and review process. This gave authors that submitted to the first round an extra opportunity to improve their work based on the comments and feedback of the reviewers. Altogether 40 submissions were received. Double-anonymous review guidelines were adopted for all papers. The reviewers assessed the submissions in terms of their novelty, significance and potential impact, and were instructed to carefully consider weightings across these criteria depending on the paper category.

Each submission was reviewed by at least three members of the Program Committee, and ultimately 20 papers were accepted for presentation. The SLE 2023 keynote was given by Crista Videira Lopes from University of California, Irvine.

As the Organization Committee, we are indebted to the hard work of the many people who contributed to the success of this year's SLE. Specifically, we would like to acknowledge the work of the Program Committee and for the timely delivery of reviews which resulted in a rich and diverse program. We are also grateful to SLE's Steering Committee and the SPLASH organization for their help. Finally, we would like to thank the authors of all submitted papers — you represent the core of the SLE conference, and it is your work that advances the state of the art in software language engineering. We hope that you will enjoy reading these proceedings and listening to the accompanying talks.

Cascais	Elizabeth Scott (PC co-Chair)
Portugal	Thomas Degueule (PC co-Chair)
October 2023	Joao Saraiva (Conference Chair)

SLE 2023 Organization

Organizing Committee

General Chair
João Saraiva (University of Minho, Portugal)

Program Co-chairs
Thomas Degueule (CNRS / LaBRI, France)
Elizabeth Scott (Royal Holloway University of London, UK)

Publicity Chair
Andrei Chiş (feenk, Switzerland)

Program Committee

Casper Bach Poulsen (Delft University of Technology, Netherlands)
Jean-Christophe Bach (IMT Atlantique, Lab-STICC (UMR 6285), France)
Jordi Cabot (Luxembourg Institute of Science and Technology, Luxembourg)
Horatiu Cirstea (LORIA / University of Lorraine, France)
Davide Di Ruscio (University of L'Aquila, Italy)
Romina Eramo (University of Teramo, Italy)
Bernd Fischer (Stellenbosch University, South Africa)
Görel Hedin (Lund University, Sweden)
Felienne Hermans (Vrije Universiteit Amsterdam, Netherlands)
Robert Hirschfeld (University of Potsdam / Hasso Plattner Institute, Germany)
Zhenjiang Hu (Peking University, China)
Adrian Johnstone (Royal Holloway University of London, UK)
Dimitris Kolovos (University of York, UK)
Ivan Kurtev (Eindhoven University of Technology, Netherlands)
Julien Lange (Royal Holloway University of London, UK)
Ralf Lämmel (Universität Koblenz, Germany)
Stefan Marr (University of Kent, UK)
Marjan Mernik (University of Maribor, Slovenia)
Gunter Mussbacher (McGill University, Canada)
Oscar Nierstrasz (feenk, Switzerland)
Bruno C. d. S. Oliveira (University of Hong Kong, China)
Juri Di Rocco (University of L'Aquila, Italy)
Bernhard Rumpe (RWTH Aachen University, Germany)
Neil Sculthorpe (Nottingham Trent University, UK)
Tamás Szabó (GitHub Next, Germany)
Mauricio Verano Merino (Vrije Universiteit Amsterdam, Netherlands)
Manuel Wimmer (JKU Linz, Austria)
Vadim Zaytsev (University of Twente, Netherlands)
Philipp Zech (University of Innsbruck, Austria)

Luis Eduardo de Souza Amorim (Australian National University, Australia)
L. Thomas van Binsbergen (University of Amsterdam, Netherlands)
Mark van den Brand (Eindhoven University of Technology, Netherlands)
Tijs van der Storm (CWI / University of Groningen, Netherlands)

Contents

Exceptions all Over the Shop

Modular, Customizable, Language-Independent Exception Handling Layer

Walter Cazzola
Università degli Studi di Milano, Italy
cazzola@di.unimi.it

Luca Favalli
Università degli Studi di Milano, Italy
favalli@di.unimi.it

Abstract

The introduction of better abstractions is at the forefront of research and practice. Among many approaches, domain-specific languages are subject to an increase in popularity due to the need for easier, faster and more reliable application development that involves programmers and domain experts alike. To smooth the adoption of such a language-driven development process, researchers must create new engineering techniques for the development of programming languages and their ecosystems. Traditionally, programming languages are implemented from scratch and in a monolithic way. Conversely, modular and reusable language development solutions would improve maintainability, reusability and extensibility. Many programming languages share similarities that can be leveraged to reuse the same language feature implementations across several programming languages; recent language workbenches strive to achieve this goal by solving the language composition and language extension problems. Yet, some features are inherently complex and affect the behavior of several language features. Most notably, the exception handling mechanism involves varied aspects, such as the memory layout, variables, their scope, up to the execution of each statement that may cause an exceptional event—*e.g.*, a division by zero. In this paper, we propose an approach to untangle the exception handling mechanism dubbed the *exception handling layer*: its components are modular and fully independent from one another, as well as from other language features. The exception handling layer is language-independent, customizable with regards to the memory layout and supports unconventional exception handling language features. To avoid any assumptions with regards to the host language, the exception handling layer is a stand-alone framework, decoupled from the exception handling mechanism offered by the back-end. Then, we present a full-fledged, generic Java implementation of the exception handling layer. The applicability of this approach is presented through a language evolution scenario based on a Neverlang implementation of JavaScript and LogLang, that we extend with conventional and unconventional exception handling language features using the exception handling layer, with limited impact on their original implementation.

CCS Concepts: • **Software and its engineering** → **Abstraction, modeling and modularity**; **Compilers**; **Extensible languages**.

Keywords: Language Modularization, Exception Handling.

ACM Reference Format:
Walter Cazzola and Luca Favalli. 2023. Exceptions all Over the Shop: Modular, Customizable, Language-Independent Exception Handling Layer. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3623476.3623513

1 Introduction

Programming language development is a complex activity. It involves the development of an ecosystem of varied software artifacts, such as, parsers, optimizers, translators and development environments. The traditional approach towards language development is monolithic: language constructs and their semantics are planned during the design phase and rarely change overtime. The monolithic approach is considered easier to develop and more performant; however, the final products are hard to change, update and evolve. Developing different languages with similar constructs can provide reuse opportunities that are only possible if the implementation is modularized, so that it is easier to extract and reuse in different contexts [41]. *Language workbenches* [22, 25] are a common approach to this problem.

However, tool support does not suffice due to the inherent complexity of some language features. Take exception handling as an example. Introducing an exception handling mechanism in an existing language implementation has a drastic impact on the way a program executes, because each statement might throw an exception: exception handling is a *crosscutting feature.* Crosscutting features are language features whose code is scattered across the implementation: they are known to reduce the flexibility and maintainability of software systems [19], and can also affect parse-tree rewriting contexts [32]. Moreover, each language has a different memory layout and handles exceptions in a different way.

SLE '23, October 23–24, 2023, Cascais, Portugal
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0396-6/23/10…$15.00
https://doi.org/10.1145/3623476.3623513

This makes a strictly modular and reusable implementation of exception handling challenging. In this work, we discuss a framework for the generalization of the exception handling crosscutting feature, to implement it without affecting the original implementation of the language and without making any assumptions on the structure of the original language.

Our contribution is a general, customizable, reusable, and extensible exception handling conceptual framework—dubbed as *exception handling layer*—that can be adopted by language workbenches to untangle the exception handling concern from the code of other language features.

The applicability of the proposed framework is demonstrated through a full-fledged Java implementation, then used to refactor the exception handling mechanism of a JavaScript [12] interpreter written in Neverlang [8, 14, 48] Moreover, we present the flexibility of our proposal by implementing unconventional exception handling language features such as the retry and resume statements, and implementing a recovery procedure for the LogLang [13] declarative domain-specific language. On each step of the language evolution scenario, we keep track of the required development effort in terms of lines of code and modified files. This work is validated by answering these research questions:

RQ$_1$. How hard is it to refactor an existing language implementation so that it can be used in tandem with the exception handling layer?
RQ$_2$. How hard is it to add exception handling support to a language implementation using the exception handling layer?
RQ$_3$. How much is the achieved modularization reliant on Neverlang-specific mechanisms?
RQ$_4$. Does the conceptual framework support varied exception handling mechanisms and their language features?

The remainder of this paper is structured as follows. Sect. 2 contains any background information relevant to this work, including language workbenches, their capabilities and the basics of exception handling. Sect. 3 presents the exception handling layer as the main contribution of this work. In Sect. 4 we present language evolution scenarios based on the introduction and extension of the exception handling layer. Finally, in Sect. 5 and Sect. 6 we will respectively discuss any related work and draw our conclusions on this research.

2 Background

In this section, we discuss the background information on language workbenches and exception handling.

2.1 Language Workbenches

Modular language development benefits from the creation of *sectional compilers* [8] defined in terms of independently developed language features. Each language feature is a reusable piece of a language specification, formed by a syntactic asset and a semantic asset, representing a language construct and its behavior respectively. Language workbenches [23] embrace this philosophy to improve reusability and maintainability of linguistic assets. The term language workbench was firstly introduced by Fowler [25] for the tools suited to support the language-oriented programming paradigm [55], in which complex software systems are built around a set of domain-specific languages, each used to express the problems and the solutions of a portion of the complex system. Nowadays, language workbenches are used to facilitate the development of modular programming languages and the reuse of software artifacts through better abstractions. These abstractions are designed to support five different composition mechanisms among programming languages: language extension, language restriction, language unification, self-extension, and extension composition [21]. There are several language workbenches in literature, each proposing its own flavor of language composition. Some examples (among many others) are: Melange [20], MPS [52], MontiCore [33], Neverlang [48], Rascal [31], Silver/Copper [51], and Spoofax [54].

2.2 Exception Handling

Software exceptions are anomalies that can occur during the execution of any instruction of a program. When an exception occurs, the application state does not conform to the continuation of its normal execution flow [27]. Instead, exceptions are handled through dedicated language control structures—called exception mechanisms—that replace the standard continuation with an exceptional continuation. Most modern programming languages provide such exception mechanisms, yet adequate exception handling has been proven difficult [18]. Sub-optimal exception handling practices are associated to low software quality and post-release defects [44]. Therefore, it is vital that each programming language provides the exception mechanisms that are the most appropriate with respect to the intended behavior. Relying on the abstractions provided by the backend is still the most common practice in the development of DSLs but this limits the capabilities of the exception handling mechanism to those offered by the back-end. Each language has its own constructs and uses a different notation, although three common elements have been identified [27]:

— a part of a program or an operation that brings an exceptional event to the attention of the caller; this is called *throwing* or *raising* an exception and can either be implicit (e.g., a division by zero) or explicit (e.g., a `throw` call in Java);

— the *handler* is a part of a program that must be executed to handle an exceptional event; exception handling can either be explicitly defined or provided by default;

— the handler's *reach* is a syntactic construct or part of a program (such as a block) that can launch the associated handler if the activation point of the exceptional event falls within it.

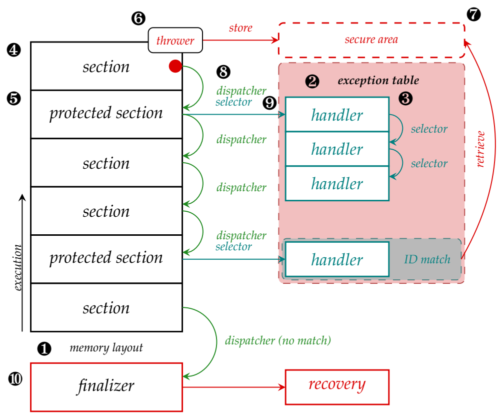


Figure 1. EHL general architecture and process.

3 The Exception Handling Layer

In this section, we discuss the conceptual framework for the implementation of a portable exception handling mechanism whose code is not scattered across the language implementation. This conceptual framework provides language developers with a template to design modular exception handling language extensions. We dubbed this conceptual framework as *exception handling layer* (EHL). We discuss EHL's fundamental data structures and procedures. We also present a fully modular decomposition that decouples exception handling from the other language features.

3.1 Architecture

Fig. 1 depicts the general architecture of EHL, its components and their interaction. Note how an exception-unaware language is a corner case of this architecture where all elements except the memory layout are omitted.

Memory layout (❶). The memory layout abstracts the stack of the machine the program is running on. We do not make any assumptions with regards to data structure used to represent the memory layout. Instead, the memory layout is split into *sections*.[1] According to this architecture, sections are nodes arranged within a directed graph that abstracts the entire memory. For simplicity, Fig. 1 shows the memory layout as a stack, a common memory layout among general purpose programming languages [1]. In fact, a stack can be viewed as a directed acyclic graph whose nodes are arranged in a chain—*i.e.*, each section is connected to the previous element of the stack. Arbitrary memory layouts allow to abstract unconventional exception handling mechanisms such as the `exit` function in Erlang: if a process calls `exit(kill)` and does not catch the exception, it will terminate and emit exit signals to all linked processes.[2]

Each section of the memory layout is either protected or normal, depending if it is within a handler's reach or not. However, the memory layout itself does not hold this piece of information. In fact, to properly modularize the exception handling concern, the memory layout is unaware of the existence of exceptions at all. Instead, any information regarding exceptions—such as, any exception handlers that can reach a section—resides in a distinct data structure. The same memory layout can therefore be used in a programming language without exception support. Notice that the memory layout of an exception-unaware language implementation contains only normal sections.

Exception table (❷). The *exception table* contains all references to the location of the exception handlers so that the correct handler can be executed for each exception type. In this context, the exception type is not a data type provided by the back-end but a more general and arbitrary descriptor. There is no assumptions wrt. the implementation of this table, e.g., it may be a list with an index associated to each handler or a bi-dimensional map associating each pair (section, exception ID) to a handler.

Exception handler (❸). As in [27], an *exception handler* refers to the code to be executed when an exception is caught, e.g., a **`catch`** block in Java. The nature of the handler depends on the language the exception mechanism is plugged on: in a compiler, it can be the method or the function to be executed; in an interpreter it can directly hook the AST node representing the code to execute. A custom implementation of the exception handler could even store the instructions to be executed directly inside the exception table.

Normal section (❹). Each *normal section* coincides with the memory reserved to the execution of a function or method call, or, in some languages, to a block of code—*e.g.*, a delimited sequence of instructions. Each section also serves as a namespace: it contains a symbol table with an arbitrary number of scopes, each with the named constants, variables, structures and procedures that are visible within that scope. A normal section is unaware of the existence of exceptions and is not within any exception handler's reach. The memory layout contains a reference to the currently executing section. In languages using a single stack, the current section coincides with the top of the stack, whereas in other cases it can be set by an external program, e.g., by the scheduler.

Protected section (❺). *Protected sections* represent the handler's reach from [27] and are parts of the program that are capable of capturing and handling exceptions, such as a **`try`**-**`except`** block in Python. A protected section is identical to a normal section but it is within the reach of an exception handler. If a section is normal or protected is determined by a procedure called *selector* (more on that later).

Thrower (❻). The *thrower* is a section that threw an exception, according to [27]. The thrower may be implicit, such as a section that caused a division by zero, or explicit, such as a section containing a **`throw`** statement in Java.

[1]Please refer to the corresponding paragraphs for more details on sections.
[2]https://www.erlang.org/doc/man/erlang.html#exit-1

Secure area (❼). The *secure area* is a special buffer reserved to the thrower to store all the relevant information (if any) for the exception handling mechanism. The secure area is needed to avoid any assumptions on the exception handling mechanism: the EHL treats exceptional events as simple signals, whereas any additional information is carried by the secure area. E.g., in Java the secure area would hold an instance of the `Throwable` class, whereas JavaScript applications can throw the result of any expression. The secure area is also used to store other information, such as the list of all sections visited during the exception handling event.

Dispatcher (❽). The *dispatcher* is an arbitrary graph traversal algorithm that is fired when an exception is thrown. Starting from the current section, the dispatcher navigates the memory layout. On each section, the dispatcher runs a secondary procedure called *selector* to determine if the current section is protected. The traversal ends when a protected section has a viable exception handler, as determined by the selector. Otherwise—*i.e.*, when there are no more sections to be visited in the queue—the dispatcher terminates abruptly and delegates to a procedure called *finalizer*. For instance, a stack-based implementation of the dispatcher may pop frames from the stack until finding a handler.

Selector (❾). The *selector* is an arbitrary procedure returning a viable handler for the thrown exception when a section is protected. Its result may be determined by inspecting the contents of the symbol table—*i.e.*, a section is protected if the exception table maps that section to at least one handler. The selector could be customized to implement implicitly protected sections—*i.e.*, sections with a default exception handler without a need for the programmer to declare one—even without inspecting the exception table.

Finalizer (❿). The *finalizer* is an arbitrary procedure that runs when no viable handler is found and the dispatcher cannot reach any more sections within the memory layout. The finalizer implements the ultimate recovery procedure and allows the runtime environment to smoothly shut down the application when a thrown exception cannot be handled by any handler of any protected section. Finalizers can also be used to attempt restoring the application to a suitable state without stopping the execution (see Sect. 4).

3.2 Process

In this section, we discuss the life-cycle of an exceptional event according to EHL. This process evolves according to four sequential phases: *normal execution*, *exception throwing*, *exception carrying* and *exception handling*.

Normal execution. During normal program execution, the memory expands and shrinks according to the creation and destruction of sections. Traditionally—*i.e.*, when the memory is a stack—a new section is created and pushed on the top of the stack upon entering a new scope, such as at the beginning of a block or on function calls. Sections are then popped from the stack after their code completes its execution. Different languages may use a different memory layout, adding and removing sections accordingly. For instance, if a section spawned several threads, the memory layout graph can contain several sections with an edge towards that section, one for each thread. In EHL, each time a new protected section is added to the memory layout, any handler for that protected section is added to the exception table. When a protected section ends its execution, it is removed from the memory layout and its handlers are (optionally) unregistered from the exception table. This execution flow is continued until a thrower is encountered, as shown by the red dot in Fig. 1, then an exceptional event occurs and the execution proceeds to the exception throwing phase.

Exception throwing. The system halts the normal execution when an exception is thrown. Such an exception is identified with a descriptor—e.g., its class in Java and the secure area is populated. Finally, a signal is sent to the EHL runtime to start the exception carrying phase.

Exception carrying. The thrown exception travels across the system according to the dispatcher algorithm until the correct handler is found, if it exists. During this phase, it must be possible to inspect the memory state, to feed handlers with any relevant information—*e.g.*, the variables in scope. Fig. 1 shows that the exception carrying phase is handled by the dispatcher and the selector. The dispatcher traverses the memory layout graph; on each visited section, the dispatcher delegates to the selector to determine if the current section is protected and if any of its handlers can handle the carried exception. This is usually done by inspecting the exception table, but some languages, particularly DSLs, may implement default handler procedures that are not held within the exception table. For instance, if the layout is a stack, the dispatcher may iteratively inspect the section on top of the stack, popping any normal section and any protected section with incompatible handlers. The process proceeds to the exception handling phase if a compatible handler is found (as shown by the blue box in Fig. 1). Otherwise the dispatcher is resumed to continue browsing the memory layout according to the traversal algorithm. The exception carrying mechanism fails and control is given up to the finalizer when the dispatcher ends its execution—*i.e.*, when there are no more sections to be visited. The finalizer performs any procedure needed to ensure that the system is safely shut down or recovered, possibly reporting any failures to the user. For instance, in Java the finalizer prints the stack frame before terminating the execution. A finalizer could also be used to roll back to a globally-known safe state.

Exception handling. The exception handling procedure starts when the selector finds a handler that is compatible to the thrown exception. The handling procedure retrieves the

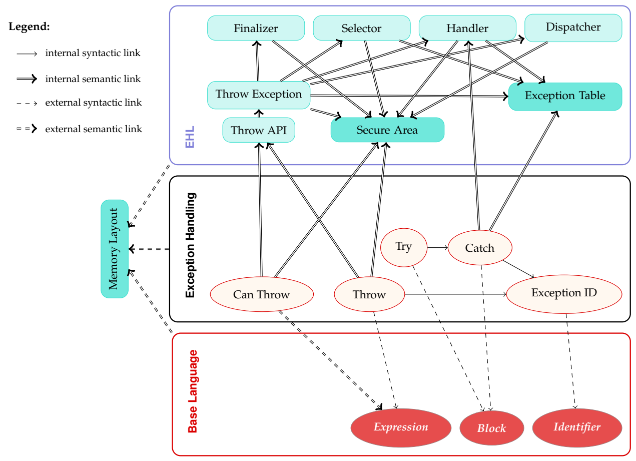


Figure 2. Language modularization according to the EHL.

information stored inside the secure area and—optionally—the scope of the protected section. There is no requirement on how the secure area is implemented: two possible options are either a globally accessible object or a instance that is created upon exception riding and then tunneled across the dispatcher, the selector and eventually the handler or the finalizer. Once the exception has been handled, the normal application flow is resumed. In many programming languages the execution flow is resumed after the end of the protected section, but other resumption mechanisms may be in place. Some examples are resumption from the instruction in which the exception has been thrown (resume) and from the first instruction of the thrower (retry). These mechanisms usually assume the handler changed the memory to a safe state or with additional information before resuming the normal execution. The EHL is agnostic wrt. exception handling mechanism chosen by the language and supports these mechanisms (see next sections).

3.3 Exception Handling Modular Decomposition

A language can be decomposed to leverage the EHL, untangling the exception handling code from the code of unrelated language features. Fig. 2 depicts such a modularization. Fig. 2 is comprised of two main components: the base language (red box) and the exception handling implementation built on its top. Fig. 2 also splits the exception handling mechanism into exception handling language features (black box) and EHL (blue box). The part about the base language implementation is not relevant to this discussion and is omitted. Each node in Fig. 2 is either a language feature (oval shape) or a component of the EHL architecture (rectangular shape, data structures are represented with a darker color and algorithms with a lighter color). The EHL components mirror the architecture discussed in Sect. 3.1. Each arrow represents a dependency between coupled components. Double and single arrows represent semantic and syntactic dependencies respectively. Dashed and normal arrows represent dependencies to external and internal components respectively.

The key element of this decomposition is dependency management. To minimize coupling between components and to maximize reuse, the decomposition uses EHL data structures as adapters [26], so that exception handling language features are not directly coupled with the underlying exception handling mechanism and the semantics can be changed freely. In fact, no feature from the base language depends on the exception handling, neither syntactically nor semantically. Exception handling features can instead depend on features of the base language, which they can extend, override and specialize depending on the exception handling mechanism to be implemented. For instance, in Fig. 2, the `Throw` language feature syntactically depends on the `Expression` language feature, because in this case the throw statement can throw an exception based on the return value of an expression. Similarly, the `CanThrow` language feature extends the semantics of any expression by declaring that its evaluation may cause an exceptional event—*e.g.*, a division by zero. Notice how such a decomposition is completely modular, so that exception handling features can be used independently. E.g., it is possible to create languages where i) the **`throw`** statement is present but expressions can never cause an exception, ii) expressions can cause exceptions but the **`throw`** statement is absent, and iii) exceptions can be thrown but never caught (no **`try-catch`** statements).

Internal dependencies among components of the EHL are similarly structured: the `Throw Exception` component acts as glue code that depends on all other elements of the exception handling mechanism, namely exception table, dispatcher, selector, handlers, finalizer and secure area. A different exception handling mechanism can be deployed by swapping the `Throw Exception` component with a similar component that shares the same interface but connects different elements. For instance, it is possible to create a new exception handling in which the dispatcher is replaced whereas all other elements remain the same. Similarly, elements can be shared across several `Throw Exception` components, possibly pertaining different programming languages. To ensure that data is properly carried throughout the entire exception handling process, all elements depend on the exception table, on the secure area, and on the memory layout. Thus, in the EHL the dependencies between components are limited to the data representation (darker color), rather than on the behavior: as long as the data representation stays the same, the exception handling mechanisms can be extended and replaced at will. Some specific language features may break this rule. For instance, the `Catch` language feature depends on the `Handler`, because it needs to register a new handler within the exception table upon entering a protected section. However, such dependencies are always limited to one language feature and do not impact the rest of the language: in

```java
public class JEL {
  public static void raise ( ExceptionID exceptionID,
      ExceptionTable exceptionTable, Memory memory,
      Section thrower, Dispatcher dispatcher,
      Selector selector, Optional<Finalizer> finalizer,
      Optional<SecureArea> secureArea) {
    var handler = dispatcher.dispatch (
      exceptionID, exceptionTable, memory,
      thrower, selector, secureArea);
    handler.ifPresentOrElse (
      h -> h.handle(secureArea),
      () -> finalizer.ifPresent (f ->
        f.finalize(secureArea, exceptionID)));
  }
}
```

Listing 1. Glue code that connects all the elements of JEL. Please note that redundant generic data types are omitted to save space; refer to the text of this section for the generic data types associated to each element.

Fig. 2, the `Catch` feature can be replaced with a different one without affecting the `Throw` and `CanThrow` features.

3.4 Exception Handling Layer in Java

In this work, we implemented the EHL as a library dubbed as *Java exception layer* (JEL). JEL behaves like an intermediate layer between the running application and the underlying JVM execution environment. The JEL library is intended to be used instead of the default exception handling mechanism offered by the JVM to support additional language features, such as retry/resume operations and handlers for arbitrary types—instead of just members of the `Throwable` hierarchy. While the underlying Java exceptions still exist within the runtime environment, they should be transparent for the user: assuming the language provides a full-fledged implementation of its exception handling mechanism using JEL, all Java exceptions are captured and translated into a JEL exceptional event. According to the modularization constraints discussed in Sect. 3.3, notice how JEL does not refer to any language-specific implementation aspect, such as the supported operations and their syntax. JEL is implemented as a library of generic interfaces with a default implementation.

Generic Data Types. Since the dependencies among components is EHL are based on the data representation, JEL data structures and algorithms can be customized according to five different data types (classes, in Java):

- `EX_ID` the exception identifier;
- `SEC_ID` the unique identifier for a section;
- `VAR_NAME_TYPE` the type used for variables identifiers;
- `VAR_TYPE_TYPE` the type used for variables types;
- `PAYLOAD` the type of data carried by the secure area.

Default implementation. Mirroring the EHL architecture, JEL provides the interfaces for three data structures: i) the memory layout, ii) the exception table and iii) the secure area, as well as their default implementations.

The *memory layout* is shared between the base language and the exception handling module. To fit the modularization requirements, the memory layout is implemented in an agnostic way wrt. the exception handling and is populated by generic `Section` objects. JEL provides two default implementations for the memory layout interface: a graph and a stack sharing the same `Section` objects. Both implementations can be adapted to the language by specifying the `SEC_ID`, `VAR_NAME_TYPE` and `VAR_TYPE_TYPE` generic data types.

The default *exception table* is implemented as a two-dimensional look-up table that takes the ID of the exceptional event and the ID of the thrower `Section` and maps them to the respective handler method. The exception table can be interacted with to register and unregister exception IDs and exception handlers upon entering and exiting sections during execution. The default exception table can be customized according to all five generic data types.

The default *secure area* is implemented as a wrapper for an object with store and retrieve operations. The type of wrapped data is set by specifying the `PAYLOAD` generic data type. The responsibility of correctly populating this data structure is delegated to the thrower, which will change depending on the language the EHL is being plugged on.

JEL also provides a default implementation for dispatcher and selector routines that can be customized according to all five generic data types. The default *dispatcher* is a traversal algorithm for a stack-based memory layout, that pops elements from the top, delegating to the selector on each element, until finding a handler or reaching the bottom. The default *selector* queries the two-dimensional exception table for the handler for a (`SEC_ID`, `EX_ID`) pair, if any.

Finally, JEL provides a static `raise` method that is in charge of starting the exception throwing event and that acts as the glue code connecting all the elements, as shown in Listing 1. Given this code, the execution of an exceptional event from the perspective of a language feature coincides with a call to the `raise` method with the correct arguments.

4 Case Study and Discussion

This section presents and discusses a language evolution scenario that uses EHL to add exception handling support to a base exception-unaware language. The Neverlang language workbench [48] is used to implement both the exception-unaware base language and its exception-aware variants.

4.1 Neverlang Overview

Neverlang [48] is a language workbench for the modular development of programming languages and their ecosystems. It is based on the language feature concept [9], each developed as separate units called *slices* that can be independently compiled, tested, and distributed. Syntactic and semantic

```
 1 module nl.jel.JELTryStatement {
 2   reference syntax {
 3     try_part: TryPart <-- "try" ProtectedSection;
 4     protected: ProtectedSection <-- Block;
 5   }
 6   role (evaluation) {
 7     try_part: .{
 8       eval $try_part[1];
 9       Section<Long,String,JSReference> section =
10         $try_part[1].section;
11       JELSymbolTable jst = (JELSymbolTable)$$SymbolTable;
12       Stack<Long, String, JSReference> stack =
13          jst.getStack();
14       stack.push(section);
15       stack.peek().get().getCode().execute();
17       if($try_part[1].shouldRaise) {
18         JSSecureArea secure = $$SecureAreaBuilder.build (
19             $try_part[1].error;
20         );
21         JSJEL.raise($$JSExceptionTable, stack,
22             section, $$JSDispatcher, $$JSSelector,
23             $$JSFinalizer, secure
24         );
25       }
26       stack.pop();
27     }.
28     protected: .{
29       /*Code to generate the Section executable object*/
30     }.
31   }
32 }
33 endemic slice nl.jel.JELEndemic {
34   declare {
35     static JSExceptionTable: nl.jel.JSExceptionTable;
36     static SecureAreaBuilder: nl.jel.SecureAreaBuilder;
37     static JSDispatcher: nl.jel.JSDispatcher;
38     static JSSelector: nl.jel.JSSelector;
39     static JSFinalizer: nl.jel.JSFinalizer;
40   }
41 }
42 language nl.jel.JSLangJEL {
43   slices nl.jel.JELTryStatement /* ... */
44   endemic slices nl.jel.JELEndemic /*...*/
45   roles syntax <+ evaluation
46 }
```

Listing 2. Syntax and semantics for the JavaScript try block language feature in Neverlang.

assets are contained in a compilation unit called *module*. A module contains a **reference syntax** block—in which the productions are defined—and any number of roles. Each role, is preceded by the **role** keyword, and represents a visit of the parse tree: each role is made by one or more semantic actions [1] that are executed when some nonterminal symbol is encountered in the parse tree. Syntactic definitions and semantic roles are exogenously composed using slices. Please refer to [48] for a full Neverlang overview.

Basic capabilities. Listing 2 shows a modular implementation of the **try** statement and part of the composition mechanisms offered by Neverlang. The Try module (lines 1-32) declares a reference syntax for the try part of a **try-catch** block (lines 2-5), made of two production rules. The first is labeled "try_part" (line 3) and the second is labeled "protected" (line 4). The semantics are declared within a **role** block (lines 6-31) defining several semantic actions; each action is attached to a nonterminal of any of the productions of the reference syntax by referring to their label[3]—*e.g.*, the semantic action at line 7 refers to the production at line 3, whereas the semantic action at line 28 refers to the production at line 4. Nonterminals within a production are accessed using square brackets, in an array-like fashion as highlighted by the red arrows in Listing 2. Following the syntax directed translation technique [1], attributes are accessed from nonterminals by dot notation as done for retrieving the `section` attribute on line 10. Neverlang semantic actions are written in Java with some syntactic sugar. The semantic action at lines 7-27 retrieves the `Section` object from a child node (line 10), pushes it on the stack (line 14), and tries to execute it (line 15). A new exception is thrown (line 21) if any error occurs. Regardless of the result, the section is eventually popped from the stack (line 26).

Other capabilities. Neverlang supports composition between **module** units using other units called **slice** and **bundle**, hereby not shown for brevity. Neverlang **endemic slices** units can be used to declare instances that are globally accessible throughout all semantic actions within the language. E.g., lines 33-41 of Listing 2 declare several instances needed for the correct execution of JEL, one for each of its customizable elements, as discussed in Sect. 3.4. These instances can be accessed using the $$ operator, as done when throwing an exception on lines 21-24. Thanks to this mechanism, the exception handling process can be customized simply by swapping the `nl.jel.JELEndemic` endemic slice with a different endemic slice that re-declares the same instances by changing their class. Instead, the semantic action of module `nl.jel.JELTryStatement` remains unaltered. Modules, bundles, slices, and endemic slices are composed into a complete and executable language specification using the **language** unit, as done in lines 42-46. Neverlang supports language product line engineering [10, 11] through AiDE [34, 35, 49, 50], FeatureIDE [24], and the Gradle build tool.

4.2 JavaScript Evolution Scenario

We present a three-staged evolution for a JavaScript interpreter written in Neverlang [12] conform to the EcmaScript 3 specification, with the exception of part of the standard library. The variant V_1 is the base language with its own implementation of both the memory layout and of the exception handling. Variant V_2 removes exception handling support and replaces the original implementation of the memory layout with a JEL-based symbol table. Variant V_3 adds several exception handling language features on top of variant V_2.

[3]Neverlang also provides an alternative mechanism, based on absolute position of nonterminals within the reference syntax, not discuss for brevity.

Table 1. The effort to evolve JavaScript from V_1 to V_2 and from V_2 to V_3 wrt. the memory layout (Memory) and exception handling features (Exceptions). Data are collected by using the `svn diff` and the `diffstat` Linux commands.

Code edit	Evolution step effort		
	Change type	$V_1 \rightarrow V_2$	$V_2 \rightarrow V_3$
Files changed	**Total**	7	1
	Memory	6	0
	Exceptions	0	0
	Glue	1	1
Files added	**Total**	7	20
	Memory	5	0
	Exceptions	0	17
	Glue	2	3
Insertions (LoC)	**Total**	369	489
	To new files	292	485
	Memory	322	0
	Exceptions	0	448
	Glue	47	41
Deletions (LoC)	**Total**	71	0
	Memory	70	0
	Exceptions	0	0
	Glue	1	0

On both evolution steps, we measured the implementation effort, as summarized in Table 1. The first step is intended to measure the effort needed to render an existing language compliant to JEL and, by extension, to the EHL, thus answering RQ_1. The second evolution step aims at measuring the effort needed to implement varied exception handling language features in JEL, thus answering RQ_2. In both steps, we discuss how much this refactoring is affected by Neverlang, thus answering RQ_3. The resulting implementation of JavaScript V_3 is available on Zenodo.[4]

JavaScript V_1. JavaScript V_1 supports many of the most important features offered by the language, including (but not limited to):

— numeric, boolean, string and reference types;
— prototype-based classes and constructors;
— expressions between basic, reference and object types;
— if-else, switch, while, and for statements;
— standard output;
— functions declaration and invocation;
— throw, try, catch, and finally statements.

Most notably, JavaScript V_1 uses a custom memory layout based on a linked list; since this interpreter runs on the Neverlang runtime and therefore on JVM, it leverages the default exception handling mechanisms provided by Java to implement **`throw`** and **`try-catch`** statements. While this allows for a easy solution, the end result is hard to extend, due to Java not supporting unconventional exception handling language features by default.

[4]https://doi.org/10.5281/zenodo.8328246

```java
public class JSStack
    extends Stack<Long, String, JSReference>
    implements JSEnvironment.Instance<JSStack> {
  @Override
  public Class<JSStack> genericType() {
      return JSStack.class;
  }
}
```

Listing 3. Adapting generic JEL datatypes to the needs of a specific language interpreter.

Overall, JavaScript V_1 is comprised of 144 Neverlang units—for a total of 5,409 lines of code (LoC)—and 73 Java classes—for a total of 6,475 LoC: 318 LoC are needed to implement the memory layout, with an additional 1,983 LoC to represent types and variables within memory. 43 LoC are needed to implement the **`throw`** statement, and 135 LoC are needed to implement **`try`**, **`catch`**, and **`finally`**, with an additional 109 LoC to implement the errors of various types.

This is the baseline against which the following variants will be evaluated, to measure the effort of replacing this implementation with a JEL-based one.

JavaScript V_2. The JavaScript V_2 interpreter replaces the default implementation of the linked list symbol table provided by JavaScript V_1 with the default stack memory layout provided by JEL. Moreover, it removes any support for exception handling, meaning that JavaScript V_2 programs cannot throw nor catch any exceptions. To minimize the impact on the original code, the symbol table was implemented as an adapter [26]—dubbed `JELSymbolTable`—that extends the `LinkedListSymbolTable` class, so that the old implementation still works just by changing the runtime class of the symbol table. Then, calls to the `JELSymbolTable` are delegated to the actual JEL stack. As shown in Listing 3, the implementation of this stack is minimal because it does not provide any functionality, but simply specifies the generic types introduced in Sect. 3.4 according to data types needed by JavaScript, in particular:

— `SEC_ID` is instantiated to `Long`;
— `VAR_NAME_TYPE` is instantiated to `String`;
— `VAR_TYPE_TYPE` is instantiated to `JSReference`.

The `JSReference` class is particularly important in this context, because we could reuse most of the existing types with the new data structures. The `JSEnvironment.Instance` interface replaces the naïve singleton `LinkedListSymbolTable` instance with a more customizable alternative that allows instances of any subclass to be registered as the singleton.

Although limited, some modifications were required to change the original implementation of JavaScript V_1. Table 1 reports the effort required to make these changes. The refactoring required the modification of 7 files (6 Java classes and 1 Neverlang unit) and the creation of an additional 7 files (5 Java classes and 2 Neverlang units). All three Neverlang units are a form of glue code: we implemented a new language unit and two endemic slices, but no modules. This refactoring required 369 insertions and 71 deletions, for a total of 440 modifications; removing the existing implementation of the exception handling mechanisms required no modifications. Considering the size of the whole JavaScript V_1 project, changing the memory layout to support JEL required a modification of $440/(5,409 + 6,575) = 3.67\%$ of the project. Moreover, Table 1 shows that out of 369 insertions, 292 were made to newly created files, therefore changes to existing files are limited to 71 deletions and 77 insertions. In fact, using the `diffstat` command with the `-m` flag, reveals that the actual results are 306 insertions, 8 deletions and 63 modifications. We can now answer RQ_1.

> **How hard is it to refactor an existing language implementation so that it can be used in tandem with the exception handling layer?**

Refactoring an existing and full-fledged implementation of a language interpreter such as JavaScript V_1 so that it can be used in tandem with a EHL implementation for Java requires modifying about 3.67% of its code. We can conclude that an existing memory layout can be replaced with a memory layout based on EHL with limited effort, especially if part of the default implementation can be reused. Of course, different languages may require a different effort—*e.g.*, a smaller project may require more changes wrt. the project total size.

JavaScript V_3. The JavaScript V_3 interpreter adds several exception handling language features on top of JavaScript V_2. This includes the generic types specification for the JEL exception table, dispatcher, selector, finalizer and secure area according to the following types:

— `EX_ID` is instantiated to String;
— `PAYLOAD` is instantiated to `JSExceptionPayload`.

The remaining three generic types must conform to the memory layout definition and therefore they are the same used in JavaScript V_2. We used the default JEL implementation for all data structures and algorithms (akin to what shown in Listing 3), with the exception of the finalizer and the secure area. The finalizer stops the application, whereas the secure area holds a `JSExceptionPayload` that keeps the stack trace during the exception handling process. We also implemented the following exception handling language features: **`throw`** statement, **`retry`**, and **`resume`** statements, and **`try catch`** block, **`finally`** blocks. Most notably, **`throw`**, **`try`**, **`catch`**, and **`finally`** are fairly common exception handling language features, whereas **`retry`** and **`resume`** are rather unconventional. Both are resumption mechanisms that drive the execution of the program after running an exception handler. The **`retry`** (inspired by design by contract [40]) continues the execution from the first statement of the thrower and the **`resume`** (inspired by hardware pipelines) continues the execution from the next statement after the one causing the exceptional event. The implementation of the **`try`** block was already shown in Listing 2 and discussed in Sect. 4.1. We do not report the implementation of all other features for brevity and

```
module neverlang.js.jel.exceptions.JELStatementList {
  reference syntax from neverlang.js.JSStatementList
  role (evaluation) {
    s_list_0: .{
      ▶baseActionList;
      JSCompletionValue s = $s_list_0[0].cvalue;
      if (s.getType() == JSCVType.THROW)
        $$JELResumeArea.push($s_list_0[2]);
    }.
    s_list_1: .{ ▶baseAction; }.
  }
}
slice neverlang.js.jel.exceptions.JELResumeBlock {
  concrete syntax from neverlang.js.JSStatementList
  module neverlang.js.jel.exceptions.JELStatementList
    with role evaluation delegates {
      baseActionList ⇒
        neverlang.js.JSStatementList ▶ evaluation[0],
      baseAction ⇒
        neverlang.js.JSStatementList ▶ evaluation[3]
    }
}
module neverlang.js.jel.exceptions.JELResumeStatement {
  reference syntax {
    stat: Statement ⇽ ResumeStatement;
    resume: ResumeStatement ⇽ "resume" SemiColonOpt;
  }
  role(evaluation) {
    resume: .{
      ASTNode resume = $$JELResumeArea.peek();
      $ctx.eval(resume);
      $resume.cvalue = resume.getValue("cvalue");
    }.
  }
}
```

Listing 4. Throw statement using Neverlang and JEL.

```
var a = 1;
try {
  throw 1;
  a = a + 42;
} catch (x) {
  a = a + x;
  resume;
}
```

(a) JavaScript program using the **`resume`** statement.

```
var a = 1;
try {
  if ( a <= 10 )
    throw 1;
} catch (x) {
  a = a + x;
  retry;
}
```

(b) JavaScript program using the **`retry`** statement.

Listing 5. Unconventional exception handling language features in JavaScript V_3.

instead we focus on the most interesting aspects. With regards to the **`catch`** and **`finally`** blocks, both are registered as handlers for the corresponding protected section within the exception table, with the difference that the handler for the finally block is always executed, regardless of an exception being thrown or not. From an implementation standpoint, this was achieved by creating a composite [26] handler that runs both the catch part and the finally part when an exception is caught whereas only the finally part is executed if no exception occurs. To implement the **throw** statement in a way that also supports the **resume** statement, we leveraged the original implementation provided by the JavaScript V_1 interpreter: the node of the parse tree associated to each statement is assigned an attribute called `cvalue` that marks if that statement keeps the normal flow (`JSCVType.NORMAL`) or not, such as upon execution of a **break** or a **continue**. Similarly, when a throw statement is found, the exception is not thrown right away, rather the `cvalue` attribute is set to `JSCVType.THROW`. If that value is found when executing a statement, then all other statements within the same block are skipped. Listing 4 shows how to achieve reuse for this implementation. The `JELStatementList` imports its syntax from the JavaScript V_1 statement lists (line 2) and it overrides its semantics by leveraging the delegation operator [4] (lines 5 and 10). Upon encountering the delegation operator, the actual semantic actions to be executed are specified within the slice unit, as shown at lines 17 and 19. Therefore, the same code may use different delegates by using a different slice with a different **delegates** block. While the semantics of the semantic action labeled as `s_list_1` stay the same (it calls the delegate but it adds no code), the semantic action labeled as `s_list_0` is overridden. After the delegation, if the current statement of the list was a **throw** (line 7), then the parse tree node for the next statement is pushed to an endemic instance called `JELResumeArea`: this information is the resumption point of any **resume** statements within the exception handlers (line 8). This information is then retrieved (line 30) and executed (line 31) in the current context by the semantic action of the **resume** statement itself. Listing 5(a) shows a JavaScript V_3 program in which variable a evaluates to 44 because the execution of the **try** block is resumed after executing the exception handler. The **retry** statement leveraged a similar technique: the node associated to the first statement of a protected section is pushed to the `JELRetryArea` endemic instance and can be retrieved upon executing the **retry**. In Listing 5(b) the protected section is retried until $a > 10$. For both the `JELResumeArea` and the `JELRetryArea` the latest resumption point is popped upon exiting the protected section.

Table 1 reports the effort associated to the implementation of the language features hereby discussed. Overall, the implementation required the creation of 20 new files and the modification of just one existing file—*i.e.*, the Neverlang **language** unit was updated to include the new language features. Notice how no files had to be modified to achieve these results, therefore this evolution step required no deletions. This result is important because it shows that once the memory layout for JEL is in place, the exception handling language features can be implemented without further modifications to existing code. Moreover, out of the 489 insertions (only 81.6 LoC per exception handling language feature on average), none was used to add features to the memory layout–*e.g.*, by adding additional information within the sections on the stack—instead insertions were limited to the implementation of the language components and their interaction with JEL. We can now answer RQ_2 and RQ_3.

How hard is it to add exception handling support to a language implementation using the exception handling layer?

Adding an exception handling language feature to a language that uses a memory layout compliant to the EHL such as JavaScript V_2 took 81.5 LoC per feature on average. In total, we implemented 6 different language features: **try**, **catch**, and **finally** blocks, and **throw**, **retry**, and **resume** statements using 489 LoC and without affecting any of the pre-existing implementation, except for 4 lines of glue code. The total effort may increase if the interpreter needs to implement different exception handling mechanisms, such as a dispatcher different from the JEL default. However, such a change can also be achieved without changing the original code, but simply by adding pieces of glue code such as Neverlang endemic slices.

How much is the achieved modularization reliant on Neverlang-specific mechanisms?

In the first evolution step ($V_1 \longrightarrow V_2$), we achieved the refactoring by writing only 3 Neverlang units used as glue code. Most of the modifications involved the reliance of Java classes on the singleton instances; we refactored this into a more customizable mechanism using only Java, as exemplified in Listing 3. Thus, we believe that the same refactoring could be performed in other language workbenches with similar results. In the second evolution step ($V_2 \longrightarrow V_3$) we could add the exception handling features without changing the original code partly thanks to Neverlang features, especially the delegation operator shown in Listing 4. Although delegation can be used to compose semantic actions [4], in this context it was simply used as an overriding mechanism for semantic actions, a feature that is supported by most language workbenches such as MontiCore [29], MPS [7], Lisa [42], Spoofax [30], and Melange [20]. The same can also be achieved with aspect-oriented superimposition [36]. Similarly, Listing 2 shows the composition among JEL elements using the Neverlang endemic slice construct. However, the same result can be achieved with an additional layer between the semantic action and the JEL interface, as shown in Fig. 2 with the `Throw Exception` component: instead of performing the composition directly within Neverlang, the same composition could be performed within a Java class. We conclude that the reliance of our implementation on Neverlang was very limited and was a form of opportunistic reuse rather than an actual requirement.

4.3 LogLang Evolution Scenario

The EHL and JEL by extension are intended to be used across different languages with different characteristics. Since this model is meant to work in tandem with the modularization options offered by language workbenches, it is particularly

```
task SomeTask {
  backup "/foo/bar.txt" "/backup/bar.bak"
  remove "/foo/bar.txt"
}
```

(a) Exemplary task written in LogLang.

```
./gradlew runLogLang
> Task :runLogLang
executing task SomeTask
File ./foo/bar.txt does not exist, do you want to create it?
```

(b) LogLang finalizer in action.

Listing 6. LogLang with exception handling support.

relevant that it is compliant to the needs of domain-specific languages (DSL), that are typically the main output of language workbenches. To test this, we stretched JEL capabilities to implement a recovery procedure for the LogLang DSL, used for file system tasks declarations [13]. LogLang tasks are declarative, do not support any form of exception handling nor run on any memory layout. The DSL also does not include any language features to catch errors, nor to define handlers. In this case, the idea was to verify if the same model is applicable to a scenario in which most elements of the EHL can be omitted. To achieve this goal, we extended LogLang, so that LogLang tasks throw a JEL exception whenever the file on which the task must be performed does not exist. Such an example is shown in Listing 6. Since there are no memory, no sections, and no handlers, both the dispatcher and the selector always fail their search, therefore according to Listing 1, control is eventually taken by the finalizer. The finalizer is a custom procedure that prompts the users by asking them if they want to create the file. For instance, when running the task reported in Listing 6(a), file "foo/bar.txt" does not exist, therefore the user is prompted accordingly, as in Listing 6(b). The users can either accept or decline: in the former case the file is created and the task execution is resumed normally; in the latter the finalizer stops the application. This language evolution required the creation of just one class[5] of 70 LoC, added 1 LoC to three different Neverlang modules—*i.e.*, the code needed to throw the exception—and modified an endemic slice to include the JEL-related endemic instances. In total, the refactoring took 77 LoC. We can now answer RQ$_4$.

Does the conceptual framework support varied exception handling mechanisms and their language features?

Thanks to JEL we could implement varied exception handling language features. Some features were fairly traditional, such as the **`throw`** statement, and the **`try`**, **`catch`**, and **`finally`** blocks in JavaScript, while others are rather unconventional, such as the **`retry`** and **`resume`** statements. Most notably, these features are not supported by the exception handling mechanisms offered by the JVM, but they could be easily implemented with JEL in a modular way. Moreover, we applied the same architecture for the implementation of an unconventional recovery procedure for a DSL without any memory layout and that does not support any exception handling by default. We believe that these two evolution scenarios prove the applicability of the EHL to the creation of varied exception handling mechanisms and their features.

[5]For simplicity, we created only one class that implements the `Dispatcher`, `Selector`, and `Finalizer` interfaces at the same time.

4.4 Threats to Validity

External validity. The language evolution scenarios are based on Neverlang and Java and they may not be possible to reproduce it in different contexts. To prevent this issue, we implemented JEL without making any assumptions neither with regards to the base language nor to the language workbench. In particular, JEL does not rely on the Neverlang runtime and can be used by any program running on the JVM. We also tried to limit our reliance on Neverlang to perform the language evolution, as discussed in the answer to RQ$_3$. Similarly, JEL was created from scratch and does not rely on specific characteristics of Java to work. Even the infrastructure based on the five generic types is just a programmer convenience to improve error messages at compile time, since those types are affected by type erasure and do not exist at runtime. In summary, we believe that a similar library with data structures and algorithms compliant to the EHL could be implemented in any other general purpose programming language other than Java.

Construct validity. The answer to RQ$_1$ and RQ$_2$ is based on a specific evolution scenario, therefore a similar implementation of the EHL on other languages may require additional effort. To avoid this issue, we strictly defined the EHL first, then developed JEL following the EHL and finally performed the evolution experiment. We never reiterated on any prior step to accommodate the evolution experiment. Whenever a change was necessary to perform the evolution, it was made to the host language implementation and never to the library nor to the EHL therefore our measures should be able to represent the actual development effort.

Internal validity. We defined the EHL and answered RQ$_3$ and RQ$_4$ based on our experience with exception handling mechanisms and with language workbenches. This may cause an issue due to exotic exception handling mechanisms and language workbenches we may be unaware of. To prevent this issue, we did not make any assumptions on the language, even allowing for arbitrary memory layouts; we also tested an unconventional exception handling mechanism in which most of the elements of the EHL are optional. This convinced us of the generality of our architecture, as well as of its implementation.

5 Related Work

Modular language development is a popular research topic. Many language workbenches have been proposed, each with its own take on language composition. Their contribution is related to ours due to their focus on providing ways to avoid the monolithic approach to language implementation. Melange [20] integrates tools from the Eclipse Modeling Framework (EMF) ecosystem [46]and supports language extension and language merge [20]. Meta Programming System (MPS) [52] is a development environment for non-textual DSLs based on projectional editing [53] using concepts (abstract syntax nodes) and behaviors (semantics). MontiCore [33] generates abstract data types for the parse tree and uses Java visitors for the semantics. It supports reuse through the extension of abstract data types and grammar inheritance. Rascal [31] is a meta-programming language that supports the implosion of parsed text and parse tree transformations. The evaluation leverages the pattern-based dispatch technique [3]. Spoofax [54] provides several DSLs for language development; the semantics are called rules and strategies and can be defined as a sequence of functions over the AST. To the best of our knowledge, there are no contributions using language workbenches to directly address the exception handling mechanism and its portability, but, as we discussed in Sect. 4.2, we believe that the EHL is applicable to all aforementioned language workbenches and Java-based ones could even use the JEL library.

Development of crosscutting features such as exception handling have been discussed mainly with regards to aspect-oriented programming. For instance, Liebig *et al.* use the superimposition operator to handle crosscutting features in Mobl [37]. Hadas and Lorenz switch the perspective by introducing language oriented modularity [28]: instead of tackling the problem of crosscutting features in languages, they leverage the ease of use of language workbenches to create several DSLs, each tackling a different crosscutting concern in other systems. However, interactions among language-based tools are hard to understand without good integration [5]; compared to the EHL, their work is not applicable to the definition of modular exception handling language features in a unique language.

On the topic of exception handling, some contributions focus on exception handling in management systems: Chiu *et al.* [17] address the importance of reusing exception handlers to deal with workflow exceptions and propose the ADOME exception handling environment for the definition of dynamic bindings for exception handlers, run-time modifications of exception handlers and exception handler reuse. Similarly, the VIEW scientific workflow management system provides customizable and hierarchical exception handlers; the authors also propose a language for user-defined exception handling mechanisms [45]. Celovic and Soukouti [15] describe the proper use of exception handlers for the development of large scale enterprise systems. In their work, they defined six groups of responsibilities, including the thrower and the catcher; our conceptual framework is similar and reflects these responsibilities. In all these cases, the applicability to traditional programming languages is not discussed. The contribution from Ogasawara *et al.* [43] addresses on the optimization of stack unwinding and stack cutting in Java and could be used to create an optimized version of the exception handling layer. More in general, using an intermediate layer to abstract the memory layout and exception handling introduces an overhead that requires optimizations such as by limiting the costs of metaprogramming capabilities used by the language workbench [39] and using optimized AST interpreters with partial evaluation [38].

Cabral and Marques implement retry semantics on languages lacking this language feature using aspect-oriented programming [6]. Bagge *et al.* [2] present a layer that can be used on top of any platform-specific error reporting to generalize error reporting and handling through the alert concept. The proposed implementation also supports retry semantics, but it is implemented as an extension to the C language, therefore replication in other languages requires the development of a similar extension. Chase [16] also discussed exception handling in C, although his remarks are general enough to be valid for any language. Chase observes that the exception handling mechanism should be smoothly integrated with the rest of the host programming language, but the contribution focuses on low level details instead of defining a general and abstract framework such as the EHL.

In their contribution, Brinke *et al.* [47] discuss a tailorable control flow, including exceptional flow. In their view, all exception handling mechanisms should be supported within the same language and application programmers should be able to choose which kind of exception handling mechanism they want to use. Their work is closely related to ours, since they propose an intermediate layer to customize exception handling, but the code is written using continuations, thus compared to the EHL the base language must support first-order functions and the resulting code may be less readable.

6 Conclusions

Exception handling is a collection of language features whose implementation is usually hard to reuse because scattered across several parts of the implementation. The EHL framework permits to untangle the code of the exception handling language features from the code of other language features. The EHL architecture is very flexible, allows for arbitrary memory layouts, dispatching algorithms and handling procedures, and most of its elements are optional. We proved the EHL applicability by developing the JEL library and using it to add exception support to a full-fledged implementation of JavaScript without changing its implementation. Our experience shows that several exception handling language features—both conventional and unconventional—can be achieved with an high degree of modularity and with limited development effort.

Acknowledgments

This work was partially funded by the MUR project "T-LADIES" (PRIN 2020TL3X8X).

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (second ed.). Addison-Wesley, Boston, MA, USA.

[2] Anya Helene Bagge, David Valentin, Magne Haveraaen, and Karl Trygve Kalleberg. 2006. Stayin' Alert:: Moulding Failure and Exceptions to Your Needs. In *GPCE'06*. ACM, Portland, OR, USA, 265–274.

[3] Bas Basten, Jeroen van den Bos, Mark Hills, Paul Klint, Arnold Lankamp, Bert Lisser, Atze van der Ploeg, Tijs van der Storm, and Jurgen Vinju. 2015. Modular Language Implementation in Rascal—Experience Report. *Science of Computer Programming* 114 (Dec. 2015), 7–19.

[4] Francesco Bertolotti, Walter Cazzola, and Luca Favalli. 2023. On the Granularity of Linguistic Reuse. *Journal of Systems and Software* 202 (Aug. 2023). https://doi.org/10.1016/j.jss.2023.111704

[5] Barret Bryant, Jean-Marc Jézéquel, Ralf Lämmel, Marjan Mernik, Martin Schindler, Friedrich Steinmann, Juha-Pekka Tolvanen, Antonio Vallecillo, and Markus Völter. 2015. Globalized Domain Specific Language Engineering. In *Globalizing Domain-Specific Languages (Lecture Notes in Computer Science 9400)*, Benoît Combemale, Betty H.C. Cheng, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe (Eds.). Springer, 43–69.

[6] Bruno Cabral and Paulo Marques. 2009. Implementing Retry—Featuring AOP. In *LADC'09*. IEEE, João Pessoa, Brazil, 73–80.

[7] Fabien Campagne. 2016. *The MPS Language Workbench*. Vol. 1. CreateSpace Independent Publishing.

[8] Walter Cazzola. 2012. Domain-Specific Languages in Few Steps: The Neverlang Approach. In *SC'12 (Lecture Notes in Computer Science 7306)*. Springer, Prague, Czech Republic, 162–177.

[9] Walter Cazzola, Ruzanna Chitchyan, Awais Rashid, and Albert Shaqiri. 2018. μ-DSU: A Micro-Language Based Approach to Dynamic Software Updating. *Computer Languages, Systems & Structures* 51 (Jan. 2018), 71–89. https://doi.org/10.1016/j.cl.2017.07.003

[10] Walter Cazzola and Luca Favalli. 2022. Towards a Recipe for Language Decomposition: Quality Assessment of Language Product Lines. *Empirical Software Engineering* 27, 4 (April 2022). https://doi.org/10.1007/s10664-021-10074-6

[11] Walter Cazzola and Luca Favalli. 2023. Scrambled Features for Breakfast: Concept, and Practice of Agile Language Development. *Commun. ACM* (Nov. 2023).

[12] Walter Cazzola and Diego Mathias Olivares. 2016. Gradually Learning Programming Supported by a Growable Programming Language. *IEEE Transactions on Emerging Topics in Computing* 4, 3 (Sept. 2016), 404–415. https://doi.org/10.1109/TETC.2015.2446192

[13] Walter Cazzola and Davide Poletti. 2010. DSL Evolution through Composition. In *RAM-SE'10*. ACM, Maribor, Slovenia.

[14] Walter Cazzola and Edoardo Vacchi. 2013. Neverlang 2: Componentised Language Development for the JVM. In *SC'13 (Lecture Notes in Computer Science 8088)*. Springer, Budapest, Hungary, 17–32.

[15] Dino Celovic and Nader Soukouti. 2004. *About Effective Exception Handling*. White Paper. Sanabel Solutions.

[16] David Chase. 1994. Implementation of Exception Handling. *The Journal of C Language Translation* 5, 4 (June 1994), 229–240.

[17] Dickson Chiu, Qing Li, and Kamalakar Karlapalem. 2000. A Logical Framework for Exception Handling in ADOME Workflow Management System. In *CAiSE'00 (Lecture Notes in Computer Science 1789)*. Springer, Stockholm, Sweden, 110–125.

[18] Roberta Coelho, Lucas Almeida, Georgios Gousios, Arie van Deursen, and Christoph Treude. 2017. Exception Handling Bug Hazards in Android. *Empirical Software Engineering* 22 (June 2017), 1264–1304.

[19] Adrian Colyer, Awais Rashid, and Gordon Blair. 2004. *On the Separation of Concerns in Program Families*. Technical Report 107. Lancaster University, Lancaster, United Kingdom.

[20] Thomas Degueule, Benoît Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. 2015. Melange: a Meta-Language for Modular and Reusable Development of DSLs. In *SLE'15*. ACM, Pittsburgh, PA, USA, 25–36.

[21] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. 2012. Language Composition Untangled. In *LDTA'12*. ACM, Tallinn, Estonia.

[22] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerrtsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, and Eelco Visser. 2013. The State of the Art in Language Workbenches. In *SLE'13 (Lecture Notes on Computer Science 8225)*. Springer, Indianapolis, USA, 197–217.

[23] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Alex Kelly, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2015. Evaluating and Comparing Language Workbenches: Existing Results and Benchmarks for the Future. *Computer Languages, Systems and Structures* 44 (Dec. 2015), 24–47.

[24] Luca Favalli, Thomas Kühn, and Walter Cazzola. 2020. Neverlang and FeatureIDE Just Married: Integrated Language Product Line Development Environment. In *SPLC'20*. ACM, Montréal, Canada, 285–295.

[25] Martin Fowler. 2005. Language Workbenches: The Killer-App for Domain Specific Languages? Martin Fowler's Blog. http://www.martinfowler.com/articles/languageWorkbench.html

[26] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Ma, USA.

[27] John B. Goodenough. 1975. Exception Handling: Issues and a Proposed Notation. *Commun. ACM* 18, 12 (Dec. 1975), 683–696.

[28] Arik Hadas and David H. Lorenz. 2016. Toward Pratical Language Oriented Modularity. In *Modularity'16*. ACM, Málaga, Spain, 94–98.

[29] Robert Heim, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Andreas Wortmann. 2016. Compositional Language Engineering Using Generated, Extensible, Static Type-Safe Visitors. In *ECMFA'16 (Lecture Notes in Computer Science 9764)*. Springer, Vienna, Austria, 67–92.

[30] Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *OOPSLA'10*. ACM, Reno, Nevada, USA, 444–463.

[31] Paul Klint, Tijs van der Storm, and Jurgen Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *SCAM'09*. IEEE, Edmonton, Canada, 168–177.

[32] Jan Kort and Ralf Lämmel. 2003. Parse-Tree Annotations Meet Re-Engineering Concerns. In *SCAM'03*. IEEE, Amsterdam, The Netherlands, 161–170.

[33] Holger Krahn, Bernhard Rumpe, and Steven Völkel. 2010. MontiCore: A Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer* 12, 5 (Sept. 2010), 353–372.

[34] Thomas Kühn and Walter Cazzola. 2016. Apples and Oranges: Comparing Top-Down and Bottom-Up Language Product Lines. In *SPLC'16*. ACM, Beijing, China, 50–59.

[35] Thomas Kühn, Walter Cazzola, and Diego Mathias Olivares. 2015. Choosy and Picky: Configuration of Language Product Lines. In *SPLC'15*. ACM, Nashville, TN, USA, 71–80.

[36] Ralf Lämmel. 2003. Adding Superimposition to a Language Semantics. In *FOAL'03*. Boston, MA, USA, 65–70.

[37] Jörg Liebig, Rolf Daniel, and Sven Apel. 2013. Feature-Oriented Language Families: A Case Study. In *VaMoS'13*. ACM, Pisa, Italy.

[38] Stefan Marr and Stéphane Ducasse. 2015. Tracing vs. Partial Evaluation: Comparing Meta-Compilation Approaches for Self-Optimizing Interpreters. In *OOPSLA'15*. ACM, Pittsburgh, PA, USA, 821–839.

[39] Stefan Marr, Chris Seaton, and Stéphane Ducasse. 2015. Zero-Overhead Metaprogramming: Reflection and Metaobject Protocols Fast and without Compromises. In *PLDI'15*. Portland, OR, USA.

[40] Bertrand Mayer. 1992. Applying 'Design by Contract'. *Computer* 25, 10 (Oct. 1992), 40–51.

[41] David Méndez-Acuña, José A. Galindo, Thomas Degueule, Benoît Combemale, and Benoît Baudry. 2016. Leveraging Software Product Lines Engineering in the Development of External DSLs: A Systematic Literature Review. *Computer Languages, Systems & Structures* 46 (Nov. 2016), 206–235.

[42] Marjan Mernik. 2013. An Object-Oriented Approach to Language Compositions for Software Language Engineering. *Journal of Systems and Software* 86, 9 (Sept. 2013), 2451–2464.

[43] Takeshi Ogasawara, Hideaki Komatsu, and Toshio Nakatani. 2001. A Study of Exception Handling and Its Dynamic Optimization in Java. *Sigplan Notices* 36, 11 (Oct. 2001), 83–95.

[44] Guilherme de Pádua and Weiyi Shang. 2018. Studying the Relationship between Exception Handling Practices and Post-Release Defects. In *MSR'18*. ACM, Gothenburg Sweden, 564–575.

[45] Dong Ruan, Shiyong Lu, Aravind Mohan, Xubo Fei, and Jia Zhang. 2012. A User-Defined Exception Handling Framework in the VIEW Scientific Workflow Management System. In *SC'12*. IEEE, Honolulu, Hawaii, USA, 274–281.

[46] Dave Steinberg, Dave Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modeling Framework*. Addison-Wesley.

[47] Steven te Brinke, Mark Laarakkers, Christoph Bockisch, and Lodewijk Bergmans. 2012. An Implementation Mechanism for Tailorable Exceptional Flow. In *WEH'12*. Zürich, Switzerland, 22–26.

[48] Edoardo Vacchi and Walter Cazzola. 2015. Neverlang: A Framework for Feature-Oriented Language Development. *Computer Languages, Systems & Structures* 43, 3 (Oct. 2015), 1–40. https://doi.org/10.1016/j.cl.2015.02.001

[49] Edoardo Vacchi, Walter Cazzola, Benoît Combemale, and Mathieu Acher. 2014. Automating Variability Model Inference for Component-Based Language Implementations. In *SPLC'14*. ACM, Florence, Italy, 167–176.

[50] Edoardo Vacchi, Walter Cazzola, Suresh Pillay, and Benoît Combemale. 2013. Variability Support in Domain-Specific Language Development. In *SLE'13 (Lecture Notes on Computer Science 8225)*. Springer, Indianapolis, USA, 76–95.

[51] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. 2010. Silver: an Extensible Attribute Grammar System. *Science of Computer Programming* 75, 1-2 (Jan. 2010), 39–54.

[52] Markus Völter and Vaclav Pech. 2012. Language Modularity with the MPS Language Workbench. In *ICSE'12*. IEEE, Zürich, Switzerland, 1449–1450.

[53] Markus Völter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. 2014. Towards User-Friendly Projectional Editors. In *SLE'14 (Lecture Notes in Computer Science Volume 8706)*. Springer, Västerås, Sweden, 41–61.

[54] Guido H. Wachsmuth, Gabriël D. P. Konat, and Eelco Visser. 2014. Language Design with the Spoofax Language Workbench. *IEEE Software* 31, 5 (Sept./Oct. 2014), 35–43.

[55] Martin P. Ward. 1994. Language Oriented Programming. *Software—Concept and Tools* 15, 4 (Oct. 1994), 147–161.

Received 2023-07-07; accepted 2023-09-01

An Executable Semantics for Faster Development of Optimizing Python Compilers

Olivier Melançon
olivier.melancon.1@umontreal.ca
Université de Montréal
Montréal, Canada

Marc Feeley
feeley@iro.umontreal.ca
Université de Montréal
Montréal, Canada

Manuel Serrano
Manuel.Serrano@inria.fr
Inria/Université Côte d'Azur
Sophia Antipolis, France

Abstract

Python is a popular programming language whose performance is known to be uncompetitive in comparison to static languages such as C. Although significant efforts have already accelerated implementations of the language, more efficient ones are still required. The development of such optimized implementations is nevertheless hampered by its complex semantics and the lack of an official formal semantics. We address this issue by presenting an approach to define an executable semantics targeting the development of optimizing compilers. This executable semantics is written in a format that highlights type checks, primitive values boxing and unboxing, and function calls, which are all known sources of overhead. We also present semPy, a partial evaluator of our executable semantics that can be used to remove redundant operations when evaluating arithmetic operators. Finally, we present Zipi, a Python optimizing compiler prototype developed with the aid of semPy. On some tasks, Zipi displays performance competitive with that of state-of-the-art Python implementations.

CCS Concepts: • **Software and its engineering** → **Translator writing systems and compiler generators**.

Keywords: compiler, dynamic programming language, optimization, executable semantics, partial evaluation, python

ACM Reference Format:
Olivier Melançon, Marc Feeley, and Manuel Serrano. 2023. An Executable Semantics for Faster Development of Optimizing Python Compilers. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 14 pages.
https://doi.org/10.1145/3623476.3623529

1 Introduction

Python is a dynamic language known for its extensive standard library, object-oriented approach and admittedly poor performance in comparison to static languages such as C and dynamic languages such as JavaScript [12]. It is nonetheless among the most popular languages in use today and its popularity shows no sign of decline [32].

The Python language specification is *The Python Language Reference* [31]. While the syntax is formally specified, this is not the case of the semantics, leaving room for ambiguities and making it difficult to reason about programs [21].

In our experience, these challenges are related. Python's complex semantics and absence of formal specification complicate the development of a compiler compatible with CPython, the reference implementation. The effort spent getting the semantics right leaves little time for optimization.

Furthermore, part of the semantics makes Python implementations susceptible to execute redundant type checks, extensive boxing and unboxing of primitive values, and abundant method calls, which affects performance [14, 34]. Optimization of operations on atomic types (such as `int` and `float`) has been suggested to resolve this issue [34].

This paper offers two main contributions. First, we describe an executable semantics for Python that is written in a Python syntax to allow reuse in existing compilers. Second, we present a tool that applies partial evaluation to remove redundant type checks, boxing and unboxing, and method calls from arithmetic operations on atomic types. We use this executable semantics to automate the implementation of arithmetic operations in an optimizing compiler and demonstrate that it provides run time performance competitive with those of PyPy [5], a state-of-the-art Python implementation.

This paper is organized as follows. In Section 2, we provide an overview of Python's semantics. In Section 3, we define an executable semantics that describes the behavior of various Python operations. In sections 4 and 5, we present a technique for partial evaluation of our executable semantics that focuses on removing redundant type checks, boxing and unboxing, and method lookups and invocations from Python operations. In Section 6, we show how we reused our executable semantics in the implementation of Zipi, our partial implementation of a Python optimizing compiler. Finally, in Section 7 we provide an overview of performance.

2 Overview of Python's Semantics

Python's semantics is highly dynamic. This is an obstacle to the implementation of an optimizing compiler. This section gives an overview of this problem.

As of today, Python does not have an official formal semantics. The reference manual [31] uses prose instead of a formal specification, which leaves room for ambiguities. When such ambiguities arise, we refer to the behavior of CPython [28], the reference implementation.

2.1 Data Model

Python's abstraction for data are *objects*, which are entirely defined by their *identity*, *type* and *value*. The *identity* is a unique integer value that never changes across the life of the object and is available by calling `id(obj)`. The *value* is the data represented by the object, for example an integer, a floating point number or a pointer to another data structure. Finally, the *type* determines the operations allowed on the object. An object's type is itself an object that can be obtained with `type(obj)`. Under certain conditions, an object's type can be modified. However, this is not possible for objects whose type is a built-in type such as booleans, floats, integers, strings, lists, tuples, sets and dictionaries.

All values in a Python program are objects. For example, Python boolean values are represented by the singleton objects `True` and `False`, which belong to the `bool` type, a subtype of the `int` type. This contrasts with other object-oriented languages such as JavaScript where primitive data types such as `number` and `boolean` exist [20]. In the absence of primitive values, the operations allowed on an object are defined entirely by the methods available on its type. Such methods governing operations are called *magic methods*.

Figure 1 shows the semantics of the operation `(x + y)`. It attempts to invoke the `__add__` magic method of `x`'s type. If the `__add__` method is found, it is invoked with `x` and `y` as arguments and returns the result of `(x + y)`. If `__add__` is not found, or if it is found but returns the special singleton object `NotImplemented`, addition falls back on the `__radd__` method of the type of `y` (the `r` in the name stands for *reflected*). Otherwise, it raises a `TypeError` exception with an explicative message (which is omitted for brevity).

The semantics of other arithmetic operations are similar, only the names of the required methods change. For example, the semantics of the subtraction operator is identical to that of addition, but calls the methods `__sub__` and `__rsub__`.

The only operators that cannot be overloaded are the "`is`" operator, which compares objects by identity, and the "`and`", "`or`" and "`not`" boolean operators. All other operations are governed by magic methods. For example, iteration in a `for`-loop calls the `__iter__` method, which returns an iterator. The syntactical form `obj.attr` for attribute access calls the `__getattribute__` method. The same applies for function invocation, truthiness, type casting and so on.

```
py_add(x, y): # semantics of x + y
  if type(x) has a method __add__:
    result = type(x).__add__(x, y)
    if result is the object NotImplemented:
      return py_radd(y, x)
    else:
      return result
  else:
    return py_radd(y, x)

py_radd(y, x): # reflected addition
  if type(y) has a method __radd__:
    result = type(y).__radd__(y, x)
    if result is the object NotImplemented:
      raise TypeError
    else:
      return result
  else:
    raise TypeError
```

Figure 1. Pseudocode for the semantics of the + operator

2.2 Method Resolution Order

The Python language supports multiple inheritance. Inheritance expands the features of a type by enabling it to access its parents' magic methods. When recovering a method on a type, such as the `__add__` method in Figure 1, Python executes an ordered search across the type and its parents. The expression `type(x).__add__` first looks for `__add__` on `type(x)` itself. If no such method is found, it is looked up recursively on the parents of `type(x)`.

To avoid inconsistencies in the context of multiple inheritance, searching for a magic method (or any attribute) requires an order in which to traverse the parents, called the *method resolution order* (MRO). The MRO is a property of a type computed at the creation of the `type` object by using the *C3 superclass linearization* algorithm [23]. The MRO cannot be altered afterward, but the types contained in the MRO are often mutable. Their attributes may be updated, new ones may be introduced, or existing ones may be removed. This prevents determining a result of the lookup of each magic method at the creation of a type.

An important exception is that attributes of all built-in types are read-only. That is the case both in CPython and PyPy [27], another popular implementation of the language. Immutability of built-in types is part of Python's semantics.

2.3 Dynamic Environments and Attributes

Python incorporates features such as dynamic typing, late binding, and dynamic code evaluation. It also offers a deep level of introspection that allows altering the behavior of a program in ways that a compiler can hardly predict through static analysis [17].

Python supports modular programming through `module` objects. No syntactical distinction is made between a code file intended to be run directly and one intended to be imported. When a file is executed, an object of type `module` is created. All global assignments executed in its code are stored as attributes of the module. Conversely, any modifications applied to a module's attributes is reflected on its global environment. Other Python programs can then import this module to access and also update its attributes.

Python also allows the global scopes of its modules to be reified by invoking the `globals()` built-in function. This function returns a dictionary (a hash table) that allows the global environment to be read and written. Since the returned dictionary is a Python object, any program can keep a reference to it and update it. The prospect of dynamically loaded code updating the environment at any point of the execution always remains. This makes static analysis of global variables impracticable.

2.4 Dynamic Type Checks

In Figure 1, we showed that two objects can be added if the left-hand operand's type has a `__add__` method that does not return `NotImplemented` for the given right-hand operand. This process requires two forms of type checks[1]: (1) looking up whether the left-hand operand's type has an `__add__` method and (2) checking if the type of the right-hand operand is suitable for the operation. No explicit type check is required on the left-hand operand since the method `__add__` was recovered on its type. This ensures that the magic method receives a first argument of a suitable type.

However, magic methods are not private and can be called with unexpected arguments. Figure 2 shows the result of directly calling `int.__sub__` and `float.__rsub__` with various arguments. The first call returns the expected result of the operation `(43 - 1)`. The second call shows that `int.__sub__` does not know how to handle an argument of type `float`. Subtraction between an `int` and a `float` is handled by `float.__rsub__` (third call). The last call shows that if the first argument of `int.__sub__` is not an `int` then a `TypeError` is raised. We conclude that the `int.__sub__` method contains a type check of its first argument. When computing the result of the "-" operator, this check is redundant since the left-hand operand is already known to be an instance of `int`. We observed similar behaviors for other built-in magic methods.

2.5 Sources of Overhead

The features presented in this section explain the poor performance of a naive implementation of Python. An operation as simple as subtracting an integer and a floating point number requires two method searches in the MRO of `int` and `float` respectively. Both methods are called due to the first returning `NotImplemented`. In both cases, the magic methods apply a redundant type check on their first argument.

In particular, implementing number arithmetic with method calls introduces a major overhead on operations that could otherwise be computed with a single assembly instruction as C would do. In the case of CPython, function and method calls are the primary source of overhead [34].

Furthermore, in the context of arithmetic operations, magic methods are required to extract the values from `int` and `float` objects and generate a new object to store the result. This procedure, known as *boxing and unboxing*, leads to additional overhead [14].

```
>>> int.__sub__(43, 1)
42
>>> int.__sub__(43, 1.0)
NotImplemented
>>> float.__rsub__(1.0, 43)
42.0
>>> int.__sub__(43.0, 1.0)
TypeError: descriptor '__sub__' requires a
          'int' object but received a 'float'
```

Figure 2. Results of direct calls to magic methods in CPython

3 Executable Semantics for Python

We now present an executable semantics aimed at developing optimizing Python compilers. Our goal is for such a formalization to (1) automate the implementation of a Python compiler, (2) be easily reusable by existing Python compilers and (3) yield performant implementations.

Writing the numerous magic methods of Python's built-in types by hand is tedious and error-prone. We ought to automate this process to accelerate development, including that of existing compilers, independently of the language and tools chosen for its implementation. We achieve this by writing the semantics in the syntax of Python. Hence, it is possible to interface with the semantics by using the parsing infrastructure of an existing compiler.

Our strategy is similar to that of RPython, which implements a subset of Python with limited dynamic features [1]. It differs in that we instead use a superset of Python to highlight parts of the semantics causing overhead such as boxing and unboxing of primitive values, type checking, and method calls. In this section, we introduce this superset of Python and use it to write an executable semantics. We will show how this semantics can be read by a compiler to implement optimized versions of various operators in sections 5 and 6.

[1]We employ *type check* in a broad sense to refer to any operation that requires a test on the type of an object, including magic method dispatches.

3.1 The Compiler Intrinsics Statement

To express the semantics of operators, we extend Python with the *compiler intrinsics* statement. Its syntax is the same as that of an `import` statement, except that the module name must be `__compiler_intrinsics__` followed by a sequence of names. The imported names correspond to low-level primitives that we call *intrinsics* (we detail all intrisincs in Appendix A). Intrinsics imported with the compiler intrinsics statement are static, they cannot be shadowed by another assignment or assigned to a variable. Since the compiler intrinsics statement reuses the syntax of Python's `import`, its implementation requires no change to the parser.

The compiler intrinsics statement has been sufficient to implement all arithmetic operators, unary operators, comparison operators, truthiness, length, type casts, attribute access and assignment, subscript access and assignment, and context managers [15]. These operators are magic-method-dependent, which the compiler intrinsics statement is well suited to implement. We have yet to extend our executable semantics to describe control flow, scoping rules and other features that do not rely on magic methods.

In sections 3.2 and 3.3, we provide two examples of operators for which an executable semantics can be written with the compiler intrinsics statement: addition and truthiness. These examples effectively illustrate why seemingly simple operations incur a significant overhead.

3.2 Example: Semantics of Addition

In Figure 3, we translate the addition semantics from Figure 1. We import three intrinsics: (1) `define_semantics`, which indicates that a decorated function is not a Python function, but rather the definition of an operator's semantics, (2) `class_getattr`, which implements the MRO lookup of a magic method, and (3) `absent`, a sentinel value returned by `class_getattr` if no corresponding magic method is found.

The addition semantics in Figure 3 defines the nested function `normal` (line 6) and `reflected` (line 17). Since those are in the scope of a `define_semantics`, the compiler can avoid the allocation of `function` objects and define low-level procedures instead. It is also possible to apply *lambda-lifting* to prevent the creation of closures capturing 'x' and 'y'. All arithmetic operators can be defined in similar fashion.

When a function is decorated with `define_semantics`, we refer to it as *a semantics* or the *semantics of* a given operator.

```
 1 from __compiler_intrinsics__ \
 2 import class_getattr, define_semantics, absent
 3
 4 @define_semantics
 5 def add(x, y):
 6   def normal():
 7     magic_method = class_getattr(x, "__add__")
 8     if magic_method is absent:
 9       return reflected()
10     else:
11       result = magic_method(x, y)
12       if result is NotImplemented:
13         return reflected()
14       else:
15         return result
16
17   def reflected():
18     magic_method = class_getattr(y, "__radd__")
19     if magic_method is absent:
20       raise TypeError
21     else:
22       result = magic_method(y, x)
23       if result is NotImplemented:
24         raise TypeError
25       else:
26         return result
27
28   return normal()
```

Figure 3. Semantics of the + operator written with the compiler intrinsics statement

3.3 Example: Semantics of Truthiness

An object's truthiness is computed when it is used as the condition of an `if` statement or `while` statement, or if converted to a boolean using `bool(x)`. Objects considered to be falsy include `False`, `None`, zeros of numeric types and empty sequences (e.g., an empty list or string).

The operation of truthiness is especially convoluted since it falls back on recovering the length of objects whose type does not have a `__bool__` magic method. Computing the length of an object has its own semantics, which must assert that the resulting length is a *small* integer.[2] In Figure 4, we implement the truthiness operation. It attempts to call the `__bool__` method, but may fall back on the `maybe_length` operation. The latter computes the length of an object, but returns `absent` if the object's type does not have a `__len__` method (in which case the object is always truthy).

The `maybe_length` semantics must assert that the computed length is a small integer. This operation is implemented by the `index` semantics in Figure 5. To abstract the notion of *small* integers, we introduce two intrinsic types: `sint` and `bint`, which respectively stand for small and big integer. Those are abstract subtypes of `int` that differentiate between small and big integers using the `isinstance` built-in function (fig. 5, lines 12 and 14) while leaving room for implementation-dependent details. The result of the `index` semantics is returned and compared to zero. The object is truthy only if its length is non-zero.

[2] *Small* is implementation-dependent, but typically means an integer that fits in a machine word.

```python
@define_semantics
def truth(obj):
  magic_method = class_getattr(obj, "__bool__")
  if magic_method is not absent:
    result = magic_method(obj)
    if type(result) is bool:
      return result
    else:
      raise TypeError
  else:
    len_result = maybe_length(obj)
    if len_result is not absent:
      return len_result != 0
    else:
      return True

@define_semantics
def maybe_length(obj):
  magic_method = class_getattr(obj, "__len__")
  if magic_method is absent:
    return absent
  else:
    len_result = magic_method(obj)
    index_result = index(len_result)
    if index_result < 0:
      raise ValueError
    else:
      return index_result
```

Figure 4. Semantics of computing the truthiness of an object

```python
from __compiler_intrinsics__ \
import class_getattr, define_semantics, absent, \
       sint, bint

@define_semantics
def index(obj):
  magic_method = class_getattr(obj, "__index__")
  if magic_method is absent:
    raise TypeError
  else:
    result = magic_method(obj)
    if isinstance(result, sint):
      return result
    elif isinstance(result, bint):
      raise OverflowError
    else:
      raise TypeError
```

Figure 5. Semantics of casting an object to an index-sized integer

3.4 Magic Methods

We cannot fully describe Python's semantics without describing the magic methods of its built-in types. For instance, the add semantics from Figure 3 fails to predict the specific result of the expression `(41 + 1.0)`. In this section, we introduce intrinsics to describe magic methods.

Applying an operation requires boxing and unboxing objects' values. An unboxed value is not a Python object. Its exact format depends on the *host* language used by a compiler, we thus call it a *host value*. To write magic methods, we need to express how host values are manipulated. Therefore, we introduce a family of intrinsic functions that are named `X_to_host` and `X_from_host`.

The intrinsic function `X_to_host` takes a single argument of type `X` and returns the host value of that argument. For example, the expression `int_to_host(42)` returns the numerical representation of 42 in the host language. If the argument is not an instance of `X`, then the behavior of the function is undefined.

The intrinsic function `X_from_host` is the inverse of `X_to_host`. It takes a host value as argument and returns an object of type `X` that encapsulates this value. While `X` could be any built-in type, we limit ourselves to numerical types such as `int` and `float` for now.

We also introduce the `builtin` intrinsic, which is similar to the `define_semantics` decorator. It is used as a *class decorator* and indicates that a given class definition is the definition of the corresponding built-in type.

In Figure 6, we use these new intrinsics to implement the `__add__` and `__floordiv__` (floor division) magic methods of `int`. Notice the redundant type check of both methods on lines 7 and 18 (yet, they are necessary when calling a magic method directly). In the case of `__floordiv__`, we also check that there is no division by zero on line 20. These magic methods introduce arithmetic operations in the host languages. In the expressions on lines 10, 20 and 22, the left-hand and right-hand sides are all host values. However, the usage of `int_to_host` can be detected statically, allowing to generate code for host integers addition. Throughout the remainder of this paper, examples will frequently show overloading of operators to execute arithmetic in the host language.

Magic methods defining the behavior of arithmetic operators are numerous, but they can be generated from templates to automate writing down the executable semantics [15].

For non-numerical types, it is straightforward to extend our pool of intrinsics to manipulate other types of host values. For example, we introduce the `str_len_to_host` intrinsic function, which takes a Python string as argument and returns a host integer representing its length. In Figure 7, we use it to implement the `__len__` method of `str` (string type).

```python
from __compiler_intrinsics__ \
import builtin, int_from_host, int_to_host

@builtin
class int:
  def __add__(self, other):
    if isinstance(self, int):
      if isinstance(other, int):
        return int_from_host(
                    int_to_host(self) +
                    int_to_host(other))
      else:
        return NotImplemented
    else:
      raise TypeError

  def __floordiv__(self, other):
    if isinstance(self, int):
      if isinstance(other, int):
        if int_to_host(other) != int_to_host(0):
          return int_from_host(
                      int_to_host(self) //
                      int_to_host(other))
        else:
          raise ZeroDivisionError
      else:
        return NotImplemented
    else:
      raise TypeError
```

Figure 6. The `__add__` and `__floordiv__` methods of `int`

```python
1  from __compiler_intrinsics__ \
2  import builtin, int_from_host, str_len_to_host
3
4  @builtin
5  class str:
6    def __len__(self):
7      if isinstance(self, str):
8        return int_from_host(str_len_to_host(self))
9      else:
10       raise TypeError
```

Figure 7. The `__len__` magic method of `str`

3.5 Redundant Operations in the Semantics

Now that we defined some magic methods for `int` and `str` we can analyze the extent of the semantics's overhead. Consider what happens if we recover the truthiness value of a string. The `truth` semantics looks up for the `__bool__` method (fig. 4, line 3). Since this method cannot be found on `str`, the `__len__` method is looked up (fig. 4, line 19). So is the `__index__` method later on (fig. 5, line 7). Both the `__len__` and `__index__` methods are invoked.

Once we know that the object is a string, multiple checks are superfluous. For example, the `__len__` method checks the type of its argument (fig. 7, line 7). Furthermore, the length of a string will always be a positive small integer. Thus the whole invocation of the `index` semantics is unneeded, as well as the assertion that the length is positive (fig. 4, line 25).

A naive implementation of the `truth` semantics would execute these redundant operations. Yet, once we know that the object is a string, only recovering the length of the string (fig. 7, line 8) and checking whether it is non-zero (fig. 4, line 13) is relevant. The required computation boils down to `int_from_host(str_len_to_host(obj)) != 0`.

The same exercise with the expression `(1 + 2)` reveals redundant operations despite the required computation boiling down to `int_from_host(int_to_host(1) + int_to_host(2))`. Expressing the semantics of primitive operators using our formalism enables a compiler to implement that sort of optimization.

4 Behaviors

A compiler can implement arithmetic operators from the semantics defined in Section 3. Yet, by doing so in a naive way, that is calling each magic method, the implementation would likely offer poor performance.

We pointed out that the magic methods of built-in types cannot be changed. Given an operator and built-in types for its operands, we can thus predict which magic methods will be looked up and which of these will contribute to computing a result. This makes looking up or calling some magic methods superfluous, for instance if a method is known to be `absent` or if it can be predicted that it will return `NotImplemented`. We exploit that fact to generate optimized versions of Python operators.

We define a *behavior* to be a procedure that describes how to compute the result of an operator *for a given combination of built-in types* without redundant type checks or superfluous method calls. Behaviors are written in a similar fashion to operators' semantics by using the `define_behavior` intrinsic decorator, which behaves identically to `define_semantics`, but labels functions differently.

In Figure 8, we implement the behaviors for addition of an integer and a float (`add_intX_floatY`), floor division between two integers (`floordiv_intX_intY`) and truthiness of a string (`truth_strX`).

We use *`operation`*`_`*`ltype`*`X_`*`rtype`*`Y` as naming convention for behaviors, where *`operation`* is the short-circuited semantics, *`ltype`* is the required type of the left-hand operand and *`rtype`* is the required type of the right-hand operand. We also include the types in the annotation of the behavior (annotations are the types written after each argument and are part of Python's syntax) as it is more

```
1  @define_behavior
2  def add_intX_floatY(x: int, y: float):
3    return float_from_host(
4              int_to_host(x) + float_to_host(y))
5
6  @define_behavior
7  def floordiv_intX_intY(x: int, y: int):
8    if int_to_host(y) != int_to_host(0):
9      return int_from_host(
10               int_to_host(x) // int_to_host(y))
11   else:
12     raise ZeroDivisionError
13
14 @define_behavior
15 def truth_strX(x: str):
16   return int_from_host(str_len_to_host(x)) != 0
```

Figure 8. Behaviors for addition of an integer and a float (`add_intX_floatY`), integer floor division (`floordiv_intX_intY`) and string truthiness (`truth_strX`)

convenient for a compiler to read them from the annotation than from the behavior's name. Unary behaviors are written by omitting the right-hand type, for example `truth_strX`.

We can use partial evaluation to generate all behaviors for arithmetic operations on numeric types by identifying which methods return a result for each operator. This is made possible by the fact that a built-in magic method returns `NotImplemented` for a value of a given type if and only if it returns `NotImplemented` for all instances of that type.

Within a given magic method, most `if` statements' conditions are type checks that can be resolved from the operands' types. The only exceptions are division and bitwise-shift, which respectively check for zero division and negative shift. These are left to be evaluated at run time (see Figure 8, line 8).

5 A Partial Evaluator to Generate Behaviors

This section presents semPy, a Python tool for generating behaviors by removing redundant type checks, boxing and unboxing, and method calls whenever possible.[3]

semPy is a Python partial evaluator supporting the compiler intrinsics statement. It takes as inputs a semantics and a *context* that consists of built-in types for each of the arguments. It outputs a specialization of the semantics given that context, which is a behavior. The behaviors presented in Figure 8 were generated by semPy. For example, the `add_intX_floatY` was generated from the `add` semantics (Figure 3) in a context where the left-hand operand is an `int` and the right-hand operand is a `float`.

The structure of operators and built-in magic methods is sufficiently homogeneous that behaviors can be generated by using only three transformations: (1) aggressive inlining of method calls, (2) branch resolution based on type information and (3) removal of redundant boxing and unboxing.

[3]The semPy source code is available online [16].

```
def __pos__(self):
  if isinstance(self, int):
    return int_from_host(int_to_host(self))
  else:
    raise TypeError
```

Figure 9. The `__pos__` magic method of `int`

5.1 Inlining

When a semantics or magic method is invoked, semPy systematically inlines the callee's code at the call site. This removes method calls from semantics specializations. Magic methods are returned by invocations of the `class_getattr` intrinsic function. This function is always called on the arguments of a semantics, whose types are provided in the type context, so it is always possible to resolve which method is to be called, or if that method is absent.

5.2 Branch Resolution

When semPy successfully computes the truthiness of the condition of an `if` statement, we can get rid of the branch that is not executed. Since semantics are written without using Python dynamic features, we can resolve the value of expressions that would normally be hard to evaluate statically. We can resolve conditions such as `isinstance(X, Y)`, which checks whether `X` is an object of type `Y`. Comparisons of the form `(magic_method is absent)` can always be resolved since built-in magic methods are immutable. We can also resolve comparisons of the form `(result is NotImplemented)`. In this case, we usually cannot infer the exact value of `result`, but we can at least infer that it is not the object `NotImplemented`.

Most branches are removed by resolving the aforementioned conditions. Some branches may still depend on the value of an object and can only be resolved if its origin provides sufficient information (such as lengths being non-negative). If not, the branch must be evaluated at run time.

5.3 Removal of Redundant Boxing and Unboxing

A naive implementation of Python's semantics sometimes causes unnecessary boxing and unboxing. For example, the pos semantics, which corresponds to unary +, is equivalent to the identity operation when applied to an integer. Yet, the magic method `__pos__` of `int` applies boxing and unboxing to account for the possibility that the argument is of a strict subtype of `int` in which case the result should be cast to an `int` (see Figure 9). A simple example is that of the expression `+True`, which must return 1.

```
@define_behavior
def pos_sintX(x: sint):
  return x

@define_behavior
def pos_boolX(x: bool):
  return int_from_host(int_to_host(x))
```

Figure 10. Removal of unboxing in unary + of `int` by semPy

In the context of a behavior where the argument is known to be of type `int`, semPy removes this type conversion. When the cast must occur, for example in the case of unary + on a `bool`, semPy preserves it as shown in the generated behaviors of Figure 10. This simplification occurs after inlining and branch resolution. At this point, unnecessary boxing manifests as chains of calls to primitives that are one another inverses and can be removed from the behavior.

5.4 The `test` Behavior

We present another example of unnecessary boxing removal. Consider the semantics of the `if` statement where Python evaluates the truthiness of a value and branches accordingly. This truthiness is determined by the `truth` semantics (see Figure 4), which returns either `True` or `False`. We show the behavior for truthiness of an integer in Figure 11. This behavior requires the `bool_from_host_bool` intrinsic function, which maps booleans in the host language to the corresponding Python boolean objects.

```
@define_behavior
def truth_intX(obj: int):
  return bool_from_host_bool(
           int_to_host(0) != int_to_host(obj))
```

Figure 11. Behavior for truthiness of `int`

In the condition of an `if`, this behavior takes the host result `int_to_host(0) != int_to_host(obj)` and converts it to a Python `bool` that the `if` statement immediately needs to convert back to a host boolean. Thus, the call to the intrinsic `bool_from_host_bool` is a form of unnecessary boxing.

We solve this by introducing the `test` semantics (fig. 12, line 2) and the intrinsic `bool_to_host_bool`, which acts as the inverse of `bool_from_host_bool`. The purpose of the `test` semantics is solely to express a variant of the `truth` semantics where we prefer the output to be a host boolean rather than a Python object. This semantics can be fed to semPy to return behaviors in which the unnecessary boxing was removed, such as the `test_intX` behavior (fig. 12, line 6).

This strategy is expandable to other cases where a condition is tested but a Python `bool` is not required, for example when the condition of an `if` statement is the result

```
1 @define_semantics
2 def test(obj):
3   return bool_to_host_bool(truth(obj))
4
5 @define_behavior
6 def test_intX(obj: int):
7   return int_to_host(0) != int_to_host(obj)
```

Figure 12. The `test` semantics and behavior of `test` for `int`

of a comparison. This would generally invoke one of the comparison semantics (`eq`, `ne`, `lt`, `le`, `gt` or `ge`), then check the truthiness of the result (Python comparison operators can return a value other than `True` or `False`). Instead, semPy can generate behaviors for those specific cases. To those behaviors we assign the names `test_`*`comp`*`_`*`ltype`*`X_`*`rtype`*`Y` where *comp* is the partially evaluated comparison semantics.

Figure 13 shows the result of semPy partial evaluation of `le(x, y)` (semantics of the <= operator) and its counterpart, the `test_le(x, y)` comparison where `x` and `y` are respectively an `int` and a `float`.[4]

```
@define_behavior
def le_intX_floatY(x: int, y: float):
  return bool_from_host_bool(
           float_to_host(y) >= int_to_host(x))

@define_behavior
def test_le_intX_floatY(x: int, y: float):
  return float_to_host(y) >= int_to_host(x)
```

Figure 13. The `le` and `test_le` behaviors for `int` and `float`.

6 Zipi: a Compiler Using Behaviors

We now detail how the tools described in this paper can be used to implement an optimizing Python compiler. We present Zipi, a compiler prototype that implements arithmetic operators and magic methods using the compiler intrinsics statement.

6.1 Zipi

The Zipi compiler [15] is an ahead-of-time (AOT) compiler from Python to Scheme [6]. It implements arithmetic operations using behaviors generated by semPy and extends this strategy to other operators. Zipi compiles Python to Scheme code, which is then compiled to an executable using either the Bigloo [22] or Gambit [10] Scheme compilers.

[4]In Figure 13, the `le_intX_floatY` and `test_le_intX_floatY` behaviors use the >= operator instead of the expected <= operator. Comparison magic methods can also return `NotImplemented`, which may lead to their reflected magic method to be called. In that case, the `__le__` method of `int` returns `NotImplemented` and the comparison resorts to the `__ge__` method of `float`.

```python
# A Python program that sums a list of integers
s = 0
for x in [1, 2, 3]:
    s = s + x
```

```scheme
1  ; The main body of the code generated
2  ; by Zipi from the above program
3  (define x #f) ; #f indicates the variable
4  (define s #f) ; is not yet bound
5  (global-register! global (& "x")
6                    (lambda () x)
7                    (lambda (v) (set! x v)))
8  (global-register! global (& "s")
9                    (lambda () s)
10                   (lambda (v) (set! s v)))
11 (set! s (py-int-from-scheme 0))
12 (py-for-each
13   #:target
14   (py-make-list (py-int-from-scheme 1)
15                 (py-int-from-scheme 2)
16                 (py-int-from-scheme 3))
17   (begin
18     (set! x #:target)
19     (set! s
20       (py-add (or s (global-get (& "s")))
21               (or x (global-get (& "x")))))))
```

Figure 14. An example of Scheme code generated by Zipi

In Figure 14, we present a snippet of code generated by Zipi from a small Python program. The compiler maps most operations directly to a procedure or macro provided by Zipi's runtime system. For example, the forms `py-for-each` (line 12), `py-make-list` (line 14) and `py-add` (line 20) are all Scheme macros whose expansions implement `for`-loops, list allocations and addition, respectively. Only relevant parts of the generated code are shown and variable names have been demangled for readability.

Zipi supports all compiler intrinsics statement. In Figure 15, we show the Scheme version of the `add` semantics from Figure 3. Note that Zipi compiles the semantics to the `py-add-fallback` macro. The full semantics is used only as a fallback when no specialized behavior exists. Behaviors are generated and compiled once, at Zipi's build time.

When generating behaviors for Zipi, we distinguish between small integers (`sint`) and big integers (`bint`). This allows semPy to generate more specialized behaviors and the Zipi runtime system to further optimize integer arithmetic by representing small integers with Scheme fixnums.

In Figure 16, we show the compiled `add` behaviors for small integers. The `fx+?` operator applies small integer addition with an overflow check. In case of overflow, the `+2` operator applies addition and returns a Scheme big

```scheme
(define-macro (py-add-fallback x y)
  `(let ((x ,x) (y ,y))
     (py-add-fallback:normal x y)))

(define (py-add-fallback:normal x y)
 (let ((magic_method (getattribute-from-obj-mro
                      x (&& "__add__"))))
   (if (py-test-is magic_method py-absent)
       (py-add-fallback:reflected x y)
       (let ((result (py-call magic_method x y)))
         (if (py-test-is result py-NotImplemented)
             (py-add-fallback:reflected x y)
             result)))))

(define (py-add-fallback:reflected x y)
 (let ((magic_method (getattribute-from-obj-mro
                      y (&& "__radd__"))))
   (if (py-test-is magic_method py-absent)
       (py-raise-binary-TypeError-fallback
         (&& "+") x y)
       (let ((result (py-call magic_method y x)))
         (if (py-test-is result py-NotImplemented)
             (py-raise-binary-TypeError-fallback
               (&& "+") x y)
             result)))))
```

Figure 15. Scheme version of the `add` semantics

```python
# Python add behavior for small integers
@define_behavior
def add_sintX_sintY(x: sint, y: sint):
    return int_from_host(int_to_host(x) +
                         int_to_host(y))
```

```scheme
; add behavior compiled by Zipi
(define-macro (py-add-sintX-sintY-inline x y)
  `(let ((x ,x) (y ,y))
     (or (fx+? x y) (py-bint-to-scheme (+2 x y)))))

(define (py-add-sintX-sintY-fallback x y)
  (or (fx+? x y) (py-bint-to-scheme (+2 x y))))
```

Figure 16. `add_sintX_sintY` behavior compiled to Scheme

integer. The `py-bint-to-scheme` procedure is the Scheme equivalent of the `int_to_host` intrinsic for big integers.

6.2 Behaviors in Zipi

To dispatch an operation to a specific behavior at run time, Zipi stores the procedures of each behavior within an array, called a *behavior array*. Each operator has its own behavior array, for example the *add behavior array* contains behaviors of addition. Once the type of each operand is known, it is possible to recover the corresponding behavior from that operand's array and invoke it.

To recover behaviors from a behavior array, we assign a unique identifier to each type. We call this identifier the *class index* of a type. Since we generated separate behaviors for small integers and big integers, those have separate class indices despite having the same Python type. For example, small integers may have the class index `1`, big integers the index `2`, `bool` the index `3` and so on. We reserve the index `0` for all types that have no specialized behavior.

When an arithmetic operator is applied, we recover the class index of the types of each operand. In the case of unary operators, this index is the position of the corresponding behavior in that operator's behavior array. In the case of binary behaviors, we apply the formula `(right + N * left)` where `right` and `left` are the class indices of both operands and `N` is the number of existing class indices (Zipi currently has 17). The procedure at that computed index can be safely invoked without further type-checking.

In some cases, the procedure stored at the class index is not a behavior, but rather the full semantics without specialization. For example, if we add two objects whose type is user-defined, the resulting index will be `0`. The `add` behavior array contains the `py-add-fallback:normal` procedure (fig. 15, line 5) at that index.

A special case of the dispatch of a behavior happens when operands are both small integers or both `float` objects, which are represented by Scheme `fixnum` and `flonum` values respectively. Since those are common arithmetic operations, we inline the corresponding behavior for those cases. We limit inlining to those frequent use cases to avoid code bloat.

Figure 16 shows that Zipi generates two versions of each behavior. The `inline` version is a macro allowing to invoke a behavior inline while the `fallback` version is a first-class procedure, which we store in the behavior array.

In Figure 17, we show this inlining process with the `py-pos` macro, which implements the unary + operation. Whenever the operand `x` is either a `fixnum` (line 4) or a `flonum` (line 5), we execute the corresponding inline behavior. Otherwise, we recover the class index of the object and invoke the corresponding behavior from the `add` behavior array (line 7).

```
1 (define-macro (py-pos x)
2   `(let ((x ,x))
3      (cond
4        ((fixnum? x) (py-pos-sintX-inline x))
5        ((flonum? x) (py-pos-floatX-inline x))
6        (else
7         ((vector-ref py-pos-behaviors-array
8                      (py-obj-class-index x))
9          x)))))
```

Figure 17. Dispatch of the behavior for unary +

The code for dispatching behaviors is similar in the case of binary operators. Behaviors are inlined when both operands are either `fixnums` or `flonums`, otherwise the behavior is recovered from the corresponding behavior array. The same happens for comparisons, and the `truth` and `test` semantics.

7 Performance

In this section, we discuss Zipi's performance in comparison to CPython and PyPy [27], the current state-of-the-art implementation performance-wise. Performance was measured through microbenchmarks as well as regular benchmarks implementing well-known algorithms.

Initialization and compilation times vary across CPython, PyPy, and Zipi. CPython compiles code ahead-of-time (AOT) into bytecode that is then interpreted by a virtual machine. PyPy uses a tracing just-in-time (JIT) compiler [5]. Lastly, Zipi is AOT and has a deep pipeline that compiles Python code to Scheme, then to C, and finally to machine code[5]. As this occurs before execution, we do not consider it in this evaluation report. We configured our benchmarks to only measure the run time performance after initialization. We also allow PyPy's JIT to warm up by executing a dry run that does not count toward execution time for each benchmark. Benchmarks measure real time using the Python `time` module, which all implementations provide.

Results were generated by Forensics, our tool for tracking performance. Both Forensics' source code [8] and the benchmarks results are available online [9]. Benchmarks were executed on a machine with an Intel Core i7-7700K, 48 GB of RAM, and under Debian 10.13 with kernel version SMP Debian 4.19.269-1. We used CPython 3.9.0 with *profile guided optimization* enabled [29]. As for PyPy, we used version 7.3.5. Each PyPy release implements more than one version of Python, we used the newest version at the time, which was Python 3.7. Zipi was compiled with Gambit 4.9.3-1380, with `single-host` enabled, and GCC 10.3.

7.1 Microbenchmarks

We use microbenchmarks to evaluate the performance of individual operations and determine whether a targeted optimization, such as behaviors for arithmetic operators, is effective. The microbenchmarks have been useful to focus our optimization efforts on operations suffering from poor performance. The operation being evaluated is wrapped in a loop to reach a measurable time on the order of one second on CPython. To minimize the loop overhead, its body contains several repetitions of the measured operation (typically 20). The microbenchmarks allow a direct comparison between Zipi and CPython on individual operations. Unfortunately,

[5]To provide an idea of compilation time, the `deltablue` program discussed in Section 7.2 contains 440 lines of code and takes about 50 seconds to compile. This compilation time could be improved by compiling Python code directly to machine code.

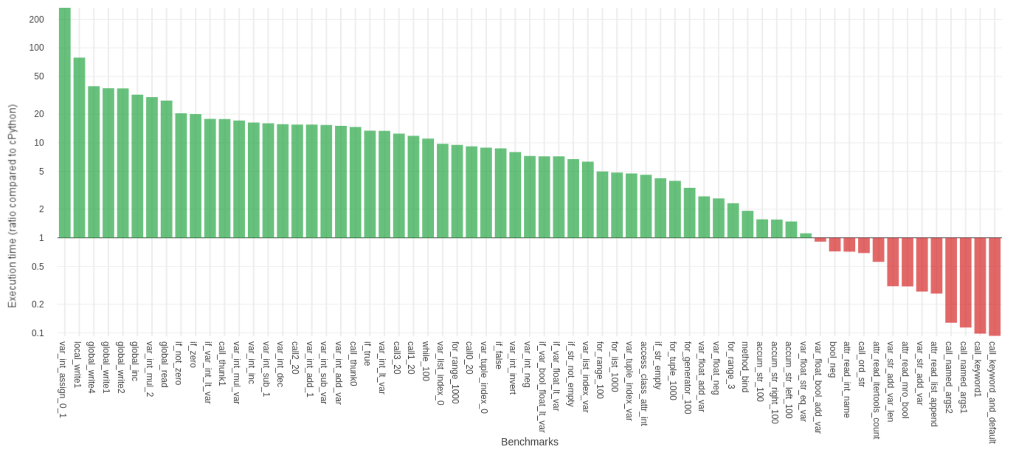


Figure 18. Microbenchmarks results of Zipi compared to CPython v3.9.0. A ratio higher than 1 (green) indicates an execution faster than CPython. A ratio lower than 1 (red) indicates a slower execution.

it does not allow a comparison with PyPy, which treats the kernel of many of our benchmarks as dead code. Neither Zipi nor CPython do this, so every operation is actually executed. Figure 18 shows the results of our microbenchmarks. All microbenchmarks are described in more details in [15].

Microbenchmarks indicate that behavior optimizations provide a significant performance boost for binary operators on small integers (between 15× and 30× faster) and `floats` (between 3.0× and 7.2×), truthiness of `bools` (between 8.7× and 14×), `ints` (20x) and `strs` (between 4.2× and 6.7×), and comparison between `ints` (18×) and `floats` (7.2×).

Performance improvements from other optimizations unrelated to behaviors also show up in the microbenchmarks. For instance, assignment to global variables, function calls and iteration on built-in types are all faster than with CPython. On the other hand, some microbenchmarks display poor performance. Those are unoptimized features that we implemented in a naive way, such as function calls with keyword arguments.

7.2 Benchmarks

We compared Zipi to CPython and PyPy using custom benchmarks and benchmarks from PyPerformance, an authoritative suite of benchmarks for Python [26]. Zipi being at an early development stage, only four benchmarks from PyPerformance are supported at the moment, hence the need for custom benchmarks.

Our custom benchmarks include `ack`, `fib`, `queens`, `bague` and `sieve`. The code for all custom benchmarks is available in [15]. Benchmarks from PyPerformance include `deltablue`, `fannkuch`, `richards` and `float` and are available online [30]. Each benchmark is executed once using parameters that result in a run time on the order of one second on CPython. Figure 19 compares the execution time of Zipi and PyPy using the CPython execution time as a baseline.

Zipi fares especially well on programs that extensively use small integer arithmetic: `ack` (38× faster than CPython), `fib` (24×) and `queens` (14×) execute faster than with PyPy. The `bague` (3.9×) and `sieve` (1.2×) benchmarks are slightly faster than CPython with Zipi. These benchmarks use small integer arithmetic, but also list and attribute access. The behavior optimization has a noticeable but limited effect in those cases. Finally, `fannkuch` (0.8×), `richards` (0.7×), `deltablue` (0.5×) and `float` (0.3×) execute slower than with CPython. These benchmarks make extensive use of user-defined types, which we did not optimize, our focus being on built-in types.

Overall, Zipi's performance on benchmarks making intensive use of small integer arithmetic rival with PyPy. Yet, this speedup does not translate to benchmarks that make a limited use of arithmetic. This is expected since behaviors specifically target arithmetic. We wish to extend the behavior optimization to other operations in the future to further analyze its impact on performance.

7.3 Threats to Validity

The validity of our results faces the common potential issues of assessing the performance of a prototype compiler.

Despite implementing Python's core features, including those identified as the main source of overhead in CPython (see Section 2), Zipi only supports a subset of the language. It lacks features such as threads, `async` functions, and most of the standard library. It remains to measure the impact of introducing these features in our prototype.

Our benchmarks show a clear performance increase when executing arithmetic-heavy programs. Nonetheless, the absence of most modules from Python's standard library limits our ability to measure the extent of this speed up on real-life programs. The PyPerformance benchmark suite also makes use of external libraries (such as `django`, a high-level web framework written in Python) [25], which prevents executing some of its benchmarks with Zipi.

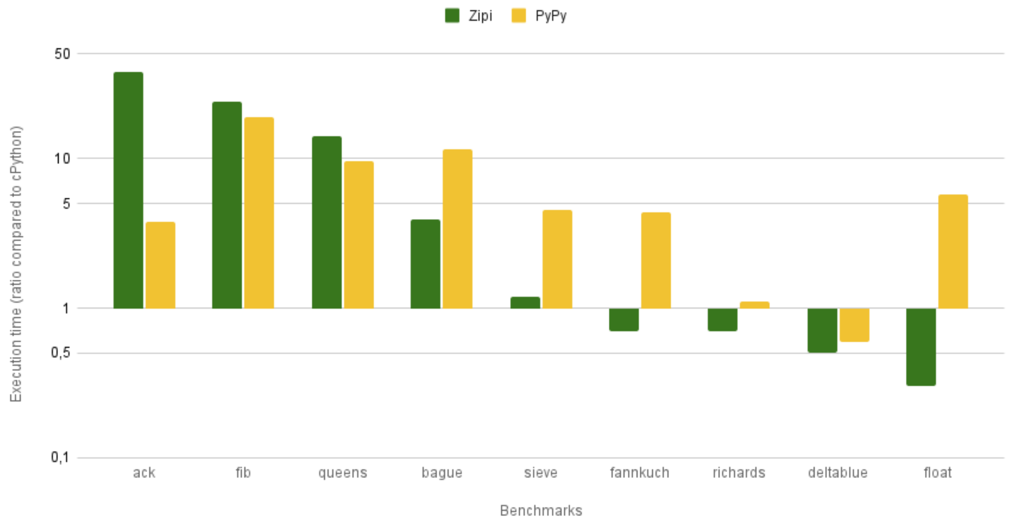


Figure 19. The execution time of Zipi (green) and PyPy3.7 v7.3.5 (yellow) is compared to that of CPython v3.9.0. A ratio higher than 1 indicates an execution faster than CPython. A ratio lower than 1 indicates a slower execution.

8 Related Work

We attribute the first instance of compiler generation from a formal semantics to Mosses [18], who developed a compiler generator based on denotational semantics. However, the generated compilers were inefficient. Mosses later outlined the pragmatic issues of denotational semantics for compiler generation. First, extension to a language's semantics often requires to completely reformulate the denotational semantics. Furthermore, denotational semantics fail to convey how a program must be executed, hindering the generation of performant compilers [19].

Executable semantics have been implemented for various languages, including C [7], Java [4], JavaScript [2], LLVM IR [33], Lua [24], PHP [11], POSIX shell [13], Python [21], and R [3]. Nowadays, these typically employ frameworks such as K [4, 7, 11], Redex [24], or a proof assistant such as Coq to extract an executable semantics [2, 3, 13, 33]. This generally results in significantly slower implementations than modern, hand-optimized compilers.

Politz et al. [21] proposed an alternative strategy for defining a Python executable semantics. It involves translating code into a lambda calculus equipped with key features such as method lookup. While not focused on performance, the technique demonstrates how the semantics can be described by desugaring code into key features.

Our approach was inspired by the RPython experiment, which allows to express high-level details about a language's semantics while remaining easy to analyze statically [1].

9 Conclusion

We presented an approach to define an executable semantics for Python operators allowing reuse in optimizing compilers. We expressed this semantics using a syntax similar to that of Python for seamless integration to an existing compiler. Our approach enhances Python with primitive functions to describe operations at a lower level. This allows us to define the notion of *behavior*, a specialization of an operator for a given combination of built-in types. In particular, we showed how behaviors remove redundant type checks, magic method calls, boxing and unboxing.

We implemented semPy, a tool for partial evaluation of the semantics, to generate behaviors automatically. The overall structure of Python's operators and magic methods allows to generate behaviors using straightforward function inlining and branch resolution.

We integrated these behaviors to Zipi, an AOT optimizing Python compiler prototype. Zipi dispatches operations to their corresponding behaviors at run time. This increases execution speed, offering performance that rivals PyPy. Although this speedup is limited to arithmetic-heavy programs, behaviors could be extended to other operations or serve alongside other optimization techniques.

We hope semPy and the behavior optimization can contribute to the ongoing optimization efforts of Python implementations. It appears to us that they would be well suited for CPython, as they specifically address the known overhead of this implementation.

Data Availability Statement

The semPy source code is available online at [16].

Acknowledgements

This work was supported by the Natural Sciences and Engineering Research Council of Canada.

A Compiler Intrinsics

This appendix describes the full specification of all intrinsics encountered in this paper. Some intrinsics' names have been shortened in the context of this paper for readability.

`define_semantics`

The `define_semantics` intrinsic is used as a function decorator. It indicates that a given definition is not that of a Python function, but rather the definition of a semantics. Thus, the compiler does not need to allocate a `function` object and is free to store the semantics in its preferred format (such as a function in the host language). The targeted semantics is defined by the name of the function. Labelling a function with `define_semantics` declares to the compiler that, within the code of the semantics, the compiler can assume that:

1. Built-in names such as `int` and `isinstance` have their standard binding;
2. The built-in functions `globals()`, `locals()`, `vars()` and `super()` are never called;
3. No global variable is used except to refer to other semantics

This precludes the use of problematic Python features, which in turn allows the compiler to apply optimizations such as inlining built-in function calls. Preventing the usage of global variables allows the compiler to skip the creation of a module altogether as it removes the need for a dynamic global environment. The behavior of `define_semantics` is undefined if not used as a function decorator.

`class_getattr`

The `class_getattr` intrinsic function takes a Python object and a string literal as arguments. It traverses the MRO of the object's type to recover the attribute specified by the string literal. If the attribute is found, it is returned. Otherwise, the value `absent` is returned to indicate that the attribute was not found. Figure 20 shows a pseudocode implementation of `class_getattr`. In most cases, the result of a call to `class_getattr` is a magic method. However, due to Python's dynamic nature, any object could be returned in which case calling the returned value may raise an exception. The behavior of `class_getattr` is undefined if it is called with anything but the aforementioned arguments.

```
class_getattr(obj, name):
  for each class in the mro of type(obj):
    if class has an attribute name:
      return class.name
  return absent  # instrinsic value 'absent'
```

Figure 20. Pseudocode for the `class_getattr` intrinsic

`absent`

The `absent` intrinsic is a primitive value similar to the JavaScript `undefined` [20]. It has an *identity* and can be compared with the `is` operator. It is not a Python object and so any other operation on it is undefined.

`sint`

An abstract subtype of `int` representing *small* integers. It is not a proper Python type, but allows to differentiate between small and big integers using the `isinstance` built-in function, while leaving room for implementation-dependent details regarding the exact threshold between small and big integers. For instance, `isinstance(x, sint)` returns `True` if `x` has type `int` and is a small integer, and returns `False` otherwise. Usage of `sint` in another context than as second argument of `isinstance` is undefined.

`bint`

Similar to `sint`, but for *big* integers.

`builtin`

The `builtin` intrinsic is used as a *class* decorator. It indicates that a class definition is that of a built-in type. Similarly to the `define_semantics` decorator, it declares that the class body does not use Python's most dynamic features.

`define_behavior`

The `define_behavior` intrinsic is used as a function decorator. It indicates that a function is the definition of a behavior. Similarly to the `define_semantics` decorator, it declares that the function does not use Python's most dynamic features.

`X``_from_host`

A family of primitive functions where *`X`* can be any built-in type, although we limit ourselves to `int` and `float` in the scope of this paper. The primitive *`X`*`_from_host` takes the host representation of an object of type *`X`* and returns the corresponding Python object of type *`X`*. This applies the operation of boxing a native value into an object.

`X``_to_host`

The primitive *`X`*`_to_host` takes a Python object of type *`X`* and returns its corresponding native representation. This applies the operation of unboxing a native value from an object.

There exists one case where *`X`*`_to_host` does not behave as the inverse of *`X`*`_from_host`. The `bool` type is a subtype of `int` and the boolean values, `True` and `False`, are respectively equal to `1` and `0`. Thus, `int_from_host(int_to_host(True))` must in fact return `1`. This is why the `bool_from_host_bool` and `bool_to_host_bool` primitive functions are required.

`bool_from_host_bool`

A primitive function that maps booleans in the host language to the corresponding Python boolean. It does not allocate a new object, since Python booleans are singleton objects.

`bool_to_host_bool`

A primitive function that maps Python booleans to the host language representation of booleans. This function is the inverse of the `bool_from_host_bool` intrinsic.

`str_len_to_host`

A primitive function that returns the length of a Python string as an integer in the host language.

References

[1] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. 2007. RPython: A Step towards Reconciling Dynamically and Statically Typed OO Languages. In *Proceedings of the 2007 Symposium on Dynamic Languages (DLS '07)*. New York, NY, USA, 53–64. https://doi.org/10.1145/1297081.1297091

[2] Martin Bodin, Arthur Chargueraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A Trusted Mechanised JavaScript Specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. 87–100. https://doi.org/10.1145/2535838.2535876

[3] Martin Bodin, Tomás Diaz, and Éric Tanter. 2020. A Trustworthy Mechanized Formalization of R. *SIGPLAN Not.* 53, 8 (apr 2020), 13–24. https://doi.org/10.1145/3393673.3276946

[4] Denis Bogdanas and Grigore Roşu. 2015. K-Java: A Complete Semantics of Java. *SIGPLAN Not.* 50, 1 (jan 2015), 445–456. https://doi.org/10.1145/2775051.2676982

[5] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems* (Genova, Italy) *(ICOOOLPS '09)*. 18–25. https://doi.org/10.1145/1565824.1565827

[6] Dybvig, Kent. 2009. *The Scheme Programming Language* (fourth edition ed.). The MIT Press. https://www.scheme.com/tspl4/.

[7] Chucky Ellison and Grigore Rosu. 2012. An Executable Formal Semantics of C with Applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. 533–544. https://doi.org/10.1145/2103656.2103719

[8] Marc Feeley and Marc-André Bélanger. 2023. Forensics. https://github.com/udem-dlteam/forensics/.

[9] Marc Feeley and Marc-André Bélanger. 2023. Zipi Forensics. https://zipi-forensics.gambitscheme.org/.

[10] Feeley, Marc. 2023. Gambit. https://www.iro.umontreal.ca/~gambit/doc/gambit.pdf.

[11] Daniele Filaretti and Sergio Maffeis. 2014. An Executable Formal Semantics of PHP. In *ECOOP 2014 – Object-Oriented Programming*, Richard Jones (Ed.). 567–592.

[12] Gouy, Isaac. 2023. The Computer Language Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/.

[13] Michael Greenberg and Austin J. Blatt. 2019. Executable Formal Semantics for the POSIX Shell. *Proc. ACM Program. Lang.* 4, POPL, Article 43 (dec 2019), 30 pages. https://doi.org/10.1145/3371111

[14] Mohamed Ismail and G. Suh. 2018. Quantitative Overhead Analysis for Python. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. Raleigh, NC, USA, 36–47. https://doi.org/10.1109/IISWC.2018.8573512

[15] Olivier Melançon. 2022. *Reusable Semantics for Implementation of Python Optimizing Compilers*. M.Sc. Thesis. https://papyrus.bib.umontreal.ca/xmlui/handle/1866/26538.

[16] Olivier Melançon. 2023. semPy. https://doi.org/10.1145/3580421

[17] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. 2020. Static Type Analysis by Abstract Interpretation of Python Programs. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 17:1–17:29. https://doi.org/10.4230/LIPIcs.ECOOP.2020.17

[18] Peter D. Mosses. 1975. *Mathematical semantics and compiler generation.* Ph.D. Thesis. University of Oxford. https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.466424

[19] Peter D. Mosses. 1996. Theory and Practice of Action Semantics. *MFCS '96* 1113, 37–61. https://doi.org/10.1007/3-540-61550-4_139

[20] Mozilla. 2023. Primitive. https://developer.mozilla.org/en-US/docs/Glossary/Primitive.

[21] Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. 2013. Python: The Full Monty — A Tested Semantics for the Python Programming Language. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. New York, NY, USA, 217–232. https://doi.org/10.1145/2509136.2509536

[22] Serrano, Manuel. 2023. Bigloo. https://www-sop.inria.fr/mimosa/fp/Bigloo.

[23] Simionato, Michele. 2023. The Python 2.3 Method Resolution Order. https://www.python.org/download/releases/2.3/mro.

[24] Mallku Soldevila, Beta Ziliani, and Bruno Silvestre. 2022. From Specification to Testing: Semantics Engineering for Lua 5.2. *J. Autom. Reason.* 66, 4 (nov 2022), 905–952. https://doi.org/10.1007/s10817-022-09638-y

[25] Stinner, Victor. 2023. Benchmarks — Python Performance Benchmark Suite. https://pyperformance.readthedocs.io/benchmarks.html.

[26] Stinner, Victor. 2023. The Python Performance Benchmark Suite. https://pyperformance.readthedocs.io.

[27] The PyPy Team. 2023. PyPy. https://www.pypy.org.

[28] The Python Software Foundation. 2023. cPython 3.9. https://www.python.org.

[29] The Python Software Foundation. 2023. Performance Options. https://docs.python.org/3/using/configure.html#performance-options.

[30] The Python Software Foundation. 2023. PyPerformance. https://github.com/python/pyperformance.

[31] The Python Software Foundation. 2023. The Python Language Reference. https://docs.python.org/3/reference/index.html.

[32] TIOBE Software BV. 2023. TIOBE Index for August 2023. https://www.tiobe.com/tiobe-index/.

[33] Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. 2021. Modular, Compositional, and Executable Formal Semantics for LLVM IR. *Proc. ACM Program. Lang.* 5, ICFP, Article 67 (aug 2021), 30 pages. https://doi.org/10.1145/3473572

[34] Qiang Zhang, Lei Xu, Xiangyu Zhang, and Baowen Xu. 2022. Quantifying the Interpretation Overhead of Python. *Science of Computer Programming* 215, C (March 2022). https://doi.org/10.1016/j.scico.2021.102759

Received 2023-07-07; accepted 2023-09-01

Adaptive Structural Operational Semantics

Gwendal Jouneaux
gwendal.jouneaux@irisa.fr
Univ. Rennes, Inria, IRISA
Rennes, France

Damian Frölich
dfrolich@acm.org
University of Amsterdam
Amsterdam, The Netherlands

Olivier Barais
Benoit Combemale
firstname.lastname@irisa.fr
Univ. Rennes, Inria, IRISA
Rennes, France

Gurvan Le Guernic
gurvan.le_guernic@inria.fr
DGA Maîtrise de l'Information, Univ. Rennes, Inria, IRISA
Rennes, France

Gunter Mussbacher
gunter.mussbacher@mcgill.ca
McGill University, Inria
Montreal, Canada

L. Thomas van Binsbergen
ltvanbinsbergen@acm.org
University of Amsterdam
Amsterdam, The Netherlands

Abstract

Software systems evolve more and more in complex and changing environments, often requiring runtime adaptation to best deliver their services. When self-adaptation is the main concern of the system, a manual implementation of the underlying feedback loop and trade-off analysis may be desirable. However, the required expertise and substantial development effort make such implementations prohibitively difficult when it is only a secondary concern for the given domain. In this paper, we present ASOS, a metalanguage abstracting the runtime adaptation concern of a given domain in the behavioral semantics of a domain-specific language (DSL), freeing the language user from implementing it from scratch for each system in the domain. We demonstrate our approach on RobLANG, a procedural DSL for robotics, where we abstract a recurrent energy-saving behavior depending on the context. We provide formal semantics for ASOS and pave the way for checking properties such as determinism, completeness, and termination of the resulting self-adaptable language. We provide first results on the performance of our approach compared to a manual implementation of this self-adaptable behavior. We demonstrate, for RobLANG, that our approach provides suitable abstractions for specifying sound adaptive operational semantics while being more efficient.

CCS Concepts: • **Software and its engineering** → **Specification languages**; **Semantics**; *Source code generation.*

Keywords: DSL, Operational Semantics, Self-Adaptation

ACM Reference Format:
Gwendal Jouneaux, Damian Frölich, Olivier Barais, Benoit Combemale, Gurvan Le Guernic, Gunter Mussbacher, and L. Thomas van Binsbergen. 2023. Adaptive Structural Operational Semantics. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3623476.3623517

1 Introduction

In a constantly evolving world, the need to design programs malleable to varying execution conditions (*i.e., self-adaptive* programs) is an important issue that has been worked on for many years by the adaptive systems community [16]. While there exist many frameworks (*e.g.,* [15, 37]) and architectural approaches (*e.g.,* [5, 21]), one of the difficulties in the field of domain-specific languages is for the average domain experts to be in a position to have to handle this concern, that is naturally outside their expertise. Indeed, this goes against the intuition that a domain-specific language (DSL) allows a domain expert to focus on their domain and generally leads to incorporating dedicated concepts for implementing self-adaptation in domain-specific languages.

To solve this problem, we propose a metalanguage, named **ASOS** (Adaptive Structural Operational Semantics), aimed at freeing the domain expert from the task of implementing self-adaptation. ASOS allows to define context-adaptable semantics for domain-specific languages. Built as an extension of MSOS [25], our approach aims to:

- Provide modular abstractions for defining and applying adaptations in the DSL's operational semantics;
- Leverage a static introduction mechanism for composing the definition of an adaptation;
- Ensure that the adaptations introduced within the semantics of a DSL do not break the fundamental properties of programs written in the DSL, such as determinism, completeness, and termination.

The contributions of this paper encompass (1) the definition of the ASOS metalanguage, (2) ASOS formal semantics for reasoning over self-adaptable operational semantics, and (3) a first implementation through a translational semantics from ASOS to SEALS [17], a framework for self-adaptable languages. This paper illustrates the concepts of the approach in the definition of a DSL for robot behavior (named RobLANG).

We then present the applicability of the approach on RobLANG. Next, we define the formal semantics of the ASOS metalanguage and discuss the alignment of the implementation with the formal semantics. Furthermore, we hint at the capability of verifying properties such as determinism, termination, and completeness of a program bound to an adaptive semantics. Lastly, we evaluate the performance overhead associated with the approach. We conclude that ASOS provides the foundations for specifying sound adaptive operational semantics while introducing little performance overhead compared to a manual implementation.

The remainder of the paper is as follows. Section 2 provides background and motivation, and introduces RobLANG. The following section gives an overview of our approach, including the definition of the abstract syntax and semantics of a self-adaptable language and the configuration of the self-adaptation loop. Section 4 details the syntax and semantics of ASOS. The last three sections present the evaluation, related work, and our conclusion and future work.

2 Motivation and Illustrative Example

While a manual implementation of the self-adaptation concern may be desirable when it is the main concern of the system, the required expertise and substantial development effort often does not justify a manual approach when it is only a secondary concern for the given domain. A more automated approach is needed. Hence, we first provide background information on self-adaptive system design and describe the limitations of the current approaches in the context of domain-specific languages (DSLs). Finally, we present the RobLANG DSL as an illustrative example that would benefit from the abstraction of self-adaptation at the language level.

2.1 Design of Self-Adaptive Systems

Self-adaptive systems are designed to adjust their behavior based on changes in the system or its environment. This changement of behavior is called an adaptation. They use sensors to gather data, analyze the current situation, decide on a new behavior if necessary, and implement the changes through effectors. This process is known as the feedback loop scheme, which consists of four main functions: Monitoring, Analysis, Planning, and Execution (MAPE-K) [19].

Previous work helps in the design of self-adaptive systems providing architectural solutions (*e.g.,* three layer architecture [21], MORPH [5], PLASMA [35]) and frameworks (*e.g.,* Executable Runtime Megamodels [37], DCL [27], ActivFORMS [15], Ponder2 [36]). While architectural solutions give guidelines, they generally do not support designers during the implementation. Meanwhile, frameworks provide support to the designer. However, those frameworks are restricted to the languages they were designed for, likely requiring re-implementation for other languages.

Another way to design a self-adaptive system is to define self-adaptation at the meta-level. SEALS [17], *e.g.,* supports language engineers in the implementation of self-adaptable virtual machines, providing abstractions and avoiding to re-implement a known framework for self-adaptation from scratch. SEALS provides facilities to support the definition of the language abstract syntax and operational semantics, the feedback loop and associated trade-off reasoning, and the adaptation semantics and the predictive model of their impact on the trade-off (Impact Model). However, SEALS does not readily enable formal verification of properties such as determinism, completeness, and termination.

2.2 Limitations of Current DSL Approaches

In the context of DSLs, the re-implementation of frameworks for self-adaptation can be tedious or impossible, depending on the expressiveness of the language. E.g, the difficult task of implementing a constraint solving algorithm for trade-off analysis or monitoring the execution environment (*e.g.,* CPU, RAM) may not be supported. Moreover, this implementation requires the language user to be an expert in the design of self-adaptive systems, which is often not the case. However, designing self-adaptation at the meta-level offers appropriate tooling to language engineers for the implementation.

2.3 Illustrative Example: Energy-Aware RobLANG

RobLANG[1] is a representative of procedural DSLs used, in this case, for specifying the actions of a robot. The domain concepts manipulated in RobLANG include speed changes, movement (forward and backward), orientation (turn left-/right), and use of the sensors (*e.g.,* battery, distance). To orchestrate these actions, RobLANG also has functions, control structure (if and while), arithmetic and boolean expressions.

Often, the domain of robotics requires developers to implement the behavior of the robot with energy efficiency in mind to avoid battery depletion. One way to reduce the energy consumption of a robot is to reduce the speed of the motors. This is due to the exponential increase in motor energy costs as a function of speed [2].

However, speed is often also an important factor in the robot mission. Therefore, it is necessary to dynamically apply this speed reduction, depending on the trade-off between energy consumption and speed, taking into account various potentially dynamic pieces of information (*e.g.,* availability of the power supply, current level of the battery, time estimation to complete the current task, importance of the task).

In this context, RobLANG would benefit from the abstraction of this recurrent adaptive behavior in a metalanguage

[1]Implementation : https://www.gwendal-jouneaux.fr/SLE2023/RobLANG

to free the language user from the implementation of the feedback loop and trade-off analysis.

3 Approach Overview

This section presents a general overview of our approach and describes ASOS (Adaptive Structural Operational Semantics), a metalanguage to specify and reason on self-adaptable operational semantics. Figure 1 depicts the design of a language with adaptive operational semantics, *i.e.,* a Self-Adaptable Language (SAL), and the generation of an interpreter in SEALS [17], an implementation framework for building self-adaptable virtual machines (see Section 2.1). SEALS is used as the target of the generation because of the suitable abstraction. The goal is to allow, through ASOS, to reason about such semantics, rather than on SEALS Java code. The execution of the ASOS semantics rules of the SAL are presented in Figure 2, starting from the *Evaluate* step, with first the feedback loop and then the rule executions.

The definition of a self-adaptable language, as in other languages, includes the definition of its abstract syntax and semantics. The abstract syntax specifies the domain concepts of the language and their relations, whereas the semantics define the meaning of those concepts. In this paper, with SEALS as the target implementation framework, we focus on operational semantics as a way to define language semantics.

3.1 Abstract Syntax Definition

In the modeling community, the abstract syntax is often expressed with a metamodel. Since SEALS relies on a Java-based definition of abstract syntax, we choose to use Ecore [34] as the metalanguage to express the metamodel. In addition, a generator from Ecore to Java classes using EMF [34] exists and can be modified to generate the SEALS-based interpreter. This allows a language engineer to define the abstract syntax as they would normally do for any Ecore-based DSL definition (*e.g.,* EcoreTools [34]). Figure 1 represents this by the language engineer defining the abstract syntax conforming to the Ecore metamodel, and the SEALS-based interpreter being generated for the language implementation.

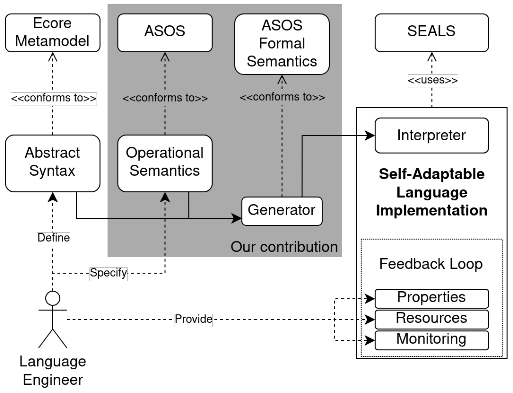


Figure 1. Approach overview

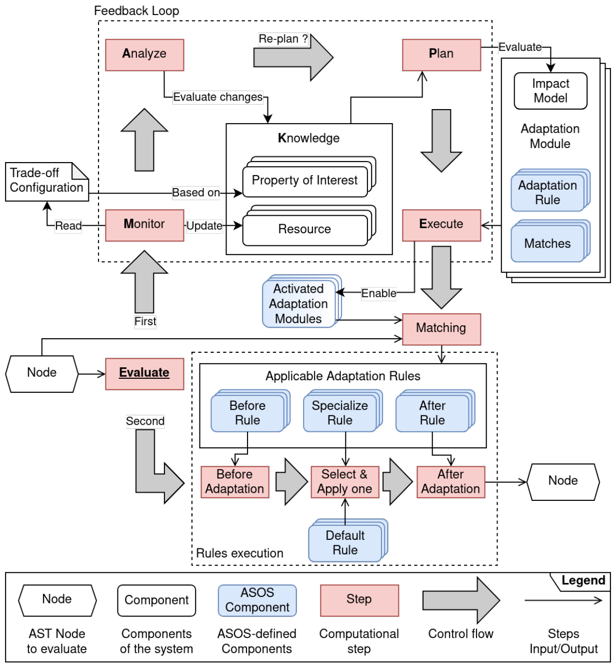


Figure 2. Overview of ASOS semantics execution

3.2 Operational Semantics Definition

Operational semantics define the meaning of the language concepts by expressing the computational steps (evolution of the state of the execution at runtime). The operational semantics of a language can be expressed using metalanguages such as Structural Operational Semantics (SOS) [29], and is typically implemented through an interpreter or a compiler.

In particular, SEALS defines operational semantics in the form of an interpreter composed of three components: a default semantics for the language, a feedback loop performing analyses, and modular adaptations modifying the default semantics. SEALS, through this decomposition, ensures a default behavior and allows the delegation of the development and/or configuration of adaptations to the language users, or even the end-users. ASOS keeps this decomposition for the same reasons. These three components are represented, in Figure 2, through the stack of *Default Rule* at the bottom, the *Feedback Loop* dashed box, and the stack of *Adaptation Module* on the right, respectively.

To support a modular approach and the ability to reason on operational semantics, we choose to base our definition of default and adaptation semantics on Modular SOS (MSOS) [25] and its extension I-MSOS [26]. Among the potential frameworks to formally specify semantics, such as $\mathbb{K}$ [32] and its matching logic [31], we choose MSOS for its focus on modular definition of operational semantics and the implicit propagation of auxiliary components of its extension I-MSOS, reducing redundancy in the rule writing and fitting the propagation of models through programs execution.

3.2.1 Default Semantics (Rules). The definition of the default semantics is a set of transition rules for all the concepts of the language. I-MSOS transition rules can be decomposed into several components, presented in Figure 3. First, the conclusion of the rule represents the effect of the transition on the state. This transition from a pattern used to match a particular term structure, *i.e.,* the *input pattern*, results in a new term to be evaluated or a computed value to return, *i.e.,* the *result* of the rule. When a transition is performed, we say that the term matching the *input pattern progressed* to the *result.* In addition, this transition can affect auxiliary entities such as the memory. In I-MSOS, these auxiliary entities are implicitly propagated and only expressed when a rule makes use of them. This conclusion is conditioned by two other components of the rule: side-conditions and premises.

Side-conditions allow to limit the application of the rule to a subset of the terms matching the structure defined in the conclusion of the rule. For example, a rule performing a division would only apply if the divisor is not zero. These conditions are also used to describe computations over computed values. For instance, a rule performing an addition would define a condition $n = n1 + n2$ such as $n1$ and $n2$ are the computed values of the left and right expressions. In this case, rather than evaluating the predicate, those conditions require an instance of n to make the predicate true, thus computing the value for n.

On the other hand, premises define assertions on the ability of subterms (terms contained in the main term) to *progress*, *i.e.,* perform transitions. It differs from the side-condition computations, as it represents computations of terms using rules.For example, a rule in charge of evaluating the condition of an if statement progresses to a term representing the same statement but replacing the original condition by the *result* of its progression. However, this makes sense only if the condition can progress. Hence, this rule is conditioned by a premise on the ability of the condition to progress.

ASOS reuses these concepts in the definition of its rules, which are detailed in Section 4.1. The key difference with I-MSOS is the clear separation between side-conditions and computations present in ASOS. In ASOS, computations are explicit and grouped, with assignment of values to/from the propagated auxiliary entities, in a dedicated section of the rule. Hence, making the side-conditions, as for the pattern matching, has no side effects. In the proposed implementation, these auxiliary entities are defined using a metamodel representing the structure of the semantic domain. The merge of the abstract syntax metamodel and the metamodel for the semantic domain forms the execution metamodel. Instances of this execution metamodel (execution models) represent the runtime state of the program, which is implicitly propagated in the runtime in the same way as the I-MSOS auxiliary entities.

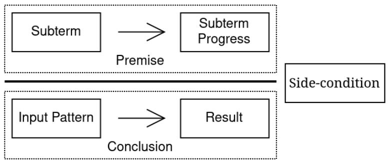


Figure 3. Structure of I-MSOS rule

3.2.2 Feedback Loop and Trade-Off Reasoning. At the core of a self-adaptable language, there is a feedback loop selecting the adaptations to perform depending on monitored resources and the desired trade-off between the addressed properties of interest. The resources represent the environment upon which the decision to adapt is taken, while properties of interest denote the specific properties that we seek to maximize through adaptations. To implement the feedback loop, SEALS requires the implementation of the Monitor, Analyze, Plan, and Execute phases (red boxes of the feedback loop in Figure 2) and Knowledge base, providing the resources to monitor, the properties of interest, and the function reading the desired trade-off. ASOS provides an implementation for the Analyze, Plan, and Execute phases of the feedback loop based on the modeling approach provided by the framework. However, the feedback loop still requires configuration. The resources and properties of interest needs to be configured in SEALS, and monitoring hooks must be implemented to update resources values. They are represented in the *Knowledge* box of the *Feedback Loop*, and by the "read" and "update" arrows in Figure 2 (see also the Feedback Loop in Figure 1). In this first version of ASOS, we do not provide abstractions for the configuration and monitoring hooks to retain flexibility to implement various strategies.

3.2.3 Defining Adaptations Modules. Finally, to express the adaptations of the operational semantics, ASOS requires adaptation modules.An adaptation module can be defined by the language engineer, or delegated to other stakeholders (*e.g.*, language users, end-users). However, while the inclusion of external adaptation modules is facilitated by SEALS and generated code from ASOS, the method used to manage external adaptations (*e.g.,* link in command line, dedicated folder) is left to the language engineer to implement.

An adaptation module is defined by three components: adaptation rules, matching clauses, and a model of expected impacts of the adaptation on the properties of interests. Adaptation rules are defined similar to default rules but with an additional description on how to introduce them in the operational semantics dynamically and under which conditions. We propose three ways to introduce adaptation rules: (i) *specialization*, where the new rule replaces an existing rule, (ii) *before*, where the new rule is executed before another rule, and (iii) *after*, where the new rule is executed after another rule. It means that before executing the default rule, as depicted in the *Rules execution* box of Figure 2, the applicable before adaptations are called, then either one of the specialization rules or the default rule is applied, and finally the applicable after adaptations are called.

Of course, an adaptation may not always apply. To specify when the conditions are met and the adaptation can be introduced, we propose a matching system based on a structural matching on the Abstract Syntax Tree and conditions on the runtime values in the execution model. As shown in Figure 2, every adaptation module define *Matches*, that are used during a *Matching* step verifying if the adaptation is applicable on the current AST node.

In addition to this matching system, an adaptation also needs to be activated by a feedback loop evaluating its relevance to the current trade-off and environment. To do so, the adaptation module must declare a predictive model of its impact on the properties of interest, the *Impact Model*. This model is used in the *Plan* phase of the *Feedback Loop* (see Figure 2) to select the set of adaptations based on the trade-off given the current context. The adaptations are then enabled in the *Execute* phase.

The SEALS implementation for adaptation modules is derived from the ASOS specification for this module. However, the impact model for this module still needs to be defined. It is left as future work to provide the appropriate abstractions for the impact models in ASOS because there are multiple alternatives to implement the *Analyze* and *Plan* phases and define the impact models (*e.g.,* Goal Modeling, Machine Learning). When using the base implementation of the feedback loop provided by ASOS, the language engineer will have to define a goal modeling-based impact model using the constructs provided by the SEALS framework.

4 The ASOS Metalanguage

ASOS[2] is a declarative language to specify operational semantics based on MSOS [25]. ASOS extends MSOS by providing abstractions to define runtime dynamic adaptation of the operational semantics of the language. The definition of the operational semantics is done through transition rules in the same fashion as MSOS. The additional adaptation concern is managed using adaptation rules, *i.e.,* transition rules with additional information on how and when to introduce them in the set of applicable transition rules. This is defined using the ASOS matching system because it allows to express structural patterns that are not possible to express using typical MSOS rule format [9].

4.1 ASOS Syntax

This section describes the abstract syntax of the ASOS metalanguage. Figure 4 shows the main concepts of the ASOS metamodel. `Adaptive Operational Semantics` is the top-level concept representing the adaptive operational semantics of the implemented DSL, and is composed of a set of rules and a set of adaptation modules representing the default semantics and the adaptation semantics of the language. We use RobLANG[3] as an illustrative example with the concrete syntax provided by the implementation (Section 4.2).

[2]Implementation: https://www.gwendal-jouneaux.fr/SLE2023/ASOS

4.1.1 Structure of Transition Rules.

The `Rule` concept is at the core of ASOS, describing the computation to perform for a given concept. These computations are mainly described using the `Transitions` components of the rule. Two types of transitions exist, the conclusion of the rule and the premises. Both represent the same concepts as the ones from MSOS described in Section 3.2.

```
1  rule IfCond,
2       RobLANG.If(cond, then, else)
3       ->
4       RobLANG.If(newcond, then, else)
5  resolve
6       cond -> newcond
7
8  rule IfTrue,
9       RobLANG.If(sd.ValueBool(b), then, else) -> then
10 where
11      b == true
12
13 rule IfFalse,
14     RobLANG.If(sd.ValueBool(b), then, else) -> else
15 where
16     b == false
```

Listing 1. Transition rules to compute an if condition

Listing 1 presents the definition of three `Rules` for the *If* concept. `Transitions` are represented using an arrow (⟶), with the conclusion defined as the first transition in the rule (*e.g.,* lines 2-4 for rule *IfCond*) and premises defined in the *resolve* section (*e.g.,* line 6). The LHS and RHS of transitions are `Terms` except that the LHS of the conclusion transition (*e.g.,* line 2) must be a `Configuration` defined in the abstract syntax of the DSL. Such a `Configuration` is prefixed with *RobLANG* and defines the concept on which to execute the rule. A concept prefixed with *sd* is also a `Configuration` but defined in the semantic domain structure and represents the computed values. The constructor notation[4] is used to denote a `Configuration` (*e.g., RobLANG.If(...)*), whereas `Symbols` are represented as identifiers (*e.g., cond*, *then*, *else*).

The subterms in the parenthesis of the constructor notation correspond to the elements contained in this concept as defined in the DSL's abstract syntax. This could represent a computed value constructor (*e.g., sd.ValueBool(...)* in line 9) or it could bind a name to the subterm of a configuration for further use in the premise (*e.g., cond*). The subterms allow us to represent part of the state of the evaluation of the concept, and update it. When defining a computed value constructor (prefixed with *sd*) for a subterm (see line 9), this implies that this subterm has been computed. Moreover, premises assert that a subterm, via a transition, can change state. In line 6 of *IfCond*, the state of evaluation of the if condition changed, and the *newcond* `Symbol` is bound to this new state.

[3]Specification: https://www.gwendal-jouneaux.fr/SLE2023/ASOSRobLANG
[4]A *constructor notation* is the pattern : `prefix.Concept(subterms...)`

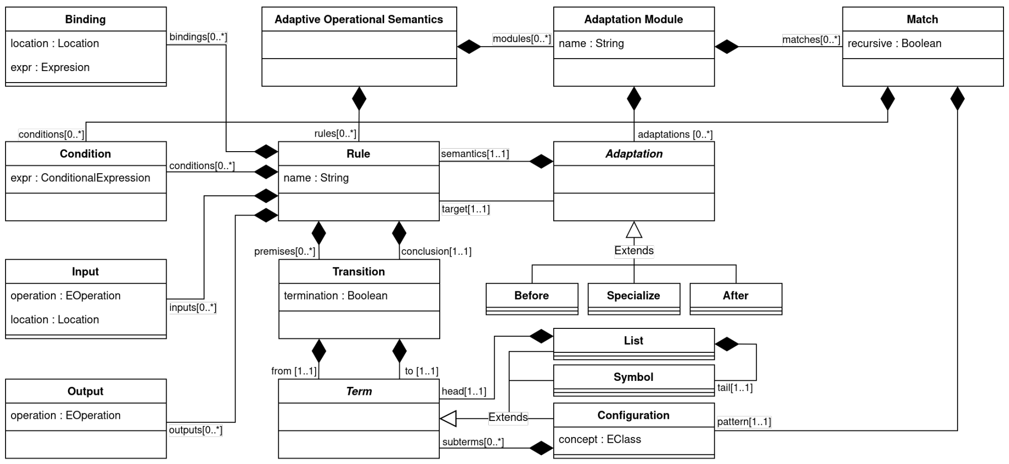


Figure 4. ASOS Metamodel (ConditionalExpression, Expression, and Location omitted to focus on main concepts)

The constructor notation akin to algebraic data allows us to express the impact of this change in the evaluation state of the *If* concept by changing *cond* to *newcond* in the output term (RHS of transition) using the Symbol as a reference. Just like in I-MSOS, memory and other auxiliary entities are propagated implicitly and do not need to be represented.

In addition to the premises assertion, rules can be conditional. The *IfTrue* rule includes a Condition (line 13) in the *where* section of the rule. A condition could be used to define first-order logic predicates over values. For instance, the choice of the branch of the if statement to execute depends on the condition truth value.

```
1  rule Break,
2      RobLANG.BreakLoop()
3      ->
4      termination sd.BreakSignal()
5
6  rule LoopTrueBreak,
7      RobLANG.Loop(sd.ValueBool(b), body)
8      ->
9      sd.NilValue()
10 where
11     b == true
12 resolve
13     body -> termination sd.BreakSignal()
```

Listing 2. Abrupt termination using loop breaking

To ease handling abrupt termination (*e.g.*, errors, breaks, returns), transitions can emit and receive abrupt termination signals using the keyword termination. Conclusion transitions using the termination keyword emit a signal containing the usual output (e.g., line 4 in Listing 2). On the other hand, premises with a termination nature (e.g., line 13) are the only premises matching a transition emitting such signal. This allows the language engineer to receive and handle this abrupt termination, while remaining oblivious of the upward propagation of these signals when not managed. Furthermore, support of termination allows for default handling to be embedded in the ASOS metalanguage.

Transitions may require computations and/or storing capabilities to describe the arrival state (*e.g.*, arithmetic expressions, assignments). In ASOS, both are managed using the Binding construct. Bindings associate the result of an expression to a location. This location can be either a new Symbol (e.g., line 6 in Listing 4), allowing reuse of the expression result in the output of the transition, or a "dot" notation allowing to set values in the execution model propagated during the execution. Expressions support the "dot" notation to access the execution model, symbol reference, the usual arithmetic and boolean operators, and constants.

Finally, some concepts involve an external component either for *Input* or *Output*. For example, print statements require access to a communication interface, *i.e.*, the console. This access is managed through external functions conforming to a predefined signature in the semantic domain structure definition. In addition, the *on* keyword can be used to specify the object on which to call the function. An Input is denoted through the assignment of a function result (e.g., lines 8-9 in Listing 4), while an Output is just a function call (e.g., lines 10-12 in Listing 4).

4.1.2 Adaptation Modules Definition.

Adaptations are defined in modules. An Adaptation Module groups a set of transition rules representing the adaptation of a concept. These transition rules are defined like other transition rules as explained earlier. However, an adaptation developer needs to additionally define the conditions to apply the adaptation at the module level, and how the adaptation rules are added to the operational semantics at the rule level.

```
recursive match RobLANG.Loop(
    RobLANG.GreaterEqual(
        lhs,
        RobLANG.DoubleConstant(d)),
    body)
where
    d == 0.0
```

Listing 3. Match clause of an adaptation module

The conditions are defined using a `Match` clause. This match clause is composed of a recursive (or not) nature, a structural match, and conditions, the latter two being similar to the input pattern (LHS of conclusion transition) and *where* section of a rule, respectively. Listing 3 gives an example of an adaptation module's match clause. In this example, the structural match targets a while loop of the form: *while(lhs ≥ d){body}*. Moreover, the *where* section condition ensures that the constant d is equal to 0. The match describes a configuration which leads to the dynamic introduction of the module adaptation rules in the operational semantics when the current term to evaluate is matched. The new rules introduced can be used for the evaluation of this term and/or all of its descendant in the AST. If a descendant of the matched term happens to be of the same nature (here the *Loop* concept) but is not valid with respect to the match clause, there are two possibilities. If the match clause possesses the recursive nature, nothing changes and the introduced rules remains. However, if the match is not recursive, the introduced adaptation rules are removed from the operational semantics for the descendant and the associated sub-tree of the AST.

To define how to introduce adaptation rules in operational semantics, we propose three types of adaptation rules presented in Listing 4. A `Specialization` adaptation defines a rule that will replace a target rule in the operational semantics. For instance, the adaptation rule *HalfSpeedForward* is a rule that replaces the *ForwardAct* rule to perform the moving forward action at half of the speed. `Before` and `After` adaptation also target an existing rule, but the rule defined is executed respectively before or after the target rule.

```
Specialize ForwardAct rule HalfSpeedForward,
    RobLANG.MoveForward(sd.ValueDouble(d))
    -> sd.NilValue()
bind
    half = 0.5 * s
IO
    ctx = RobLANG.WithContext.getContext();
    s = sd.Context.getNominalSpeed() on ctx;
    sd.Context.setSpeed(half) on ctx;
    sd.Context.moveRobot(d) on ctx;
    sd.Context.setSpeed(s) on ctx

Before TargetRule rule BeforeTargetRule,
    ...
After TargetRule rule AfterTargetRule,
    ...
```

Listing 4. Three types of adaptation rules

4.1.3 Well-Formedness Rules.

To specify a well-formed ASOS semantics, additional constraints on the abstract syntax need to be followed. First, all transitions in a rule are not defined in the same way. The transition representing the conclusion of the rule requires a `Configuration` defined in the abstract syntax of the DSL as left `Term` (from), which is not the case for premises. Second, `List` terms, representing subterms with cardinality greater than 2, can only be used as subterms to decompose, for instance, the list of statements in a loop. Finally, adaptation rules require well-formedness constraints to ensure their applicability. Thus, a `Before` adaptation requires, as result, a valid term that can be executed by the adapted rule.

4.2 ASOS Translational Semantics

In this section, we detail the current implementation of the ASOS language. This implementation takes the form of a translational semantics to a Java implementation based on Ecore [34] and the SEALS framework [17].

4.2.1 Derive Java Code from ASOS Transition Rules.

To derive a Java implementation of a transition rule, we divide the rule into two parts: effects and guards. Effects represent the effects of the rule on the state, such as the resulting term of the rule, bindings, inputs, and outputs. Guards represent the conditions to apply a rule, such as the input pattern, premises, and conditions. The overview of the rule guards and effect generation is presented in Figure 5. Solid arrows represent the generation of one element of the rule, while dashed arrows represent the generic generation repeated for all instances in a section. Arrows pointing to adaptations represent the call site of this type of adaptation.

To generate the effect of the rule, we first generate the inputs, then the bindings, the computation of the resulting term, and finally outputs. `Input` and `Output` are defined as callable functions in ASOS. In the implementation, we use Ecore *EOperations* to model these functions. We first generate the processing of the function arguments, then create a call to the appropriate *EOperation* for each input and each output. This is presented in Figure 5 by the dashed arrows from the `Input` and `Output` of the *IO* section. For `Input`, the result of each function is stored either in a temporary variable if the `Location` is a `Symbol`, or in the execution model by resolving the associated "dot" notation. Each `Binding` generates, as for `Input`, an assignment of the computed expression to a temporary variable or a call to the appropriate setter of the execution model. The expression of the `Binding` is translated to its Java equivalent, with execution model accesses resolved as calls to the appropriate getters.

Finally, we generate the computation of the output `Term` depending on its form: a `Configuration` of the same concept as the input, a `Symbol`, or a `Configuration` with a different concept. In the case of a `Configuration` of the same concept, the rule is an update of the state of evaluation of the current concept. The update of the current state is generated from the difference between the two configurations. For instance, the `Term` resulting from a premise can be retained in the new state of evaluation of the current concept, as is the case in the *IfCond* rule (line 6 in Listing 1). If the transition resolves to a `Symbol`, we do not know what it represents (term or value). To manage this, we generate a conditional assignment to a *return* variable, that will affect the value if computed, or the term if not. At the end of the rule's semantics, the con-

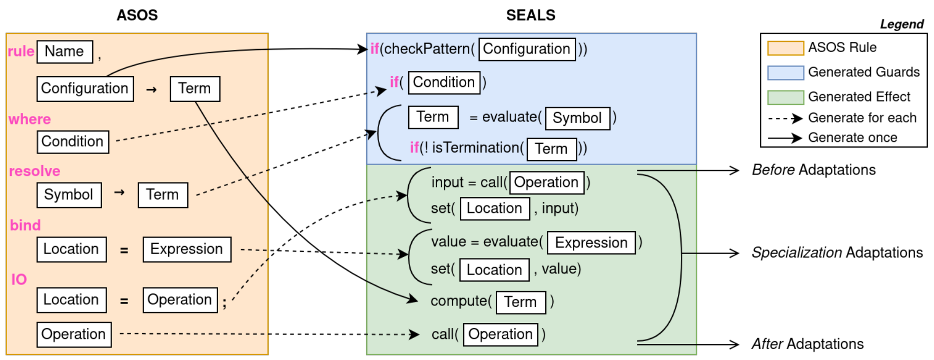


Figure 5. Overview of rule generation (ASOS to Pseudo-code)

tents of this variable are returned or executed, depending on whether it is a value or a term. Finally, if the output is another `Configuration`, we generate the expected structure using the object factory provided by the Ecore model. Subterms represented with a `Configuration` generate a new concept instance and subterms represented with a `Symbol` are resolved and set in the correct concept instance. This structure is assigned to the *return* variable, and is returned or executed depending on whether it is a value or a term.

To generate the guards, we first generate the input pattern matching condition, then check the conditions of the *where* section, and finally the premises. The generation of the input pattern condition is done by checking the type of the subterms of the associated `Configuration`. `Symbols` does not impose constraint, hence generating no condition. For `Configuration` and `List`, we generate the condition on the associated type, and recursively generate conditions for their subterms. This guard is the first generated, ensuring the structure required for valid symbol resolution in the remaining of the rule. The generation of a `Condition` is done by translating its expression to its Java equivalent. If there are multiple conditions, they are represented as nested in the order of definition in the ASOS rule.

Finally, the `Premise` generation produces three statements: a condition checking that the subterm has not been evaluated, a call for the evaluation of the `Symbol`, and a verification of abrupt termination signals. In Figure 5, the first statement is implicitly shown in the *evaluate* method as values cannot be evaluated. This condition is necessary as a premise is an assertion on the transitions of subterms, whereas a computed value can never transition. If it has not been computed, we compute it and store its result for reuse in the effect of the rule. If the right hand side of the premise is a `Configuration`, we also check that the resulting term is matching its pattern A final condition is generated to check the normal or abrupt termination of the premise computation.

If the premise does not expect a termination signal, or if this signal is different than the expected one, the rule is not applied, and the termination signal is stored for potential propagation if no other rules handle this termination. To avoid re-executing a premise when its result is the only difference in rule guards (*e.g., termination vs normal execution*), the result of premise evaluation is shared across those conflicting rules, as a failing guard does not apply a rule, hence does not change the state of execution.

4.2.2 Generate Default Semantics and Feedback Loop.

In SEALS, the default semantics is defined through *Operation* classes, providing the complete semantics for one concept in an "*execute*" function. From those *Operation* classes, SEALS provides a visitor who can evaluate the AST of the language. Moreover, SEALS allows adaptations on those *Operation* semantics by explicitly defining it as an *AdaptableOperation* and providing its interface with adaptations, requiring ASOS to generate one pair for each concept. Finally, SEALS requires the specialization of its *FeedbackLoop*, *AdaptationContext*, and *SelfAdaptableLanguage* concepts, generated by ASOS.

To generate the content of the *execute* function, presented by Figure 6, all the ASOS rules defined for the concept are retained from the set of rules defining the default semantics.

Since we consider the provided semantics definition as deterministic, the generated rules, presented in the yellow box in Figure 6, preserve the ASOS definition order. To store the computed value during the execution of the concept, we automatically generate a data class that contains one field for each subterm and the associated getters and setters. We have two instances of this data class, one for the data kept across rules, and one to store the result of computed premises.

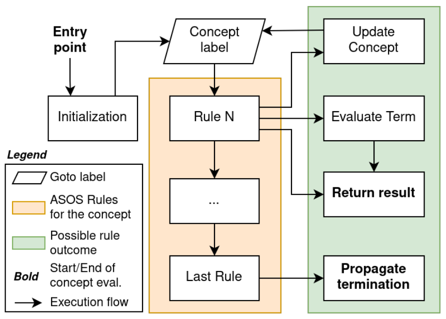


Figure 6. AdaptableOperation *execute* function overview

At the end of a rule execution, there are three cases: the rule returns a value, the rule reduces to a term to evaluate, or the current concept state is updated and needs to be executed further. All of these cases are shown in the green box on the right of Figure 6. In the first case, the computed value is simply returned as a result of the execution of the concept. In the second case, the visitor is called on this term to execute it, then the resulting computed value is returned. Finally, if the current concept requires more rule applications to complete, we jump back to the top of the rules list. In the case that no rule can be applied and because the semantics is considered complete, we know that a termination signal was not handled at this concept level and should be propagated to the calling concept. This is represented by "Propagate termination" at the end of the last rule in Figure 6.

Because adaptations are based on the introduction of new rules in semantics, we generate a language-wide interface defining three `adaptationrule` fields for each rule of the default semantics, corresponding to the `Before`, `After`, and `Specialization` adaptations. These `AdaptationRules` represent a callable adaptation rule, and therefore, will be detailed in the next section. The arrows on the right part of Figure 5 denote the call to these `AdaptationRules` if they exist for this rule in the current context. This is done inside the guards to ensure that we call adaptation only if the rule is effectively called. The effect of the original rule is generated in a way that either the specialization or the effect will be executed but not both. Finally, the `After` adaptations are called on the output of the current rule.

In addition, generic implementations of the three main classes of a SEALS language (*AdaptationContext*, *FeedbackLoop*, and *SelfAdaptableLanguage*) are generated. The *FeedbackLoop* implementation make use of SEALS proposed modeling approach for impact and trade-off analysis. However, the *AdaptationContext* still requires the definition of the resources and properties of interests, and implementation of the trade-off monitoring function.

4.2.3 Generate SEALS Adaptation Modules.

In ASOS, adaptation rules achieving part of the same adaptation logic are grouped in modules. These modules contain, in addition to the set of adaptation rules, a matching expression enabling the adaptation to occur on the term and its subterm. On the other hand, the concept of adaptation module of SEALS provides the *adapt* and *trigger* method. The *trigger* method verifies if the *adapt* method should be called on the current term, while the *adapt* method performs the adaptation.

To map the behavior specified in ASOS, we chose to generate a set of `AdaptationRule` classes representing each adaptation rule, implementing an *adapt* function with the code for the adaptation rule. To evaluate the adaptation rule, the implementation requires an access to the subterms, and a way to evaluate premises of the adaptation rule. The first point is managed by passing the node and the current state of execution of the node. The second is achieved by providing the store of computed premises and the visitor, giving access to computed premises and a way to evaluate the others.

These rules are the ones used in the pattern managing adaptation, at the right on Figure 5. When a match occurs, an instance of these classes will be created and added to the pool of executable rules by adding it to the interface for adaptations. However, SEALS recreates an interface at each step of the evaluation, hence this new set of rules is not propagated. This issue can be resolved by adding some information at the *AdaptationModule* level. Rather than performing the matching for the current node like SEALS, we save the state of the matching at the module level. When evaluating a term that matches the `Match` clause of the module, we save this information in a boolean in the module. Then for every node, if an ancestor node matched the clause, we add the `AdaptationRule` instance to the interface. Depending on the `Match` recursive nature, we potentially invalidate this match when going deeper in the structure. In the end, only the impact model of the adaptation remains for the adaptation designer to specify.

4.3 ASOS Formal Semantics

In this section, we introduce a formalization for the ASOS meta-language and its alignment with the implementation (Section 4.3.1). The formalization allows us to reason about certain properties, in particular, how the adaptation process affects determinism (Section 4.3.2), termination (Section 4.3.3), and completeness (Section 4.3.4). To formalize ASOS we build upon earlier work of generalized transition systems (GTSs) as defined for MSOS by Mosses [25].

Definition 4.1. A GTS is a tuple $\langle \Gamma, \mathbf{A}, \longrightarrow, T \rangle$, where $\mathbf{A}$ is a category with morphisms A, $\longrightarrow \subseteq (\Gamma \times A \times \Gamma)$ is the transition relation, and $T \subseteq \Gamma$ are terminal configurations. $\langle \Gamma, A, \longrightarrow, T \rangle$ is a labeled terminal transition system[5] (LTTS) [30].

The category is referred to as a label category and is an (indexed) product category of component categories $\prod_{i \in I} \mathbf{A}_i$. For a full account of the different types of component categories we refer the reader to [25].

With the category, it is ensured that the labels of subsequent transitions compose. So when we have $\gamma \xrightarrow{\alpha_1} \gamma_1 \xrightarrow{\alpha_2} \gamma_2$ for some $\gamma, \gamma_1, \gamma_2 \in \Gamma$ and $\alpha_1, \alpha_2 \in A$, we have that $\alpha_1; \alpha_2$ holds in the category $\mathbf{A}$. Using the theory of MSOS, we define our formalization as follows.

Definition 4.2. Let $B = \langle \Gamma, \mathbf{A}, \longrightarrow, T \rangle$ be a GTS, then we define our formalization as a tuple $\langle B, \boldsymbol{\Delta}, \pi, \kappa, \zeta, \rightsquigarrow \rangle$ such that the discrete category with $\mathcal{P}(\Delta)$ as its objects is one of the component categories of $\mathbf{A}$; $\boldsymbol{\Delta} = (\Delta, lab : \Delta \rightarrow \{0, 1\})$ is a structured set of symbols that we call adaptation signals, and lab is a labeling function which maps adaptation

[5]An LTTS is simply an LTS with an extra component representing the terminal configurations, i.e. the configurations for which there are no transition.

signals to either 1 or 0, denoting to apply or not apply recursive activation, respectively; $\pi : \Gamma \rightarrow (\mathcal{P}(\Delta) \rightarrow \mathcal{P}(\Delta))$ is an adaptation projection function; $\kappa : O \rightarrow \mathcal{P}(\Delta)$ is an adaptation activation function with O being the set of objects; $\zeta : \mathcal{P}(\Delta) \rightarrow \delta$ is an adaptation selection function; and $\rightsquigarrow \in (\Gamma, \Delta) \times A \times \Gamma$ is an adaptable transition relation. Furthermore, we define the $\twoheadrightarrow$ relation inductively as follows, where $D \diamond D' = \{\delta \mid \delta \in D \cup D' \land lab(\delta) = 1\}$ with $D, D' \in \mathcal{P}(\Delta)$, and X denotes the label of the $\twoheadrightarrow$ transition, so X is a morphism of the category $\mathbf{A}$.

$$\text{adaptation}\,\frac{\begin{array}{c}\pi(\gamma)(\kappa(source(X))) = D' \\ \zeta(D \cup D') = \delta' \\ D \diamond D' \vdash (\gamma, \delta') \rightsquigarrow \gamma'\end{array}}{D \vdash \gamma \twoheadrightarrow \gamma'}$$

$$\text{default}\,\frac{\begin{array}{c}\pi(\gamma)(\kappa(source(X))) = D' \\ \zeta(D \cup D') = \delta' \\ D \diamond D' \vdash (\gamma, \delta') \not\rightsquigarrow \\ D \diamond D' \vdash \gamma \longrightarrow \gamma'\end{array}}{D \vdash \gamma \twoheadrightarrow \gamma'}$$

These rules state that when there is an active adaptation for the current configuration and the current adaptation signal, then that rule is picked. Otherwise, the transition in the base GTS is used. In addition, we require that the $\rightsquigarrow$ relation respects the terminal configurations of the base GTS. I.e., for all $\gamma \in T$ and for all $\delta' \in \Delta$ we have $(\gamma, \delta') \not\rightsquigarrow$.

4.3.1 Alignment with the Implementation.

To explain our formalization in a bit more detail, we discuss the alignment of the ASOS model and our formalization.

Γ and T represent the terms to evaluate and is reflected in the implementation by a metaclass in the language metamodel. While Γ ranges over configuration with an arbitrary metaclass representing the term, the terminal configurations T are the subset that use metaclasses from the semantic domain structure definition. Adaptation signals(Δ), represent all the combinations of activated adaptation modules. Moreover, before and after adaptations obtain an annotated adaptation signal to differentiate between them within the $\rightsquigarrow$ relation. An adaptation signal is reflected in the implementation by an adaptation interface instance that contains adaptation rules. The addition of two module rules to the interface makes it contain the union of the two sets of rules, hence representing the same set of rules as the adaptation signal of the merge of the signals of the two modules. Thus, the composition in the interface is similar to the adaptation signal merge for composition in formal semantics.

The three functions κ, π, ζ model the feedback loop, adaptation activation, and adaptation selection. The matching on terms in the meta-language to activate an adaptation corresponds to the π function. To ensure that only one signal is active at a time, the ζ function selects one signal based on the current active adaptation signals.

The $\longrightarrow$ relation makes up the original semantics of the programming language and corresponds to the rules outside of adaptation modules. The $\rightsquigarrow$ corresponds to the specialization adaptations as defined in the adaptation modules. The syntax of $\rightsquigarrow$ is similar to $\longrightarrow$ with the addition of the adaptation component, corresponding to the adaptation module that captures the ASOS rule. The Δ component of the $\rightsquigarrow$ ensures that the specialization activates iff the module is activated. For both relations, the arrows in the premises correspond to the $\twoheadrightarrow$ relation. For the $\rightsquigarrow$ relation, premises can also contain before and after steps that model the before and after adaptations. The final semantics of the language is defined by the $\twoheadrightarrow$ relation, which combines the $\longrightarrow$ and $\rightsquigarrow$ relations. The $\twoheadrightarrow$ relation thus corresponds to the *Adaptive Operational Semantics* component in Figure 4.

Finally, the category $\mathbf{A}$ models the auxiliary entities available to rules. This corresponds to the *Binding*, *Input*, and *Output* components in Figure 4. In the implementation, we implicitly propagate these entities represented by the execution model, the feedback loop state, and the modules state. Therefore, successive modification of these elements forms a trace similarly to label composition in the category.

4.3.2 On Determinism in an ATS.

In our model, determinism is controlled by the designer and not introduced by the adaptation process. In other words, iff the three relations $\Rightarrow$, $\rightsquigarrow$ and $\longrightarrow$ are deterministic, then $\twoheadrightarrow$ is deterministic.

To demonstrate this, we give a proof sketch that shows that for every configuration and for all $\alpha \in A$ we have either $\gamma \not\twoheadrightarrow$ and $\gamma \in T$ or $\exists! \gamma' \in \Gamma$ such that $\gamma \twoheadrightarrow \gamma'$. This is only true whenever we have no derivation tree or we have a unique derivation tree. The first case holds by definition. For the second case, we show that under the assumption, the *adaptation* and *default* rules are deterministic. Let us assume they are not deterministic, then we can construct multiple derivations trees for some $\gamma \in \Gamma$ and some $\alpha \in A$. For the *adaptation* rule we can construct multiple derivation trees whenever one of premises $\pi(\gamma) \circ \kappa(source(X))$, $\zeta(D \cup D')$, or $D \diamond D' \vdash (\gamma, \delta') \rightsquigarrow \gamma'$ has multiple solutions and the resulting conclusion configurations are distinct. By definition, both the first and second component have exactly one solution. So, for some $\delta' \in \Delta$ we have $\exists \gamma_1, \gamma_2 \in \Gamma$ and $\gamma_1 \neq \gamma_2$ such that $D \diamond D' \vdash (\gamma, \delta') \rightsquigarrow \gamma_1$ and $D \diamond D' \vdash (\gamma, \delta') \rightsquigarrow \gamma_2$. However, this contradicts our assumption that $\rightsquigarrow$ is deterministic. The same process can be used for the *default* rule. Finally, the *adaptation* and *default* rule are non-overlapping by definition due to the conflicting premise of the $\rightsquigarrow$ relation. Hence, under the assumption, the $\twoheadrightarrow$ is deterministic.

4.3.3 On Non-Termination in an ATS.

Non-termination in an ATS can arise due to the interplay between the $\rightsquigarrow$ and $\longrightarrow$ relations. For example, we might have $\gamma \rightarrow \gamma'$ and then $(\gamma', \delta) \rightsquigarrow \gamma$ for some $\delta \in \Delta$, which can result in non-termination. It might not, because the outside environment, *e.g.,* sensors, can change resulting in a different adaptation or no adaptation being performed. In addition, such occurrences might be intentional. For example, when a *while* term is adapted but a term in the body of the *while* is not adapted.

So far, we have not yet identified a reasonable restriction on the adaptations that prevents this from occurring. Nevertheless, using our formal model, we can reason about such occurrences, and we aim to utilize model checking to identify adaptations that introduce such sequences.

4.3.4 On Completeness of the Original GTS. With our model, we wanted to retain the completeness of the original GTS This means that for all $\gamma \in \Gamma$ we either have $\exists \gamma'$ such that $\gamma \twoheadrightarrow \gamma'$ or $\gamma \in T$. This holds trivially in our formalization due to the $\twoheadrightarrow$ relation definition and by our requirement that the $\rightsquigarrow$ relation respects the terminal configurations.

5 Evaluation

To evaluate the proposed implementation, we discuss: the applicability of the approach on RobLANG, and the performance overhead.To discuss the applicability , we compare the number of actions performed by the robot with and without adaptations. Finally, we assess the performance of our approach, *i.e.,* self-adaptation at language level, to the performance of the same adaptive behavior written in the program, *i.e.,* self-adaptation at program level.

5.1 ASOS Applicability to RobLANG

To assess the applicability of ASOS to RobLANG, we compare the number of actions performed by the robot until battery depletion with and without adaptations, to show that we can express meaningful adaptations despite the fact that we abstracted the adaptation at language level. In addition, we also evaluate the ability to react to change in the environment by dynamically changing the trade-off at run-time. The adaptation used in this case is the reduction in motor speed discussed in Section 2.3, with the motor running at 75% speed. The action performed by the robot is a movement in square pattern, repeated until the battery depletion. The number of squares completed is 49898 without adaptation and 87475 with adaptation always active. In addition we also manually verified the change of semantic used when changing the trade-off from energy focused to performance focused, later refered as "Switch" configuration. The results shows that the application of the adaptation allowed the robot to perform 1.75 times the number of actions and that the language-level adaptations correctly change based on the context.

5.2 Assessing ASOS Performance

For this experiment, we choose RobLANG, presented in Section 2.3, as object of study. We use two implementations of this language, a self-adaptive one using ASOS, and a classic one using well known tools for DSLs implementation. For both implementations, the abstract syntax of the language was defined using an Ecore [34] metamodel and the concrete syntax using an Xtext [13] grammar. Since these are implementations of the same language, only small changes were made in the syntax, due to the operational semantics implementation method. Both metamodels define concepts that include: (1) Functions definition and calls, (2) Simple arithmetic and boolean predicates, (3) Access to the robot sensors, (4) Effectors to move the robot. The grammars for these concepts are the same for both implementations. However, in the case of ASOS, the structure of the semantic domain (*e.g.,* metaclasses of runtime values, attribute storing dynamic information) needs to be defined in the form of an Ecore metamodel, that is merged with the abstract syntax to define the execution metamodel. In addition, the abstract syntax of the ASOS RobLANG also defines a new statement allowing a language user to define their trade-off and change it at run-time. For the definition of operational semantics, ASOS is used to specify the adaptive version of RobLANG, while the classic version uses Xtend [4] dispatch to define a visitor.

5.2.1 Experimental Setup. To evaluate the overhead of our approach, we compare execution time of a program requesting adaptations. We compare the RobLANG ASOS implementation (*ASOS*) to a manual implementation of the self-adaptation concern at the program level (Program) using the classic implementation. The adaptation used is a reduction in motor speed in robot movement. This adaptation applies depending on the trade-off selected. If *Energy* is more important, the adaptation is applied. If *Time* is more important, it is not applied. We use a program that iteratively moves the robot in a square pattern, and we measure the overhead for three configurations: (i) the adaptation never applies (*Without*), (ii) the adaptation always applies (*With*), and (iii) the adaptation is activated and deactivated periodically at runtime (*Switch*). For each configuration, we measure 30 executions in a row, repeated three times with reboot between each repetition to mitigate the effect of the initial state [18]. Measurements were performed on a computer with 31Gb of RAM and an Intel(R) Core(TM) i7-10850H CPU (12 cores at 2.70GHz) with Manjaro 22.0.5. The language runtimes are executed using the OpenJDK Runtime Environment 11.0.18, and run alone on the computer.

Table 1. Mean time(ms) and 95% confidence intervals, relative speedups, and speedups geometrical mean

	Implementation		Speed-
Conf.	Program	ASOS	up
Without	1245.72 ms [1239.12, 1252.32]	1120.92 ms [1109.65, 1132.20]	x0.90
With	2075.72 ms [2065.50, 2085.95]	1979.91 ms [1968.42, 1991.40]	x0.95
Switch	1817.19 ms [1807.57, 1826.81]	1445.49 ms [1429.55, 1461.42]	x0.80
	Speedups Geometrical Mean		x0.88

5.2.2 Results.

Table 1 summarizes the performance of both the ASOS implementation and the manual adaptation implementation for each configuration. We compute the mean execution time, the speedup for each configuration, and provide the geometrical mean of these speedups when comparing the two implementations. With an overall speedup of 0.88, the implementation of the self-adaptation concern is more efficient with ASOS. The biggest speedup comes from the *Switch* configuration with a factor of 0.80, followed by the *Without* configuration (x0.90) and finally the *With* configuration (x0.95). These results shows that ASOS does not introduce problematic performances pitfalls, and can even surpass a manual implementation for non-optimized DSLs.

5.2.3 Discussion.

First, we can observe the speedup of ASOS compared to the handcrafted adaptation. With the biggest speed-up coming from the *Switch* configuration, we deduce that the implementation of the feedback loop is more efficient using ASOS. This is probably because the usual RobLANG interpreter running the feedback loop is not optimized, whereas the JVM optimizes the feedback loop when using ASOS. The second observation is that the *With* configuration speedup is less important than the *Without* configuration. These two configurations make the same use of the feedback loop, as their trade-off does not change. In both cases, the handcrafted version performs a call to the speed-setter statement. Hence, this difference in the speedups comes from the difference in performance to call an adaptation compared to the original rule. To conclude that these hypotheses are true, further experimentation needs to be done.

6 Related Work

Adaptable Systems. In this paper, we have introduced the idea of adaptable structural operational semantics to capture the essence of adaptable languages. Earlier work on describing adaptable systems exists. Adaptable interface automata [6] are an extension of interface automata [11] with atomic propositions that model state observations. Adaptations are then transitions where the two states of the transition give different results for some (or all) proposition(s). Self-Adaptive Abstract State Machines [3] use multi-agent abstract state machines to formalize self-adaptable systems. Compared to our approach, the adaptive system is not centralized but is distributed among several agents. There are two types of agents: managing and managed. Synchronous Adaptive Systems [1] is another approach to the formalization of adaptive systems. In this approach, a system is divided into several modules which can be in different configurations — each representing a different behavior. Configurations are activated and deactivated via guards — somewhat resembling the matching in our approach. A rewriting approach [7] is used by modeling self-adaptive systems in Maude [10] relying on computational reflection [22]. The approach takes an unbounded layered approach in which (partial) knowledge flows downwards and effects flow upwards. A layer can modify the rules of the layer below it, modeling adaptation in the approach. Recurring in these different approaches is a separation of an adaptable system in two or more layers. This idea is also present in our approach, exemplified by the two transition relations in our approach. The idea behind ASOS clearly falls within the line of adaptable interpreters [8] that enable the creation of dynamic systems. The ASOS approach, by proposing semantics based on MSOS, additionally enables the construction of verification tools for language engineers.

Abstraction at Language Level. The quest to abstract non-functional properties (*e.g.*, adaptability, security) and incorporate their effects in the behavior of an existing application has been a longstanding endeavor. This approach is rooted in Model-Driven Engineering methodologies that aim to provide a software creation process through a series of transformations, enabling the specialization of such properties [33]. It is also rooted in modern programming languages where annotations/attributes may be used to abstract non-functional features as first level entities [28]. Additionally, this is also observed in the community of dynamic software product lines [12, 14] and their implementation based on Aspect-Oriented Programming (AOP) [20], thereby allowing the weaving of non-functional concerns based on software design decision. In these approaches, we recognize the quest to abstract non-functional concerns from the design phase carried out by a domain expert. While certain approaches have focused on investigating the correctness of system behavior for various configurations [24], the ability to reason compositionally about this correctness remains limited [23]. Abstracting the adaptation concern at the language level, while providing a clear semantics for the composition of adaptation modules with a base program, allows the language designer to reason about the impact of an adaptation module on a set of behavioral properties of a base program written using an ASOS-defined DSL.

7 Conclusion and Future Work

This paper proposes the ASOS framework to define modular and adaptable semantics of a DSL. ASOS paves the way for checking determinism, completeness, and termination properties based on the proposed formal semantics. ASOS also provides the possibility of generating an implementation of a modular and adaptable interpreter based on SEALS, an implementation framework for adaptable interpreters.

Perspectives of this work are (1) evaluating the complexity for a language designer to use ASOS , (2) allowing the definition of correctness envelopes at the rule level, (3) allowing the configuration of the feedback loop in ASOS, and (4) showing that the declarative nature of ASOS rules allows language composition, facilitating the construction of self-adaptable language fragments, enabling the scenarios where a DSL is built by assembling existing language fragments.

References

[1] Rasmus Adler, Ina Schaefer, Tobias Schüle, and Eric Vecchié. 2007. From Model-Based Design to Formal Verification of Adaptive Embedded Systems. In *Formal Methods and Software Engineering, 9th International Conference on Formal Engineering Methods, ICFEM 2007, Boca Raton, FL, USA, November 14-15, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4789)*, Michael J. Butler, Michael G. Hinchey, and María M. Larrondo-Petrie (Eds.). Springer, 76–95. https://doi.org/10.1007/978-3-540-76650-6_6

[2] Anwar Al-Mofleh, Soib Taib, Wael Salah, and Mokhzaini Azizan. 2008. Importance of Energy Efficiency: From the Perspective of Electrical Equipments. In *Proceedings of the 2nd International Conference on Science and Technology (ICSTIE)*.

[3] Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra. 2015. Modeling and Analyzing MAPE-K Feedback Loops for Self-Adaptation. In *10th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015, Florence, Italy, May 18-19, 2015*, Paola Inverardi and Bradley R. Schmerl (Eds.). IEEE Computer Society, 13–23. https://doi.org/10.1109/SEAMS.2015.10

[4] Lorenzo Bettini. 2011. A DSL for writing type systems for Xtext languages. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java, PPPJ 2011, Kongens Lyngby, Denmark, August 24-26, 2011*. 31–40. https://doi.org/10.1145/2093157.2093163

[5] Victor Braberman, Nicolas D'Ippolito, Jeff Kramer, Daniel Sykes, and Sebastian Uchitel. 2015. Morph: A reference architecture for configuration and behaviour self-adaptation. In *Proceedings of the 1st International Workshop on Control Theory for Software Engineering*. 9–16.

[6] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch Lafuente, and Andrea Vandin. 2013. Adaptable transition systems. In *Recent Trends in Algebraic Development Techniques: 21st International Workshop, WADT 2012, Salamanca, Spain, June 7-10, 2012, Revised Selected Papers 21*. Springer, 95–110.

[7] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch-Lafuente, and Andrea Vandin. 2015. Modelling and analyzing adaptive self-assembly strategies with Maude. *Sci. Comput. Program.* 99 (2015), 75–94. https://doi.org/10.1016/j.scico.2013.11.043

[8] Walter Cazzola and Albert Shaqiri. 2016. Dynamic software evolution through interpreter adaptation. In *Companion Proceedings of the 15th International Conference on Modularity*. 16–19.

[9] Martin Churchill and Peter D Mosses. 2013. Modular bisimulation theory for computations and values. In *Foundations of Software Science and Computation Structures: 16th International Conference, FOSSACS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 16*. Springer, 97–112.

[10] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott (Eds.). 2007. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. Lecture Notes in Computer Science, Vol. 4350. Springer. https://doi.org/10.1007/978-3-540-71999-1

[11] Luca de Alfaro and Thomas A. Henzinger. 2001. Interface automata. In *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001, Vienna, Austria, September 10-14, 2001*, A Min Tjoa and Volker Gruhn (Eds.). ACM, 109–120. https://doi.org/10.1145/503209.503226

[12] Tom Dinkelaker, Ralf Mitschke, Karin Fetzer, and Mira Mezini. 2010. A dynamic software product line approach using aspect models at runtime. In *5th Domain-Specific Aspect Languages Workshop*. Citeseer.

[13] Moritz Eysholdt and Heiko Behrens. 2010. Xtext: implement your language faster than the quick and dirty way. In *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. 307–309. https://doi.org/10.1145/1869542.1869625

[14] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. 2008. Dynamic software product lines. *Computer* 41, 4 (2008), 93–95.

[15] M Usman Iftikhar and Danny Weyns. 2014. Activforms: Active formal models for self-adaptation. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. 125–134.

[16] Paola Inverardi and Massimo Tivoli. 2009. The future of software: Adaptation and dependability. *Software Engineering: International Summer Schools, ISSSE 2006-2008, Salerno, Italy, Revised Tutorial Lectures* (2009), 1–31.

[17] Gwendal Jouneaux, Olivier Barais, Benoit Combemale, and Gunter Mussbacher. 2021. SEALS: a framework for building self-adaptive virtual machines. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering*. 150–163.

[18] Tomas Kalibera, Lubomir Bulej, and Petr Tuma. 2005. Benchmark precision and random initial state. In *Proceedings of the 2005 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2005)*. 484–490.

[19] J. O. Kephart and D. M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (Jan 2003), 41–50.

[20] Gregor Kiczales. 1996. Aspect-oriented programming. *ACM Computing Surveys (CSUR)* 28, 4es (1996), 154–es.

[21] Jeff Kramer and Jeff Magee. 2007. Self-managed systems: an architectural challenge. In *Future of Software Engineering (FOSE'07)*. IEEE, 259–268.

[22] Pattie Maes. 1988. Computational reflection. *Knowl. Eng. Rev.* 3, 1 (1988), 1–19. https://doi.org/10.1017/S0269888900004355

[23] Andreas Metzger and Klaus Pohl. 2014. Software Product Line Engineering and Variability Management: Achievements and Challenges. In *Future of Software Engineering Proceedings* (Hyderabad, India) *(FOSE 2014)*. Association for Computing Machinery, New York, NY, USA, 70–84. https://doi.org/10.1145/2593882.2593888

[24] Brice Morin, Olivier Barais, Gregory Nain, and Jean-Marc Jézéquel. 2009. Taming dynamically adaptive systems using models and aspects. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 122–132.

[25] Peter D Mosses. 2004. Modular structural operational semantics. *The Journal of Logic and Algebraic Programming* 60 (2004), 195–228.

[26] Peter D Mosses and Mark J New. 2009. Implicit propagation in structural operational semantics. *Electronic Notes in Theoretical Computer Science* 229, 4 (2009), 49–66.

[27] Hiroyuki Nakagawa, Akihiko Ohsuga, and Shinichi Honiden. 2012. Towards dynamic evolution of self-adaptive systems based on dynamic updating of control loops. In *2012 IEEE Sixth International Conference on Self-Adaptive and Self-Organizing Systems*. IEEE, 59–68.

[28] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2016. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience* 46, 9 (2016), 1155–1179.

[29] Gordon D Plotkin. 1981. *A structural approach to operational semantics*. Aarhus university.

[30] Gordon D. Plotkin. 2004. A structural approach to operational semantics. *J. Log. Algebraic Methods Program.* 60-61 (2004), 17–139.

[31] Grigore Roşu, Chucky Ellison, and Wolfram Schulte. 2010. Matching logic: An alternative to Hoare/Floyd logic. In *International Conference on Algebraic Methodology and Software Technology*. Springer, 142–162.

[32] Grigore Roşu and Traian Florin Şerbănuţă. 2010. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434.

[33] Douglas C Schmidt et al. 2006. Model-driven engineering. *Computer-IEEE Computer Society-* 39, 2 (2006), 25.

[34] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: Eclipse Modeling Framework.* Pearson Education.

[35] Hossein Tajalli, Joshua Garcia, George Edwards, and Nenad Medvidovic. 2010. PLASMA: a plan-based layered architecture for software model-driven adaptation. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 467–476.

[36] Kevin Twidle, Naranker Dulay, Emil Lupu, and Morris Sloman. 2009. Ponder2: A policy system for autonomous pervasive environments. In *2009 Fifth International Conference on Autonomic and Autonomous Systems*. IEEE, 330–335.

[37] Thomas Vogel and Holger Giese. 2012. A language for feedback loops in self-adaptive systems: Executable runtime megamodels. In *2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 129–138.

Received 2023-07-07; accepted 2023-09-01

A Reference GLL Implementation

Adrian Johnstone
a.johnstone@rhul.ac.uk
Royal Holloway University of London
UK

Abstract

The Generalised-LL (GLL) context-free parsing algorithm was introduced at the 2009 LDTA workshop, and since then a series of variant algorithms and implementations have been described. There is a wide variety of optimisations that may be applied to GLL, some of which were already present in the originally published form.

This paper presents a reference GLL implementation shorn of all optimisations as a common baseline for the real-world comparison of performance across GLL variants. This baseline version has particular value for non-specialists, since its simple form may be straightforwardly encoded in the implementer's preferred programming language.

We also describe our approach to low level memory management of GLL internal data structures. Our evaluation on large inputs shows a factor 3–4 speedup over a naïve implementation using the standard Java APIs and a factor 4–5 reduction in heap requirements. We conclude with notes on some algorithm-level optimisations that may be applied independently of the internal data representation.

CCS Concepts: • **Software and its engineering** → **Parsers**; • **Theory of computation** → **Grammars and context-free languages**.

Keywords: Programming language syntax specification, GLL parsers, GLL implementation

ACM Reference Format:
Adrian Johnstone. 2023. A Reference GLL Implementation. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3623476.3623521

1 Introduction

GLL was introduced at the 2009 LDTA workshop [10] in the form of a generator that produces generalisations of simple recursive descent recognisers; this is extended to a full parser in [11]. A variety of extensions to the basic GLL algorithm have been reported, including direct handling of EBNF constructs [13] and multi-parsing with applications to generalised lexing [14, 16]. The control flow within GLL parsers naturally lends itself to combinator-style GLL implementations which are particularly suited to functional programming languages, and several combinator GLL implementations have been reported [4, 17, 21]. The Rascal meta-programming language [7] deploys GLL style parsers.

Performance optimisations of classical GLL include the FGLL variant which improves performance on left-factored grammars, and the RGLL variant which requires fewer independent processing threads [12]. Space and performance optimisations arising from encoding derivation forests using Binary Subtree Representation sets are explored in [15] leading to a discussion of *clustering* and the development of the Clustered Nonterminal Processing (CNP) variant. Machine-level approaches to optimising the data structures required by the GLL algorithm are explored in [5].

This paper provides a new presentation of GLL as a fixed-form 'interpreted' parser which is parametrised by an in-memory representation of the grammar. We have two main objectives (i) to provide an easily accessible 'reference' version of GLL that we hope will facilitate adoption, and (ii) to provide a baseline implementation that allows the throughput and memory consumption of GLL variants to be compared in a principled fashion.

The approach taken here is avowedly procedural in style. The code is written in Java, but is trivially portable to ANSI-C. We have done this so as to provide a reference implementation that minimises dependencies on particular programming styles such as object orientation or functional combinators.

At the datastructure level we provide two implementations (i) gllBL (baseline) and (ii) gllHP (hash pool). The first uses the standard Java API methods to implement sets and lists; this allows for a more readable presentation. The gllHP variant explains how we use low level memory management to enhance performance in our production parsers. We have previously compared and contrasted an idiomatically 'pure' object oriented implementation to an optimised procedural implementation. In that study we found the OO variant to impose a performance overhead [6]. However, there have been significant advances in Java compiler and Java Virtual Machine performance in the intervening years, and it would be interesting to revisit that work.

By way of introduction, we review the way in which standard recursive descent parsers can be extended to a wider class of grammars by incorporating backtracking. We call our particular approach Ordered Singleton Backtracking Recursive Descent (OSBRD), and note some of its failure modes. We then show how GLL generalises recursive descent by directly handling the parse function call stacks in a combined graph, and by recording parser configurations in process descriptors, allowing all parsing choices to be explored in worst case cubic time and space.

In the rest of this section we summarise the background material needed to describe our implementation. In Section 2 we focus on control flow, moving from a compiled (non-general) OSBRD parser to the state-based interpretive style that we use for GLL. In Section 3 we give a detailed account of gllBL, our baseline GLL algorithm. In Section 4 we present an efficient data representation for the GLL data structures, and in Section 5 we evaluate the performance of these GLL variants on a small number of large and diverse examples. Section 6 contains notes on opportunities to improve the performance of our baseline algorithm, and we conclude in Section 7 with an informal 'practicality' test.

Software artefacts corresponding to this presentation are available in a public repository at
https://github.com/AJohnstone2007/referenceImplementation.

Grammar notation A *Context Free Grammar* (CFG) is a 4-tuple $\Gamma = (N, T, S, P)$ denoting respectively, a set of nonterminals, a set of terminals, a start nonterminal and a set of productions with $N \cap T = \emptyset, S \in N, P \in N \times (N \cup T)^*$. An *Extended Context Free Grammar* (ECFG) has productions $P \in N \times \rho$ where ρ is a regular expression over $(N \cup T)^*$.

For small examples, we use the following conventions. ϵ denotes the empty string; a, b, c, x, y, z are elements of T; X, Y, Z are elements of N (along with S); u, v, w are elements of T^*; and $\alpha, \beta, \gamma \in (N \cup T)^*$.

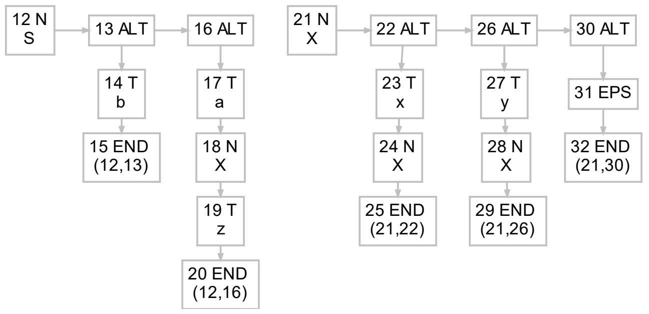
We may specify a grammar simply by enumerating its set of productions in the form $X \longrightarrow \alpha$ with the convention that the left-hand side of the first production is the start symbol; where we have multiple alternate productions with the same left hand side $X \longrightarrow \alpha \quad X \longrightarrow \beta \quad X \longrightarrow \ldots$ we may use the shorthand $X \longrightarrow \alpha \mid \beta \mid \ldots$

The *derivation step* relation $\Rightarrow$ captures the notion of language generation from a grammar; if we have $\alpha X \gamma$ and $X \longrightarrow \beta \in P$, then $\alpha X \gamma \Rightarrow \alpha\beta\gamma$. We write $\overset{*}{\Rightarrow}$ to denote the *derives* relation: for instance $S \overset{*}{\Rightarrow} \alpha$ is the set of *sentential forms* derivable from the start symbol, and $S \overset{*}{\Rightarrow} u$ is the set of *sentences* derivable from the start symbol, that is $L(\Gamma)$ the *language* of Γ.

The special symbol \$ (with $\$ \notin T$) denotes the *end of input string* marker which is appended to putative sentences before input to our parsing and recognition algorithms.

Grammar representation This paper is about implementation, so we must present concrete representations of grammars. We use a set of trees, one for each nonterminal in N. Grammar tree nodes are labelled with a unique integer node index *ni*, a grammar element *el* and two child references named *alt* and *seq*. An appropriate Java declaration is
class GNode{**int** ni; GElement el; GNode alt, seq;}

Using instances of GNode, the grammar
$\Gamma_1 = \{S \longrightarrow b \quad S \longrightarrow aXz \quad X \longrightarrow xX \quad X \longrightarrow yX \quad X \longrightarrow \epsilon\}$
is represented as



In this visualisation, *alt* references are shown as horizontal arrows, and *seq* references as vertical arrows. The instance numbers for grammar nodes are allocated sequentially from a base value which equates to $|T| + |N| + E$ (in this example $5 + 2 + 5 = 12$) where E is the number of element types in the grammar — we shall enlarge on this in Section 4.

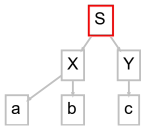
Each rule's right-hand side is represented by a sequence of nodes linked by their seq reference, terminated with an END node and headed by an ALT node; all ALT nodes for a given nonterminal are linked via their alt references and headed by an LHS node labelled with the left hand side nonterminal. For END nodes, seq references the nearest ALT ancestor and alt references the nearest ancestor ALT-header, which for ordinary (non-extended) CFGs will be an LHS node. To avoid cluttering the visualisation, END references are shown as an ordered pair of reference numbers rather than arrows.

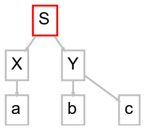
Grammar elements are tuples (*ei, kind, str*) where *ei* is a unique element index, *kind* is one of EOS, T, EPS, N, ALT, END (for end-of-string, terminal, empty string, nonterminal, alternate and end of production) and *str* is a nonterminal or a terminal appropriately. In Java these may be declared as
class GElement {**int** ei; Kind kind; String str;}
enum Kind {EOS, T, EPS, N, ALT, END}

Derivation trees and ambiguity Any production with more than one nonterminal on its right hand side gives rise to multiple derivations in an uninteresting way. For instance the grammar $\Gamma_3 = \{S \longrightarrow XY \quad X \longrightarrow x \quad Y \longrightarrow y\}$ can generate xy in two ways; $S \Rightarrow XY \Rightarrow xY \longrightarrow xy$ and $S \Rightarrow XY \Rightarrow Xy \longrightarrow xy$. A *leftmost derivation* contains only derivation steps in which the first nonterminal in a rule is expanded, and by convention we shall use only leftmost derivations.

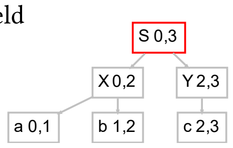
A *derivation tree* is a useful graphical representation of a class of derivations of some string. A derivation tree is an ordered tree in which the root node is labelled with S, each interior node is labelled with an element of N and each leaf is labelled with some element of T or ϵ. If an interior node X has children labelled $x_1, x_2, \dots x_n$ then $X \longrightarrow x_1, x_2, \dots x_n \in P$. The leftmost derivation of $u \in L(\Gamma)$ corresponds to the preorder traversal of the derivation tree of u.

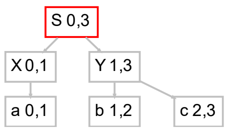
A CFG Γ is *ambiguous* if there is some $u \in L(\Gamma)$ which has more than one leftmost derivation, and thus more than one derivation tree. For instance the grammar $\Gamma_4 = \{S \longrightarrow XY \quad X \longrightarrow a \mid ab \quad Y \longrightarrow bc \mid c\}$ generates the string abc in two leftmost ways; $S \Rightarrow XY \Rightarrow aY \Rightarrow abc$ and $S \Rightarrow XY \Rightarrow abY \Rightarrow abc$. The corresponding derivation trees are





The *yield* of a derivation tree node is the substring whose terminal nodes are descendants of that node. In the example above, we see that the terminal nodes and the root node have the same yield in each derivation, but the nodes labelled X and Y have different yields in the two derivations. In any derivation tree, we can find the yield of a node by descending to its leaves, but when dealing with ambiguities it is convenient to annotate each node with its yield. An *annotated derivation tree* is a derivation tree whose node labels have been extended by the left and right indices of the node's yield





With these annotations we can directly see that the start and terminal nodes of both derivations are 'the same' because their yields are the same in each case, but that the X and Y nodes are different because their corresponding derivation steps generate different substrings of the input.

Recognisers, partial parsers and general parsers A *recogniser* for Γ tests a string u for language containment, i.e. whether $u \in L(\Gamma)$. A *partial parser* tests a string for language containment and returns at least one derivation for at least one string in $L(\Gamma)$. A *general* parser returns all derivations. Most current programming language processors employ partial parsers which (i) admit only a subset of the context free grammars and (ii) return at most one derivation. For many parsing algorithms, (i) is well characterised, but the constraints imposed by (ii) are less well understood in practice for non-trivial cases, which can lead to puzzling outcomes.

As programming languages become more complex, anecdotal evidence shows that implementers increasingly struggle with classical near-deterministic parser generators and resort to hand written front ends. There is a useful discussion at [18] which notes that GCC abandoned Bison based parsers nearly twenty years ago [2, 3]. We hope that the availability of well engineered GLL parsers will allow a return to principled engineering of compiler front ends.

Compiled vs. interpreted parsers We distinguish between *interpreted* parsers which are fixed pieces of code that are parameterised by a data structure encoding the grammar, and *compiled* parsers where the parser code itself encodes the grammar. Classical recursive descent parsers are compiled parsers in that they have a parse function for each nonterminal, and the body of a parse function reflects the nonterminal's productions.

Traditionally, shift-reduce parsers are implemented as interpreted parsers operating over a table that represents an automaton derived from the grammar. However, Penello [9] describes *Recursive Ascent* (a compiled LALR parser), and Aho and Ullman give a table-driven predictive LL(k) parser [1, pp338–341].

Parser context Parsing is a *search* problem in which we traverse both a grammar and an input string to locate derivations. The algorithms we shall examine in this paper all work by processing a current parser *context* comprising an index i into the input string, a current grammar node gn, a current stack, whose top is stack node sn, and a current derivation whose most recent step is derivation node dn.

In detail (i) compiled parsers do not have an explicit gn since the grammar positions correspond to locations in the code; (ii) recognisers do not generate derivations and so do not require a dn and (iii) some parsers make use of the host languages call stack, and thus do not need an explicit sn.

Lexicalisation Most programming language grammars are defined over terminals which have internal structure rather than simple characters, and the input character string is usually *lexicalised* into a sequence of non-overlapping substrings called *lexemes* each of which belongs to a lexical class which is given a number called the *token*. There are several advantages to this scheme — whitespace may elided from the grammar rules; the alphabet of the grammar is the set of lexeme classes not the set of characters and that improves the resolution of lookahead tests; and for many languages regular recognisers may be used for lexicalisation rather than the full power of a context free parser, and that improves performance. The parser itself then works with strings of tokens, not strings of characters.

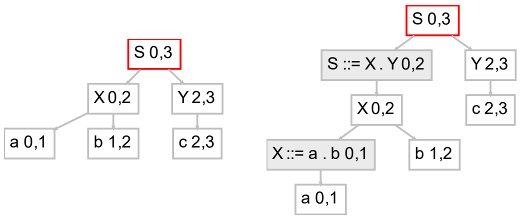
The algorithms presented here all assume that inputs have been lexicalised into a string of tokens using longest match.Token number zero is reserved for the end of string symbol \$.

Representing derivation forests A general parser must return all derivations for any string in the language. Simply returning individual derivation trees is impractical as the number of derivations can grow very quickly. Consider the grammar $\Gamma_5 = \{Z \longrightarrow S \mid SZ \quad S \longrightarrow XY \quad X \longrightarrow a \mid ab \quad Y \longrightarrow bc \mid c\}$ which is Γ_4 extended with a new start rule, allowing one or more abc substrings to be generated. Each instance of S will have two ways to generate its substring, hence the total number of derivations will be 2^k where k is the number of substrings.

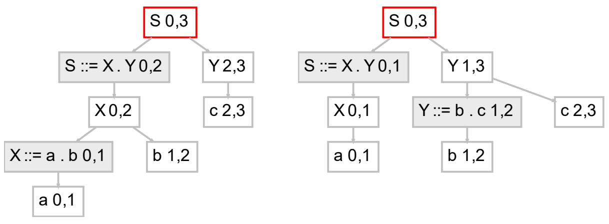
We noted above that some of the elements of the two derivations in Γ_4 are the same in that they have matching labels, and thus generate the same substring of the element. Tomita [20] observed that derivation trees may be combined by *sharing* and *packing* into a directed acyclic graph called a Shared Packed Parse Forest (SPPF). We say that an SPPF *embeds* derivation steps and derivations. If $u \in L(\Gamma)$ then the SPPF for u in Γ will contain a node labelled $S, 0, n$ where S is the start symbol and n is the length of u, in other words a node labelled with the start symbol whose yield is the entire string.

In Tomita's original formulation, SPPFs embed standard derivation trees in which internal nodes are labelled with a nonterminal and have out-degree k_p where k_p is the length of some production p. Whilst building derivations, we need to know both the production we are working on and the position within it. If we were to use this form of SPPF, then in general the dn element of our parser context would need to be a pair $p, j(0 < j < k)$ where j specifies a position within the production. Instead, we binarise our derivation trees by adding additional intermediate nodes.

This binarisation of derivations allows us to encode a complete grammar position into a single SPPF node, so that the dn field in our parser context can comprise a single node. For the grammar $\Gamma_6 = \{S \longrightarrow abc\}$ the single derivation in 'flat' and binarised forms are

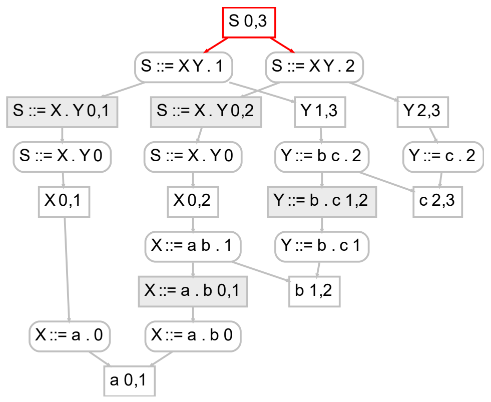


The binarised annotated derivation trees for Γ_4 are



The binarisation or 'intermediate' nodes are labelled with a position *within* a production. We use Knuth's 'item' notation and mark the position with a dot. In visualisations, we shade the binarisation nodes.

The corresponding SPPF constructed from the two annotated derivation trees for Γ_4 is



2 Backtracking Recursive Descent Parsers

To motivate the GLL approach, we begin by examining the simplest useful CFG parser we know of; a compiled non-general backtracking parser with only a single thread of control. We then present the same algorithm implemented in two interpreted styles; the first a 'folding' of the compiled parser and the second a state-machine implementation closely related to the interpreted GLL parser to be discussed in the next section.

Compiled style - osbrdG Here is one style of backtracking parser for
$\Gamma_1 = \{S \longrightarrow b \mid aXz \quad X \longrightarrow xX \mid yX \mid \epsilon\}$

```
boolean p_S() { // Attempt to match nonterminal S
  int eI = i; DNode eDN = dn; //store global variables at entry

  if (input[i]==2/*b*/) {i++; du(13); return true;}

  i = eI; dn = eDN; //recall global variables for next production
  if (input[i]==1/*a*/) {i++; // bump input pointer on success
    if (p_X()) { // call parse function for nonterminal Z
      if (input[i]==5/*z*/) {i++; // bump input pointer on success
          du(16); return true;}}} // end; update derivation
  return false;
}

boolean p_X() { // Attenmpt to match nonterminal S
  int eI = i; DNode eDN = dn; //store global variables at entry

  if (input[i]==3/*x*/) {i++;
  if (p_X()) { du(22); return true;}}

  i = eI; dn = eDN; //recall global variables for next production
  if (input[i]==4/*y*/) {i++;
  if (p_X()) { du(26); return true;}}

  i = eI; dn = eDN; //recall global variables for next production
  /* epsilon */ du(30); return true; // epsilon always matches
}
```

A parse begins by loading the input with the lexicalised sequence of tokens, setting `i` to zero, `dN` to `null` and then calling `p_S()`. The string is accepted if on return, `input[i]` is the end of string symbol.

We call this algorithm *Ordered Singleton Backtrack Recursive Descent* (OSBRD) to emphasise that it (i) treats the productions of a nonterminal in an ordered fashion (which may cause some productions to be ignored) and (ii) returns at most one derivation.

This particular implementation is called `osbrdG` because the code has been *generated* from an input grammar. If we change the grammar, then we must regenerate a new parser. The generated code is effectively a bidirectional 'pretty print' of the grammar, in the sense that one may read the grammar in a single pass and output the parser code, and one may read the code in a single pass and output the grammar. Nigel Horspool calls this the *Recursively Decent* property.

Each nonterminal Z has a corresponding boolean parse function `p_Z()`. On entry, parse functions record the input index at entry `eI` and the entry derivation node `eDN`. The productions $Z \longrightarrow \alpha$ are then examined in the order they were written. The sequence of elements in α is tested using a nest of `if` statement. Nonterminals are tested by calling the corresponding parse function, and terminals by testing the current input character against that terminal's token number (with the current index being incremented on success). If all of a production's tests return true, then the derivation is extended by one step using function `du(n)` and the parse function returns true, otherwise the current index and derivation node are reset to their entry values and the next production is tested. If no production matches, the parse function returns false.

The leftmost derivation is encoded as a linked list of production number — function `du(int n)` updates the current derivation by head-inserting a node; in detail the production number is the number of the corresponding `ALT` node in our representation. A suitable Java declaration for derivation nodes is **class** DNode {**int** altn; DNode next;}

There are three major deficiencies in the OSBRD algorithm (i) on some inputs it does not terminate; (ii) on some inputs it will require exponential time to terminate; and (iii) on some inputs it will terminate but incorrectly reject strings that are in the language.Versions of this algorithm have been reported many times. The most comprehensive treatment is in Aho and Ullman's 1972 monograph [1, pp.56–469] where the technique is called *TDPL*; they note that *It can be quite difficult to determine what language is defined by a TDPL program* which should be read as a warning. We do not recommend the approach for serious work; we use it here merely as a stepping stone to understanding GLL parsing.

Interpreting via a function - osbrdF There is a closely related interpreted implementation that 'folds' all of the parse functions into a single function `osbrdF` which takes a grammar node representing the nonterminal to be tested — in our implementation, we use the corresponding `LHS` node from our representation.

Instead of laying out the productions as nests of `if` statements, we have an outer loop over the `alt` references and an inner loop over the `seq` references enclosing a switch statement which performs the appropriate action for each kind of grammar node. Java (and many other languages) use the dot operator to specify fields from composite data structures hence, for instance, `gn.s` refers to the string field of the current grammar node. We make use of Java's named-continue feature to allow a failed match to immediately proceed to the next `alt` iteration. An implementation in C or C++ might use a `goto` to achieve the same effect.

```
boolean osbrdF(GNode lhs) {
 int ei = i; DNode edn = dn;
 altLoop: for (GNode alt = lhs.alt; alt != null; alt = alt.alt) {
  i = ei; dn = edn;
  GNode gn = alt.seq;
  while (true) {
   switch (gn.el.kind) {
   case T: if (mt(gn)) {i++; ; gn = gn.seq; break;}
                else continue altLoop; // failure; next alternate
   case N: if (osbrdF(lhs(gn))) {gn = gn.seq; break;}
                else continue altLoop; // failure; next alternate
   case EPS: gn = gn.seq; break; // epsilon always matches
   case END: du(alt.ni); return true; // end; update derivation
   }}}
 return false;
}
```

Interpreting with explicit stack management - osbrdE A general parser must handle non-determinism; during processing of some parser context we may identify more than one successor context that must be explored. OSBRD handles some (but not all) nondeterminisms through limited backtracking. Both the `osbrdG` and `osbrdF` implementations rely on the host language's function call mechanism to implictly manage a single stack of nonterminal instances. This is a fundamental weakness, because the sequence of parser contexts that may be examined is limited to those that conform to a last-in, first-out discipline. In a conventional procedural language such as Java without continuations, there is no way of loading the function call stack with a particular state.

A GLL parser works by saving parser contexts for later processing in a way that allows any context to be processed independently of the others, and not surprisingly it uses an explicit stack data structure to allow switching between contexts. Our next step (`osbrdE`) is to implement OSBRD using explicit stack management in a style that matches our GLL baseline implementation. At this stage we still only have a single stack for all contexts since this is only an OSBRD implementation.

As before, derivations are also developed within a linked list of `DNode`s, with the most recent derivation step held in variable `dn`.

Stack entries must contain all of the information in the stack frame for function `osbrdF()`; that is a return position in the grammar and the local variables `eI` and `eDN`. A suitable

Java declaration is
class SNode {GNode rN; **int** eI; SNode next; DNode eDN;}

We model the stack with a linked list of `SNodes` and hold the stack top in variable `sn`. A perhaps unexpected side-effect of removing reliance on the runtime function call stack is that we may no longer use the unwinding of recursive calls to handle some aspects of backtracking. Instead, we explictly traverse the grammar representation, executing stack pops as we go, and this adds complexity to the match fail code at lines 7–13 in `osbrdE`.

Termination of a parse is triggered by popping the root element. If this occurs during back tracking (line 10) then the parse has failed and `osbrdE()` returns false. If the root element is popped whilst processing an `END` node (line 18) then we have reached the end of a production in *S*, and `osbrdE()` returns true; the caller must then check to see whether the entire string has been consumed.

```
1  boolean osbrdE() {
2   initialise();
3   while (true)
4   switch (gn.el.kind) {
5   case T:
6    if (mt(gn)) {i++; gn = gn.seq; break;}
7    else { while (true) { // On failure, backtrack
8     while (gn.el.kind != Kind.END) gn = gn.seq;
9     if (gn.alt.alt == null) { // No more productions; return
10     gn = ret();
11     if (sn == null) return false;} // No more stack frames; fail
12    else { // restore context
13     i = ((StackNode) sn).ei; dn =((StackNode) sn).edn;
14     gn = gn.alt.alt.seq; break;}}}
15   break;
16  case N: call(gn); break;
17  case EPS: gn = gn.seq; break;
18  case END:
19     du(gn.alt.ni); gn = ret(); if (sn == null) return true; break;
20 }}
```

3 A Baseline Interpreted GLL Parser - gllBL

The GLL algorithm takes the basic control flow patterns of our recursive parse functions and recasts them as a collection of separate threads that may be independently executed. The only data items required by a thread are the unchanging input and the four context elements identified earlier — a grammar node `gn`, an input index `i`, a top-of-stack node `sn` and a most-recent derivation step node `dn`.

Each parse thread may thus be uniquely characterised by a 4-tuple *descriptor*, declared in the style we have been using as **class** Descriptor {GNode gn; **int** i; SNode sn; DNode dn;} A descriptor captures the starting context for a thread which is loaded into global variables `gn`, `i`, `sn` and `dn`. These values then evolve during execution of the thread, and often potential new starting contexts are identified for later processing.

At the outermost level, GLL is a worklist algorithm that selects descriptors from a collection of awaiting descriptors. During execution of a thread, new descriptors may be created. For instance when an instance of a nonterminal is encountered, the algorithm will create descriptors for each of that nonterminal's productions.

A GLL parse begins with a single awaiting descriptor `(lhs(S), 0, gssRoot, null)`, that is the `LHS` grammar node for the start nonterminal, input index zero, a reference to the GSS base node and an empty derivation step. The parse terminates when the descriptor collection is empty.

Good performance of the algorithm relies on efficient implementation of the collection of descriptors, the collection of stacks and the collection of derivations. We delay consideration of those mechanisms until after we have examined the control flow aspects of the algorithm.

```
1  void gllBL() {
2   initialise();
3   nextDescriptor: while (dequeueDesc())
4   while (true) {
5    switch (gn.el.kind) {
6     case T: if (input[i] == gn.el.ei)
7               {du(1); i++; gn = gn.seq; break;}
8               else // abort thread on mismatch
9                 continue nextDescriptor;
10    case N: call(gn); continue nextDescriptor;
11    case EPS: du(0); gn = gn.seq; break;
12    case END: ret(); continue nextDescriptor;
13 }}}
```

This top level control flow is pleasingly simple. It has the same structure as for `osbrdE`, but is relieved of the complex backtracking code. This is because descriptors are created for each production by the `call()` function. When a terminal mismatch occurs, we can simply abandon the current thread by executing **continue** nextDescriptor. We do not need to backtrack to find the next viable alternate because it will already either have been processed, or be in the queue for processing. The `call()`, `ret()` and `du()` functions update the SPPF and GSS and are described below.

Thread management In a context free parser, a context once processed need never be processed again (although see notes on *contingent pops* below). The key to achieving a cubic upper bound on performance is to maintain a set of previously encountered descriptors `descS` in addition to a set of descriptors `descQ` awaiting processing. The `call()` function (line 9 above with definition below) then uses function `queueDesc()` to only load a descriptor to `descQ` if it has never been seen before. In this implementation we use the Java double-ended queue `Deque` for `descQ`. Since additions to `descQ` are guarded by checks on `descS`, descriptors can never appear more than once within `descQ`.

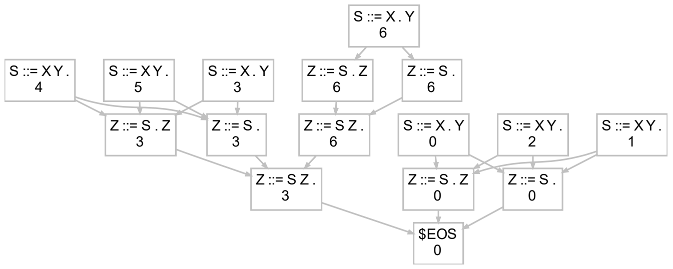
```
Set<Desc> descS; Deque<Desc> descQ;
GNode gn;
GSSN sn;
SPPFN dn;

```

```
void queueDesc(GNode gn, int i, GSSN gssN, SPPFN sppfN) {
  Desc tmp = new Desc(gn, i, gssN, sppfN);
  if (descS.add(tmp)) descQ.addFirst(tmp);
 }

boolean dequeueDesc() {
 Desc tmp = descQ.poll();
 if (tmp == null) return false;
 gn = tmp.gn; i = tmp.i; sn = tmp.sn; dn = tmp.dn;
 return true;
}
```

The `dequeueDesc()` function polls `descQ` to remove the head element and unloads its fields into the global context variables, or returns `false` if no descriptors are available.

Stack management In the parsers discussed here, the purpose of the stack is to store a return point from the call to a nonterminal. A general parser starts with a single stack but new stacks may diverge from it. The shared bases of these stacks naturally need only be represented once, but Tomita [20] observed that when two stacks for a context free parser show the same stack top, their future behaviour will be indistinguishable. As a result, stacks may be *merged* together when they have the same top state, and the whole set of stacks may be represented by a directed acyclic graph called a Graph Structured Stack (GSS). (As an aside, the structure is perhaps more reasonably called a stack structured graph, but the term GSS is in common use.)

This visualisation shows the GSS that results from a GLL parse of Γ_5 and string *abcabc*. Each node has a grammar position (shown as a production with a dot after the nonterminal that was called) and a left index which is value of the input index at the point of call. The GSS is initially loaded with a *stack root* node labelled (EOS, 0) which ensures that all stacks are tied to a common base node.



Paths between the top and bottom node represent the result of successful matches to the whole string. The other paths result from parses which ultimately failed to match the entire string.

GSS nodes contain a grammar node which acts as the return point; an input index `i` which will be the leftmost index for the substring matched by this call; a set of out edges which references the stacks converging on this node; and a set of *contingent pops* which will be discussed below. A suitable Java declaration is

```
class GSSN {GNode gn; int i;
              Set<GSSE> edges; Set<SPPFN> pops;}
```

In a parser GSS edges are also labelled with the derivation node that was current at the time of their creation. A GSS edge also requires a reference to the destination of the edge. An appropriate Java declaration is

```
class GSSE {GSSN dst; SPPFN sppfnode; }
```

The `call()` and `ret()` functions manage the evaluation of threads involving nonterminal instances. The basic requirement is to extend a stack on call, and to retrieve the return point in the grammar on return. In addition, a `call()` must queue descriptors for the start of each production in the called nonterminal (lines 18 and 19 below), and a `ret()` must check to see if we have reached the bottom of the stack, in which case we check for acceptance (lines 23–25).

```
1  Map<GSSN, GSSN> gss;
2  GSSN gssRoot;
3
4  GSSN gssFind(GNode gn, int i) {
5   GSSN gssN = new GSSN(gn, i);
6   if (gss.get(gssN) == null) gss.put(gssN, gssN);
7   return gss.get(gssN);
8  }
9
10 void call(GNode gn) {
11  GSSN gssN = gssFind(gn.seq, i);
12  GSSE gssE = new GSSE(sn, dn);
13  if (!gssN.edges.contains(gssE)) {
14   gssN.edges.add(gssE);
15   for (SPPFN rc : gssN.pops)
16    queueDesc(gn.seq, rc.ri, sn, sppfUpdate(gn.seq, dn, rc));
17  }
18  for (GNode p = rules(gn).alt; p != null; p = p.alt)
19   queueDesc(p.seq, i, gssN, null);
20 }
21
22 void ret() {
23  if (sn.equals(gssRoot)) {
24   if (accepting(gn)) accepted |= (i == input.length - 1);
25   return;
26  }
27  sn.pops.add(dn);
28  for (GSSE e : sn.edges)
29   queueDesc(sn.gn, i, e.dst, sppfUpdate(sn.gn, e.sppfnode, dn));
30 }
```

Perhaps the most subtle part of the GLL algorithm is the handling of *contingent pops*. When we perform a return at a GSS node `sn` we want the parse to continue at all of the return points encoded by the out-edges of `sn`. Edges can be added to a GSS node *after* a pop has occurred, and in such a case the earlier pop (or pops) must be applied to that new edge. Thus each GSS node contains a *pop set* which records pop actions (line 27) so that they may be applied to any new edges that subsequently may be added (lines 15–16 above).

Derivation updates Derivations, as discussed above, are represented using binarised SPPFs.

```
  class SPPFN {GNode gn; int li; int ri;
              Set<SPPFPN> packNS;}
class SPPFPN {GNode gn; int pivot; SPPFN lC; SPPFN rC;}
```

A derivation update may be triggered in two ways (i) the main gllBL algorithm adds elements via function `du()` when a terminal or an ϵ-rule is matched, and (ii) the `call()` and `ret()` functions add elements associated with nonterminals.

The SPPF is modelled as a set of SPPF nodes that carry a set of pack node children which in general will be updated during a parse. A Java quirk is that the standard `Set` API does not easily support updates to set elements. Now, Java sets are implemented as maps from elements to themselves, and so without loss of performance we can explicitly use such a map to allow retrieval of elements.

```
Map<SPPFN, SPPFN> sppf;

SPPFN sppfFind(GNode dn, int li, int ri) {
  SPPFN tmp = new SPPFN(dn, li, ri);
  if (!sppf.containsKey(tmp)) sppf.put(tmp, tmp);
  return sppf.get(tmp);
}

SPPFN sppfUpdate(GNode gn, SPPFN ln, SPPFN rn) {
  SPPFN ret = sppfFind(gn.el.kind == Kind.END ? gn.seq : gn,
                        ln == null ? rn.li : ln.li,
                        rn.ri);
  ret.packNS.add(
    new SPPFPN(gn, ln == null ? rn.li : ln.ri, ln, rn));
  return ret;
}

void du(int width) {
    dn = sppfUpdate(gn.seq, dn, sppfFind(gn, i, i + width));
}
```

Function `sppfFind()` returns an existing SPPF node, or adds a new SPPF node with empty pack node set.

Function `sppfUpdate()` takes a grammar node and two existing SPPF nodes and either identifies or creates the corresponding subtree.

Initialisation The core data structures are re-initialised for each parse. Lines 2-3 create new, empty, data structures, lines 4-5 creates the base node for the GSS, line 6 initialises the global context variables and lines 7-8 load desriptors for each production of the start symbol.

```
1 void initialise() {
2   descS = new HashSet<>(); descQ = new LinkedList<>();
3   sppf = new HashMap<>(); gss = new HashMap<>();
4   gssRoot = new GSSN(grammar.endOfStringNode, 0);
5   gss.put(gssRoot, gssRoot);
6   i = 0; sn = gssRoot; dn = null;
7   for (GNode p = grammar.startNode.alt; p != null; p = p.alt)
8     queueDesc(p.seq, i, sn, dn);
9 }
```

4 Data Structure Optimisation

In *The Design and Evolution of C++* [19, p.211] Stroustrup wrote

Many programs create and delete a large number of small objects of a few important classes ... The allocation and deallocation of such objects with a general-purpose allocator can easily dominate the run time and sometimes the storage requirements of the programs.

The GLL algorithm is a perfect exemplar of this situation since it performs very little actual calculation since the algorithm is dominated by the conditional creation of small elements under control of tests based on simple integer comparisons. When built with the native Java APIs in an object oriented fashion as we have done in gllBL, runtime is dominated by allocation of small objects.

Stroustrup advocated the use of custom heap management for these kinds of programs, and claimed speedups of as much as an order of magnitude for applications in which the heap became heavily fragmented. GLL does not fragment the heap in that our core data structures (the GSS, the SPPF and the descriptor set) only ever grow — essentially the only deallocations that occur are the worklist elements that reference descriptors (which themselves are not deallocated). As a result we might not expect Stroustrup's fragmentation-driven factor ten speedup, but as we shall show in section 5, the scheme discussed in this section does yield speedup factors of 3–4 over the matching Java API implementation, whilst reducing memory demands by a factor of between 4 and 5.

The general approach is to (i) map all grammar objects including grammar positions onto the integers in a single sequence and (ii) to allocate elements sequentially from a pool of memory blocks. Essentially we eschew the use of Java objects, and make each grammar 'object' a small sequence of integers with contiguous memory addresses.

Enumeration of grammar elements The primitive objects manipulated by the GLL algorithm are: The end of string symbol, the terminals, the epsilon symbol, the nonterminals the elements not otherwise accounted for (such as `ALT` and `END`) and finally the grammar nodes. We arrange all of these in a sequence, and number them from zero. So, for Γ_1 the sequence is 0:EOS 1:a 2:b 3:x 4:y 5:z 6:EPS 7:S 8:X 9:ALT 10:END

The sequence then continues with the grammar nodes, the first of which is always the node which labels the GSS root (11 in this case), followed by the nodes representing the grammar rules as shown on page 1. Hence the node numbers there start at 12.

The purpose of this enumeration is to avoid the need to carry type information around. We can tell, for instance, if an element of the sequence is a terminal simply by checking that it is greater than zero and less than the sequence value for ϵ. Terminals appear first in the sequence because algorithms employing lookahead test input tokens against sets of terminals, and a bit vector representation is more space efficient if the elements in the set have small values.

Pool based memory management gllBL maintains six sets — GSS nodes, GSS edges, pop elements, SPPF nodes, SPPF packed nodes and descriptors. Our main goal is to reduce the volume of calls to the system's heap allocation routines for these sets. We do this by explictly coding C-like structures for each set element and allocating them sequentially into a pool of memory blocks. The elements require 5, 5, 5, 7 and 6 integers each respectively, and our memory blocks are 2^{26} integers long (that is 256MByte each for 32-bit integers), so the number of calls to the system allocator is negligible.

Our memory references are 32-bit integers which can manage up to 4G integer locations, or 16Gbyte of memory. Since our blocks sizes are always a power of 2, we can use shift and mask operations to extract block number and address within the block

```
int poolGet(int index) {
 return
 pool[index >> poolAddressOffset][index & poolAddressMask];
}

void poolSet(int index, int value) {
 pool[index >> poolAddressOffset][index & poolAddressMask] =
 value;
}
```

The sets themselves are implemented as hash tables using separate chaining. The chain link is at offset zero in each of the structure elements. As always with hash tables, the effectiveness of the hash function and the table's load factor will dictate performance. We will illustrate the load factor impact in Section 5.

The GLL Hash Pool implementation – gllHP Our final implementation is gllBL modified to use this *Hash Pool* memory management scheme. The full code is available in the repository. This is the top level control flow. It is essentially identical to gllBL except that all context elements are integers which are used as references into the pool data; the lookup table `kindOf` is used to encode the grammar element's type; and advancing gn to the next sequence element simply requires gn to be incremented, since grammar nodes are numbered sequentially.

```
void gllHP() {
 initialise();
 nextDescriptor: while (dequeueDescriptor())
 while (true) {
  switch (kindOf[gni]) {
  case T: if (input[i] == elementOf[gni])
                 {d(1); i++; gni++; break;}
                 else continue nextDescriptor;
   case N: call(gni); continue nextDescriptor;
   case EPS: d(0); gni++; break;
   case END: ret(); continue nextDescriptor;
}}}
```

5 Performance Evaluation

Our intention is that gllBL and gllHP will provide reference performance data for future studies. To that end, the repository includes large corpora of Java 18 and Standard ML code with associated grammars, and we expect to expand those holdings in future. In this paper we restrict ourselves to six grammars and only five strings, but these strings are large — in the range 84–130 kBytes. We take this approach because we want to concisely present throughput and memory consumption figures for large real-world examples.

As well as gllBL and gllHP, we present some data for gllOpt which is an older implementation of GLL that is compiled, uses lookahead on both descriptor creation and pops and has the same hash pool data structure mechanisms as gllHP. gllOpt is not as tightly engineered for performance since it has support for trace messages and statistics gathering code, as well as support for some of the other GLL variants mentioned above. Results in all categories improve as we move from gllBL to gllHP to gllOpt; we would expect to see further improvements in gllOpt when a performance engineered version is available.

We use a range of programming language grammars — the ANSI-C grammar from the Kernighan and Ritchie textbook; the ANSI C++ grammar from the 1997 Public Review Document which underpinned C++98; the C# version 1.2 grammar and the Java Language Specification version 1 and 2 grammars. Larger studies using Java 18 and Standard ML may be found in the repository. The JLS2 grammar uses extended CFG constructs. We produced two variants by expanding closure using left or right recursion; we would expect the left recursive version to be more demanding of a GLL parser.

Test strings include the full source code for the parser generators RDP (C), GTB (C) and ART (C++) along with a Twitter client (C#) and multiple concatenations of an implementation of Conway's Game of Life (Java).

Data structure cardinalities We begin by looking at the cardinalities of the various GLL data structures (Table 2). Of course, gllBL and gllHP generate exactly the same cardinalities so we only compare gllBL to gllOpt which uses lookahead. As expected, the lookahead significantly reduces the number of descriptors and indeed the cardinalities of the other sets too. It is clear from this table that the amount of ambiguity encountered drives the size of these sets. That is as we should expect since were the grammars LL(1) then there would be no nondeterminism and we might hope to approach a linear cost. GLL performance is worse case cubic in the length of the input, and as the level of nondeterminism goes up we would expect to move towards that cubic bound. Thus the rightmost column is important. We can see that ANSI C++ is much more ambiguous than ANSI C when run on gtbSrc and rdpSrc.

Throughput Speed measurements in Table 3 were made using a Dell XPS 15 9510 laptop with 16GByte of installed

Table 1. Heap utilisation for ANSI C++

String	Heap BL	Heap HP	Factor	Pool
artSrc	3,165,999,856	545,260,192	5.81	562,135,472
gtbSrc	3,637,287,192	817,889,952	4.45	666,862,111
rdpSrc	2,814,125,624	545,260,192	5.16	506,684,413

memory and an Intel Core i7-11800H eight-core processor running at 2.3GHz. The experiments were run from the command line under Microsoft Windows 10 Enterprise version 10.0.19042 using Oracle's Java HotSpot(TM) 64-Bit Server VM (build 14.0.2+12-46, mixed mode, sharing).

The nanosecond timing routines in the Java System API do not accurately reflect computational load in multicore systems and can even return negative values. As a result we used the System.currentTimeMillis() to measure runtimes even though its resolution is only around 0.03 seconds; for each experiment we made 10 runs and report here the mean run time in milliseconds.

We deliberately disabled the resizing of hash tables in gllHP and gllOpt, and set the size of the tables to be twice that required to handle the gtbSrc input when run with the ANSI-C grammar (the highlighted first line). The throughput in tokens per second ranges from 113,458 for JLS2 right down to 6,701 for gtbSrc and ANSI C++. Looking at the cardinailities table, it is clear that much of this variation arises from the hash table load factor.

Hash table load factor To further investigate this effect, Table 4 shows partial histograms of bucket occupancy in the hash tables. The ANSI C++ examples show significant hash table congestion, with in one case more than 10% of the bucket lists having four or more elements.

Heap utilisation We wanted to get a measure of memory consumption, comparing gllBL to gllHP. It is quite difficult to measure heap utilisation in Java but we can approximate it by asking for the free heap size before and after a run. Table 1 shows the results for the three pieces of C/C++ source code running with the ANSI C++ grammar. As a consistency check we have also included (in column Pool) the *computed* size of all of the data structure elements which we can derive from the known data structure cardinalities and the size of their elements. These figures should clearly be treated with great caution, but they do indicate that a factor four reduction in memory footprint is achieved with the HashPool implementation. This is unsurprising.

6 Potential Future Work on Optimisation

GLL was introduced at the 2009 LDTA workshop [10]. A variety of improvements to the basic GLL algorithm have been reported since then which we summarise here along with some ideas that have not (as far as we know) appeared yet in the literature. In future work, we shall present implementations of some of these ideas in the same style as our baseline (gllBL) and hashpool (gllHP) implementations so as to produce a consistent evaluation of their strengths and weaknesses. In what follows, each paragraph is a separate optimisation opportunity.

Thread management gllBL and gllHP do not use lookahead. Wherever there is a break in control flow, we can reduce the number of descriptors being created by using lookahead to suppress descriptors for threads that will immediately terminate; the lookahead effectively allows us to pre-compute whether the first match operation in a thread will fail. In fact the effect can be quite large, since where we have rules of the form $X \longrightarrow \alpha Y \beta \quad Y \longrightarrow Z\gamma \quad Z \longrightarrow \delta$ and gn corresponds to $X \longrightarrow \alpha \cdot Y\beta$, a failing lookahead test will suppress descriptor creation for both Y and Z. We can also use lookahead in the `ret()` function to suppress descriptor creation when the current input symbol is not in the FOLLOW set of the left hand side of the current rule. We give some initial results from lookahead implementations in the next section.

Scott McPeak reported on *Elkhound* [8] which is a table-driven generalised LR parser which runs deterministically on those parts of the table which are context free. In GLL, descriptors also only need to be created when true nondeterminism is detected. The conditions under which descriptor creation may be suppressed are yet to be fully studied, but we note that a combination of lookahead and FIFO-style short circuiting effectively ensure deterministic execution for calls to rules which are LL(1), since the alternate rules will have disjoint FIRST sets so at most one new thread can exist.

The presentation here uses the language of threads to describe GLL control flow, and it is natural to wonder whether a truly multi-threaded implementation running on a modern multi-core processor would demonstrate significant speedups. Initial experiments using Java threads have not been encouraging, which is perhaps to be expected since as we have already noted, GLL performs very little actual computation since nearly all actions are conditional data structure updates, and those data structures are global to all threads. Hence the ratio of inter-thread communication to in-thread computation is high. However, our experience with gllHP has shown that the general purpose Java libraries cannot compete with a tuned implementation, and so we might imagine that there are highly tuned approaches to distributing the GLL algorithm over multiple processors that might be worthwhile. The current ubiquity of multi-core hardware makes this an attractive goal for further research.

In principle, the number of contingent pop actions may be reduced by choosing a suitable execution order for the descriptors. In practice it is not clear whether there are significant gains to be had since the work associated with the pop has to be performed under any ordering, and the overhead of scanning the pop list for each GSS node is not great. There may be cache effects that can be exploited if we achieve spatial locality of actions, and that suggests that processing

Table 2. Effect of lookahead on data structure cardinalities

Grammar	String	Tokens	Mode	Descriptors	GSS Node	GSS Edge	Pops	Symbol	Packed	Ambig
ANSI C	gtbSrc	36,828	Opt	4,178,345	564,437	2,042,843	559,859	297,677	261,401	515
			gllBL	6,578,603	946,975	2,989,166	776,934	881,128	829,463	526
ANSI C	rdpSrc	26,552	Opt	3,122,638	417,204	1,510,486	425,730	222,206	195,799	138
			gllBL	4,803,532	699,089	2,219,720	567,128	637,043	602,230	139
ANSI C++	artSrc	36,445	Opt	9,493,519	1,036,075	4,755,333	874,868	473,257	475,542	27,362
			gllBL	20,250,528	2,496,038	8,069,723	1,227,578	1,310,876	1,306,193	49,682
ANSI C++	gtbSrc	36,828	Opt	13,061,222	1,270,903	6,392,785	1,110,400	561,139	562,843	26,081
			gllBL	24,091,341	2,647,731	9,650,268	1,430,990	1,531,742	1,513,670	50,039
ANSI C++	rdpSrc	26,552	Opt	9,687,071	942,742	4,709,390	841,963	425,385	426,291	18,925
			gllBL	18,294,156	2,056,206	7,427,350	1,061,367	1,142,989	1,125,019	35,806
C# 1.2	twitter	33,841	Opt	2,024,014	443,304	1,056,916	390,990	255,343	225,052	2,670
			gllBL	4,659,342	1,140,430	2,170,525	555,474	639,254	604,999	11,460
JLS1	life	36,976	Opt	2,302,532	505,179	1,249,154	402,501	260,377	223,401	0
			gllBL	4,175,967	883,950	1,948,492	599,751	710,803	655,277	0
JLS2 left	life	36,976	Opt	858,335	262,104	395,704	316,328	343,729	313,678	24,725
			gllBL	3,475,876	699,565	1,022,346	462,108	745,514	620,736	30,650
JLS2 right	life	36,976	Opt	783,455	266,303	375,252	296,777	336,505	305,478	23,500
			gllBL	3,196,289	642,062	822,362	413,005	719,486	582,283	30,650

Table 3. Throughput using hash table tuned for load factor 2 on ANSI C parsing gtbsrc

Grammar	String	Characters	Tokens	CPU seconds			Speedup		Throughput tokens s^{-1}		
				BL	HP	Opt	HP	Opt	BL	HP	Opt
ANSI C	**gtbsrc**	**117,557**	**36,828**	**4.14**	**1.32**	**0.85**	**3.13**	**4.90**	**8,888**	**27,801**	**43,537**
ANSI C	rdpsrc	84,778	26,552	2.76	0.92	0.63	3.00	4.39	9,605	28,833	42,126
ANSI C++	artsrc	118,922	36,445	14.67	5.30	3.87	2.76	3.79	2,485	6,870	9,425
ANSI C++	gtbsrc	117,557	36,828	18.82	6.92	5.50	2.72	3.42	1,957	5,321	6,701
ANSI C++	rdpsrc	84,778	26,552	13.95	4.64	3.84	3.01	3.64	1,903	5,728	6,923
C# 1.2	twitter	131,323	33,841	3.38	0.84	0.67	4.02	5.06	9,998	40,239	50,554
JLS 1	life	125,594	36,976	2.67	0.81	0.62	3.31	4.30	13,871	45,899	59,639
JLS2 left	life	125,594	36,976	1.89	0.56	0.35	3.40	5.46	19,543	66,539	106,651
JLS2 right	life	125,594	36,976	1.75	0.49	0.33	3.58	5.36	21,182	75,802	113,458

descriptors in a first-in, first-out manner might be advantageous. In the gllBL implementation we have used a double ended queue so as to explore such effects. gllHP uses a stack to hold the descriptors, and thus is FIFO.

If we are using FIFO descriptor scheduling then we can short circuit the enqueue/dequeue operations for the final descriptor in a call action, since once loaded it will be immediately unloaded, so we might as well directly load the context variables.

Derivation representation The binarised SPPF as described here has obvious redundancies. In the binarisation scheme above, the last element of each production has an intermediate node parent with only one child, and this can be suppressed with the element directly attached as the left child of the preceding intermediate node, so in general we would only need $k-2$ binarisation nodes for sequences of length $k (k > 2)$, and no binarisation node for a sequence of length 1 or 2.

Terminal nodes themselves can be omitted since the parent pack node and its parent symbol node contain the indices into the string for that terminal.

Pack nodes are only required where there is ambiguity, and as we shall see in Table 2 below, for current programming language grammars the proportion of symbol and intermediate nodes that are ambiguous is small, thus large potential savings are possible.

If pack nodes are labelled with the left and right indices from their parent symbol, then they contain all of the information required to encode the derivation forest and the parent symbol and intermediate nodes are redundant. This is

Table 4. Effect of suboptimal hash table load factor

Grammar	String	Mode	Pool bytes	Pool/tok	1	2	3	≥ 4	% ≥ 4
ANSI C	gtbSrc	Opt	119,434,167	3,243	5,249,823	1,065,880	139,023	25,498	0.39
		gllHP	191,663,728	5,204	7,937,594	1,960,621	320,110	44,516	0.43
ANSI C	rdpSrc	Opt	89,192,035	3,359	4,235,001	701,905	69,906	11,063	0.22
		gllHP	140,115,278	5,277	6,652,636	1,193,682	143,439	14,323	0.18
ANSI C++	artSrc	Opt	266,870,915	7,323	7,738,658	2,994,531	757,995	257,778	2.19
		gllHP	560,677,098	15,384	8,812,860	5,823,056	2,723,507	1,381,572	7.37
ANSI C++	gtbSrc	Opt	364,085,364	9,886	8,237,055	4,063,565	1,352,693	581,128	4.08
		gllHP	665,091,529	18,059	8,216,549	6,298,134	3,450,289	2,182,890	10.83
ANSI C++	rdpSrc	Opt	270,071,141	10,171	7,638,987	2,995,175	762,042	260,622	2.24
		gllHP	505,374,437	19,033	9,034,932	5,393,298	2,293,843	1,019,367	5.75
C#	twitter	Opt	60,884,465	1,799	3,495,829	397,677	30,893	2,880	0.07
		gllHP	138,587,616	4,095	6,672,188	1,269,676	163,048	17,001	0.21
JLS1	life	Opt	68,910,630	1,864	3,820,275	485,504	43,484	5,234	0.12
		gllHP	125,525,127	3,395	6,319,822	1,108,786	128,263	12,761	0.17
JLS2 left	life	Opt	29,385,111	795	2,204,270	132,437	6,585	242	0.01
		gllHP	102,413,841	2,770	5,227,440	761,699	82,727	6,671	0.11
JLS2 right	life	Opt	27,315,451	739	2,102,990	121,011	5,915	251	0.01
		gllHP	93,771,492	2,536	4,848,491	651,856	65,324	5,844	0.10

the basis if the *Binary Subtree Representation* (BSR) described in [15]. As well as reducing memory requirements, this approach reduces derivation updates to set-addition of BSR elements which simplifies and speeds up the operation.

The use of the Kleene and Positive closures can act as hints to the parser to use iteration rather than recursion which may yield performance improvements. Extended context free grammars offer opportunities to both reduce stack activity and compress derivations. Extending a GLL *recogniser* to handle extended CFG constructs is straightforward, but correctly embedding all derivations in the SPPF for a GLL *parser* requires care. A complete scheme is given in [13] in which extended constructs are referred to as 'bracketed' constructs.

7 Concluding Remarks

We have discussed two reference implementations of GLL (i) gllBL which uses standard Java API objects to implement the core data structures and (ii) gllHP which uses explicit memory management.

The key question is whether GLL is a plausible engineering option compared to classical approaches. There is no question that the approach is *very* expensive compared to the near-deterministic techniques developed in the 1970s; in some cases gllHP needs as much as 8kbytes of memory per input character. However even for very long inputs of over 100kByte characters parsed using the ANSI C++ grammar with its many ambiguities, gllHP needs no more than 0.8Gbyte of memory which is only 5% of the memory on a typical modern 16GByte laptop computer, thus these gargantuan memory demands are not a practical problem.

Throughput is also much less than for a classical parser, but similarly manageable on modern hardware. We propose the informal metric *Good Enough for Gnu* (GEG) as a threshold test for utility. We imagine that the current GNU C compiler is re-engineered with a GLL parser. Assuming that the existing classical parser takes negligible resources, a parser is GEG if it slows GNU C down by no more than 10%.

The underlying source code for the gtbSrc string comprises 996,776 characters which when compiled with GNU C++ in its default mode requires 10.5s, and when compiled with -Ofast, 18.8s. Hence we would like our general parsers to process this string in at most 1–2 seconds.

gtbHP processes gtbSrc in 1.35s using the ANSI C grammar, and 7.02s for the much more challenging ANSI C++ grammar. However gtbHP is a baseline implementation (albeit with efficient memory management), is interpreted and is written in Java. Informal experiments indicate that adding in lookahead and using a compiled parser will improve throughput by a factor of around two; that converting the code to ANSI C will produce another factor two improvement; and that setting the hash table load factors appropriately for ANSI C++ (rather than ANSI-C as here) may give a further factor 1.5 improvement.

References

[1] Alfred V. Aho and Jeffrey D. Ullman. 1972. *The Theory of Parsing, Translation, and Compiling.* Prentice-Hall, Inc., USA.

[2] GNU. 2023. New C parser. https://gcc.gnu.org/wiki/New_C_Parser. Accessed: 2023-07-06.

[3] GNU. 2023. New C parser [patch]. https://gcc.gnu.org/legacy-ml/gcc-patches/2004-10/msg01969.html. Accessed: 2023-07-06.

[4] Anastasia Izmaylova, Ali Afroozeh, and Tijs van der Storm. 2016. Practical, General Parser Combinators. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (St. Petersburg, FL, USA) *(PEPM '16)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2847538.2847539

[5] Adrian Johnstone and Elizabeth Scott. 2011. Modelling GLL Parser Implementations. In *Software Language Engineering*, Brian Malloy, Steffen Staab, and Mark van den Brand (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 42–61.

[6] Adrian Johnstone and Elizabeth Scott. 2015. Principled software microengineering. *Science of Computer Programming* 97 (2015), 64–68. https://doi.org/10.1016/j.scico.2013.11.018 Special Issue on New Ideas and Emerging Results in Understanding Software.

[7] Paul Klint, Tijs van der Storm, and Jurgen Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. 168–177. https://doi.org/10.1109/SCAM.2009.28

[8] Scott McPeak and George C. Necula. 2004. Elkhound: A Fast, Practical GLR Parser Generator. In *Compiler Construction*, Evelyn Duesterwald (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 73–88. https://doi.org/10.1007/978-3-540-24723-4_6

[9] Thomas J. Pennello. 1986. Very Fast LR Parsing. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction* (Palo Alto, California, USA) *(SIGPLAN '86)*. Association for Computing Machinery, New York, NY, USA, 145–151. https://doi.org/10.1145/12276.13326

[10] Elizabeth Scott and Adrian Johnstone. 2010. GLL Parsing. *Electronic Notes in Theoretical Computer Science* 253, 7 (2010), 177–189. https://doi.org/10.1016/j.entcs.2010.08.041 Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009).

[11] Elizabeth Scott and Adrian Johnstone. 2013. GLL parse-tree generation. *Science of Computer Programming* 78, 10 (2013), 1828–1844. https://doi.org/10.1016/j.scico.2012.03.005 Special section on Language Descriptions Tools and Applications (LDTA'08 & '09).

[12] Elizabeth Scott and Adrian Johnstone. 2016. Structuring the GLL parsing algorithm for performance. *Science of Computer Programming* 125 (2016), 1–22. https://doi.org/10.1016/j.scico.2016.04.003

[13] Elizabeth Scott and Adrian Johnstone. 2018. GLL syntax analysers for EBNF grammars. *Science of Computer Programming* 166 (2018), 120–145. https://doi.org/10.1016/j.scico.2018.06.001

[14] Elizabeth Scott and Adrian Johnstone. 2019. Multiple Lexicalisation (a Java Based Study). In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering* (Athens, Greece) *(SLE 2019)*. Association for Computing Machinery, New York, NY, USA, 71–82. https://doi.org/10.1145/3357766.3359532

[15] Elizabeth Scott, Adrian Johnstone, and L. Thomas van Binsbergen. 2019. Derivation representation using binary subtree sets. *Science of Computer Programming* 175 (2019), 63–84. https://doi.org/10.1016/j.scico.2019.01.008

[16] Elizabeth Scott, Adrian Johnstone, and Robert Walsh. 2023. Multiple Input Parsing and Lexical Analysis. *ACM Trans. Program. Lang. Syst.* 45, 3, Article 14 (jul 2023), 44 pages. https://doi.org/10.1145/3594734

[17] Daniel Spiewak. 2023. gll-combinators. https://index.scala-lang.org/djspiewak/gll-combinators. Accessed: 2023-09-05.

[18] StackOverflow. 2023. Are GCC and Clang parsers really handwritten? https://stackoverflow.com/questions/6319086/are-gcc-and-clang-parsers-really-handwritten. Accessed: 2023-07-06.

[19] Bjarne Stroustrup. 1995. *The Design and Evolution of C++*. ACM Press/Addison-Wesley Publishing Co., USA.

[20] Masaru Tomita. 1985. An Efficient Context-Free Parsing Algorithm for Natural Languages. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 2* (Los Angeles, California) *(IJCAI'85)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 756–764.

[21] L. Thomas van Binsbergen, Elizabeth Scott, and Adrian Johnstone. 2018. GLL Parsing with Flexible Combinators. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering* (Boston, MA, USA) *(SLE 2018)*. Association for Computing Machinery, New York, NY, USA, 16–28. https://doi.org/10.1145/3276604.3276618

Received 2023-07-07; accepted 2023-09-01

Sharing Trees and Contextual Information: Re-imagining Forwarding in Attribute Grammars

Lucas Kramer
krame505@umn.edu
University of Minnesota
USA

Eric Van Wyk
evw@umn.edu
University of Minnesota
USA

Abstract

It is not uncommon to design a programming language as a core language with additional features that define some semantic analyses, but delegate others to their translation to the core. Many analyses require contextual information, such as a typing environment. When this is the same for a term under a new feature and under that feature's core translation, then the term (and computations over it) can be shared, with context provided by the translation. This avoids redundant, and sometimes exponential computations. This paper brings sharing of terms and specification of context to forwarding, a language extensibility mechanism in attribute grammars. Here context is defined by equations for inherited attributes that provide (the same) values to shared trees. Applying these techniques to the ableC extensible C compiler replaced around 80% of the cases in which tree sharing was achieved by a crude mechanism that prevented sharing context specifications and limited language extensibility. It also replaced all cases in which this mechanism was used to avoid exponential computations and allowed the removal of many, now unneeded, inherited attribute equations.

CCS Concepts: • **Software and its engineering** → **Translator writing systems and compiler generators**.

Keywords: compilers, attribute grammars, modular and extensible languages, static analysis, well-definedness

ACM Reference Format:
Lucas Kramer and Eric Van Wyk. 2023. Sharing Trees and Contextual Information: Re-imagining Forwarding in Attribute Grammars. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 14 pages.
https://doi.org/10.1145/3623476.3623520

```
1  for i in 0 to f(x) {  g(i);  }
```

```
1  { var i : int = 0;
2    var _v0 : int = f(x);
3    while i < _v0 {  g(i);  i := i + 1;  }  }
```

Figure 1. A for-loop example (top), introduced as a feature that translates to the code with a while-loop (bottom).

1 Introduction

Defining all semantic analyses, optimizations, and translations for all constructs in a full-featured programming languages can be a daunting task. One way to address this is to design the implementation with a smaller *core* language containing some collection of the essential language constructs and semantic analyses. Additional features are then layered on top of this core and provide a translation of the new feature into the core language, *e.g.* a for-loop with integer bounds may translate down to a while-loop in the core, as shown in Figure 1. This alleviates the need to specify certain semantic analyses or tasks, such as code generation, on non-core features by delegating them to the translation.

While other tasks, such as type checking, can also be delegated to the translation, this leads to reporting error messages on generated code instead of the code written by the programmer. If, *e.g.* the expression `f(x)` for the upper-bound on the for-loop in Figure 1 is not of type integer then an error about a type-mismatch on a variable declaration (line 2) may be reported and be nonsensical to the programmer. Thus, it is helpful to explicitly manage some aspects of compilation, such as type checking and error reporting, but dispatch other tasks, such as code generation, to the translation. Note that this requires managing the contextual information needed by sub-terms by passing this information down the syntax tree, *e.g.* an environment mapping variables to their types. In doing so, the language indicates that some semantic aspects of the non-core feature are *equivalent* to those of its translation but that some are not.

Not only does designing languages in this way save effort, it also leads to a more modular development as work on the core language (once it is established) can be isolated from work on non-core features and be carried out by different language developers. With proper tool support, modular

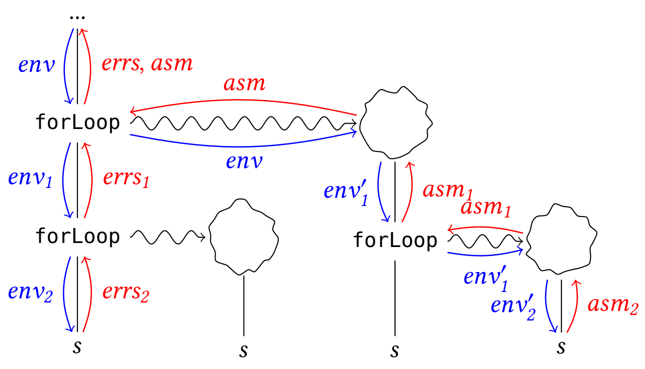


Figure 2. A portion of the decorated tree resulting from `for ... in ... { for ... in ... { s }}`. For clarity, the loop variable and bounds are not shown.

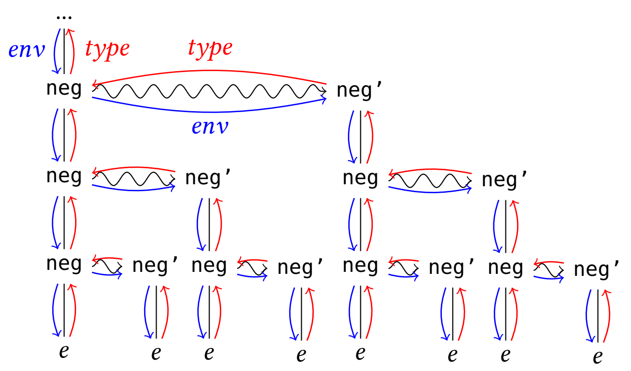


Figure 3. The resolution of operator overloading on a unary negation expression `~ (~ (~ e))` resulting in an exponential number of trees being created and traversed when computing `type`. `neg'` is the specialized implementation of `neg` for the type of e. Some labels are omitted for clarity.

design can enable a particular form of *extensible languages* in which new, often domain-specific, language constructs and static analyses can be developed independently, and composed together to form a language for problems that involve multiple domains. ExtendJ [3] and SugarJ [5] allow new features to be added to Java, and XOC [2] and ableC [11] allow new features to be added to C in a modular way.

Figure 2 shows aspects of the syntax trees created in error checking and code generation of the for-loop example in Figure 1; it also illustrates a potential problem with this approach. For two nested for-loops, the original tree with body s is shown on the left (the loop variables and bounds are not shown). The propagation of a typing environment for error checking (`errs`) is indicated by `env` that is propagated along those edges. The squiggly edges facing right indicate the translation of the loops, with the clouds representing the while-loop code containing s. Note that the translation of the outer for-loop includes the inner for-loop, an approach used in the *forwarding* technique [26] used in ableC to handle independent language extensions. This avoids inappropriately translating-away constructs from other independent extensions. The assembly language translation process to construct `asm` (which may also depend on types and thus `env`) is dispatched from the for-loops to their translations and this process takes place on right-most trees, the ones whose nodes have an `env` propagated down (or over) to them. This will provide two copies of s with an environment, but the middle two instances of s would not be constructed nor visited in these two compiler tasks.

In some cases, this incremental translation may even result in an exponential number of trees to traverse. This can happen in some instances of type-based operator overloading, in which the type of the result of an overloaded construct is not determined explicitly, but is instead computed on the overloaded construct's translation, through forwarding. Consider an overloaded negation operator `~` and the type-checking of the expression `~ (~ (~ e))`. Following this pattern, type checking results in creating and traversing an exponential number of trees, 8 in this case, as illustrated in Figure 3. The overloaded `~` operator, represented as `neg`, will type-check its sub-expression since that type is used to determine the translation, the `neg'` nodes in Figure 3. The type of a `neg` expression is determined by its `neg'` translation and thus we need to type check e under both operators. An enclosing (middle) `neg` expression queries its child to determine its translation and that translation will do the same. The results in type checking e 4 times. Another enclosing negation operator repeats the process and then e is checked 8 times, an exponential growth in the number of trees created and traversed results. In fact, this phenomenon can also occur with seemingly predictable constructs like the for-loop when, *e.g.* another extension uses both `errs` and `asm` results from a child for-loop to compute `asm`.

A second problem faced in specifying language features in this manner is the need to explicitly manage the flow of contextual information down to the components of the new language feature. In some cases this can be difficult, in others only tedious. In Figure 2, a `forLoop` construct extends its incoming environment *env* to include the declaration of the loop variable. This extended environment is passed as env_1 to the nested loop, which does the same to pass env_2 for the body `s`. It is important to make sure that the environments supplied by a production are the same as, or at least compatible, with those determined on the translation. Here, this entails ensuring that env_1 (or env_2) is compatible with env'_1 (or env'_2) as determined on the translation. As we will see, this is straightforward for a `forLoop`, but in the more sophisticated language extensions found in ableC this can be more difficult.

A solution to both of these problems is for component trees to be shared by the non-core feature and its translation. This is depicted on the left in Figure 4, in which the loop

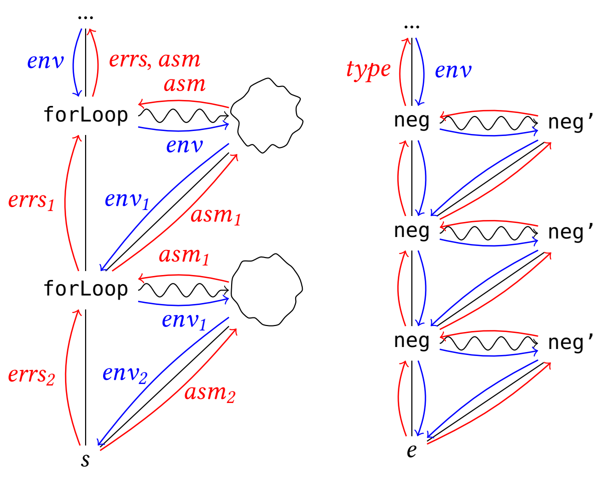


Figure 4. Alternative versions of the trees in Figure 2 and Figure 3 in which decorated children are shared between the forwarding and the forwarded-to trees.

body *s* exists only once and the env context is specified by the translation. (The lower and upper bounds of the for-loop are shared in a similar way.) Thus, the for-loop construct can still access type information on those expressions and generate appropriate error messages when those expressions do not have an integer type. Tree sharing also eliminates the exponential number of trees in the negation example, as seen on the right in Figure 4. The ABLEC system frequently uses a crude approach to sharing that avoids this duplication of trees but limits the extensibility of the language feature, the primary goal of ABLEC.

In addition to these concerns, there are also instances when a new language construct needs to supply contextual information to its components that differs from that supplied by the translation. For example, a pretty-printing task may provide an indentation level to its sub-terms and this value would differ for the loop body under these two constructs. If for a language feature the translation is not "macro-like" (as in the for-loop example) but instead must be computed from semantic information, such as the types of sub-terms, then these explicit contextual specifications are required. Additionally, there are situation in which sharing of sub-components may not be feasible and two versions of the tree are needed. For example in translating "`repeat` *body* `until` *cond*" to "*body* `; while (not` *cond*`) do` *body*" the *body* tree cannot be shared in both places; some attribute (*e.g.* a data-flow analysis) may need to have different values for each instance of *body*.

This paper examines this problem and poses solutions in the context of attribute grammars (AGs) with *forwarding* [26] — a technique that constructs translations and automatically copies contextual information to the translation. Computed semantic information on the translation is automatically copied back to the original construct, when it is not overridden (*e.g.* error messages) by an explicit definition on the "forwarding" construct.

The primary contribution of the paper is a new mechanism for sharing trees and their attribution under a new (forwarding) construct and its translation (forwarded-to) construct. In attribute grammars, a production defines synthesized attributes for the left-hand side nonterminal and inherited attributes for the right-hand side nonterminals. Forwarding provides default/implicit equations for synthesized attributes. We extend this so that for shared trees, forwarding can now do the "other half" of this work and now provide default-/implicit equations for inherited attributes for right-hand nonterminals too. Critically, this mechanism does not limit the extensibility of the language like the crude mechanism currently used in ABLEC. This is discussed in Section 3 after Section 2 continues the discussion of the shortcomings of the existing approach to forwarding. Section 3 also introduces *translation attributes*, a means for sharing context when translation trees are constructed over a number of productions in a higher-order attribute.

Section 4 validates the techniques by implementing them in the SILVER [25] attribute grammar system and applying them in the large ABLEC specification and several extensions to it, finding that around 80% of the uses of the crude non-extensible technique (and all exponential cases) could be replaced by the new extensible approach.[1]

Section 5 describes how the modular well-definedness analysis [12] can be extended to handle this new feature to ensure that there will be no missing or duplicate equations in an attribute grammar composed from independently-developed language extensions. It also discusses challenges in ensuring non-circularity with this approach.

We discuss limitations of this approach, and alternatives to it, in Section 6 before discussing related in Section 7 and future work and concluding in Section 8.

2 Background

In this section we provide background on attribute grammars, forwarding [26], its use in extensible languages, its limitations, and its realization in the SILVER AG system [25]. Background on the modular well-definedness analysis [9, 12] and its extension in this paper is discussed in Section 5.

2.1 Attribute grammars and SILVER

Attribute grammars are a declarative formalism for specifying the semantics of context-free languages [15, 16] and can be formally defined as a four-tuple $\langle G = \langle NT, T, P \rangle, A, O, E \rangle$ where G is a context free grammar with nonterminal symbols NT, terminal symbols T, and production rules P. A is a set of synthesized (A_S) and inherited (A_I, $A = A_S \cup A_I$)

[1] SILVER, ABLEC, extensions and other examples are available at https://melt.cs.umn.edu and archived at https://doi.org/10.13020/badh-qf44.

attributes that may decorate nodes of trees in the language of G, and O is a mapping of which attributes in A *occur on* which nonterminal symbols in NT. E is the set of equations on productions, that define the values of attributes on trees.

Since AGs process abstract syntax, as opposed to specifying concrete syntax, a grammar G supports a wider variety of types for tree nodes than simply nonterminal and terminal symbols. Thus, productions in P have a signature of the form $x_0 :: NT_0 ::= x_1 :: V_1 \dots x_n :: V_n$, with $n \geq 0$ in which V includes NT, T, primitive types such as integers, and others. These signature items are labeled with names so that nodes in a syntax tree can be referred to by these labels instead of their position in the production.

Attribute grammars have been extended in a wide variety of ways since their introduction to better support the specification of tasks common in language implementation. For example, higher-order attributes [29] hold tree values that are passed to new locations in the syntax tree where they are then provided with inherited attribute (that is, decorated) so that synthesized attributes can be computed on them. Another commonly used extension is reference [7] or remote [1] attributes. These can be seen as pointers, or references, to the root node of remote decorated trees somewhere in the syntax tree from which attributes can be accessed.

In Silver decorated trees in reference attributes are known to have been provided with a set of inherited attributes called a *reference set*. When decorated tree types are written as `Decorated` *NT* this set is, by default, the inherited attributes occurring on *NT* that were declared in the same grammar module as *NT*. The reference set can be given explicitly to override this default; *e.g.* `Decorated Expr with {env}` identifies the environment attribute `env` as sole attribute in the reference set for this type of decorated expression.

Figure 5 show an implementation of the for-loop construct as seen in Figure 1 in the Silver AG system. Declarations of attributes, and their occurrences, are not shown but can be inferred from the `forLoop` production. This production computes a synthesized `errors` attribute on the left-hand side `s`, defines the inherited environment attribute `env` on the three child trees, computes a local fresh variable `upperVar`. This is used in the construction of the while-loop translation, seen in Figure 1, that the for-loop will forward to. The productions `decl` for declaring and initializing variables, `seq` for statement sequencing, etc. should be clear from the example in Figure 1. The `forLoop` production takes (undecorated) terms of type `Expr` and `Stmt` in constructing the syntax tree. In the body of the production (in the curly-braces) these labeled terms are decorated by inherited attribute equations, *e.g.* lines 7–9, and thus the labels refer to trees of type `Decorated Expr` and `Decorated Stmt`. Similarly, local tree-valued *production attributes* can also be declared and defined as an undecorated nonterminal type (*e.g.* `Expr`) and decorated using inherited attribute equations in a production body to be typed as decorated (*e.g.* `Decorated Expr`).

```
1  production forLoop  s::Stmt ::=
2   iVar::String lower::Expr upper::Expr body::Stmt
3  { s.errors =
4      checkInt(lower.type, "loop lower bound") ++
5      checkInt(upper.type, "loop upper bound") ++
6      lower.errors ++ upper.errors ++ body.errors;
7    lower.env = s.env;
8    upper.env = s.env;
9    body.env  = addEnv(iVar, intType(), s.env);
10   local upperVar::String = freshName(s.env);
11   forwards to block(seq(
12     decl(iVar, intType(), new(lower)),
13     seq(decl(upperVar, intType(), new(upper)),
14       while(intLt(var(iVar), var(upperVar)),
15         seq(new(body), assign(iVar,
16           intAdd(var(iVar), intConst(1))))))));
17 }
```

Figure 5. A `forLoop` implementation. The children are explicitly decorated with the environment to support error checking, and are decorated again in the forwarded to tree.

2.2 Forwarding and extensible languages

Forwarding is a technique developed to support the modular definition of languages [26] and has been used extensively in the ableC extensible C compiler and a wide variety of composable extensions to it [10, 11]. Any queries for synthesized attributes on the production's left-hand side nonterminal that are not explicitly defined by equations are "forwarded" to the `forwards to` tree to be answered there. Likewise, any queries for inherited attributes on the forwarded-to tree are passed back to production's left hand side and their values are retrieved from there. In Figure 5, `s` defines a value for the `errors` attribute using the function `checkInt` that reports a message when a type is not an integer. This takes precedence over the value for `errors` on the while-loop construct, thus providing proper error messages that reference the code written by the programmer, not the code to which it translates. A query to `s` for an assembly language translation `asm` attribute would automatically and implicitly copy the value from of `asm` from the forwards-to tree back to `s`. Any queries of an inherited `env` attribute on the outermost `block` construct in the forwards-to tree would get is value from the `env` attribute passed down to the left-hand side symbol `s`.

Forwarding supports the automatic composition of independent extension specifications. If another extension defines, *e.g.* a new translation to Web Assembly in a `wasm` attribute, then all computations involved in that effort take place on the forwards to tree and automatically provide a value for `wasm` for `s`, even though the author of the `forLoop` extension knew nothing of this Web Assembly extension.

```
production neg   e::Expr ::= n::Expr
{ n.env = e.env;
  forwards to case n.type of
    | intType()  -> intNeg  ( new(n) )
    | boolType() -> boolNeg ( new(n) )
    | _ -> errorExpr ("incorrect types")
    end;  }
```

```
production decExpr
e::Expr ::= de::Decorated Expr with {env}
{ e.type = de.type;
  e.errors = de.errors;  }

production neg  e::Expr ::= n::Expr
{ n.env = e.env;
  forwards to case n.type of
    | intType()  -> intNeg  ( decExpr(n) )
    | boolType() -> boolNeg ( decExpr(n) )
    | _ -> errorExpr ("incorrect types")
    end;  }
```

Figure 6. The exponential neg production (top) and the efficient but crude "decExpr" hack (bottom).

2.3 Limitations of forwarding

While forwarding provides implicit attribute definitions for synthesized attributes for a production's left-hand side symbol, it provides no support for the "other half" of what equations associated with a production do: provide values of inherited attributes to the child trees. This is shown in Figure 5 where equations for env are required for child trees since their synthesized attribute errors (whose computation depends on an environment) is demanded on line 6. An important consideration when overriding errors on forLoop is that any problems in the forward tree should still be reflected in the new errors equation. To ensure this, the environment given to lower, upper and body must match the environment computed through the equations of the productions enclosing these children in the forward tree; *e.g.* body must receive an env containing iVar bound to the appropriate type. This requires extension developers to familiarize themselves with host language details in order to write the appropriate equations. Doing so may be especially burdensome in a more sophisticated host language where the inherited dependencies of errors could be more than just env.

Performance is another concern; see the duplication of for-loop (Figure 2) and negation trees (Figure 3). In the specification of the forwards-to tree for a for-loop in Figure 5 the child trees lower, upper, and body must be replicated (using new) This "undecorates" these trees, retrieving the original terms prior to decoration, and re-decorates them with new attribute values under the forwards-to tree. For the for-loop in Figure 2 this is not so expensive, but it is a significant problem with the negation operator in Figure 3. The specification of the overloaded negation operator is given at the top of Figure 6. It queries its child term's type (n.type) to determine which type-specific negation production to forward to. Since there is no explicit equation for e.type that value is determined on (and copied from) the forward tree, resulting in the duplication in Figure 3.

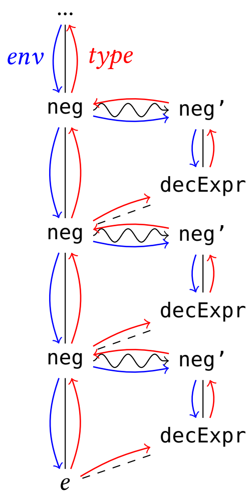


Figure 7. The efficient but non-extensible implementation of overloaded negation from the bottom of Figure 6.

The specifications at the bottom of Figure 6 demonstrate the crude "decExpr" hack that avoids the exponential duplication of trees. The decExpr production wraps up a Decorated expression tree that has been already provided with its env attribute (written with {env}), thus allowing the evaluation of the type and errors attribute on de. This production is used to wrap up the decorated child n in the optimized neg production instead of using new. This results in the tree shown in Figure 7 in which each negation child is shared between the original and forwarded-to trees.

While this is efficient, it severely limits the extensibility of the language. Recall the extension adding equations for a synthesized wasm attribute to host language productions. It would define wasm on intNeg and boolNeg but any new inherited attributes needed for this computation will not be propagated down past the decExpr node. The needed inherited attribute equations cannot be added to the host-language decExpr production since they would have no effect — its child is a reference to a tree that was decorated elsewhere, and inherited attribute equations are not permitted here. This technique is used frequently in the ABLEC specification and, while efficient, limits the kinds of language features that can be developed as composable language extensions.

3 Forwarding with Tree Sharing

Here we describe a new "tree-sharing" operator @ that allows trees, and the specification of inherited attributes, to be shared between a forwarding and a forwarded-to tree, thus avoiding the duplication and redecoration seen in Figure 2 and Figure 3 and instead producing shared trees like those in Figure 4. We also describe different scenarios in which this can be used.

```
1  production forLoop  s::Stmt ::=
2   iVar::String lower::Expr upper::Expr body::Stmt
3  { s.errors =
4      checkInt(lower.type, "lower bound") ++
5      checkInt(upper.type, "upper bound") ++
6      lower.errors ++ upper.errors ++ body.errors;
7    local upperVar::String = freshName(s.env);
8    forwards to block(seq(
9      decl(iVar, intType(), @lower),
10     seq(decl(upperVar, intType(), @upper),
11       while(intLt(var(iVar), var(upperVar)),
12         seq(@body, assign(iVar,
13           intAdd(var(iVar), intConst(1)))))))); }
```

Figure 8. An alternative version of Figure 5 in which children are shared with the forward tree, avoiding the need to specify inherited env equations.

3.1 Sharing with a static forward tree

In order to achieve the pattern of sharing seen in Figure 4, we introduce the *tree-sharing* operator @, which takes a decorated tree and wraps it as an undecorated term. Decorating this term simply yields the original tree, updated with any newly-supplied attributes added; thus the syntax @a can be read as "a gets decorated with more attributes here." This operator can be seen as a improved, built-in version of decExpr and similar "wrapper productions", except that there is no intermediate node for @ in the decorated tree, like the decExpr one seen in Figure 7. The tree-sharing operator can be used in specifying the forward for the forLoop production, as seen in Figure 8. With the nested forLoop scenario, this gives rise to the tree in Figure 4 instead of Figure 2; now the innermost statement *s* is only decorated once.

When a shared child or local appears beneath statically specified productions in the forward tree, inherited attributes supplied to the child/local by these productions can be utilized in the original forwarding production. This effectively permits child inherited attributes to be implicitly computed through forwarding. For example on lines 4–6 of Figure 8, type and errors can be accessed on the children without supplying explicit equations for env since, as seen on the left in Figure 4, env is defined for those trees by the forward tree.

Silver uses a demand-driven approach to attribute evaluation [8], which now presents some complications. Traditionally, one knows what equations will be used to compute inherited attributes on a tree before any attributes are evaluated. This is now no longer the case, as inherited attributes supplied to a child shared in a forward tree are only defined when the portion of the forward tree containing the child is demanded. For example in Figure 8, the equation for s.errors does not directly depend on forwarding; demanding lower.errors seemingly would not cause the forward tree to be created and decorated, which would lead to a missing equation for env on lower.

```
production neg   e::Expr ::= n::Expr
{ n.env = e.env;
  forwards to case n.type of
    | intType()  -> intNeg  ( @n )
    | boolType() -> boolNeg ( @n )
    | _ -> errorExpr ("incorrect types") end;  }
```

Figure 9. An implementation of operator overloading, in which the forward tree is determined based on the operand types. Computing these types requires supplying the environment to the operands.

To avoid this, any use of the original child tree must demand the corresponding portion of the forward tree. This is conceptually like pattern matching on the forward tree, with a pattern that mirrors the term containing the child. For example, the access of lower.errors could be translated as

```
case forward of
| block(seq(decl(_, _, lower))) -> lower.errors
end
```

In reality, this can be implemented more efficiently than pattern matching, as the forward tree is known to have been built with these constructors and we do not need to check that it has the expected shape. Note that this problem would not exist if a similar approach of decoration through a shared tree was used in an ordered attribute grammar [14]; an attribute supplied through sharing (like env) could be fully computed before being used on the shared tree by another attribute (like errors.)

In prior versions of Silver, children and locals could be referenced with an undecorated type, implicitly un-decorating trees in these cases. For example, the calls to new on lines 4–6 of Figure 5 could have been omitted. We now recognize this to be a language design flaw, as children can be inadvertently un-decorated, and included in the forward tree to be decorated again, without any indication of a potential problem. To address this, we simplify the type semantics of Silver so that any reference to a child or local tree gives a decorated type. This requires one to explicitly write new or @ when incorporating a sub-tree that has previously been decorated into a new term.

3.2 Dynamic forwarding

Sometimes, a child may appear in the forward tree under different productions, depending on the result of some analysis. This is the case for the negation operator from Figure 6, where n.type is computed to determine the target

```
production neg   e::Expr ::= n::Expr
{ local nVar::String = freshName(e.env);
  local impl::Expr = case n.type of
    | intType()  -> intNeg  ( var(nVar) )
    | boolType() -> boolNeg ( var(nVar) )
    | _ -> errorExpr ("incorrect types")   end;
  forwards to let_(nVar, @n, @impl);  }
```

Figure 10. An alternate version of Figure 9, in which the portion of the forward tree containing the child `n` is static.

production. Computing `type` on an expression depends on `env`, which must be supplied to `n`; to avoid a circularity, this must be done with an explicit equation rather than through forwarding.

However, one would still wish to share `n` in the forward tree, to avoid the exponential explosion seen in Figure 3; this can be done using the tree sharing operator as seen in Figure 9. For this to be possible, the writer of a forwarding production such as `neg` must ensure that the values of any explicit inherited equations on a shared child match the values that would otherwise be supplied through forwarding.

3.3 Partially dynamic forwarding

Often, an intermediate approach between static and dynamic forwarding is possible: children can be shared (and receive attributes) beneath a static portion of the forward tree, while other portions of the forward tree are computed dynamically. For example in Figure 10, the operand to `neg` can be bound to a fresh temporary variable in a let-expression. The implementation, dynamically determined from the type of `n`, can then refer to the variable instead of using `n` directly. Lazy evaluation means that the portion of the forward tree containing `n` can be decorated with `env` to compute `type`, before `impl` is computed. Note that `impl` is also marked as being shared, since it is a local that gets decorated implicitly.

Note that without care this approach can give rise to circularities, between computing an analysis on a child that is needed to determine part of the forward, and decorating the forward tree to determine inherited attributes on the child. To avoid this one must sometimes supply some inherited equations explicitly, which take precedence over equations supplied in the forward tree.

For example, the ABLEC-CLOSURE extension [10] introduces lambda functions, *e.g.* `lambda (int x) -> x + y`, where free variables such as `y` referenced in the body can be captured. To implement this, a lambda function is implemented as a function pointer, paired with a struct containing the values of captured variables. Thus, the above lambda expression would forward to a function pointer to the following function that is lifted to the global scope along with the following `struct` declaration:

```
var res : bool = table { b1 && b3 : T F
                         ~ b2     : T *
                         b2 || b3 : F T };
```

```
var res : bool =
  let _v0 : bool = b1 && b3 in
  let _v1 : bool = ~b2 in
  let _v2 : bool = b2 || b3 in
    (_v0 && _v1 && ~_v2) || (~_v0 && _v2);
```

Figure 11. A simple language extension for condition tables (top), provides an alternate concise notation for complex boolean expressions (bottom)

```
1  struct _lam_env_19 { int y; };
2  int _lam_fn_19(struct _lam_env_19 _env, int x) {
3    const int y = _env.y;
4    return x + y;  }
```

The free variables from the body to be captured are computed as a synthesized attribute `freeVariables`, which on expressions depends on `env`. However, the `env` given to the body in the forward of the lambda production depends on the variable definitions (*e.g.* line 3 in the above) generated from the free variables. This circularity can be avoided by the lambda production supplying an explicit `env` equation to its body expression. For correctness, this equation must match the `env` supplied through forwarding, in this case by making all captured variables constant.

3.4 Computing a forward over multiple productions

Sometimes, an extension may introduce its own nonterminals to provide richer syntax, and the computation of the translation it will forward to is spread across the productions for these new nonterminal symbols. For example, the condition tables extension, seen at the top of Figure 11, provides convenient syntax for writing complex Boolean expressions. A condition table expression is true if there is a column where the expression is true for every row with a `T`, and is false for every row with an `F`, while * indicates that we don't care if that expression is true or false. This expression translates to the code seen in the bottom of Figure 11, creating a let-binding for every expression, with the conditions translated into conjunctive normal form as the body.

A portion of the implementation of this extension is seen in Figure 12. Table rows are represented by the `TRows` nonterminal (line 7), with an inherited attribute `conds` to construct the needed Boolean result expression. We require in the syntax of the extension that the table has at least one row, such that `rs.conds` is non-empty in `nilRow`. On line 16, the `trans` attribute then wraps the result in the needed let bindings for the row expressions.

```
 1  production condTable  e::Expr ::= rows::TRows
 2  { e.errors = rows.errors;
 3    rows.conds = [];
 4    forwards to @rows.trans;  }
 5  inherited attribute conds::[Expr];
 6  translation attribute trans::Expr;
 7  nonterminal TRows with errors, conds, trans;
 8  production consRow  rs::TRows ::=
 9    e::Expr tf::TruthFlags rest::TRows
10  { rs.errors = e.errors ++ tf.errors ++ rest.errors
11      ++ checkBoolean(e.type, "row expression");
12    local eVar::String = freshName(rs.trans.env);
13    tf.rowExpr = var(eVar);
14    rest.conds = if null(rs.conds) then tf.rowConds
15      else zipWith(andOp, rs.conds, tf.rowConds);
16    rs.trans = let_(eVar, @e, @rest.trans);
17  }
18  production nilRow  rs::TRows ::=
19  { rs.errors = [];
20    rs.trans = foldr1(orOp, rs.conds);  }
```

Figure 12. A portion of the implementation of condition tables, computing a forward tree containing children decorated across multiple productions using a translation attribute.

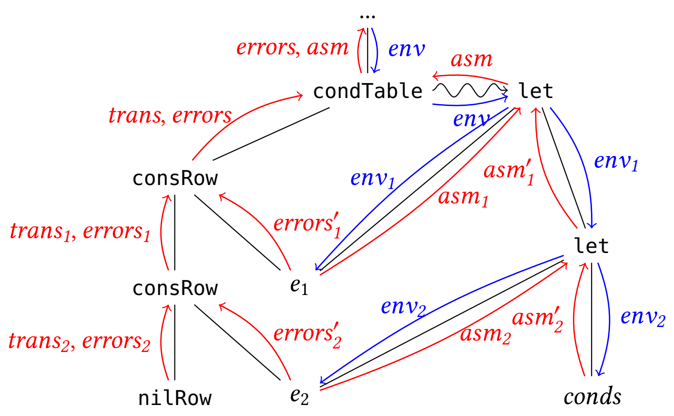


Figure 13. A tree corresponding to the condition tables extension in Figure 12. `trans` and `errors` is computed on the `TRows` nodes on the left, while `env` and `asm` are computed on the decorated form of `trans` on the right.

As seen previously with `neg` in Figure 10, we would like to decorate these host-language `Expr`s with inherited attributes supplied through their translation. However, `consRow` is not a forwarding production since its left-hand side is not a host language nonterminal. If `trans` were an ordinary higher-order attribute, we would construct an undecorated translation term and decorate it in the forward of `condTable`. This would then decorate the condition expressions in the table rows again, thus returning to the situation in Figure 2.

Instead, we would like to create the pattern seen in Figure 13. Here we first construct the translation of the table rows into `let`-expressions, and decorate it as the forward of `condTable`. The environment flows down the forward tree to the condition expressions, where it is used to compute `errors`, which are collected up the original `TRows` tree.

This can be achieved using *translation attributes*, which are synthesized attributes that serve as decoration sites for terms, similar to locals. Translation attributes are similar to higher-order attributes in that their equations create undecorated terms; the declared type of a translation attribute must be a nonterminal. However like reference or remote attributes, they hold decorated trees and may be supplied with inherited equations. On line 4 of Figure 12, accessing `rows.trans` gives back a `Decorated Expr`. On line 10, `consRow` can access `e.errors`, which depends on `e.env`, using the `env` supplied through the equation for `rs.trans`.

Synthesized and inherited attributes occurring on `Expr` may be treated like additional synthesized and inherited attributes occurring on `TRows`; for example, the `consRow` production uses `rs.trans.env` to compute `eVar`, which must have been supplied to `rs.trans` by the parent of this production. We can again use the `@` tree-sharing operator to indicate that the tree constructed by a translation attribute should be shared with (and receive inherited attributes from) another decorated tree. In `condTable`, the `env` supplied to `rows.trans` (through the use of `@` and forwarding here; once could also write `rows.trans.env = e.env;`) flows down through the `Expr` tree built by `rows.trans`. Thus in `consRow` one can access `e.errors`, which depends on `e.env`, using the `env` supplied through the equation for `rs.trans` on line 16.

Demand-driven evaluation creates complications for translation attributes, like was seen with sharing children in Section 3.1. Any use of a tree shared in a translation attribute equation must demand the decoration of the tree constructed by the attribute, which may involve recursively demanding the decoration of the translation attribute from further up the extension tree. For example, accessing `errors` from e_2 in Figure 13 must ultimately demand the forward tree from its root decoration site in the forward of `condTable`. To achieve this, the implementation involves another implicit inherited reference attribute to pass corresponding portions of the translation tree down from its root decoration site.

4 Evaluation

We evaluated the utility of this new approach to forwarding in the ableC [11] host language specification of C. There are many different extensions to ableC [10, 11, 17] and we evaluated 10 non-trivial extensions that are representative of the other ableC language extensions. The full list of extensions and detailed results of the evaluation can be found in Appendix A. Overall, 94% of uses of decorated wrapper productions like `decExpr` in Figure 7 could be replaced by the tree sharing operator, and those that could not were not needed to avoid exponential decoration. These changes allow the removal of many complex and now-unneeded inherited attribute equations, making the extension specifications significantly easier to understand and maintain.

All unary and binary operators in ableC, as well as some expressions like function call and array index, support operator overloading. Each operator has a production that forward based on the types of the children; these explicitly supply inherited attributes and wrap the decorated children in `decExpr`, similar to `neg` in Figure 9. In total, `decExpr` is used 168 times in the ableC specification; all of these uses could be replaced with uses of the tree sharing operator.

Across the 10 considered ableC extensions, there are 74 instance of decorated wrapper productions. Of these, 39 appear in a statically-determined forward tree, where the tree-sharing operator can be used to avoid some inherited equations, and 5 appeared in a dynamically-forwarding production similar to overloading, where inherited equations are still needed. In 30 cases, wrapper productions were used in a translation passed as an attribute to be decorated elsewhere. Of these, 15 could be easily replaced using translation attributes.

The ableC-Prolog extension uses a combination of multiple higher-order synthesized and inherited attributes for constructing a translation, where some host language children may appear in the trees constructed by more than one attribute. This pattern is not amenable to translation attributes; thus there are are 15 uses of decorated wrapper productions in the Prolog extension which could not easily be replaced. However, the constructs introduced in this extension are not typically nested, and thus would not suffer from exponential recomputation if these subtrees were decorated twice.

In some ableC extensions, *e.g.* condition tables, we found instances of tree re-decoration that had not been addressed with wrapper productions. We have not yet attempted to identify all instances of trees being re-decorated that could potentially be addressed with tree sharing. This is because these specifications were developed with a prior version of Silver, with the problematic implicit undecoration semantics that we mentioned in Section 3.1. Changing the type rules to require explicit undecoration or sharing creates an error flagging every potential place where a tree could be re-decorated; however this constitutes a major breaking change to all existing Silver code, and we had not yet completed this change as of performing this evaluation.

For this reason we have not attempted a comprehensive refactoring of ableC and its extensions to use the new sharing mechanisms, or quantified the number of unaddressed re-decoration issues. Instead, we refactored 3 ableC extensions to use the new sharing mechanisms: closures, condition-tables, and algebraic datatypes. In all cases this led to simpler code and the removal of inherited attributes and equations. In the datatypes extension, the use of translation attributes to translate pattern matching saved 36 lines of specification (out of a total of 900.)

The tree-sharing operator provides a small optimization over the old approach of wrapper productions, as the new approach does not require introducing an extra node in the decorated tree, as seen in Figure 7 vs Figure 4. The performance was evaluated by comparing the runtime of building the examples for the closure and datatype extensions before and after refactoring; the refactored versions yielded a roughly 4% speedup. We did not attempt to compare the performance of specifications with exponential re-decoration behavior to ones in which sharing is used, as prior to refactoring these were clearly unusable for nontrivial programs.

Overall, tree sharing has proven to be a very beneficial technique, with significant improvements in code quality, and avoiding excessive recomputation of analyses without limiting extensibility.

5 Modular Well-Definedness

Kaminski and Van Wyk [12] proposed a modular well-definedness analysis for attribute grammars with forwarding, to ensure that all potentially needed attribute equations are present in compositions of independently-designed language extensions. This analysis is based around the idea of constructing flow graphs for productions, as in Knuth's original analyses [15]. *Flow types* are inferred for every occurrence of a synthesized attribute on a nonterminal, consisting of the set of inherited attribute dependencies. Attribute equations are checked against the production flow graphs and flow types, ensuring that all needed equations are supplied.

The data structures and algorithms used in Kaminski and Van Wyk [12]'s analysis, used in Silver, must be slightly extended to accommodate the new features proposed here. While space limits preclude a full presentation of the extended analyses here, we present the issues at hand and give some intuition about the required changes below.

5.1 Avoiding duplicate equations

Since we are adding the ability to supply inherited equations to the same tree from multiple sites, we must avoid there being multiple equations for the same attribute when some precedence between them cannot be determined. First, the operand to the `@` tree-sharing operator must correspond to a *decoration site* where inherited attribute equations could ordinarily be specified, *i.e.* a nonterminal child, local, or instance of a translation attribute on a child or local.

If an explicit inherited equation is given to a child in a forwarding production, in addition to an equation through a production in the forward tree, then the explicit equation takes precedence. This is analogous to the explicit synthesized equations on the forwarding production taking precedence over those in the tree forwarded-to. We must also ensure that if a child is shared, it only appears in one place, to avoid multiple competing equations. This means that a shared child can only appear in the original production, and not in independently-introduced aspect productions. Similarly, a local tree can only be shared once and in the same module as the local, and a translation attribute instance can only be shared once per production in the same module as the attribute occurrence. It is permitted for the same tree to be shared in multiple mutually exclusive positions, such as separate branches of a pattern match as seen in Figure 9.

The expression in which a shared child appears also must not decorate the term containing the child more than once, for example a shared tree cannot appear in an arbitrary higher-order attribute equation, which may be arbitrarily used and re-decorated elsewhere. The tree-sharing operator must only appear in unique contexts, which can be a forward equation, local equation, translation attribute equation, or as an operand to a production call or conditional expression in a unique context. This is enforced by a straightforward syntactic analysis, termed the *uniqueness analysis*.

A more subtle issue exists if a production in the forward equation were to un-decorate and re-decorate its own child, perhaps intentionally to perform an analysis in a different environment. The behavior of calling `new` on a tree in Silver is to simply return the original term that was decorated to create the tree. If the term provided to this production contains a wrapped child from use of the tree-sharing operator, this may result in the child being decorated twice.

Instead, we change the semantics of un-decoration in Silver to perform a deep-copy of the term that was decorated, such that calling `new` on a decorated tree never returns a term containing a wrapped tree. Thus in the above scenario, the un-decorated and re-decorated tree does not share any subtrees. As an optimization, all constructed terms track whether they contain a wrapped tree, such that the deep copy operation only needs to happen down to the level of any wrapped subtrees.

5.2 Enforcing effective inherited completeness

Recall the intuition given in Section 3.2 for the runtime semantics of tree sharing in terms of pattern matching. The treatment of the tree-sharing operator in the modular well-definedness analysis is also similar to pattern matching, as discussed fully in previous work [9]. A new sort of flow vertex is introduced corresponding to unconditionally-decorated sub-terms of a forward or local equation. The existing analysis has a notion of *flow projection stitch points*, used to update a production's flow graph with edges corresponding to inherited equations from productions referenced in patterns. Flow projection stitch points are also used here, to add edges for inherited equations between sub-term vertices.

A synthesized attribute may now depend on an inherited attribute being supplied to a translation attribute on a tree. Thus, flow types are extended to include inherited attributes occurring on translation attributes on a nonterminal, in addition to inherited attributes occurring on a nonterminal.

For inherited attribute equations to be reliably supplied to a child through forwarding, the requirements for the term in which the child appears are somewhat stricter than imposed by the uniqueness analysis. The child must be decorated *unconditionally*, meaning that it cannot appear under any conditional expressions. For example as seen in Figure 9, `n` can be shared in the forward tree, but we cannot rely on any inherited equations supplied through forwarding.

The checks performed on equations using the inferred flow types and production flow graphs are essentially unchanged, except that we do not check for the presence of inherited equations within a production for a shared, unconditionally decorated child or local. There is an additional check required for inherited override equations on children or locals that are shared, even conditionally: the dependencies of the override equation must not exceed the dependencies of the remote equation. This is because the production constructed with the shared child will be checked with the flow graph created from the inherited equations that it supplies; if we override this equation with one that has additional dependencies, this may lead to additional transitive dependencies that were not checked, and potentially missing equations.

5.3 Limited feasibility of circularity analyses

Kaminski and Van Wyk [12]'s modular well-definedness analysis does not include a non-circularity check. This is because in practice, strict non-circularity between attributes is overly conservative; laziness means that cycles often are not present in the actual evaluation even when a strict analysis could not prove their absence. Furthermore circularity between portions of trees is often useful, such as in building the environment for mutually recursive bindings. For these reasons, attribute grammar systems such as Silver, JastAdd [4] and Kiama [21] dispense with a non-circularity analysis.

Circularities between different portions of trees have proven especially useful with tree sharing, as seen in Figure 10. We have found that this pattern does frequently give rise to actual cycles, between an inherited attribute on a shared child and a dynamic portion of the forward tree that may affect the child's attribute. Since we don't have an analysis capable of detecting these problems, these issues are typically found through crashes when developing an extension; however we have found them to be easy to diagnose and resolve by adding explicit inherited equations. These problems also typically do not appear from composing independent extensions, because the problematic dependencies involve productions explicitly specified in the forward tree. While a more sophisticated analysis could be useful for finding these issues, it is unclear if such an analysis is feasible.

6 Discussion

This section provides a discussion of tree sharing and the computation of inherited attributes through forwarding, and how this work relates to other aspects of Silver.

6.1 Specializing inherited attribute equations

A nice feature of forwarding is the freedom to specialize synthesized attributes to the new language construct by writing explicit equations for them. This allows one to report error messages specific to the new feature or define the type attribute on a new expression, *e.g.* of a extension introducing list literals may define its type to be a list type. In Section 3.3 we saw an example of needing to write explicit inherited equations to break circularities with the closure extension. These equations always had the same value as what would later be computable in the forward tree. Are cases when we want to specialize an inherited attribute with a different value, as seen with errors? We expected the new ability to specialize inherited attributes to be similarly helpful, but their actual benefit was something else. Contextual information, *e.g.* an environment mapping names to types, is some data structure containing terms for host language nonterminals, such as a `Type` nonterminal defining structured types. Thus new extension types, *e.g.* list types, naturally arise in the name bindings. It turns out there is not much need to specialize an environment, or similar inherited attributes, on the forwarding production. More frequently, we find one writes these explicit equations to break cycles in extensions with sophisticated patterns of forwarding. Since these equations are not so often intended to specialize inherited attribute values, it is also easier to ensure that the values given are compatible with those provided on the forwarded-to tree.

6.2 Why autocopy is a misfeature

Past versions of Silver and earlier presentations of forwarding [25, 26] featured *autocopy attributes*, a form of inherited attributes that were implicitly copied down the tree to children. This is convenient for attributes such as an environment that generally flow down the tree. However autocopy attributes are incompatible with the new approach to tree sharing, as we often want an environment to be supplied through forwarding, and autocopy would supply an undesired implicit copy equation for the child.

In fact, we had already recognized autocopy as a source of bugs due to undesired equations: in developing attribute grammar specifications, we sometimes use a "flow-type-driven development" approach, adding needed equations as they are flagged by the well-definedness analysis. Autocopy attributes suppress these errors by introducing implicit (and often incorrect) equations. For these reasons we have removed support for autocopy, and replaced it with a mechanism to specify where copy equations should be generated for an inherited attribute.

```
 1  production forLoop  s::Stmt ::=
 2   iVar::String lower::Expr upper::Expr body::Stmt
 3  { local localErrors::[Message] =
 4      checkInt(lower.type, "lower bound") ++
 5      checkInt(upper.type, "upper bound") ++
 6      lower.errors ++ upper.errors ++ body.errors;
 7    local upperVar::String = freshName(s.env);
 8    forward fwrd = block(seq(
 9      decl(iVar, intType(), @lower),
10      seq(decl(upperVar, intType(), @upper),
11        while(intLt(var(iVar), var(upperVar)),
12          seq(@body, assign(iVar,
13            intAdd(var(iVar), intConst(1))))))));
14    forwards to if null(localErrors) then @fwrd
15      else errorStmt(localErrors);     }
```

Figure 14. An alternative version of Figure 8 using an error production instead of overriding the equation for `errors`. A forward production attribute is used to unconditionally decorate the translation when forwarding conditionally.

6.3 Forward production attributes

Overriding attributes on forwarding productions with values differing from those on the forward tree can lead to unexpected behavior when composing independent extensions, a problem known as *interference* [13]. To avoid this, *error productions* are included in host language specifications such as ableC, which can optionally be forwarded to instead of writing an override equation. However, this pattern is incompatible with computing inherited attributes through forwarding, as the forward tree may not always be decorated.

An alternative is to use a local *forward production attribute*, as seen on line 8 of Figure 14. This allows one to specify one forward tree for unconditionally supplying context, but potentially forward synthesized attributes to a different tree. We identified 49 places in the 10 ableC extensions evaluated in Section 4 where this feature would be useful.

7 Related Work

7.1 Attribute grammars

We added tree and contextual-information sharing to the Silver [25] attribute grammar system because the notion of forwarding makes the problem of sharing an interesting one. But there are other well-used AG systems that could have been used. Kiama [21] is a Scala library that also has a notion of forwarding. Similarly, JastAdd [3] has a notion of tree-rewriting [23] that is integrated into its use of reference [19] and circular [6] attributes. This may also be an interesting candidate for tree sharing to save re-computation of attributes. Silver has strategy attributes [18] that allow one to write Stratego-style strategies to control the application or rewrite-rules [28]. The tree-sharing discussed here may be applicable in single-pass, bottom-up traversals since these are similar to what happens with translation attributes (Section 3.4). But it is not clear how well sharing can be incorporated into the more sophisticated strategies, and their ensuring traversal patterns, since it is unclear how one maintains the uniqueness requirements of the tree-sharing operator. This is certainly an area worth further study.

7.2 Tree sharing

The sharing of trees is a common practice in programming language tools. One influential example is the ATerm (Annotated Terms) system [24] for automatically sharing the representation of trees; it provides maximal sharing. Tree construction specifications will reuse existing trees if they already exist in the current collection of syntax trees. Kiama also provides an interesting notion of tree sharing in which the same tree can be decorated with two different values for the same set of attributes, an approach termed "respecting your parents" [22]. Here, the attribute values are stored separately from the tree in (unshared) attributions; they consist of a map from unique identifiers of tree nodes to attribute values. This is similar to Silver's distinguishing terms and decorated trees. In these works the aim of sharing is to represent trees more efficiently, and not to reuse or simplify the specification of computations.

Intentional Programming [20] is the most closely related work to our since forwarding was originally implemented in that system. It showed how extensions could specialize synthesized attributes, there called *questions*. But it automatically shared sub-trees under the forwarding and forwarded-to trees [27] and did not allow the specialization of inherited attributes, thus denying the language engineer the freedom to make these choices.

7.3 Attribute grammar flow analysis

Silver focuses on independent extensions so that a programmer can pick the ones they desire for their task at hand. Thus the modular well-definedness analysis [12] is used to ensure that the composition of these extensions will, in fact, work since the programmer is not in a position to debug or modify extension specifications; thus the extensions to this work in Section 5. Most related to our extensions is Boyland's analyses on remote attribute grammars [1]. That work analyses remote attributes and uses a notion of *fibers* to track dependencies not only on a remote node in the syntax tree but also the attributes that decorate it. This is similar to our extension of flow-types for synthesized attributes to also include the inherited attributes on a translation attribute on which it depends. Boyland's analysis was not a modular one and thus not directly applicable in our setting.

8 Future Work and Conclusion

8.1 Utilizing context supplied before forwarding

There are still some shortcomings with the approach to operator overloading in ableC proposed in Section 4. Every overloaded operator has both a forwarding production, and a non-forwarding default implementation production that is typically only ever constructed by its overloaded counterpart. Both productions must specify all the inherited equations needed for type checking, however with sharing, the equations on the non-forwarding production are never used.

To avoid specifying these equations multiple times, a solution is to identify productions like `intNeg` or `boolNeg` as *dispatch implementations* that can only be constructed as the forwarded-to tree of some specific *dispatching* production(s) like `neg`. The flow analysis from Section 5.2 can then be extended to consider any equations in the forwarding production as being supplied to the corresponding children in the implementation.

8.2 Data nonterminals

Sometimes nonterminals are used to represent data structures, such as optional values of type `Maybe` or an environment, that are never decorated with inherited attributes. Always automatically decorating children and locals of these types is inefficient and requires extra calls to `new`; instead we would like to mark them as *data nonterminals* that are never `Decorated`.

8.3 In conclusion

This paper introduces a new operator `@`, that permits tree-sharing without limiting extensibility. This allows language engineers to control when trees, and the specification of their contextual information, are to be shared or not. The examples given in Section 3 and the results of the evaluation in ableC indicate that tree sharing is the more common choice. But there are cases, especially when extensions are unlikely to be nested, when duplicating the tree is the right choice. This occurs in the ableC-Prolog extension discussed above, where host-language expressions appear in relations (*e.g.* in a numeric comparison goal), as well as in their translation. Since the generated code is complex, knowing *e.g.* what contextual information for a live-variable analysis is to be provided to expressions in a forwarding Prolog construct would be very difficult indeed. Here the right choice is to not share the child trees. Thus, it is important that language engineers have the freedom to make the appropriate choice, to share or not, and the new tree-sharing operator `@` provides that flexibility.

Acknowledgments

This work is partially supported by the National Science Foundation (NSF) under Grant Nos. 2123987.

Table 1. Information about the ABLEC extensions evaluated for the potential use of tree sharing.

Extension name	Description	Static	Dynamic	Translation attributes	Non-sharable	Forward prod attributes
CONDITION-TABLES	Concise syntax for boolean expressions	7				1
CLOSURE	Lambda functions that capture scoped variables		2			5
ALGEBRAIC-DATA-TYPES	Algebraic data types implemented as tagged unions, with pattern matching	3		6		3
TEMPLATING	C++-inspired templated function and type declarations		2			3
STRING	More efficient string representation type, overloadable operators for stringifying and pretty-printing various types	13				9
VECTOR	Array-backed list data structure, with overloads for +, ==, and other operators					14
CONSTRUCTOR	Syntax for constructing and deconstructing values, overloadable by other extensions such as VECTOR	1	1			5
UNIFICATION	Unification variable reference type and overloaded unify operator	5		3		3
PROLOG	PROLOG-inspired logic programming relations and queries	4		6	15	3
REWRITING	STRATEGO-inspired strategic term rewriting	7				4
Total	74 potential instances of sharing	39	5	15	15	49

A Evaluation of ABLEC extensions

Table 1 provides further details on the evaluation of opportunities for tree sharing in 10 representative ABLEC extensions. For each extension, from left to right is given

- The number of instances where tree sharing could be used in a statically-determined forward tree, supplying context and avoiding some explicit inherited equations;
- The number of instances where tree sharing could be used in a dynamic forward tree, where explicit inherited equations are still needed;
- The number of instances where tree sharing could be used in a translation attribute equation;
- The number of uses of `decExpr`-style wrapper productions that could not be replaced with the new tree sharing mechanisms, due to appearing in a non-unique context such as a higher-order inherited attribute equation;
- The number of productions in the grammar that would require a forward production attribute to enable static tree sharing.

The updated sources of these extensions can be found at https://github.com/melt-umn/ableC-<*extension-name*>; the versions evaluated here are archived at https://doi.org/10.13020/badh-qf44.

References

[1] John Tang Boyland. 2005. Remote attribute grammars. *J. ACM* 52, 4 (2005), 627–687. https://doi.org/10.1145/1082036.1082042

[2] Russel Cox, Tom Bergany, Austin Clements, Frans Kaashoek, and Eddie Kohlery. 2008. Xoc, an Extension-Oriented Compiler for Systems Programming. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 244–254. https://doi.org/10.1145/1353534.1346312

[3] Torbjörn Ekman and Görel Hedin. 2007. The JastAdd extensible Java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications (OOPSLA)*. ACM, 1–18. https://doi.org/10.1145/1297027.1297029

[4] Torbjörn Ekman and Görel Hedin. 2007. The JastAdd system - modular extensible compiler construction. *Science of Computer Programming* 69 (December 2007), 14–26. Issue 1-3. https://doi.org/10.1016/j.scico.2007.02.003

[5] Sebastian Erdweg, Tillmann Rendel, Christian Kastner, and Klaus Ostermann. 2011. SugarJ: Library-based Syntactic Language Extensibility. In *Proceedings of the Conference on Object Oriented Programming, Systems, Languages, and Systems (OOPSLA)*. ACM, 391–406. https://doi.org/10.1145/2048066.2048099

[6] R. Farrow. 1986. Automatic Generation of Fixed-Point-Finding Evaluators for Circular, but Well-Defined, Attribute Grammars. *ACM SIGPLAN Notices* 21, 7 (1986). https://doi.org/10.1145/13310.13320

[7] Görel Hedin. 2000. Reference Attribute Grammars. *Informatica* 24, 3 (2000), 301–317.

[8] T. Johnsson. 1987. Attribute grammars as a functional programming paradigm. In *Proc. of Functional Programming Languages and Computer Architecture (Lecture Notes in Computer Science, Vol. 274)*. Springer-Verlag, 154–173. https://doi.org/10.1007/3-540-18317-5_10

[9] Ted Kaminski. 2017. *Reliably Composable Language Extensions*. Ph. D. Dissertation. University of Minnesota, Minneapolis, Minnesota, USA. http://hdl.handle.net/11299/188954

[10] Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. 2017. *Reliable and automatic composition of language extensions to C — Supplemental Material*. Technical Report 17-009. University of Minnesota, Department of Computer Science and Engineering. Available at https://hdl.handle.net/11299/216011.

[11] Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. 2017. Reliable and Automatic Composition of Language Extensions to C: The ableC Extensible Language Framework. *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 98 (Oct. 2017), 29 pages. https://doi.org/10.1145/3138224

[12] Ted Kaminski and Eric Van Wyk. 2012. Modular well-definedness analysis for attribute grammars. In *Proceedings of the 5th International Conference on Software Language Engineering (SLE) (Lecture Notes in Computer Science, Vol. 7745)*. Springer, 352–371. https://doi.org/10.1007/978-3-642-36089-3_20

[13] Ted Kaminski and Eric Van Wyk. 2017. Ensuring Non-interference of Composable Language Extensions. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE)* (Vancouver, Canada). ACM, 163–174. https://doi.org/10.1145/3136014.3136023

[14] Uwe Kastens. 1980. Ordered attributed grammars. *Acta Informatica* 13 (1980), 229–256. Issue 3. https://doi.org/10.1007/BF00288644

[15] Donald E. Knuth. 1968. Semantics of Context-free Languages. *Mathematical Systems Theory* 2, 2 (1968), 127–145. https://doi.org/10.1007/BF01692511 Corrections in **5**(1971) pp. 95–96.

[16] Donald E. Knuth. 1971. Semantics of Context-free Languages: Correction. *Mathematical Systems Theory* 5, 2 (1971), 95–96. https://doi.org/10.1007/BF01702865 Corrections to [15].

[17] Lucas Kramer and Eric Van Wyk. 2019. Parallel Nondeterministic Programming as a Language Extension to C (Short Paper). In *Proceedings of the International Conference on Generative Programming: Concepts & Experience (GPCE)* (Athens, Greece). ACM, 20–26. https://doi.org/10.1145/3357765.3359524

[18] Lucas Kramer and Eric Van Wyk. 2020. Strategic Tree Rewriting in Attribute Grammars. In *Proceedings of the ACM SIGPLAN International Conference on Software Language Engineering (SLE)* (Virtual, USA). 210–229. https://doi.org/10.1145/3426425.3426943

[19] Eva Magnusson and Görel Hedin. 2007. Circular reference attributed grammars - their evaluation and applications. *Science of Computer Programming* 68, 1 (2007), 21–37. https://doi.org/10.1016/j.scico.2005.06.005

[20] Charles Simonyi, Magnus Christerson, and Shane Clifford. 2006. Intentional software. *SIGPLAN Notices* 41, 10 (2006), 451–464. https://doi.org/10.1145/1167515.1167511

[21] Anthony M. Sloane. 2011. Lightweight language processing in Kiama. In *Proceedings of the 3rd summer school on Generative and Transformational Techniques in Software Engineering III (GTTSE '09)* (Braga, Portugal) *(Lecture Notes in Computer Science, Vol. 6491)*. Springer, 408–425. https://doi.org/10.1007/978-3-642-18023-1_12

[22] Anthony M. Sloane, Matthew Roberts, and Leonard G. C. Hamey. 2014. Respect Your Parents: How Attribution and Rewriting Can Get Along. In *Software Language Engineering (Lecture Notes in Computer Science, Vol. 8706)*, Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju (Eds.). Springer, 191–210. https://doi.org/10.1007/978-3-319-11245-9_11

[23] Emma Söderberg and Görel Hedin. 2015. Declarative rewriting through circular nonterminal attributes. *Computer Languages, Systems & Structures* 44 (2015), 3 – 23. https://doi.org/10.1016/j.cl.2015.08.008

[24] Mark G.J. van den Brand and Paul Klint. 2007. ATerms for manipulation and exchange of structured data: It's all about sharing. *Information and Software Technology* 49, 1 (2007), 55–64. https://doi.org/10.1016/j.infsof.2006.08.009

[25] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. 2010. Silver: an Extensible Attribute Grammar System. *Science of Computer Programming* 75, 1–2 (January 2010), 39–54. https://doi.org/10.1016/j.scico.2009.07.004

[26] Eric Van Wyk, Oege de Moor, Kevin Backhouse, and Paul Kwiatkowski. 2002. Forwarding in Attribute Grammars for Modular Language Design. In *Proceedings of the 11th Conference on Compiler Construction (CC) (Lecture Notes in Computer Science, Vol. 2304)*. Springer-Verlag, 128–142. https://doi.org/10.1007/3-540-45937-5_11

[27] E. Van Wyk, O. de Moor, G. Sittampalam, I. Sanabria-Piretti, K. Backhouse, and P. Kwiatkowski. 2001. *Intentional Programming: a Host of Language Features*. Technical Report PRG-RR-01-21. Computing Laboratory, University of Oxford.

[28] Eelco Visser. 2001. Stratego: A Language for Program Transformation based on Rewriting Strategies. System Description of Stratego 0.5. In *Rewriting Techniques and Applications (RTA'01) (Lecture Notes in Computer Science, Vol. 2051)*, A. Middeldorp (Ed.). Springer-Verlag, 357–361. https://doi.org/10.1007/3-540-45127-7_27

[29] Harold H. Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. 1989. Higher Order Attribute Grammars. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 131–145. https://doi.org/10.1145/73141.74830

Received 2023-07-07; accepted 2023-09-01

Nanopass Attribute Grammars

Nathan Ringo
ringo025@umn.edu
University of Minnesota
USA

Lucas Kramer
krame505@umn.edu
University of Minnesota
USA

Eric Van Wyk
evw@umn.edu
University of Minnesota
USA

Abstract

Compilers for feature-rich languages are complex; they perform many analyses and optimizations, and often lower complex language constructs into simpler ones. The nanopass compiler architecture manages this complexity by specifying the compiler as a sequence of many small transformations, over slightly different, but clearly defined, versions of the language that each perform a single straightforward action. This avoids errors that arise from attempting to solve multiple problems at once and allows for testing at each step.

Attribute grammars are ill-suited for this architecture, primarily because they cannot identify the many versions of the language in a non-repetitive and type-safe way. We present a formulation of attribute grammars that addresses these concerns, called nanopass attribute grammars, that (*i*) identifies a collection of all language constructs and analyses (attributes), (*ii*) concisely constructs specific (sub) languages from this set and transformations between them, and (*iii*) specifies compositions of transformations to form nanopass compilers. The collection of all features can be statically typed and individual languages can be checked for well-definedness and circularity. We evaluate the approach by implementing a significant subset of the Go programming language in a prototype nanopass attribute grammar system.

CCS Concepts: • **Software and its engineering** → **Translator writing systems and compiler generators**.

Keywords: attribute grammars, compilers, nanopass compilers, software engineering

ACM Reference Format:
Nathan Ringo, Lucas Kramer, and Eric Van Wyk. 2023. Nanopass Attribute Grammars. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3623476.3623514

1 Introduction

Modern programming languages require sophisticated compilers. Feature-rich languages have many constructs, and a compiler typically performs several semantic analyses, optimizations, and transformations on programs. It can also be difficult to observe the behavior of different aspects of the compiler and test the results.

One approach to dealing with these complexities is to design the compiler as a sequence of several (perhaps dozens) small, clearly-defined tasks on programs in clearly-identified versions of the language. Compilers with this design are called *nanopass compilers* [18]. These tasks may be quite simple syntactic transformations such as reducing all if-then constructs in an imperative language into if-then-else constructs in which the else-clause is a skip statement. Other *lowering* transformations may replace list comprehensions with higher-order function calls or replace loops with `goto`s. These simple steps transform programs into simpler and smaller versions of the language, each one known to *not* contain the constructs that are transformed away. Other transformations may be more complex, such as type-checking in order to annotate expressions with their types; yet others may perform optimizations such as common sub-expression elimination. Some passes condense the source language down to language variations more suitable for translation. For example, a compiler may replace expressions in which binary operators are nested, as in `x + (y * z)`, with a sequence of operations that only allow atomic expressions such as variables and value literals as arguments to binary operators, as in `let temp1 = y * z in x + temp1`. Eventually, the steps transform the program to code that can be translated directly to a low-level intermediate language or to assembly, since it contains goto-statements and simple expressions.

This approach has been used successfully in both educational [19] and industrial [9] contexts. The proponents of nanopass compilers claim several benefits. The primary one is that each step is small and easy to understand. Because various language versions are clearly specified as the input and output of different passes, one can ensure that, *e.g.*, certain constructs have in-fact been transformed away and need not be considered again. Each step is also more amenable to testing as the output of each step can be inspected.

This paper adapts attribute grammars (AGs) to the nanopass approach. AGs were first specified by Knuth [10] in 1968 and are a convenient formalism for specifying computations over syntax trees. They work by decorating tree nodes with semantic information called attributes. *Synthesized* attributes propagate information up the tree, *e.g.* types on expressions. *Inherited* attributes propagate information down the tree, *e.g.* typing contexts on statements and expressions. Well-definedness analyses can ensure that all the equations needed to specify attribute values are present and are non-circular [10, 15, 16]. This provides a strong static exhaustiveness check that all language constructs have a specification for each semantic analysis. They have been extended over the years in a variety of ways, most prominently to support *higher-order* attributes [25] so that syntax trees may be passed as attributes, and *reference* [5] or *remote* [1] attributes that allow referring to remote nodes in the syntax tree.

Despite supporting a modular approach to language implementation, attribute grammars are not well-suited for the nanopass approach. The primary problem is that the context-free grammar in an attribute grammar defines a single complete language and there is no convenient mechanism in the formalism for defining a family of languages that differ in various ways such that trees in (different) input and output languages can both be safely constructed. One can, of course, define an AG for each of the dozens of different languages that arise in a nanopass compiler but this would involve the significant duplication of many grammar productions that appear in many of the language versions. The alternative is to abandon the well-definedness analyses and define equations for a particular task or transformation on only the relevant subset of productions for which that task occurs. But doing so is quite unsatisfactory.

The primary contribution of the paper is to close this gap between nanopass compilers and attribute grammars by providing a formulation of *nanopass attribute grammars* such that the various languages and their attributes can be both conveniently defined and checked for type-correctness, well-definedness, and absence of circular attribute specifications. A nanopass attribute grammar consists of 3 components:

1. $\mathcal{E}$: the collection of language elements. This is in the form of an attribute grammar from whose components different languages are to be constructed.
2. $\mathcal{L}$: the family of languages that are transformed between. A language $L \in \mathcal{L}$ is an AG that has a subset of the nonterminals, productions, attributes, etc. found in $\mathcal{E}$.
3. C: the composition of transformations into a nanopass compiler. This maps programs in the original language into some target form.

The collection of language elements $\mathcal{E}$ is statically type checked to ensure, for example, all productions are applied to the correct number of correctly-typed arguments. However, this specification may not be well-defined (some productions are intentionally missing equations for some attributes not relevant to them), and it is not meant to be used on its own.

Languages in $\mathcal{L}$ are attribute grammars and identify steps in the compilation process. They correspond to the different languages in a nanopass compiler. Terms in a language can be annotated with attribute values during construction so that they may be used, instead of recomputed, by the transformation. For example, a type-checking transformation will produce programs, if they are well-typed, that have annotations on expressions indicating their type. The annotations are just attributes that need not be computed but exist on the tree directly. Transformations are defined by *transform attributes* and correspond to passes in a nanopass compiler. The framework can be instantiated with different varieties of attribute grammars as well as different attribute evaluation mechanisms.

A second contribution is two mechanisms for *concisely* constructing the languages in $\mathcal{L}$ that maintains their type-correctness established in $\mathcal{E}$. The first specifies a language "from scratch" by identifying the productions and attribute occurrences on nonterminals to include; all other aspects, such as nonterminals in the grammar and equations for attributes are determined from the desired productions and occurrences. The second specifies a new language by extending an existing language by adding or removing components. Since each language in $\mathcal{L}$ will include different productions and attribute equations, the well-definedness and circularity analyses need to be performed on a per-language basis.

We also evaluate this notion of nanopass attribute grammars by implementing a prototype nanopass attribute grammar system and use it to implement a compiler for a significant subset of the Go programming language.[1]

Section 2 recalls the structure of attribute grammars before Section 3 provides the specification of nanopass approach to AGs and the $\mathcal{E}$, $\mathcal{L}$, and C components described above. Section 4 describes the prototype system realizing nanopass AGs and the Go compiler developed with it. Section 5 discusses related work; Section 6 discusses performance and attribute analysis, describes some future work, and concludes.

2 Background: Attribute Grammars

Attribute grammars are a formalism for defining the semantics of context free languages [10] by attributing semantic values to nodes in a syntax tree. An attribute grammar can be defined as a tuple $AG = (G, A, \Gamma_A, @, EQ)$ consisting of the context-free grammar G, attributes A, mappings of attributes to types (Γ_A) and to nonterminals on which they occur(@), and the set of attribute-defining equations EQ.

The grammar G is a tuple (NT, T, P, Γ_P, S). NT is a finite set of nonterminals and T is a finite set of terminals. T includes traditional token types with lexemes and primitive types, *e.g.* integers and strings. In some systems this includes

[1]Available at https://melt.cs.umn.edu and archived at https://doi.org/10.13020/h1qa-s993.

structured data such as lists or tuples. P is a finite set of production names and Γ_P is a total map from production names to their signatures. Signature elements are labeled so that equations can refer to the left and right-side elements using labels instead of positions. Thus, Γ_P maps P to signatures of the form $x_0 : X_0 ::= x_1 : \tau_1 \ldots x_n : \tau_n$ where $X_0 \in NT$ and each $\tau_i \in NT \cup T$ for $i \in \{1..n\}$. $S \in NT$ is the start symbol indicating the type of the root node of a program tree.

Attributes are specified by the set $A = A_I \cup A_S$ which can be partitioned into disjoint sets of inherited (I) and synthesized (S) attributes. Γ_A is a total map from attribute names to types in T, $\Gamma_A \subseteq ((A_I \cup A_S) \rightarrow T)$. The *occurs-on* relation $@ \subseteq A \times NT$ specifies which nonterminals an attribute decorates; $(a, X) \in @$ (written $a@X$ as shorthand) indicates that attribute a occurs on X. Note that S has no inherited attributes: $\forall a \in A, (a@S) \implies a \notin A_I$.

Equations, $EQ = \bigcup_{p \in P} EQ_p$, indicate how values of attributes are determined; each is associated with a production p in P. Each has the form $x.a = e$ where $a \in A$, x is label on the production, and e is an expression defining the value of $x.a$. We require that for any production p, no two equations in EQ_p have the same left hand side. Different attribute grammar systems put different requirements on the constructs in e, but generally, e is an expression that can refer to the values of attributes on the signature elements of p and construct and manipulate these values.

Since Knuth's original specification [10] attribute grammars have been extended in many ways. One variety extends the types that attributes may take (the range of Γ_A) and constructs in equations accordingly. Higher-order attributes [25] dramatically increase the usefulness of AGs by allowing trees to be passed around as attributes and then supplied with inherited attributes so that synthesized attributes can then be computed on them and accessed. These add NT to the range of Γ_A. Reference [5] and remote [1] attributes extend Γ_A with pointers (references) to decorated tree nodes somewhere else in the syntax tree. A common use is to allow variable uses in a program to have a reference attribute pointing to their declarations so that information such as the variable's type can be accessed on the remote declaration node.

Another form of extension provides means for more easily moving values up and down the tree. Kastens and Waite [8] alleviate so-called copy-rules for propagating information down the tree and described other mechanism for collecting information, such as diagnostic messages, up the tree. Variations on these are now common in AG systems.

An important aspect of attribute grammars, and many of their extensions, are static analyses to identify and validate the flow of information through different attributes. A *well-definedness* analysis in many systems determines, for each synthesized attribute, which inherited attributes may be needed to compute its value (sometimes called *flow-types* [17]) in order to determine if all required equations are present. This information can also be used to define a circularity analysis to check for cycles in attribute dependencies. These analyses were provided in Knuth's original formulation and are typically extended to accommodate new features, such as higher-order attributes [25].

There also a variety of mechanisms for computing the values for attributes on a tree. Ordered attribute grammars [7] and an extension of them [22] determine an order, applicable for all possible trees, for computing attributes. In contrast, the commonly-used demand-driven approach treats AGs similarly to lazy functional programs and computes attributes only as they are needed. Other approaches embed AGs in existing languages, often lazy functional ones, to write the specification directly as programs in those languages [12, 21]. Circular attribute grammars allow attribute dependencies to be circular as long as they are well-founded, providing a convenient means for specifying fix-point algorithms [3, 13].

This discussion of attribute grammars and their different variations is necessarily incomplete. In principle, these, and others, fit into the nanopass attribute grammar formalism presented below. That framework can be instantiated with different types of attributes and evaluation schemes.

3 Nanopass Attribute Grammars

In this section we describe the nanopass attribute grammar formalism: the language elements $\mathcal{E}$, the family of languages $\mathcal{L}$, and the compositions $\mathcal{C}$. Sections 3.1 - 3.2 specify $\mathcal{E}$ by extending the formalism specified in Section 2, discuss what is required for it to be *well-formed*, and provide its type-checking rules. Sections 3.3 - 3.5 specify $\mathcal{L}$ and transform attributes and discuss the *language checking* process performed on each language to ensure that it is well-defined and that its transformations produce terms using only the productions in their target language. Section 3.6 describes how transformations are composed to construct a nanopass compiler.

3.1 Language elements: $\mathcal{E}$

The first part of a nanopass AG specification, denoted $\mathcal{E}$, is a collection of attribute grammar elements $(G, A, \Gamma_A, @, EQ)$ as above from which different languages will be constructed. This is extended in two ways.

First, we add a new kind of attribute, transform attributes, denoted A_T. The A component of $\mathcal{E}$ is now $A = A_I \cup (A_S \cup A_T)$. These are essentially higher-order synthesized attributes for defining transformations between languages in $\mathcal{L}$ by equations also in EQ. Transform attributes always have the same type as the nonterminal they're computed on, so they are not (and need not be) included in Γ_A. These are discussed further in Section 3.4.

The second addition is *annotations*. These annotate the tree with semantic values supplied when the tree is constructed instead of being computed during attribute evaluation. To simplify the formalism, these are specified as attributes whose equations are ignored when it is demanded; instead, the value is the one provided when the tree was built. Note that Γ_A still contains the types of annotations.

To support annotations, expressions that appear on the right-hand-side of equations can supply them to trees that they construct. If we notate building a tree from production p with children t_1 and t_2 as $p(t_1, t_2)$, we might notate building the same tree while supplying annotation a with the value of the expression e as $p(t_1, t_2, a = e)$.

Annotations might be used to store, for example, the types of expressions. After the types have been computed (as an attribute), we will still continue to transform the program in the process of compiling it. To avoid needing to re-type-check the tree at any time the types of expressions might be needed later on, we can make the attribute an annotation.

As discussed in Section 3.3, each language in $\mathcal{L}$ will indicate which attributes in $A_I \cup A_S$ are to be treated as annotations and thus be predefined on trees in the language.

Due to how close $\mathcal{E}$ is to a standard AG specification, existing attribute evaluation strategies can work with only minor tweaks to support annotations. An attribute evaluation strategy that is aware of languages can take advantage of this to remove dependencies between equations that might otherwise exist.

If $\mathcal{E}$ satisfies the requirements mentioned above and for *AG* in Section 2, it is said to be *well-formed*; this is the first requirement the specifications must satisfy. Note that unlike a traditional attribute grammar specification, $\mathcal{E}$ is not expected to be well-defined. After some transformations have been applied some language constructs will have been translated into other more fundamental forms and thus no longer appear in the programs. Since later passes won't need to handle constructs that won't be present, we don't need equations on those productions for attributes that are only used after the production is eliminated.

3.2 Type checking language elements $\mathcal{E}$

Even though we cannot check well-definedness on a collection of elements $\mathcal{E}$, we can still check that they are well-typed, given some language of expressions that may appear on the right-hand side of an equation. Type checking can be done once on the language elements $\mathcal{E}$ and type-correctness will preserved for languages in $\mathcal{L}$ when they are constructed using one of the two methods described in Section 3.3. Some rules for a reasonably standard type system adapted for a NAG system are shown in Figure 1.

The type-correctness of a well-formed $\mathcal{E}$ is satisfied when for all productions $p \in P, P \in AG$, all equations $x.a = e$ in $EQ_p, EQ_p \in AG$ type check, as indicated by the judgment

$$p \vdash (x.a = e)\ \textbf{Tok}$$

This judgment in turn refers to a traditional typing judgment for expressions,

$$p \vdash e : \tau$$

In both cases, $p \in P$ acts as a context, providing the types of children. The components of *AG* are also ambiently present as the global context and referred to by names used above. Thus typing contexts such as Γ_A and Γ_P, the occurs-on relation @, and other aspects of *AG* can be used in the type checking rules. Synthesized and inherited equations have typical typing rules, T-INH-EQ and T-SYN-EQ, ensuring that attributes occur properly and the type of the expression matches that of the attribute. Following these are 3 sample rules for typing expressions. Of more interest are transform attributes; their equations are typed by T-TRANSFORM-EQ, which ensures that the type of the equation's right-hand side matches the production's left-hand side. Their access is typed similarly by T-TRANSFORM-ACCESS.

The T-PROD rule for constructing trees is more general than a typical one, since it needs to handle annotations. Note that type-checking does not check that only the annotations that should occur do occur. In $\mathcal{E}$ we do not know whether an attribute will be treated as an annotation or as an attribute to be computed. This check happens in the language-correctness checks described in Section 3.5.

3.3 Languages: $\mathcal{L}$

A nanopass AG also consists of a *family of languages*, and *transformations* between them, $\mathcal{L}$. Each language uses a subset of the grammatical and semantical features found in $\mathcal{E}$. They will use some productions and some attributes to define a language with only the desired syntax and semantics. We discuss two convenient and concise mechanisms to identify what these languages consist of so that the type-correctness established once on $\mathcal{E}$ is maintained on each language and need not be checked again.

3.3.1 Languages.

The productions, attributes, associated equations, etc., are all specified in $\mathcal{E}$ and may (or may not) be used in different languages and thus languages have (nearly) the same structure as the collection of language elements $\mathcal{E}$. We will superscript language elements by the language names and also superscript the overarching language components elements by $\mathcal{E}$. When the context is clear we will drop these superscripts.

A language $L \in \mathcal{L}$ is a 6-tuple containing:

- $G^L = (NT^L, T^L, P^L, \Gamma_P^L, S^L)$ where $NT^L \subseteq NT^{\mathcal{E}}$, $T^L = T^{\mathcal{E}}$, $P^L \subseteq P^{\mathcal{E}}$, $\Gamma_P^L \subseteq \Gamma_P^{\mathcal{E}}$, and $S^L = S^{\mathcal{E}}$.
- $A^L = A_I^L \cup (A_S^L \cup A_T^L)$, where $A^L \subseteq A^{\mathcal{E}}$
- $@^L \subseteq @^{\mathcal{E}}$
- $\Gamma_A^L \subseteq \Gamma_A^{\mathcal{E}}$
- $EQ^L = \cup_{p \in P^L} EQ_p^L$ where $EQ_p^P \subseteq EQ^{\mathcal{E}}{}_p$
- $@_A^L \subseteq @^{\mathcal{E}}$ where $@^L \cap @_A^L = \varnothing$

The first five components correspond directly to their equivalents in $\mathcal{E}$. Note that terminals and primitive values T^L and T are not scoped to a particular language and are

T-INH-EQ
$$\frac{a \in A_I \qquad a@X \qquad \Gamma_P(p) = (\ldots ::= \ldots\ x : X \ldots) \qquad p \vdash e : \Gamma_A(a)}{p \vdash (x.a = e)\ \textbf{Tok}}$$

T-SYN-EQ
$$\frac{a \in A_S \qquad a@X \qquad \Gamma_P(p) = (x : X ::= \ldots) \qquad p \vdash e : \Gamma_A(a)}{p \vdash (x.a = e)\ \textbf{Tok}}$$

T-LHS
$$\frac{p \in P \qquad \Gamma_P(p) = (x : X ::= \ldots)}{p \vdash x : X}$$

T-RHS
$$\frac{p \in P \qquad \Gamma_P(p) = (\ldots ::= \ldots\ x : \tau \ldots)}{p \vdash x : \tau}$$

T-INHSYN-ACCESS
$$\frac{a \in (A_I \cup A_S) \qquad a@X \qquad p \vdash e : X \qquad \Gamma_A(a) = \tau}{p \vdash e.a : \tau}$$

T-TRANSFORM-EQ
$$\frac{a \in A_T \qquad a@X \qquad \Gamma_P(p) = (x : X ::= \ldots) \qquad p \vdash e : X}{p \vdash (x.a = e)\ \textbf{Tok}}$$

T-TRANSFORM-ACCESS
$$\frac{a \in A_T \qquad a@X \qquad p \vdash e : X}{p \vdash e.a : X}$$

T-PROD
$$\frac{\Gamma_P(p') = (x_0 : X_0 ::= x_1 : \tau_1 \ldots x_m : \tau_m) \qquad \underset{1\le i\le n}{\forall}\ p \vdash e_i : \tau_i \qquad \underset{m<i\le n}{\forall}\ a_i@X \qquad \underset{m<i\le n}{\forall}\ \Gamma_A(a_i) = \tau_i}{p \vdash p'(e_1, \ldots, e_m, a_{m+1} = e_{m+1}, \ldots, a_n = e_n) : X_0}$$

Figure 1. Typing rules for $\mathcal{E}$.

available in all languages. This is done to simplify the presentation, but could be accommodated without much effort. Note that in all languages, the start symbol is the same $S^{\mathcal{E}}$.

The sixth component, $@^L_A$, describes the annotations that are present on each nonterminal. Values for these are provided when the L tree is constructed. Recall the use-case of saving the results of type-checking expressions. We assume the computed type is ty, the nonterminal for expressions is E, the language in which type-checking is performed is L_0, and the language in which the ty attribute is an annotation is L_1. In this case, both $ty \in A^{L_0}_S$ and $ty \in A^{L_1}_S$, but while $ty\ @^{L_0}\ E$, instead $ty\ @^{L_1}_A\ E$.

Identifying languages. A language is simply a subset of $\mathcal{E}$ along with an indication of annotations $@_A$. Identifying a specific language $L \in \mathcal{L}$ by enumerating all of the elements would be quite tedious and also open to errors from leaving out required components. For example, L might not be well-formed if the production signature map Γ^L_P does not have a signature for a production in P^L. It might also not be type-correct if in the expression for an equation in EQ^L references an attribute that is not found in A^L, $@^L$, or Γ^L_A. To avoid these problems we provide two mechanisms for specifying languages that are both concise and also result in type correct languages.

The first mechanism identifies a language L *directly*. It requires only the enumeration of production names (P^L) that are to be used and the desired occurrences of attributes ($@^L$) and annotations ($@^L_A$) to be used. All other elements of L can be inferred from these. For each production name $p \in P^L$, we include its signature in Γ^L_P. Any nonterminal appearing in Γ^L_P is added to NT^L, and $S^L = S^{\mathcal{E}}$. Thus G^L is well-formed. From the attribute and annotation occurrence relation elements identified we populate the sets of attribute A^L_I, A^L_S, and A^L_T. Also, any nonterminal X in $@^L$ or in $@^L_A$ is added to NT^L if not already there. Similarly, appropriate equations are selected for attributes in $@^L$. The equation $x.a = e$ from $EQ^{\mathcal{E}}_P$ is included in EQ^L_p when $p \in P^L$ and $a \in A^L$. Equations for annotations are *not* included in EQ^L. Γ^L_A is Γ_A restricted to attributes in A^L_S and A^L_I.

Why design things in this way? Recall the two types of expression productions discussed in Section 1: one allowed (nested) expressions as children; the other allowed only atomic children of variables and literals. The source language for the transformation that rewrites nested expressions into atomic ones needs to identify only the complex expression productions. These are the expressions that may be used to construct a tree that is input to this transformation. We would not want to include the atomic expression productions in this collection because that would indicate that they could also be used to form input terms. Doing so would not allow us to ensure that input and output languages are of the proper form. The *language checking* process described in Section 3.5 will ensure that expressions only generate trees in the appropriate language and that attributes access are in fact in the language. This check, along with ensuring that all the required equations are present, is done in the language-checking process. Specifically, see the discussion of the language-checking rule L-PROD there.

The second mechanism for identifying a language L does so by *extension*. It uses an *extends* mechanism that creates a new language by identifying elements to add to, or remove from, an existing language. Formally, a language L identified this way is specified as:

- a language L', perhaps also defined as an extension,
- a set of production names P^{L+} to add to $P^{L'}$
- a set of production names P^{L-} to remove from $P^{L'}$
- a set of occurrences $@^{L+}$ to add to $@^{L'}$

- a set of occurrences $@^{L-}$ to remove from $@^{L'}$
- a set of occurrences $@_A^{L+}$ to add to $@_A^{L'}$
- a set of occurrences $@_A^{L-}$ to remove from $@_A^{L'}$

From these, we can compute:

- $P^L = (P^{L'} \cup P^{L+}) - P^{L-}$
- $@^L = (@^{L'} \cup @^{L+}) - @^{L-}$
- $@_A^L = (@_A^{L'} \cup @_A^{L+}) - @_A^{L-}$

From here, the other elements of L are inferred using the same process as described above for creating language directly from a set of productions and occurrences. Similarly, the resulting language is well-formed and well-typed in the same manner.

3.4 Transform attributes

Transform attributes play a key role in nanopass attribute grammars, as they define the transformations from one language into the next. A transform attribute $a_T \in A_T$ is defined as having a source language $\Gamma_S(a_T) = L_S$ and a target language $\Gamma_T(a_T) = L_T$. Transform attributes have equations similar to those used for higher-order attributes.

For many transformations, the computation for many productions in the language is to simply apply the transformation to the child trees and re-build the tree with the same production and the transformed child trees. It would be quite inconvenient to write these directly and thus they are inferred when an explicit equation is not provided for a production. Consider a production *selector* for selecting a field from a record with signature *e*:*Expr* ::= *l*:*Expr* *r*:*String* with no explicit equation for the transformation a_T. When the target language has no annotations, the generated equation would be $e.a_T = \mathit{selector}(l.a_T, r)$. The same production is used, and the a_T attribute is computed on each child of nonterminal type. Children that are not of nonterminal type are passed as-is.

If the production has annotations in the target language, the generated equation copies them over from the attributes and annotations in the source language. The a_T attribute is recursively applied if the annotation has a nonterminal type — this is the same process as performed on the children. For example, if in the target language of a_T, *Expr* has two annotations, *isInLambda*:*Boolean* and *type*:*Type*, the generated equation would be:

$$e.a_T = \mathit{selector}(l.a_T, r, \mathit{isInLambda} = e.\mathit{isInLambda},\\ \mathit{type} = e.\mathit{type}.a_T)$$

Note that the $e.\mathit{type}.a_T$ call would require that there are no non-annotation inherited attributes on the *Type* nonterminal in a_T's source language, since additional attribute equations are not supplied here.

In the general case, the generated equation for a production p with signature $e{:}X_e$ ::= $x_0{:}X_0$ $x_1{:}X_1$ ... would be $e.a_T = p(x_0[.a_T], ..., a_0 = \mathit{this}.a_0[.a_T], ...)$, where the $[.a_T]$ represents the attribute only being demanded on children, attributes, and annotations of nonterminal type.

Note that this equation is generated only if:

- $p \in P^{L_T}$,
- $@^{L_S} \cup @_A^{L_S} \subseteq @_A^{L_T}$, and
- $a @_A^{L_S} X \implies \nexists a_I \in A_I$ such that $a_I @^{L_S} \Gamma_A(a)$

If these conditions are not met, the programmer is required to explicitly provide an equation. Consider the transformation replacing *if c then s* with *if c then s else skip*. Since the if-then production is not in the target language an explicit equation is required. Likewise, if the attributes and annotations in the source are not enough to define the annotations in the target then equations are required. Finally, as noted, we require that annotations on the source language do not need inherited attributes on that type.

The process of generating an equation for a transform attribute always results in a well-typed equation. If the rest of $\mathcal{E}$ passed type-checking, we know the entire specification is well-typed, and that evaluation of attributes will not result in runtime type errors.

3.5 Language checking

A language $L \in \mathcal{L}$ is language-correct if every equation in EQ^L evaluates to a term in the appropriate language. Recall that synthesized and inherited attributes produce terms in the same language as the language of the term the attributes are computed on, while transform attributes produce terms in the attribute's target language.

If $\mathcal{E}$ is type correct, then the languages $L \in \mathcal{L}$ will, if constructed using the methods described above, also be type correct and will not experience typical evaluation-time type errors. Some concerns however must be checked on the individual languages and their transform attributes. These checks ensure that a language L is *language-correct* and that the evaluation-time errors listed below do not occur.

- L is missing an equation for an attributes in $A_S^L \cup A_I^L$ for production in P^L that may be demanded during evaluation.
- L computes a tree for a higher-order attribute in A_S^L or A_I^L that is not in the language of L. That is, a production not in P^L is used in the tree.
- Similarly, L computes a tree for a transform attribute $a \in A_T$ with target language L_T ($\Gamma_T(a) = L_T$) that is not in the language of L_T.
- L has an equation for a synthesized attribute $a \in A_S^L$ for production p but also defines a as an annotation in an application of the production p, as in $p(..., a = ...)$.
- L has an equation that demands the value of an inherited attribute on a tree that does not have a parent tree that can provide the defining equations. This could happen when accessing a synthesized or transform attribute (that depends on an inherited attribute) on

a tree *stored as a higher-order attribute* since that tree has no parent.

The language-correctness of a language L is indicated by the judgment $L \vdash \textbf{Lok}$. By definition, $L \vdash \textbf{Lok}$ is established if for all nonterminals $X \in NT^L$, for all productions $p \in P^L$ with X on the left hand side, and for all attributes $a \in (A_S \cup A_I)$ where $a@^L X$ the following conditions hold:

- $(x.a = e) \in EQ_p^L$: ensuring that the attribute grammar is complete and no equations are missing, and
- $p, L \vdash (x.a = e)\ \textbf{Lok}$, ensuring L is language-correct.

Completeness is a conservative analysis and it is often more convenient to require that only the equations that would ever be needed in an attribute evaluation are present. In Section 6 we discuss why this conservative analysis is less of a concern in nanopass attribute grammars than in traditional higher-order attribute grammars. This second requirement could be replaced by a less-conservative analysis if desired.

This language-correctness analysis assumes that type-checking has already been performed. It is specified as a collection of inference rules, as was done with type checking. It is organized as a pair of judgments, one for equations and one for expressions:

$$p, L \vdash (x.a = e)\ \textbf{Lok} \quad \text{and} \quad p, L, L_T \vdash e\ \textbf{Lok}$$

These judgments are parameterized by languages. L is the language of the tree the attribute is being computed on. It is constant throughout the analysis. L_T is the language the expression computing a tree should evaluate to: the source language L for inherited and synthesized attributes and the target language L_T for transform attributes. Some of the rules establishing these judgments are given in Figure 2.

The rule L-INHSYN-EQ for synthesized and inherited equations ensures that the expression of the equation will compute a tree in the source language L. The rule L-TRANSFORM-EQ for transform equations checks the expression e using the target language L_T to ensure that trees used in a_T are in L_T. A node and its children are always in the same language as the attribute is being computed on. This allows accessing attributes that are defined on the source language.

Accessing a synthesized or inherited attribute (or an annotation) preserves the language of the term it is computed on, since we would expect the results of analyses on a term to be in the term's own language. It also requires that the attribute be present in the language. To ensure all attribute accesses are well-defined, the access may be on a name from the signature (L-INHSYN-SIG-ACCESS) where, due to the completeness check, all attributes in the language have equations defined. The access could also be of an annotation on any expression (L-ANNO-ACCESS), since the annotation must have been supplied when the term was constructed. A synthesized attribute can also be accessed on an arbitrary expression (L-SYN-NODEPS-ACCESS), for example on a higher-order attribute,

L-INHSYN-EQ
$$\frac{a \in (A_I^L \cup A_S^L) \qquad p, L, L \vdash e\ \textbf{Lok}}{p, L \vdash (x.a = e)\ \textbf{Lok}}$$

L-TRANSFORM-EQ
$$\frac{\begin{array}{cc} a \in A_T^L & p, L, L_T \vdash e\ \textbf{Lok} \\ \Gamma_S^L(a) = L & \Gamma_T^L(a) = L_T \end{array}}{p, L \vdash (x.a = e)\ \textbf{Lok}}$$

L-LHS
$$\frac{\Gamma_P^L(p) = (x{:}X ::= \ldots)}{p, L, L \vdash x\ \textbf{Lok}}$$

L-RHS
$$\frac{\Gamma_P^L(p) = (\ldots ::= \ldots\ x{:}\tau\ \ldots)}{p, L, L \vdash x\ \textbf{Lok}}$$

L-INHSYN-SIG-ACCESS
$$\frac{x{:}X \in \Gamma_P^L(p) \qquad a \in (A_I^L \cup A_S^L) \qquad a@^L X}{p, L, L_T \vdash x.a\ \textbf{Lok}}$$

L-ANNO-ACCESS
$$\frac{\begin{array}{c} a \in (A_I^L \cup A_S^L) \\ a@_A^L X \qquad p \vdash e{:}X \qquad p, L, L_T \vdash e\ \textbf{Lok} \end{array}}{p, L, L_T \vdash e.a\ \textbf{Lok}}$$

L-SYN-NODEPS-ACCESS
$$\frac{\begin{array}{c} a \in A_S^L \\ a@^L X \qquad \forall_{a'@^L X}\ a' \notin A_I^L \qquad p \vdash e{:}X \qquad p, L, L_T \vdash e\ \textbf{Lok} \end{array}}{p, L, L_T \vdash e.a\ \textbf{Lok}}$$

L-TRANSFORM-SIG-ACCESS
$$\frac{\begin{array}{c} a \in A_T \\ a@^L X \qquad x{:}X \in \Gamma_P^L(p) \qquad \Gamma_S(a)^L = L \qquad \Gamma_T(a)^L = L_T \end{array}}{p, L, L_T \vdash x.a\ \textbf{Lok}}$$

L-TRANSFORM-NODEPS-ACCESS
$$\frac{\begin{array}{c} a \in A_T \qquad a@^L X \qquad \forall_{a'@^L X}\ a' \notin A_I^L \qquad \Gamma_S(a)^L = L_S \\ \Gamma_T(a)^L = L_T \qquad p \vdash e{:}X \qquad p, L, L_S \vdash e\ \textbf{Lok} \end{array}}{p, L, L_T \vdash e.a\ \textbf{Lok}}$$

L-PROD
$$\frac{\begin{array}{c} p' \in P^{L_T} \qquad \Gamma_P(p') = (X ::= \ldots) \\ \forall_{0 \le i \le n}\ p, L, L_T \vdash e_i\ \textbf{Lok} \qquad \{a_{m+1}, \ldots, a_n\} = \{a \mid a@_A^L X\} \end{array}}{p, L, L_T \vdash p'(e_0, \ldots, e_m, a_{m+1} = e_{m+1}, \ldots, a_n = e_n)\ \textbf{Lok}}$$

L-PRIM
$$\frac{p \vdash e{:}\tau \qquad p, L, L_e \vdash e\ \textbf{Lok} \qquad \tau \notin NT \qquad L_e \in \mathcal{L}}{p, L, L_T \vdash e\ \textbf{Lok}}$$

L-IF
$$\frac{p, L, L_T \vdash c\ \textbf{Lok} \qquad p, L, L_T \vdash t\ \textbf{Lok} \qquad p, L, L_T \vdash e\ \textbf{Lok}}{p, L, L_T \vdash (\texttt{if}\ c\ \texttt{then}\ t\ \texttt{else}\ e)\ \textbf{Lok}}$$

Figure 2. Some language correctness rules for $L \in \mathcal{L}$.

when there are no inherited attributes in the language occurring on the type, as the synthesized attribute cannot possibly depend on any inherited attributes.

Accessing a transform attribute produces a term in the transform attribute's target language, from a term in the transform attribute's source language. The rules L-TRANSFORM-SIG-ACCESS and L-TRANSFORM-NODEPS-ACCESS inspect the languages L and L_T in the context of the rules. Similar rules exist for transform attributes accessed as annotation ($a@^L_A X$) but this is rarely done.

Constructing a term with a production entails more checks. The rule L-PROD checks that the production is valid in the language L_T, that its subterms are valid in that language, and that it has exactly the set of annotations it should have to be in that language.

Most of the time, L and L_T are the same language in equations for inherited or synthesized attributes (L-INHSYN-EQ), since these attributes' values are in the same language as the tree they are computed on. In contrast, transform equations (L-TRANSFORM-EQ) require that L and L_T are the source and target languages of the transform attribute.

The last issue to address is in regards to primitive types in T. They don't belong to any language, so we want to leave them unconstrained with respect to languages. Consider the equation computing a term for a transform attribute a_T using production $p \in L_T$ that uses an Boolean annotation value computed on a tree in the source language ($x.a_S$):

$$x.a_T = p(ann = x.a_S)$$

Since the Boolean value is, by definition, in the language of L_T we do not care about the languages of the trees involved in computing it. We can add the L-PRIM rule to handle this case. Note that it leaves the target language for checking e to be unconstrained: L_e can be any language in $\mathcal{L}$. [2]

With L-PRIM in place, the rules for expressions of base type can be trivial, since they can belong to any language. The only feature of note is in the rule L-IF since the *then* and *else* branches of *if* expressions must have the same L_T as the expression unless they are base types (in which case the L-PRIM rule applies). We elide similar rules for other base type expressions such as addition or numeric literals.

3.6 Composition on nanopasses: C

A nanopass attribute grammar will also specify at least one composition of a desired set of transformations, typically to lower the source language down to a version that can be easily translated to some target language. For example, we may lower an imperative language with various control flow mechanisms and expressions over various types down to a version that only uses labels and goto-statements for control flow and expressions are transformed into assignment statement sequences that directly translate into low-level intermediate code similar to assembly language instructions.

A composition, in its most primitive form, is a sequence of transform attributes $(t_0, ..., t_n)$, where each $t_i \in A_T$. This implicitly identifies the source and target languages of the overall composition; the overall source language is $\Gamma_S(t_0)$, while the overall target language is $\Gamma_T(t_n)$. In practice, a composition may check the results of on transformation before performing the next one. For example, if type errors are found on a program in a typing transformation then those errors may be output and the compilation aborted.

If $\mathcal{E}$ is type correct and the languages are all language-correct, then we can check that the compositions are also type-correct. For a composition to be type-correct the following condition must hold:

$$\forall i, i \in \{1...n\}.\ \Gamma_S(t_i) = \Gamma_T(t_{i-1})$$

This ensures that the input tree for each transformation matches the source language of the transformation. Recall that the start symbol S has no inherited attributes and thus the computation of some transform attribute t_i does not need them to be specified. If additional information is needed by in the computation of a transform attribute, then that information can be supplied as an annotation.

The output of the simple composition form above is thus a tree in $\Gamma_T(t_n)$. We can then use this tree as input to the next step in the compilation, either to compute a textual representation of a program in a target language or use higher-order attributes to construct a tree in some other language.

Multiple compositions may be defined against a single nanopass attribute grammar. For example, a production compiler may wish to define multiple optimization levels that run different sets of passes.

4 Evaluation - a Nanopass Go Compiler

To evaluate the design of nanopass attribute grammars we have developed a prototype nanopass attribute grammar system and used it to implement several passes in a compiler for Go version 1.17 that generates x86_64 assembly language code. Go is a lexically-scoped, statically-typed, imperative language and, in many respects, has control-flow statements and expressions that one might expect. We note specific points of interest below as they become relevant.

In this section we describe several of the 32 languages and transformations between them. Section 4.1 describes the simple transformation that lowers for-loops into while-loops. Section 4.2 describes the more complicated transformation (and some of its predecessors) of lowering complex numbers into records with a real and imaginary field. This demonstrates how transformations lower not only the program syntax but also that of annotations. Section 4.3 describes the last language in the sequence; one that has been sufficiently lowered to enable a direct translation to assembly language.

[2]Although this rule is non-algorithmic, it can easily be conservatively approximated and is so implemented in our prototype system.

```
 1 // for <clause> <body>
 2 prod forStmt(clause: LoopClause,
 3              body: Stmt): Stmt;
 4
 5 // for x != 0 { ... }
 6 prod whileClause(expr: Expr): LoopClause;
 7
 8 // for i := 0; i < 10; i++ { ... }
 9 prod forClause(init: Stmt, cond: Expr,
10                post: Stmt): LoopClause;
11
12 // for k, v = range m { ... }
13 prod rangeClause(lhs: ExprList,
14                  rhs: Expr): LoopClause;
```

Figure 3. A subset of the productions describing loops.

Section A in the appendix lists and briefly describes all 32 languages and 34 passes (transformations).

4.1 Lowering `for`-style loops to while-style loops

Go supports both `for` loops that have the C-like structure of `for init; cond; post { body }` and while-style ones that have only a condition. We lower the former to the latter. Figure 3 shows, in the language of our prototype, that in the abstract syntax, both loops represented by a single generic `forStmt` production (line 2) that encapsulates all the looping constructs Go supports. Interestingly, they all use the keyword `for`, so this structure is not unreasonable. There are then various clauses that can be used with the `for` keyword, which are productions of the `LoopClause` nonterminal: while-loop style on line 6, C for-loop style on line 9, and a clause for ranging over key-value pairs on line 13. There are many other productions in $\mathcal{E}$ that we do not show, but they do have the expected form.

The meta-language syntax of the prototype essentially adds concrete syntax to the constructs in the formalism in Section 3. It does interleave elements of $\mathcal{E}$ and $\mathcal{L}$ but it should be straightforward to read. Productions are written in a functional style so that the left-hand-side nonterminal appears last and the right-hand-side elements are in parens.

The lowering of C-style for-loops into while-style loops takes place after 18 previous passes, on language `L15`, and is shown in Figure 4. `L0`, the initial language in our nanopass compiler, contains the abstract syntax of Go and the intervening languages resolve names and perform type checking. The transform attribute `toL16` (line 6) produces programs without these kinds of loops in language `L16`. Language `L16` is specified by extending `L15` as shown on lines 1–5. It removes (`-=`) the `forClause` production from nonterminal `LoopClause`'s set of productions (line 2) and includes only (`:=`) occurrences of attributes `liftedInit` and `liftedPost` on `LoopClause` (line 3). These are defined as higher-order

```
 1 lang L16 extends L15 {
 2   LoopClause.prods -= {forClause},
 3   LoopClause.attrs := {liftedInit,
 4                         liftedPost}
 5 }
 6 transform attribute toL16 from L15 to L16;
 7
 8 syn liftedInit: Stmt;
 9 syn liftedPost: Stmt;
10
11 aspect forClause {
12   this.liftedInit := init;
13   this.toL16 := whileClause(cond);
14   this.liftedPost := post;
15 }
16
17 aspect default LoopClause {
18   this.liftedInit := emptyStmt();
19   this.liftedPost := emptyStmt();
20 }
21
22 aspect forStmt {
23   this.toL16 := block {appendStmt(
24     clause.liftedInit,
25     forStmt(
26       clause.toL16,
27       appendStmt(
28         body.toL16,
29         clause.liftedPost))));     }
```

Figure 4. Lowering C-style for-loops to while-style loops.

synthesized attributes holding statements (lines 8–9). The `aspect` constructs associate equations with productions and, by convention, use `this` as the name of the constructed tree node. Names of argument trees are found in the production declarations in Figure 3. These attributes lift the `init` and `post` components out of a C-style for-loop (line 12, line 14) and are the empty statement, by `default`, on other `LoopClause` productions (lines 17–20). The `forClause` can then transform itself into an `whileClause` that uses the `cond` child as the loop condition (line 13). Finally, we define an equation on the `forStmt` production to actually put the `liftedInit` and `liftedPost` statements into place in the new while-style loop (lines 22–29). For all other `Stmt` productions, *e.g.* if-then-else statements, a default equation is generated, as discussed in Section 3.4, to apply the `toL16` transformation to its components and build the same tree with those transformed results. Thus, a statement of the form `for init; cond; post { body }` is lowered to one of the form `{ init; for cond { body; post }}`. Figure 4 contains the entirety of the code for this pass.

4.2 Lowering complex numbers

Go supports complex numbers with imaginary number literals, overloaded arithmetic operators, and supporting library functions for constructing and accessing complex numbers. Here we discuss a transformation that lowers them to structs containing two floating-point numbers. (For simplicity we assume complex numbers are 64 bits; the transformation can easily be made to also handle 128-bit complex numbers.) Interestingly, complex number types and functions are not built-in syntax, but instead are library functions whose names may be shadowed by programmer-declared names, thus complicating the lowering.

This process is done in a few steps, which we will illustrate on an example function shown in its original `L0` form at the top of Figure 5. These versions are the concrete syntax versions of the results of the various transformations, lightly formatted. The ability to easily inspect the results at each step is an advantage of the nanopass approach.

By language `L8`, (see second version in Figure 5) earlier passes have performed name resolution and renamed lexical variables, so the names of all types and functions are fully qualified (shown in here with ad hoc syntax not in `L0`; the `""` package is the "universe block" in Go, and contains all predeclared identifiers). Lexically-bound names are made unique, using the `$` notation to attach unique numbers.

The first step in lowering complex numbers occurs in `L9` where we lower imaginary number literals to calls to the `complex` function (line 2 of `L9` in Figure 5). In `L11` we have recognized calls to polymorphic standard library functions, including the `complex`, `real` (access of the real component), and `imag` (accessing the imaginary component) functions, and given them their own productions, of the same names, in the language's abstract syntax. This is indicated here by bolding the constructs' names on lines 2–3. In Go 1.17, users cannot define polymorphic functions, and polymorphic functions behave unlike other variables, so it makes type-checking simpler to recognize them as productions.

In `L12` type-checking is done. The specification of `L11` adds a synthesized attribute occurrence for types, `ty: Type`, to `Expr` and the corresponding equations are then also included. Equations for type-checking complex number constructs, such as the recently added `complex` production, are also included so that these expressions are appropriately typed.

Figure 6 defines `L12` and converts the `L11` attribute `ty` into an annotation (line 4) so that typing information is retained (in `toL12`) for use in the remaining passes. As of `L14`, the `Expr` nonterminal contains the productions `complex`, `real`, `imag`, as well as the arithmetic operators that are defined on complex numbers. After `L15`, a type annotation with its `isComplex64` attribute set to true indicates an operation over complex values.

Language `L0`:

```
1  func f(x complex64) float32 {
2      y := x + 1.2i
3      return real(y) - imag(y)
4  }
```

Language `L8`:

```
1  func f(x$0 "".complex64) "".float32 {
2      y$1 := x$0 + 1.2i
3      return "".real(y$1) - "".imag(y$1)
4  }
```

Language `L9`:

```
1  func f(x$0 "".complex64) "".float32 {
2      y$1 := x$0 + "".complex(0.0, 1.2)
3      return "".real(y$1) - "".imag(y$1)
4  }
```

Language `L11`:

```
1  func f(x$0 "".complex64) "".float32 {
2      y$1 := x$0 + complex(0.0, 1.2)
3      return real(y$1) - imag(y$1)
4  }
```

Language `L15`:

```
1  func f(x$0 struct{ r "".float32,
2                     i "".float32 })
3       "".float32 {
4      var y$1 struct{ r "".float32,
5                      i "".float32 }
6      y$1 = "$builtins".AddComplex64(
7        x$0,
8        struct{ r "".float32,
9                i "".float32 }{ 0.0, 1.2 })
10     return y$1.r - y$1.i
11 }
```

Figure 5. An example function, lowered from L_0 to L_{15}.

We transform away language-level support for complex numbers when transforming from `L14` to `L15` in `toL15`, also shown in Figure 6. To get started, `L14` inherits the string attribute `complexOpName` from `L12` (line 1), to occur on binary operators (line 6) to be the name of the built-in function corresponding to the operator (line 9) to which the operator will translate. (This is the empty string for operators that don't apply to complex numbers.) Though not shown, it also inherits an attribute `isComplex64` (line 2 on type nonterminals (line 5) to indicate if a `Type` is a complex type.

The target language, `L15` on line 12, eliminates the complex number productions (line 13) and the attributes used in `L14` (lines 14–15) as part of `toL15` (line 16).

One part of toL15 translates binary operators (lines 18–20) to a call to a runtime function (line 22) whose name is based on the complex operator name (line 25) if the type of the operator is complex (line 21). The operands are also lowered (lines 27–28). The type annotation ty is set to be the lowered version of the complex type (line 30). This transformation of complex types to struct types is not shown; it produces a struct with a real and imaginary field of type float.

If the type is not complex then the transformation is applied to the child expressions and annotations (lines 32–34).

Lastly, the complex productions complex (line 38), real (line 48) , and imag (not shown), are also lowered to, respectively, struct literals (lines 39–45, or the struct selector operator (lines 49–50).

Thus, we see, that over a handful of passes, complex numbers are lowered into structure types and expressions.

4.3 L31: Nearly assembly language

After all passes, the final target language L31 is simple enough to generate assembly straightforwardly in a single pass, although register allocation has not been done. The declaration nonterminal has productions for functions and methods with stack frame layouts, opaque type definitions, and global variables without initializers. The statement nonterminal has productions for sequencing statements, for conditional and unconditional gotos, assignments (to variables or struct fields), and expressions evaluated for effect. The expression nonterminal has no productions for nested expressions, only those with atomic operands: literals, references to variables (including functions), and references to methods. The remaining productions are for those atomic operands, function calls, closure creation, casting between types, calls to the memory allocator, and references to struct fields.

5 Related Work

Naturally there are many extensions to Knuth's original attribute grammars that are related to this work. Transform attributes in Section 3 are essentially a version of higher-order attributes [25] that are constrained to have the same nonterminal type as the nonterminal on which they occur. This restriction allows for the automatic generation of equations for productions that do not define them explicitly. Reference [5] and remote [1] attributes allow trees already decorated with attribute values (or references to them) to be passed as attributes. A common use case is to link uses of a variable back to the tree that declared it. In some sense, trees with attributes converted to annotations by a transform attribute are similar in that they come with values already decorating them. Trees generated in transform attributes are more restricted, however; we only use them to construct the tree of the program output from the transformation.

```
 1  syn complexOpName: string;
 2  syn isComplex64: bool;
 3  lang L12 extends L11 {
 4    Expr.annots += { ty },
 5    Type.attrs  += { isComplex64 },
 6    BinOp.attrs += { complexOpName }, ... }
 7
 8  aspect add {      // prod add(): BinOp;
 9    this.complexOpName := "ComplexAdd";
10  } // similarly for all BinOp productions
11
12  lang L15 extends L14 {
13    Expr.prods  -= { complex, real, imag },
14    Type.attrs  -= { isComplex64 },
15    BinOp.attrs -= { complexOpName } }
16  transform attribute toL15 from L14 to L15;
17
18  // prod binOpExpr(lhs: Expr, op: BinOp,
19  //                rhs: Expr): Expr;
20  aspect binOpExpr { this.toL15 =
21      if this.ty.isComplex64 then
22        callExpr(
23          varExpr(
24            qname("$runtime",
25                  op.complexOpName),
26            ty=c64BinopType),
27          exprsCons(lhs.toL15,
28          exprsCons(rhs.toL15,
29          exprsNil())),
30          ty=this.ty.toL15)
31      else
32        binOpExpr(lhs.toL15, op.toL15,
33                  rhs.toL15,
34                  ty=this.ty.toL15);
35  }
36
37  // prod complex(r: Expr, i: Expr): Expr;
38  aspect complex { this.toL15 =
39      compositeExpr(c64Type,
40        elementsCons(fieldKey("r"),
41                     r.toL15,
42        elementsCons(fieldKey("i"),
43                     i.toL15,
44        elementsNil())),
45        ty=c64Type);   }
46
47  // prod real(c: Expr): Expr;
48  aspect real { this.toL15 =
49      selectorExpr(c.toL15, "r",
50                   ty=this.ty.toL15);  }
```

Figure 6. Lowering complex numbers in struct types and expressions.

One mechanism for transforming trees is the tree-rewriting mechanism [20] in JastAdd [2] that rewrites trees to eliminate certain syntactic forms. Another is forwarding [24] in Silver [23] used for similar purposes. These differ from transform attributes in that both of these processes are more local in nature and not used for transforming an entire program. Silver also has strategy attributes [11], in which strategies control the application of rewrite rules to transform trees in a more global manner, but the differentiation of different languages is not possible there. Perhaps most similar to transform attributes are attribute coupled grammars [4]. These were a precursor to higher order attributes and were used to link attribute grammars in a sequence to translate a tree through a series of different languages. This is essentially what transform attribute do, but again the differentiation of many languages from a common collection of language elements is not present in this formalism.

The LISA system previously explored [14] splitting an attribute grammar specification into separate languages, which are defined using an object-oriented-inspired inheritance mechanism similar to our framework's extension mechanism described in Section 3.3. In fact our prototype uses the same `extends` keyword as LISA. An important difference is that LISA could not identify and use two distinct languages in the same phase of evaluation. This is needed in computations carried out during our language transformations to isolate the safe construction of some trees in the source language and others in the target language of the transformation.

The other primary body of related work is that of nanopass compilers. The original design and implementation of nanopass compilers was done for the Scheme language for both educational [18] and industrial applications [9]. This work articulated the software engineering benefits of the approach, such as greater transparency into the working of the compiler and more direct means for testing. This is certainly useful in educational settings and Jeremy Siek's new textbook adopts this approach [19]. In that work, programs are represented as Scheme data structures that are essentially syntax trees. These lack the ability to structure computations over the tree as flexibly as attribute grammars allow. This concern was noted in the GitHub repository for the Scheme-to-C compiler.[3]

6 Discussion and Conclusion

A concern one might have about nanopass attribute grammars used production-grade compilers is runtime performance. Are the many small-scale passes much slower than a few larger-scale ones? We have not performed a rigorous evaluation of the overall performance; however, Keep and Dybvig note [9] that after rewriting the Chez Scheme compiler to use the nanopass framework, compile times remained within a factor of two despite improvements to code generation, including a slower register allocator.

Another possible concern is the traditional, and somewhat conservative, well-definedness analysis used in our formulation in Section 3. It requires equations for all synthesized attributes and inherited attributes that occur on nonterminals in a production's signature. This is overly conservative, as some equations may be written that are never actually demanded. More sophisticated analyses have been developed, such as one by Kaminski and Van Wyk [6] that checks for *effective completeness*; that is, all potentially demanded attributes have an equation. Performing this analysis and a traditional higher-order circularity analysis [25] for each language in $\mathcal{L}$ would be straightforward.

There are a number of aspects of future work that we are currently investigating. The first is the further development of the prototype nanopass AG system to make it more robust and extend it with more modern attribute grammar features such as some of those described in Sections 2 and 5. This will allow us to do a more complete evaluation of nanopass attributes grammars by applying it to more language. The current prototype is contains the features needed to experiment with the nanopass formalism described in Section 3 but it lacks many of the modern AG features that improve the usability and convenience of the paradigm.

Although, as described above, we have reason to believe that performance is not a significant problem there is one optimization that is appealing — fusing several independent passes into one to avoid an additional traversal over the tree.

Another potential extension is to specify a target language for all attributes, rather than just transform attributes. This would enable a notion of reference attributes using annotations, by defining the language of, *e.g.*, an environment attribute mapping names to definitions to have some attributes on definitions present as annotations. This may complicate language checking, as the source language of an attribute could no longer be determined from its target language.

To conclude, we have introduced nanopass attribute grammars, a formalization of their specification, and a prototype system used to define many aspects of a compiler for the Go programming language. The distinguishing feature of nanopass attribute grammars is the clear identification of many distinct, yet similar, languages drawn from the same set of language elements. This provides the linchpin on which the static *language checking* depends so that the attribute grammars for individual languages can be shown to be well-defined even when the entire collection of language elements in $\mathcal{E}$ will most likely not be. Perhaps equally important is the clarity of thought that this style of compiler design brings: one can think in terms of clearly defined languages, knowing what has, and has not, been translated away or had its structure change in some way. In a large complex software artifact such as a compiler this is a considerable benefit.

[3] https://github.com/akeep/scheme-to-c/blob/18f6cd26f/c.ss#L2576-L2578

A The passes in the Go nanopass compiler

The Go compiler we have designed consists of 32 different languages, `L0` to `L31` and 34 passes. Three passes, `renameVariables`, `makeDerefExplicit`, and `lowerSelectorMethodCalls`, have the same source and target language.

The last language has an attribute that outputs x86_64 assembly. Since there is no register allocator, this assembly is inefficient, but it is runnable. We have also not implemented the runtime support needed for concurrency.

1. `giveImportsNames`, `L0` to `L1` – Rewrites imports like `import "example.com/foo"` to `import bar "example.com/foo"`. The latter form already exists in the CST, and this removes a special case in the next pass.
2. `fullyQualifyNames`, `L1` to `L2` – Rewrites variables that refer to imported declarations to refer to the package directly. For example, `Println` might become `"fmt".Println`, `foo.Bar` might become `"example.com/foo".Bar`, and `int32` might become `"".int32`.
3. `renameVariables`, `L2` to `L2` – Renames lexical variables to have globally unique names, so that variable shadowing doesn't become an issue. For example, `x` might become `x$45`.
4. `liftTypesAndConstants`, `L2` to `L3` – Lifts declarations of types and constants in local scopes to the global scope, and renames references to them to fit. (Due to `renameVariables`, we know there will not be any name conflicts.)
5. `expandTypeAliases`, `L3` to `L4` – Expands and removes type aliases. Note that this only applies to declarations like `type a = b`, not `type a b`.
6. `expandLists`, `L4` to `L5` – Rewrites function parameters, struct fields, etc. like `func foo(a, b "".int)` to `func foo(a "".int, b "".int)`.
7. `lowerIncDec`, `L5` to `L6` – Lowers increment and decrement statements to the corresponding assignment statements.
8. `labelLoops`, `L6` to `L7` – Adds labels to loops that lack them, and makes `break` and `continue` statements explicitly refer to their loop.
9. `normalizeInterfaces`, `L7` to `L8` – Sorts the methods of interface types to be in lexicographic order, and resolves any interface inclusions. This is useful for type-checking later.
10. `lowerImaginaryLits`, `L8` to `L9` – Lowers imaginary literals to calls to the `complex` function with a zero real part.
11. `recognizeMakeAndNew`, `L9` to `L10` – Recognizes the `make` and `new` constructs, and provides errors for uses of types as function call arguments other than those constructs.
12. `recognizePolyBuiltins`, `L10` to `L11` – Recognizes the other polymorphic built-in functions.
13. `typeCheck`, `L11` to `L12` – Adds a type annotation to expressions.
14. `makeDerefExplicit`, `L12` to `L12` – Adds uses of the dereference operator that were implicit in the source language. For example, `foo$7.Bar()` might be rewritten to `(*foo$7).Bar()`.
15. `lowerSelectorMethodCalls`, `L12` to `L12` – Lowers method calls that can be statically dispatched to direct calls, and references to those methods to lambdas. For example, `x$3.Foo(n$2)` might be rewritten to `("foo".MyStruct).Foo(x$3, n$2)`, and `y$4.Foo` might be rewritten to `func(n$1321 "".int) { ("foo".MyStruct).Foo(y$4, n$1321) }`.
16. `hoistVariableDecls`, `L12` to `L13` – Lifts variable declarations to the start of their nearest enclosing function.
17. `removeIfPreStmt`, `L13` to `L14` – Removes the "pre statement" from `if` statements. For example, rewrites `if n, err = "foo".bar(); n != nil {}` to `n, err = "foo".bar(); if n != nil {}`.
18. `lowerComplex` (`toL15`), `L14` to `L15` – Lowers calls to the complex-number-related built-in functions, and operators on complex numbers, to calls to runtime functions.
19. `lowerForLoops` (`toL16`), `L15` to `L16` – Lowers for-style loops to while-style loops.
20. `lowerForRangeLoops`, `L16` to `L17` – Lowers loops using the `range` construct to use indices on arrays, slices, and strings, and runtime functions on channels and maps.
21. `lowerIfThen`, `L17` to `L18` – Lowers if-thens without an else to if-then-elses.
22. `lowerSelect`, `L18` to `L19` – Lowers `select` statements to calls to a runtime function.
23. `lowerTypeSwitch`, `L19` to `L20` – Lowers `switch` statements on the runtime type of a value to a series of `if`s.
24. `lowerExprSwitch`, `L20` to `L21` – Lowers other `switch` statements to a series of `if`s, including their `fallthrough` statements.
25. `lowerControlFlow`, `L21` to `L22` – Lowers the remaining control-flow constructs (`break`, `continue`, `for`, `if`, `return`) to gotos and labels.
26. `lowerDefer`, `L22` to `L23` – Lowers the `defer` statement to a shadow stack and adds code to the exit blocks of functions to support `defer`.
27. `liftInitFunction`, `L23` to `L24` – Recognizes the definition of `init` functions and lifts their bodies to the `Package` nonterminal.
28. `inlineConstants`, `L24` to `L25` – Inlines references to `consts` and removes the `consts`' declarations.
29. `liftInitializers`, `L25` to `L26` – Lifts initializers for global variables and constants to the `init` function.

30. `flattenExprs`, `L26` to `L27` – Lowers complex expressions to simple ones; *e.g.* `"foo".f("foo".g())` might get lowered to `tmp$1322 = "foo".g(); "foo".f(tmp$1322)`.
31. `lowerCompositeExprs`, `L27` to `L28` – Lowers assignments of composite expressions to a series of assignments. For example, `x$22 = "foo".MyStruct{1,2}` might get lowered to `x$22.Foo = 1; x$22.Bar = 2`, and `xs$23 = []"".int{1,2,3}` might get lowered to `xs$23 = make([]"".int, 3); xs$23[0] = 1; ...`, and so on.
32. `lowerPolyBuiltins`, `L28` to `L29` – Transforms calls to the polymorphic built-in functions recognized by `recognizePolyBuiltins` to calls to runtime functions.
33. `layoutStackFrames`, `L29` to `L30` – Places the local variables declared in each function and method into a single struct, such that each function has exactly one local variable. A static link is also present, containing a pointer to the parent's stack frame struct, for lambdas' stack frames.
34. `lambdaLift`, `L30` to `L31` – Lifts lambda expressions to global functions that take an additional argument for their parent's stack frame and a call to a built-in function that accepts the global function pointer and the parent's stack frame pointer and constructs the closure. Globally-defined functions and methods also get a globally-defined "closure" (that ignores the parent's stack frame). References to globally defined functions and methods are changed to refer to these closure objects instead.

References

[1] John Tang Boyland. 2005. Remote attribute grammars. *J. ACM* 52, 4 (2005), 627–687. https://doi.org/10.1145/1082036.1082042

[2] Torbjörn Ekman and Görel Hedin. 2007. The JastAdd extensible Java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications (OOPSLA)* (Montreal, Quebec, Canada). ACM, 1–18. https://doi.org/10.1145/1297027.1297029

[3] R. Farrow. 1986. Automatic Generation of Fixed-Point-Finding Evaluators for Circular, but Well-Defined, Attribute Grammars. *ACM SIGPLAN Notices* 21, 7 (1986). https://doi.org/10.1145/13310.13320

[4] H. Ganzinger and R. Giegerich. 1984. Attribute coupled grammars. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*. 157–170. https://doi.org/10.1145/502874.502890

[5] Görel Hedin. 2000. Reference Attribute Grammars. *Informatica* 24, 3 (2000), 301–317.

[6] Ted Kaminski and Eric Van Wyk. 2012. Modular well-definedness analysis for attribute grammars. In *Proceedings of the 5th International Conference on Software Language Engineering (SLE) (Lecture Notes in Computer Science, Vol. 7745)*. Springer, 352–371. https://doi.org/10.1007/978-3-642-36089-3_20

[7] Uwe Kastens. 1980. Ordered attributed grammars. *Acta Informatica* 13 (1980), 229–256. Issue 3. https://doi.org/10.1007/BF00288644

[8] U. Kastens and W. M. Waite. 1994. Modularity and reusability in attribute grammars. *Acta Informatica* 31 (1994), 601–627. https://doi.org/10.1007/BF01177548

[9] Andrew W. Keep and R. Kent Dybvig. 2013. A Nanopass Framework for Commercial Compiler Development. *SIGPLAN Not.* 48, 9 (sep 2013), 343–350. https://doi.org/10.1145/2544174.2500618

[10] Donald E. Knuth. 1968. Semantics of Context-free Languages. *Mathematical Systems Theory* 2, 2 (1968), 127–145. https://doi.org/10.1007/BF01692511 Corrections in **5**(1971) pp. 95–96.

[11] Lucas Kramer and Eric Van Wyk. 2020. Strategic Tree Rewriting in Attribute Grammars. In *Proceedings of the ACM SIGPLAN International Conference on Software Language Engineering (SLE)* (Virtual, USA). 210–229. https://doi.org/10.1145/3426425.3426943

[12] José Nuno Macedo, Marcos Viera, and João Saraiva. 2022. Zipping Strategies and Attribute Grammars. In *Functional and Logic Programming (Lecture Notes in Computer Science, Vol. 13215)*. Springer, 112–132. https://doi.org/10.1007/978-3-030-99461-7_7

[13] Eva Magnusson and Görel Hedin. 2007. Circular reference attributed grammars - their evaluation and applications. *Science of Computer Programming* 68, 1 (2007), 21–37. https://doi.org/10.1016/j.scico.2005.06.005 Special Issue on the ETAPS 2003 Workshop on Language Descriptions, Tools and Applications (LDTA '03).

[14] M. Mernik, M. Lenič, E. Avdičaušević, and V. Žumer. 1998. The template and multiple inheritance approach into attribute grammars. In *Proceedings of the 1998 International Conference on Computer Languages*. 102–110. https://doi.org/10.1109/ICCL.1998.674161

[15] Jukka Paakki. 1995. Attribute Grammar Paradigms—a High-Level Methodology in Language Implementation. *Comput. Surveys* 27, 2 (June 1995), 196–255. https://doi.org/10.1145/210376.197409

[16] Michael Rodeh and Mooly Sagiv. 1999. Finding Circular Attributes in Attribute Grammars. *J. ACM* 46, 4 (July 1999), 556–ff. https://doi.org/10.1145/320211.320243

[17] Joao Saraiva. 1999. *Purely Functional Implementations of Attribute Grammars*. Ph. D. Dissertation. University of Utrecht.

[18] Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. 2004. A Nanopass Infrastructure for Compiler Education. *SIGPLAN Not.* 39, 9 (sep 2004), 201–212. https://doi.org/10.1145/1016848.1016878

[19] Jeremy G. Siek. 2023. *Essentials of Compilation: An Incremental Approach in Racket*. The MIT Press.

[20] Emma Söderberg and Görel Hedin. 2015. Declarative rewriting through circular nonterminal attributes. *Computer Languages, Systems & Structures* 44 (2015), 3 – 23. https://doi.org/10.1016/j.cl.2015.08.008 Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014).

[21] S. D. Swierstra and P. R. Azero. 1998. *Attribute grammars in the functional style*. Springer US, Boston, MA, 180–193. https://doi.org/10.1007/978-0-387-35350-0_14

[22] L. Thomas van Binsbergen, Jeroen Bransen, and Atze Dijkstra. 2015. Linearly Ordered Attribute Grammars: With Automatic Augmenting Dependency Selection. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation* (Mumbai, India) *(PEPM '15)*. ACM, 49–60. https://doi.org/10.1145/2678015.2682543

[23] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. 2010. Silver: an Extensible Attribute Grammar System. *Science of Computer Programming* 75, 1–2 (January 2010), 39–54. https://doi.org/10.1016/j.scico.2009.07.004

[24] Eric Van Wyk, Oege de Moor, Kevin Backhouse, and Paul Kwiatkowski. 2002. Forwarding in Attribute Grammars for Modular Language Design. In *Proceedings of the 11th Conference on Compiler Construction (CC) (Lecture Notes in Computer Science, Vol. 2304)*. Springer-Verlag, 128–142. https://doi.org/10.1007/3-540-45937-5_11

[25] Harold H. Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. 1989. Higher Order Attribute Grammars. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 131–145. https://doi.org/10.1145/73141.74830

Received 2023-07-07; accepted 2023-09-01

Automated Extraction of Grammar Optimization Rule Configurations for Metamodel-Grammar Co-evolution

Weixing Zhang
weixing@chalmers.se
Chalmers | University of Gothenburg
Gothenburg, Sweden

Regina Hebig
regina.hebig@uni-rostock.de
University of Rostock
Rostock, Germany

Daniel Strüber
danstru@chalmers.se
Chalmers | Gothenburg University, Gothenburg, Sweden
Radboud University, Nijmegen, Netherlands

Jan-Philipp Steghöfer
jan-philipp.steghoefer@xitaso.com
Xitaso IT and Software Solutions
Augsburg, Germany

Abstract

When a language evolves, meta-models and associated grammars need to be co-evolved to stay mutually consistent. Previous work has supported the automated migration of a grammar after changes of the meta-model to retain manual *optimizations* of the grammar, related to syntax aspects such as keywords, brackets, and component order. Yet, doing so required the manual specification of optimization rule configurations, which was laborious and error-prone.
In this work, to significantly reduce the manual effort during meta-model and grammar co-evolution, we present an automated approach for extracting optimization rule configurations. The inferred configurations can be used to automatically replay optimizations on later versions of the grammar, thus leading to a fully automated migration process for the supported types of changes. We evaluated our approach on six real cases. Full automation was possible for three of them, with agreement rates between ground truth and inferred grammar between 88% and 67% for the remaining ones.

CCS Concepts: • **Software and its engineering** → **Software evolution**; **Model-driven software engineering**.

Keywords: meta-models, grammars, co-evolution

ACM Reference Format:
Weixing Zhang, Regina Hebig, Daniel Strüber, and Jan-Philipp Steghöfer. 2023. Automated Extraction of Grammar Optimization Rule Configurations for Metamodel-Grammar Co-evolution. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3623476.3623525

SLE '23, October 23–24, 2023, Cascais, Portugal
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0396-6/23/10.
https://doi.org/10.1145/3623476.3623525

1 Introduction

Model-driven engineering is an important software engineering paradigm, in which models are considered as primary artifacts during software development [46]. Managing the consistency of artifacts produced during model-driven engineering is a hard problem. When a meta-model is updated, associated artifacts that still refer to the old version of the meta-model, such as model instances [23], model transformations [30], and code generators [37], become outdated and need to be migrated. The overall class of problems addressed here is referred to as *co-evolution* [23] or *coupled evolution* [3] and, due to its practical significance, has led to a large body of work, focusing on automated migration support (see, e.g., [6, 9, 19, 23, 27, 28, 39, 44, 44, 45]).

We consider a scenario in which a meta-model is co-evolved with an associated grammar. Such a scenario is common in cases where the meta-model defines the underlying abstract syntax for a modeling language, and the grammar defines a concrete textual syntax for that language [34]. In the technical space of Eclipse, the meta-model and grammar could be specified using Ecore and Xtext, respectively. In this scenario, there are two situations that lead to co-evolution: First, the meta-model evolves over time, rendering previous versions of the grammar obsolete. Second, in a rapid prototyping context, the meta-model evolves quickly and then requires the grammar to be updated quickly as well.

The main challenge with this scenario stems from two core requirements that typically need to be addressed:

- The updated grammar should be consistent with the new version of the meta-model.
- The updated grammar should incorporate any manual improvements that were made to previous versions of the grammar (e.g., adding and modifying keywords, changing the order of rule components, modifying and omitting brackets).

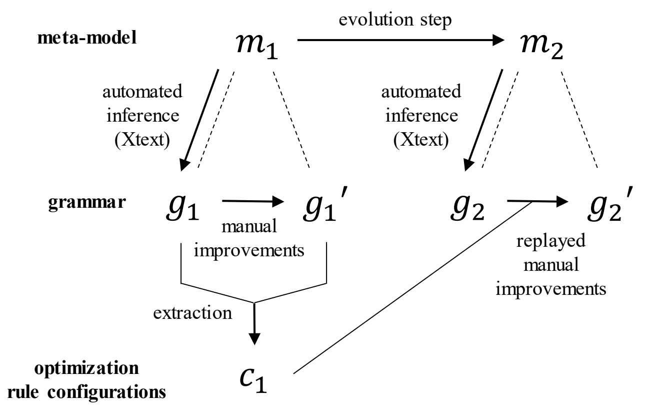

meta-model
grammar
optimization rule configurations
m_1
m_2
evolution step
automated inference (Xtext)
automated inference (Xtext)
g_1
g_1'
g_2
g_2'
manual improvements
replayed manual improvements
extraction
c_1

Figure 1. Overview of meta-model/grammar co-evolution; dashed lines indicate mutual consistency. Our contribution is to automate the extraction of c_1, previously done manually.

A tempting solution that addresses the first requirement is to automatically re-generate the entire grammar from the evolved meta-model, which is supported by platforms such as Xtext (in Fig. 1, arrow between m_1 and g_1). Yet, the second requirement renders this solution insufficient, as it leads to a laborious process, in which developers need to manually re-apply their optimizations to the re-generated grammar in every evolution step. For languages with extensive grammars like EAST-ADL, which encompasses approximately 300 grammar rules [12], this process simply seems infeasible when done manually.

We are aware of only one previous work that addresses this problem, a tool called *GrammarOptimizer* [58]. The key idea of GrammarOptimizer was to provide a catalog of grammar optimization rules that can be used to specify and then automatically perform the required changes for moving from a generated grammar to an optimized one. For example, in Fig. 1, the developers use these rules to specify a configuration c_1 that captures the improvements for moving from g_1 to g_1'. We will introduce this tool further in Sect. 2. Yet, the tool in its current version has a major drawback: manually instantiating the grammar optimization rules to specify migrations can be cumbersome and error-prone. Specifying the right configuration for the task at hand involves choosing the right set of rules together with correct values for parameters. This is a complicated configuration process.

In this paper, to considerably reduce this manual specification effort, we present an approach for automating the configuration of grammar optimization rules. Our approach assumes that two versions of the grammar are available: an automatically generated one and a manually optimized one; we call the latter the *target grammar*. In a co-evolution scenario, these grammars come from a previous state in history, and the manual optimizations evident in the target grammar should inform the migration of the grammar towards a new version of the meta-model (in Fig. 1, consider the generated grammar g_1, the target grammar g_1' and the new meta-model version m_2). Our approach can then automatically extract an *optimization rule configuration* that encodes the manual improvements. Technically, our approach works by establishing a mapping between the grammar rules from both grammars and then, per rule, performing a line-by-line comparison to extract invocations of relevant grammar optimization rules with their parametrizations. We automated this process by developing a tool named *ConfigGenerator*.

The extracted configuration can be applied to a newly-generated version of the grammar based on the evolved meta-model. For changes of the types supported by our approach, this entirely avoids any manual effort for specifying and re-applying the manual optimizations. In Fig. 1, after the evolution step that created m_2, replaying the changes between g_1 and g_1' on the generated grammar g_2 using the automatically extracted configuration c_1 leads to the target grammar g_2'. Once the target grammar is available, new and changed meta-model elements may lead to new manual optimizations on top of it. In that case, our approach can be applied after meta-model changes to capture these changes in a new version of the configuration. That way, our approach provides support for meta-model/grammar co-evolution throughout the history of an evolving language.

To evaluate our approach, we applied the *ConfigGenerator* to six cases of languages whose meta-models and grammars are available: EAST-ADL, Bibtex, Xenia, Xcore, DOT, and SML. The results show that our approach is able to extract complete configurations for three of the cases (EAST-ADL, Bibtex, and Xenia). For these languages, the target grammars yielded by replaying the optimizations are identical with an existing ground truth grammar. For the other three languages, the optimization rates — defined as the agreement between a ground truth grammar and the grammar obtained by replaying — are between 87.5% and 68%. These findings indicate the potential and effectiveness of ConfigGenerator in extracting optimization rules based on the comparison between generated grammar and target grammar.

2 Background

2.1 Xtext and DSL Generation

Eclipse Xtext is a framework for developing software languages, including modeling languages [15]. Xtext offers two approaches for implementing the grammar design of a textual DSL [42]. One approach involves creating an Ecore meta-model to represent domain concepts and their relationships, and then generating an Xtext grammar from the meta-model (in the remainder of the paper, we call it *generated grammar*). The other approach is first to create a grammar and then derive a meta-model from it. The scope addressed in this paper involves the former approach.

Listing 1. Example from Xenia: generated grammar rule `SiteWithModal`.

```
SiteWithModal returns SiteWithModal:
 {SiteWithModal}
 'SiteWithModal'
 name=EString
 '{'
  ('sites' '{' sites+=SuperSite ( "," sites+=
       SuperSite)* '}' )?
 '}';
```

In Xtext, grammars are specified in an EBNF (Extended Backus-Naur Form) format, augmented with references and annotations that specify the relationship to the Ecore meta-model. The meta-model represents the abstract syntax for language at hand (classes with their features, including names, attributes, and references), while the augmented EBNF expression describes the concrete syntax and its mapping to specific parts of the meta-model. Listing 1 shows an example of a grammar rule in Xtext, from the context of Xenia [54, 55], one of our evaluation cases. The depicted rule, `SiteWithModel`, contains both traditional EBNF elements for specifying the syntax, as well as several annotations and references. In particular, the `returns` keyword is followed by a reference to the `SiteWithModel` class, and several grammar elements are mapped to attributes (`name`) and references (`sites`) from that class, using the '=' and '+=' operands. By defining grammar rules and associating them with the corresponding meta-model elements, Xtext enables the automatic generation of a parser and other language tools. The parser uses the grammar rules to parse the input code and create an abstract syntax tree (AST) that conforms to the meta-model elements. This AST can then be further processed or used for various purposes in language development.

2.2 GrammarOptimizer and Optimization Rules

We now provide additional details for the GrammarOptimizer tool [58] that, in particular, provides the grammar optimization rules we automatically configure with our approach. Their approach includes 54 optimization rules extracted from seven sample languages, which are used to optimize the generated grammar (explained above). These optimization rules operate on various elements within the grammar, including keywords, curly braces, symbols, and optionality. For example, `AddKeywordToAttr` is used to add a new keyword to a specific attribute, `ChangeBracesToSquare` is used to transform specified curly braces into square brackets, and `RemoveRule` is used to remove unnecessary grammar rules. Their tool is an Eclipse plugin developed in Java.

To use GrammarOptimizer, language engineers need to manually select and configure the optimization rules for performing the intended changes. Given a selected rule, configuring it involves invoking methods of a Java class representing the application of that rule, with parameters such as the name of the relevant grammar rule and involved elements, such as attribute names and keywords. These parameters enable GrammarOptimizer to accurately locate the specific targets in the generated grammar that need to be modified.

Listing 2. Example from Xenia: target grammar rule `SiteWithModal`; all attributes and keywords are now on the same line.

```
SiteWithModal:
  '@' name=ID 'with' 'modal' '(' sites+=
       SuperSite (',' sites+=SuperSite)* ')'
 ;
```

As an example, consider Listings 1 and 2. Listing 1 shows the grammar rule `SiteWithModal` from Xenia's generated grammar, while Listing 2 shows the version of that rule in the target grammar. We focus on the `name` attribute, which has different types in the two grammars: `EString` and `ID` in the generated and target grammar, respectively. While editing the generated grammar manually to change the type is simple, this change cannot be recovered if the grammar is re-generated after a meta-model change, unless dedicated support is provided.

Hence, the language engineer uses GrammarOptimizer. Doing so involves identifying the relevant optimization rule, in this case, `changeTypeOfAttr`. To configure the optimization rule, the engineer instantiates the `GrammarOptimizer` class which acts as a facade and defines a public method for each of the optimization rules. The `changeTypeOfAttr` method accepts four parameters: the names of the grammar rule, of the attribute name, of the current type, and of the new type. In this example, where the instantiated `GrammarOptimizer` object is named go, the configuration of the optimization rule call to modify the type of `name` is as follows: `go.changeTypeOfAttr("SiteWithModal", "name", "EString", "ID")`.

3 Related Work

Recovery of grammars and meta-models. In legacy systems, a common situation is that the underlying grammar or meta-model is absent and has to be recovered from available instances (programs or models, respectively). Available solutions are based on using available compiler sources and language reference manuals [35], evolutionary computing [4, 25], or iteratively provided user input [38]. With a focus on supporting grammar recovery scenarios, Lämmel [32] provides a set of operators for grammar modification, focusing on refactoring, construction, and destruction. Yet, recovery approaches such as those discussed are not applicable to the scenario considered in this paper, in which the grammar instances, after an evolution step, still conform to the old, known grammar.

Co-evolution in MDE contexts. In model-driven engineering, it is well-known that evolutionary changes to an artifact

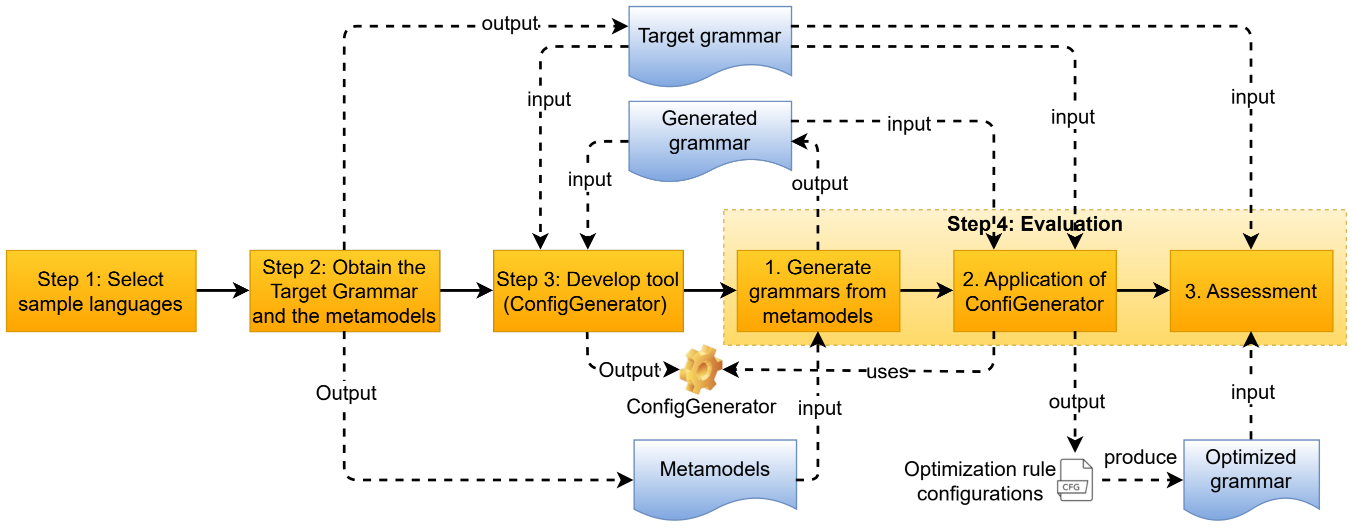

Target grammar
Generated grammar
output
input
input
input
input
input
output
Step 4: Evaluation
Step 1: Select sample languages
Step 2: Obtain the Target Grammar and the metamodels
Step 3: Develop tool (ConfigGenerator)
1. Generate grammars from metamodels
2. Application of ConfiGenerator
3. Assessment
Output
uses
ConfigGenerator
Output
input
output
input
Metamodels
Optimization rule configurations
produce
Optimized grammar

Figure 2. Schematic diagram of the whole process of the research methodology.

may affect other artifacts, which leads to several co-evolution scenarios. The most prominent one is *meta-model/model* co-evolution, in which a meta-model is evolved and corresponding instances have to be updated to stay in sync with the meta-model. This scenario has inspired a substantial body of work. Hebig et al. [23] survey 31 relevant approaches, classifying them according to their support for change collection, change identification, and model resolution. Beyond meta-model/model co-evolution, co-evolution between meta-models and other MDE artifacts have received attention as well, including associated OCL constraints [29], model transformations [30, 31], code generators [37], and graphical editor models [10]. Inconsistencies between evolved meta-models and general MDE artifacts have also been addressed in the context of *technical debt management*, with an approach that assists the modeler with the aid of interactive visualization tools [8]. However, except for *GrammarOptimizer* [58] (described in Sect. 2), on which we build and improve with our contribution, we are not aware of previous work on meta-model/grammar co-evolution.

Model federation [11, 20, 22] deals with challenges of keeping several models synchronized, which is related to our addressed co-evolution scenario. However, to the best of our knowledge, there is no previous work that applies model federation techniques to grammars. Previous work is often focused on establishing links between the different involved artifacts, which, in our scenario, is a non-issue. However, the actual modification for keeping several artifacts synchronized is often simpler if only models are involved, than in our case that deals with concrete textual syntaxes. For example, the order of attributes in the grammar does not have to be consistent with the corresponding meta-model attributes but can be changed freely according to the developer's design intention. In fact, the approach enabled by our contribution could be used to augment available model federation frameworks to make them applicable to grammars as well.

Automated rule extraction. A line of work focuses on automating the extraction of transformation rules in specific contexts. Model transformation by example [51, 53] is an important paradigm in which entire transformations are recovered from a set of user-provided examples. While the seminal work in this area mostly relied on custom heuristics, recent works have studied applications of AI, in particular, reinforcement learning [17] and deep learning [2]. Apart from these approaches for general transformation inference, there are task-specific approaches, including the refactoring of redundant rules [49, 50] and of mutation operators [47]. These approaches are orthogonal to ours, as we focus on the automated extraction of *configurations* of rules.

From meta-models to graph grammars. Beyond EBNF-style grammars as considered in this paper, grammarware in the broader sense also encompasses graph grammars, which are a rule-based approach for generating instances for a given meta-model, e.g., for testing purposes. A seminal approach by Ehrig et al. [16] supports the generation of a graph grammar in the double-pushout approach to graph rewriting, using advanced transformation features such as negative application conditions. Fürst et al. present an approach that aims to avoid the use of such advanced features that make analysis more complicated, while being sufficient for meta-models with arbitrary multiplicities and inheritance [18].

Text-based merging. Simple cases of our considered scenario could be covered by standard text merging tools, such as Git merge [7]. To this end, the user would perform manual optimizations and re-generation of the grammar in separate branches, and then merge the branches. However, text-based merging operates on the abstraction level of text rather than

grammar structures, which leads to several drawbacks: First, it easily leads to merge conflicts. For example, when the same line is manually optimized (e.g., changing a keyword) and affected by a change in the underlying meta-model (e.g., removing an attribute), a merge conflict arises, whereas our approach supports this example. Second, it does not give an easily inspectable, semantically meaningful overview of the changes, as grammar optimization rules do. In that sense, our approach can be seen as a form of semantic lifting [26] of grammar differences, focused on grammar optimizations.
Grammar convergence. Our contribution bears a connection to grammar convergence [36]. Grammar convergence aims to extract a series of transformations to make two considered grammars syntactically identical, which is similar to our goal. Yet, the grammars in their approach stem from heterogeneous sources (e.g., different parsers for the C++ language), instead of being based on the same underlying source meta-model in several versions, which gives both approaches different knowledge to rely on. A relevant scenario is metalanguage evolution [56], in which the notation used to define the considered languages, instead of the languages themselves, evolves, which necessitates changes in associated artifacts (e.g., parsers). Another one is style normalization for X-to-O mappings, which aims to bridge heterogeneity in different XML different styles when supporting their mapping to object models [33].

4 Methodology

The research methodology in this study consists of the steps shown in Figure 2. The first two steps were performed to prepare the inputs for Step 3, in which we developed the *ConfigGenerator*, and for Step 4, in which we evaluated our approach. All steps are described in the following.

4.1 Step 1: Select Sample Languages

In the first step, we selected appropriate case languages. These chosen languages served as the foundation for our solution and evaluation. Since our goal was to make our approach applicable to real-world DSLs, we needed to select a set of real-world DSLs for which both a grammar and a meta-model were available. In our previous work [58], we identified 9 such DSLs through an extensive search. We decided to directly work with a subset of six of their considered languages–Bibtex, DOT, EAST-ADL (full version), SML, Xcore, and Xenia–, which has the following benefits: First, the considered languages covered a diverse range of domains. Second, we knew from their evaluation that GrammarOptimizer could be used to optimize grammars for these languages. Since GrammarOptimizer was a baseline tool for our approach, working with these languages ensured that any observed issues stem from our approach for automated configuration extraction, and not from our baseline tool. Our reason for selecting a subset was that four of their considered languages had complications that led to a lack of full support (e.g., using OCL as part of the grammar definition). We still included one of the not-fully-supported languages, SML, to study the effect of applying our approach to one case from that category.

4.2 Step 2: Obtain the Target Grammars and Meta-models

After selecting the case languages, we obtained their meta-models and target grammars. The information regarding the meta-models (source and the number of classes) and target grammars (source and the number of grammar rules) is presented in Table 1. We noticed that the meta-models for Bibtex and SML needed adjustments to be effectively used, which we completed in previous work [58]. Therefore, we directly adopted our prepared meta-models for Bibtex, SML, and DOT, and obtained the meta-models for the other three languages from their respective sources.

ConfigGenerator takes two Xtext grammars as input: the target grammar and the generated grammar (i.e., the grammar newly generated from the meta-model). We observed that EAST-ADL and Bibtex did not have original grammars in Xtext, so we directly adopted the optimized grammars from [58] as the target grammars for these two languages. As for the other four languages, their Xtext grammars were already provided in their respective sources, and we simply copied these Xtext grammars as the target grammars.

4.3 Step 3: Develop Tool

In the 3rd step we developed the ConfigGenerator. We developed the initial version of ConfigGenerator based on EAST-ADL. The development of ConfigGenerator involved the implementation of comparisons for different grammar elements. Each time we implemented a comparison method for a specific grammar element (e.g., comparing line orders), we applied it to compare two EAST-ADL grammars and check the selected optimization rules. If the selected optimization rules differed from our expectations, we used the debug mode to identify the reasons behind the differences and fixed them. Once we had the initial version of ConfigGenerator, we applied it to Xenia to refine its implementation. In this context, we considered that target grammars might contain manual modifications that could affect line-by-line matching. For example, in Xenia's target grammar, some different attributes are placed on the same line. This situation impacts our line recognition and then matching. Consequently, when applying ConfigGenerator to Xenia, we developed handling methods for this situation.

4.4 Step 4: Evaluation

To validate our approach, we applied it to all six languages identified in Step 1. Our goal was to explore whether and to what extent the ConfigGenerator built based on EAST-ADL and Xenia, could also be applied to other DSLs. In our

Table 1. DSLs used in this paper, the sources of the meta-model and the grammar used, as well as the size of the meta-model and grammar.

	Meta-model		Target grammar		Generated grammar			Used in[2]	
DSL	Source	Classes[1]	Source	Rules	lines	rules	calls	Dev.	Eva.
EAST-ADL	EATOP Repository [12]	291	[58]	297	2839	297	3062	YES	YES
BibTex	[58]	48	[58]	43	293	43	188	NO	YES
Xenia	Github Repository [54]	15	Github Repository [55]	13	84	15	36	YES	YES
DOT	[58]	19	Dot [40]	21	125	23	51	NO	YES
Xcore	Eclipse [13]	22	Eclipse [14]	26	243	33	149	NO	YES
SML[3]	[58]	48	SML repository [21]	45	658	96	377	NO	YES

[1] The metrics are assessed after adaptations and contain both classes and enumerations.
[2] These two metrics indicate whether the language is used in the step "Development (Dev.)" or "Evaluation (Eva.)".
[3] The metrics of SML are based on excluding the embedded SML expressions.

previous work [58], we had already shown that an optimization rule configuration created to optimize one version of a language could be reused, with a few changes, for another language version. Thus, we aimed at evaluating whether ConfigGenerator could create a correct optimization rule configuration for a language version given the generated and target grammar. To do so, we performed the following steps.

4.4.1 Step 4.1: Generate Grammars From Meta-models.
When the meta-model was ready, we created an empty EMF project for the language in Eclipse and imported its meta-model. Then, utilizing Xtext, we automatically generated the Xtext grammar from the meta-model. We performed this process for each language.

4.4.2 Step 4.2: Application of ConfigGenerator.
Next, we applied the ConfigGenerator to all of these languages by comparing the target grammar with the generated grammar and extracting the optimization rule configurations. These extracted optimization rule configurations can be used by GrammarOptimizer. We then used the created optimization rule configurations with the GrammarOptimizer on the generated grammars of these languages to automatically create an optimized grammar.

4.4.3 Step 4.3: Assessment.
We conducted a comprehensive comparison between the optimized grammar and the target grammar of each language, based on a one-to-one comparison of corresponding grammar rules.

To assess the similarity between the optimized grammar and the target grammar, in our final step we decided to assess the following metrics which will be listed in Table 2: To provide an impression of the amount of manual adaptation that needed to happen to change the generated grammar to the target grammar, the 4th to 6th columns indicate the number of grammar rules that were modified, removed, and added from the generated grammar to the target grammar. Further, we show how big optimization rule configurations for these languages are. The 2nd column shows the number of lines of optimization rule configurations used in our previous work [58], which are capable of optimizing the generated grammar to achieve an identical state as the target grammar. We referred to the optimization rule configurations used in our previous work [58] as "manual" configuration, since these configurations had to be written by hand. The 3rd column represents the number of lines of the optimization rule configurations extracted by ConfigGenerator. We also listed in columns 7 to 9 the number of grammar rules that were modified, removed, and added from the generated grammar to the optimized grammar. Finally, we aimed to assess how complete the generated optimization rule configurations are. Thus, the last three columns provide statistics on the comparison of the optimized and target grammar, i.e., whether all grammar rules for these languages are the same between the optimized grammar and the target grammar. Specifically, "Same" represents the number of grammar rules that are identical in both grammars, "Diff" represents grammar rules that are not identical, and "Percent" indicates the percentage of grammar rules that are identical between the two grammars.

5 Solution

In this section we present the ConfigGenerator, which creates an optimization rule configuration based on a generated grammar and a target grammar, to enable a re-application of manually defined grammar changes after a meta-model changed and a new grammar was generated. We first introduce and reason about the assumptions we made when building our solution. Afterward, we explain how grammars are compared (rule-to-rule and line-to-line) and how the comparison result is used for generating the configuration.

5.1 Assumptions

Based on the technical reality and practice of Xtext, we made the following assumptions about our solution:

- A grammar rule name is unique across the grammar. Otherwise, Eclipse will prompt "A rule's name has to be unique." error.
- An attribute name is unique within a grammar rule, because the attributes in the generated grammar are unique.
- Attribute names are not modified by users when they manually create a target grammar out of a generated grammar. Otherwise, this may cause the grammar and the meta-model to become incompatible.

5.2 Grammar Comparison Workflow and Grammar Rule Matching

ConfigGenerator selects and parameterizes optimization rules by comparing two input grammars. The selected optimization rules form a configuration that can then be utilized by GrammarOptimizer.

Figure 3 illustrates the internal workflow of ConfigGenerator for selecting the required optimization rules by comparing two grammars. It parses the generated grammar and creates a list Rules$_{gen}$ containing instances of a data structure for each grammar rule. Each instance contains all lines of text that make up that grammar rule. ConfigGenerator does the same for the target grammar to create a list Rules$_{tgt}$.

ConfigGenerator traverses Rules$_{gen}$, taking one grammar rule at a time and searching for the grammar rule with the same name in Rules$_{tgt}$. If no match is found, it indicates that the grammar rule has been deleted in the target grammar, thereby requiring the selection and parameterization of an optimization rule for deleting that grammar rule. If a match is found, a line-by-line comparison is performed between the grammar rules to identify the required optimization rules.

Once the entire traversal of Rules$_{gen}$ is completed, ConfigGenerator performs a reverse traversal. In this reverse traversal, ConfigGenerator retrieves one grammar rule at a time from Rules$_{tgt}$ and searches for the corresponding rule in Rules$_{gen}$. If a match is found, ConfigGenerator takes no action (as the comparison has already been done in the previous traversal). If no match is found, it signifies that the grammar rule is newly added in the target grammar. In this case, an optimization rule for adding the grammar rule is selected and parameterized.

After both traversals are completed, ConfigGenerator yields an optimization rule configuration with the selected and parameterized optimization rules and writes it into a text file.

5.3 Normalization of Rules$_{tgt}$

Before performing line-by-line mapping, we need to perform normalization checks and operations on the rules of the target grammar. Because the target grammar may have traces of manual modification that are not conducive to our line-by-line matching. For example, in the generated grammar of Xenia, each of the different attributes and also the `{SiteWithModal}` action, the opening brace, and the closing brace each has its exclusive line, as shown in Listing 1. However, in the target grammar of Xenia, all attributes and keywords of the grammar rule `SiteWithModal` are placed on the same line as shown in Listing 2. This situation hinders row-to-row matching and thus needs to be normalized.

In particular, we begin by examining whether the following situations exist within a grammar rule: 1) different attributes are placed on the same line, 2) an Action with the same name as the grammar rule is combined with any other non-empty string on the same line, and 3) symbols are placed on separate and exclusive lines. If any of these situations are present, normalization is performed. During the normalization process, we gather all lines except the one containing the grammar rule name into a single string, which is then split. Using regular expressions, we separate different attributes into distinct lines, ensuring that each attribute has its own line. Similarly, if there is an Action with the same name as the grammar rule, an opening brace, and a closing brace, we allocate separate lines exclusively for each of them. Additionally, all symbols are placed in adjacent attribute lines rather than being treated as separate lines themselves, e.g., place the symbol `':'` after the attribute `name`.

5.4 Line Matching

As described in Section 5.2, ConfigGenerator matches grammar rules by traversing two lists. After completing the matching of Rules$_{gen}$ and Rules$_{tgt}$, we need to match the lines between them, forming the foundation for line-to-line comparison. With the exception of attribute lines, all other lines have only one occurrence within the same grammar rule. Therefore, ConfigGenerator only needs to find the corresponding unique line to complete the line matching. For example, to compare the main keyword of the same grammar rule, we search for the main keyword in both the Rules$_{gen}$ and Rules$_{tgt}$. We call the keyword with the same name as the grammar rule in the generated grammar "main keyword". If both sides find a line containing the main keyword, then the two lines from both sides match each other.

For the matching of attribute lines, we need to recall the assumption set earlier, which states that each attribute within the same grammar rule has a unique name, and language engineers do not modify the names of attributes when modifying the grammar. Therefore, we can use the attribute name as a unique identifier for lines to perform line matching. Specifically, when matching a line, we first take an attribute line from a grammar rule in the generated grammar, and then search for the line with the same attribute name within the corresponding grammar rule in the target grammar, thus completing the line-matching process.

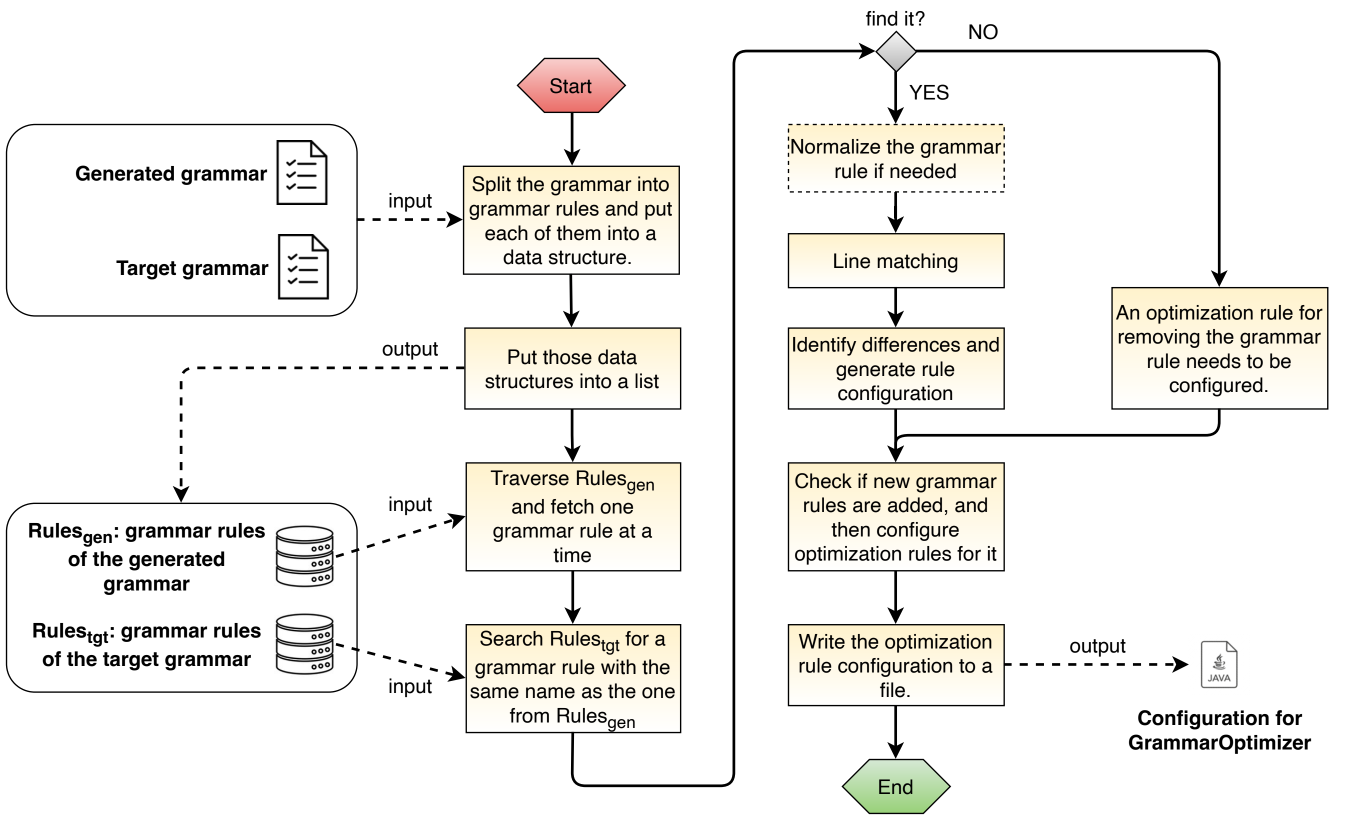

Start
Generated grammar
Target grammar
input
Split the grammar into grammar rules and put each of them into a data structure.
output
Put those data structures into a list
Traverse Rules$_{gen}$ and fetch one grammar rule at a time
Rules$_{gen}$: grammar rules of the generated grammar
Rules$_{tgt}$: grammar rules of the target grammar
input
input
Search Rules$_{tgt}$ for a grammar rule with the same name as the one from Rules$_{gen}$
find it?
NO
YES
Normalize the grammar rule if needed
Line matching
Identify differences and generate rule configuration
An optimization rule for removing the grammar rule needs to be configured.
Check if new grammar rules are added, and then configure optimization rules for it
Write the optimization rule configuration to a file.
output
JAVA
Configuration for GrammarOptimizer
End

Figure 3. The workflow of extracting optimization rule configurations based on a comparison between the generated grammar and the target grammar.

5.5 Difference Identification, Rule Selection and Parametrization

Once we completed the line-by-line matching as mentioned in the previous section, the next step is to perform the line-by-line comparison. As previously stated, except for attribute lines, the other types of lines are unique within a grammar rule. We only need to compare if they have been removed or renamed. For example, if the ConfigGenerator finds container braces (the outermost braces within a grammar rule) in the grammar rule `Model` in the generated grammar but not in the same grammar rule in the target grammar, it will select and parameterize an optimization rule for removing the braces.

Comparing attribute lines is more complex because they typically consist of multiple elements. Firstly, an attribute line always contains an attribute string which is usually in the form of e.g., `attributeName=typeName`. Additionally, it may include keywords, asterisks indicating multiplicity following parentheses, commas, and curly braces enclosed in single quotation marks, among other elements. The use of regular expressions enables us to identify and distinguish different elements, allowing for their comparison across different grammars. For example, when examining the attribute `ownedComment` in the same grammar rule on both sides, we may observe that in the generated grammar, this attribute is preceded by the keyword `'ownedComment'`, whereas in the target grammar, there is no keyword preceding it. In such a case, an optimization rule named `removeKeyword` would be selected and parameterized.

6 Evaluation

6.1 Results

Table 2 summarizes the results of applying ConfigGenerator to extract optimization rule configurations for different languages (see Table 1 for information about the sources and initially generated grammars of these languages).

The 2nd and 3rd columns show the size of the optimization rules configurations. For comparison, we first show the number of configuration lines in the manually created optimization rule configurations from our previous work [58]. Next to it are the number of configuration lines in the optimization rule configurations that were extracted by ConfigGenerator. For several languages, e.g. BibTex, the automatically extracted configuration had much more lines than the manually written counterpart (with the extreme case of EAST-ADL, where the extracted configuration is up to 100 times as long as the manually written one). This difference

Table 2. Results of applying ConfigGenerator to extract optimization rule configurations for different languages.

Language	**Configuration lines**		**Target grammar**[2]			**Optimized grammar**			**Grammar Comparison**		
	Manual	Extracted	Change	Remove	Add	Changed	Removed	Added	Same	Diff	Percent
EAST-ADL	31	3378	233	1	12	233	1	12	297	0	100%
BibTex	47	254	43	0	0	43	0	0	43	0	100%
Xenia	74	114	13	2	0	13	2	0	15	0	100%
DOT	79	134	24	3	1	16	3	1	21	3	87.5%
Xcore	307	351	20	14	7	17	14	7	28	12	70.0%
SML	421	369	40	56	8	38	56	8	51	24	68.0%

[1] The numbers in column 2, 4, 5 and 6 were obtained from the supplemental materials of our previous work. [58].
[2] Number of grammar rules that would need to be changed/be removed/be added to create the target grammar out of the generated grammar.

can be explained by the fact that the manually written optimization rule configurations make use of generalizations. I.e., instead of introducing for each grammar rule a new configuration removing curly braces, manually created configurations might just introduce one configuration that applies to all grammar rules. This generalization could in theory be imitated automatically, too. However, it would require additional analysis of the side effects of the generalized rule to make sure that no unintended changes happen. Therefore we, kept this to future work.

The columns for target grammar show the difference between the generated and the target grammar in the form of the number of grammar rules that require a change, require to be removed, and require to be added. These numbers are reported by our previous work. [58]. It can be seen that in all languages the majority of the grammar rules would need to change. This illustrates how different Xtext-generated grammars really are from grammars used in real languages and further illustrates the need to capture and preserve the manual effort made to create grammars.

Table 2 displays the number of changed, removed, and added grammar rules in the optimized grammar compared to the generated grammar in columns 7 to 9. The rightmost three columns compare the differences in grammar rules between the optimized grammar and the target grammar. For EAST-ADL, BibTex, and Xenia, we see the same amount of changed, removed, and added grammar rules as we would have expected judging from the target grammar. However, for DOT, Xcore, and SML the number of changed grammar rules is lower. This is already an indication that the generated optimization rule configuration did not perform the complete adaptation targeted for these three languages.

Finally, the last columns in Table 2 summarize how the optimized grammar compares to the target grammar. The results confirm that the grammar rules in the generated grammars of EAST-ADL, Bibtex, and Xenia have been optimized to be identical to the target grammar using the extracted optimization rule configurations. In the case of DOT, 87.5% of the grammar rules in the optimized grammar are identical to the target grammar. For Xcore and SML, the corresponding figures are 70.0% and 68.0%, respectively. Below we will discuss more in detail, when the ConfigGenerator performed well and when not.

6.2 Capabilities of ConfigGenerator

Although ConfigGenerator cannot optimize all grammar rules in the generated grammars of DOT, Xcore, and SML to achieve an identical state as their target grammars, it still provides the optimization rule configurations which perform the majority of necessary changes. Specifically, configuration rules for the following grammar changes were correctly generated in all cases:

- Removing or renaming individual keywords, including changing the value of `literals` in enumerations.
- Removing grammar rules or attributes.
- Modifying the multiplicities of attributes, including changing optional attributes to mandatory ones.
- Removing braces, including removing braces in attribute lines and container braces.
- Modifying the order of lines in a grammar rule, as long as lines can be identified by attribute names.
- Adding symbols to individual attribute lines.
- Adding rules, including adding terminal rules and primary type rules, as well as completing primary type rules which are to be implemented.
- Removing calls to other rules in grammar rules.
- Changing a specific type in the cross-reference of an attribute.

6.3 Missing Capabilities of ConfigGenerator

For the languages DOT, Xcore, and SML, there are a total of 42 grammar rules with differences between the optimized grammar and the target grammar. These differences can be found in the supplemental materials of this paper [41]. Here, we provide a list of typical cases of these differences.

- Difference in line order in some specific cases. In cases where one of the lines moved contains only a main keyword that has been changed by another configuration

Listing 3. Two attributes in the grammar rule `XOperation` in the *generated* grammar of Xcore

```
...
(unordered?='unordered')?
(unique?='unique')?
...
```

Listing 4. Two attributes in the grammar rule `XOperation` in the *original* grammar of Xcore

```
...
unordered?='unordered' unique?='unique'? |
unique?='unique' unordered?='unordered'?
...
```

rule, the reordering of the lines might not work. For example, in Xcore, the order of lines in the `XPackage` grammar rule is different between the optimized grammar and the target grammar. In the optimized grammar, the attribute `annotations` appears after the attribute `name`, while in the target grammar, it appears before the attribute `name`.

- Inconsistent attribute grouping. Listing 3 shows two attributes of the `XOperation` grammar rule in Xcore. In the generated grammar, they are listed one after the other, indicating an order of their appearance. In the target grammar, they are combined together and are in "and" and "or" relationships (as shown in Listing 4), indicating that their order is not predefined. The ConfigGenerator is not able yet, to create a configuration that replicates the occurrence of attributes in the grammar like that.
- Braces not changed to square brackets. In DOT, e.g., the container of the grammar rule `AttrList` in the generated grammar uses square brackets (i.e., '`[`' and '`]`'), while in the optimized grammar, it uses braces ('`{`' and '`}`').
- Difference in the position of optionality. In DOT, e.g., there is an attribute `attributes` in a grammar rule where optionality (i.e., `()?`) is handled differently. In the target grammar, the added comma and semicolon are surrounded by `()?`, i.e., `(',' | ';')?`. However, in the optimized grammar, the attribute string is surrounded by `()?`.

The cause of these limitations is that ConfigGenerator uses a line-based text comparison to identify which lines correspond to each other in the generated and in the target grammar and derives optimization rule configurations from this comparison. This limitation can be remedied by relying on a comparison that is based on an abstract syntax tree (AST) instead: the tool could parse both the generated and the target grammar and compare the ASTs, potentially making it more robust to changes in the order of lines and for groups that span multiple lines as these syntactical issues would not be present in the AST. Such an approach would also reduce the reliance on regular expressions which can be a limiting factor as well (see the fourth point in the list above). However, such an implementation is left for future work.

Finally, GrammarOptimizer, adopted without feature-level modification from our previous work [58] has limitations, e.g., rules to transform braces into brackets as needed for the third point mentioned above. While it would be possible to emulate this by first applying a rule that removes the braces and then adds brackets back, we have decided not to use this more complex strategy at this stage and leave the combination of different optimization rules as future work.

6.4 Usefulness of ConfigGenerator

The results in Section 6.1 indicate that the current version of ConfigGenerator can produce an optimization rule configuration that allows to modify a generated grammar fully into the target grammar for *some* of the languages we tested. In our initial work on GrammarOptimizer [58], we have shown that it is possible to find an optimization rule configuration manually to transform all of the languages we included in our evaluation to the target grammar.

We argue that ConfigGenerator is still a useful tool, even if it cannot fully derive a complete optimization rule configuration for all languages yet. SML, for instance, requires a total of 421 parameterized optimization rule invocations to be fully transformed. Creating all of them manually is a significant effort. ConfigGenerator automatically extracts 369 rules and therefore provides an excellent starting point for a language engineer to complete the optimization rule configuration.

ConfigGenerator is intended for use in scenarios where languages evolve and where they are rapidly prototyped. In such situations, speed is critical and ConfigGenerator increases the speed with which a language engineer can create an optimization rule configuration, even if it requires manual adaptations. We follow the line of argument from our previous work, that making manual changes to a reusable artifact such as an optimization rule configuration is less effort and faster than manually transforming a large grammar repeatedly.

6.5 Threats to Validity

There are several threats to the internal validity of our evaluation. The first stems from the fact that we worked with the slightly adjusted meta-models as well as Xtext versions of target grammars, which were partially not originally written in Xtext, from our previous work [58]. It is possible that these preparation steps introduce differences to the languages and, thus, might have simplified the task of changing the grammar and with that also the configurations that needed to be generated.

A further threat to internal validity concerns correctness: To which extent can we produce grammars that not only syntactically, but also semantically agree with manually changed counterparts? In our evaluation, most generated grammar rules were syntactically identical to their manually written counterparts, which indicates semantic equivalence and thus, correctness in these cases. This applies to three considered languages completely, and to three partially (68-87%). All observed differences were analyzed manually, as reported with details in Sections 6.2 and 6.3. The main practical implication of these cases is that existing grammar instances can no longer be parsed, which makes these inaccuracies easy to spot for the user.

However, analysing semantics and providing correctness guarantees during evolution are intrinsically hard problems. That is because a formalized semantics might not be available (as in our evaluation cases), and, where it is, the semantics might change over time. For example, UML 2 introduced a new Petri-net semantics for sequence diagrams. Supporting such evolution steps in semantics-sensitive way requires specialized approaches for the involved semantic representations (if available). Still, from our experience as language developers [1, 24, 43, 48, 52, 57], changes to semantics of existing language elements are exceedingly rare, and then require careful navigation on part of the developer. Specialized approaches could help, but are outside our scope. The vast majority of changes either add or remove language elements or change the syntax, which is exactly our scope.

Another threat that might make us overestimate the ability of the ConfigGenerator is that we could not build it without consulting real language examples. In consequence, the tool is very likely to work very well for the two used languages EAST-ADL and Xenia. To mitigate these two threats, we made sure to evaluate the tool on four additional languages, to also reduce the impact that changes to meta-models and target grammars of single languages might have had.

Finally, there is a threat to the external validity of generalizability. Of course, using more languages would have given us more insights into how well the ConfigGenerator already works. However, the languages we worked with are fairly different in character, which allows us to cover at least some level of language variety.

7 Conclusion

We presented an approach for supporting the co-evolution of meta-models and associated grammars. Our technical contribution is a technique for automatically extracting *grammar optimization rules*, which capture manual improvements from a previous evolution step, and allow these improvements to be replayed on future versions of the grammar. Our evaluation indicates a perfect coverage for three out of six considered cases – including a large one, namely, EAST-ADL – while showing good coverage with clearly identified limitations in the remaining ones.

We foresee several directions for future work. First, we aim to further improve the coverage of our approach. One idea is to move the grammar comparison to the level of ASTs, rather than lines, which would help to improve support for multiple-line changes. Second, a complementary co-evolution scenario to the one addressed in this paper, which requires support as well, involves migrations of the meta-model after changes to the grammar. Third, we intend to provide support for all-quantified rules (e.g., removing curly braces from all grammar rules) via automated generalization. This would allow to extract considerably more compact and easy-to-read rule configurations. Fourth, a comprehensive evaluation of our technique in concert with the baseline technique from [58] on a full-fledged co-evolution scenario would yield further insight into the practical applicability of our approach. The next step of the work [58] can be to apply GrammarOptimizer to build a language workbench that supports blended modeling [5]. If the automatic extraction capability of this paper can be integrated, it will certainly assist the textual grammar optimization ability of this workbench.

References

[1] Hugo Bruneliere, Jokin Garcia, Philippe Desfray, Djamel Eddine Khelladi, Regina Hebig, Reda Bendraou, and Jordi Cabot. 2015. On lightweight metamodel extension to support modeling tools agility. In *ECMFA*. Springer, 62–74.

[2] Loli Burgueno, Jordi Cabot, Shuai Li, and Sébastien Gérard. 2022. A generic LSTM neural network architecture to infer heterogeneous model transformations. *Software and Systems Modeling* 21, 1 (2022), 139–156.

[3] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. 2009. Managing dependent changes in coupled evolution. In *Theory and Practice of Model Transformations: Second International Conference, ICMT 2009, Zurich, Switzerland, June 29-30, 2009. Proceedings 2.* Springer, 35–51.

[4] Matej Črepinšek, Marjan Mernik, Faizan Javed, Barrett R Bryant, and Alan Sprague. 2005. Extracting grammar from programs: evolutionary approach. *ACM Sigplan Notices* 40, 4 (2005), 39–46.

[5] Istvan David, Malvina Latifaj, Jakob Pietron, Weixing Zhang, Federico Ciccozzi, Ivano Malavolta, Alexander Raschke, Jan-Philipp Steghöfer, and Regina Hebig. 2023. Blended modeling in commercial and open-source model-driven software engineering tools: A systematic study. *Software and Systems Modeling* 22, 1 (2023), 415–447.

[6] Andreas Demuth, Markus Riedl-Ehrenleitner, Roberto E Lopez-Herrejon, and Alexander Egyed. 2016. Co-evolution of metamodels and models through consistent change propagation. *Journal of Systems and Software* 111 (2016), 281–297.

[7] Git developers. 2023. Git merge. Retrieved June 2023 from https://git-scm.com/docs/git-merge Accessed June, 2023.

[8] Davide Di Ruscio, Amleto Di Salle, Ludovico Iovino, and Alfonso Pierantonio. 2023. A modeling assistant to manage technical debt in coupled evolution. *Information and Software Technology* 156 (2023), 107146.

[9] Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. 2011. What is needed for managing co-evolution in mde?. In *Proceedings of*

the 2nd International Workshop on Model Comparison in Practice. ACM, 30–38.

[10] Davide Di Ruscio, Ralf Lämmel, and Alfonso Pierantonio. 2011. Automated co-evolution of GMF editor models. In *Software Language Engineering: Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers 3*. Springer, 143–162.

[11] Bastien Drouot and Joël Champeau. 2019. Model Federation based on Role Modeling. In *MODELSWARD*. 72–83.

[12] EAST-ADL Association. 2022. EATOP Repository. https://bitbucket.org/east-adl/east-adl/src/Revison/ Accessed February, 2023.

[13] Eclipse Foundation. 2012. Xcore Metamodel. Retrieved May 2022 from https://git.eclipse.org/c/emf/org.eclipse.emf.git/tree/plugins/org.eclipse.emf.ecore.xcore/model/Xcore.ecore Accessed February, 2023.

[14] Eclipse Foundation. 2018. Eclipse Xcore Wiki. Retrieved May 2022 from https://git.eclipse.org/c/emf/org.eclipse.emf.git/tree/plugins/org.eclipse.emf.ecore.xcore/src/org/eclipse/emf/ecore/xcore/Xcore.xtext Accessed February, 2023.

[15] Eclipse Foundation. 2023. Xtext. Retrieved June 2023 from https://www.eclipse.org/Xtext/index.html Accessed June, 2023.

[16] Karsten Ehrig, Jochen Malte Küster, and Gabriele Taentzer. 2009. Generating instance models from meta models. *Software & Systems Modeling* 8 (2009), 479–500.

[17] Martin Eisenberg, Hans-Peter Pichler, Antonio Garmendia, and Manuel Wimmer. 2021. Towards reinforcement learning for in-place model transformations. In *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 82–88.

[18] Luka Fürst, Marjan Mernik, and Viljan Mahnič. 2015. Converting metamodels to graph grammars: doing without advanced graph grammar features. *Software & Systems Modeling* 14 (2015), 1297–1317.

[19] Sinem Getir, Lars Grunske, Christian Karl Bernasko, Verena Käfer, Tim Sanwald, and Matthias Tichy. 2015. CoWolf–A generic framework for multi-view co-evolution and evaluation of models. In *Theory and Practice of Model Transformations: 8th International Conference, ICMT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 20-21, 2015. Proceedings 8*. Springer, 34–40.

[20] Fahad R Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guerin, and Christophe Guychard. 2016. Addressing modularity for heterogeneous multi-model systems using model federation. In *Companion Proceedings of the 15th International Conference on Modularity*. 206–211.

[21] Joel Greenyer. 2018. Scenario Modeling Language (SML) Repository. Retrieved May 2022 from https://bitbucket.org/jgreenyer/scenariotools-sml/src/master/ Accessed February, 2023.

[22] Christophe Guychard, Sylvain Guerin, Ali Koudri, Antoine Beugnard, and Fabien Dagnat. 2013. Conceptual interoperability through models federation. In *Semantic Information Federation Community Workshop*. 23.

[23] Regina Hebig, Djamel Eddine Khelladi, and Reda Bendraou. 2016. Approaches to co-evolution of metamodels and models: A survey. *IEEE Transactions on Software Engineering* 43, 5 (2016), 396–414.

[24] Jörg Holtmann, Jan-Philipp Steghöfer, and Henrik Lönn. 2022. Migrating from proprietary tools to open-source software for EAST-ADL metamodel generation and evolution. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. 7–11.

[25] Faizan Javed, Marjan Mernik, Jeff Gray, and Barrett R Bryant. 2008. MARS: A metamodel recovery system using grammar inference. *Information and Software Technology* 50, 9-10 (2008), 948–968.

[26] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. 2011. A rule-based approach to the semantic lifting of model differences in the context of model versioning. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 163–172.

[27] Wael Kessentini and Vahid Alizadeh. 2020. Interactive metamodel/-model co-evolution using unsupervised learning and multi-objective search. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. IEEE, 68–78.

[28] Wael Kessentini, Houari Sahraoui, and Manuel Wimmer. 2019. Automated metamodel/model co-evolution: A search-based approach. *Information and Software Technology* 106 (2019), 49–67.

[29] Djamel Eddine Khelladi, Reda Bendraou, Regina Hebig, and Marie-Pierre Gervais. 2017. A semi-automatic maintenance and co-evolution of OCL constraints with (meta) model evolution. *Journal of Systems and Software* 134 (2017), 242–260.

[30] Djamel Eddine Khelladi, Horacio Hoyos Rodriguez, Roland Kretschmer, and Alexander Egyed. 2017. An exploratory experiment on metamodel-transformation co-evolution. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 576–581.

[31] Angelika Kusel, Jürgen Etzlstorfer, Elisabeth Kapsammer, Werner Retschitzegger, Wieland Schwinger, and Johannes Schönböck. 2015. Consistent co-evolution of models and transformations. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 116–125.

[32] Ralf Lämmel. 2001. Grammar adaptation. In *FME 2001: Formal Methods for Increasing Software Productivity: International Symposium of Formal Methods Europe Berlin, Germany, March 12–16, 2001 Proceedings*. Springer, 550–570.

[33] Ralf Lämmel. 2007. Style normalization for canonical X-to-O mappings. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. ACM, 31–40.

[34] Ralf Lämmel. 2018. *Software Languages*. Springer.

[35] Ralf Lämmel and Chris Verhoef. 2001. Cracking the 500-language problem. *IEEE software* 18, 6 (2001), 78–88.

[36] Ralf Lämmel and Vadim Zaytsev. 2009. An introduction to grammar convergence. In *Integrated Formal Methods: 7th International Conference, IFM 2009, Düsseldorf, Germany, February 16-19, 2009. Proceedings 7*. Springer, 246–260.

[37] Tiziano Lombardi, Vittorio Cortellessa, Alfonso Pierantonio, and In Model. 2021. Co-evolution of Metamodel and Generators: Higher-order Templating to the Rescue. *J. Object Technol.* 20, 3 (2021), 7–1.

[38] Jesús J López-Fernández, Jesús Sánchez Cuadrado, Esther Guerra, and Juan De Lara. 2015. Example-driven meta-model development. *Software & Systems Modeling* 14 (2015), 1323–1347.

[39] Bart Meyers, Manuel Wimmer, Antonio Cicchetti, and Jonathan Sprinkle. 2012. A generic in-place transformation-based approach to structured model co-evolution. *Electronic Communications of the EASST* 42 (2012), 13 pages.

[40] miklossy, nyssen, prggz, and mwienand. 2020. Dot Xtext grammar. Retrieved May 2020 from https://github.com/eclipse/gef/blob/master/org.eclipse.gef.dot/src/org/eclipse/gef/dot/internal/language/Dot.xtext Accessed February, 2023.

[41] OSF. [n. d.]. *Replication Package*. https://osf.io/6b2mf/?view_only=5cda50c7a6a9497eb65136fc443266cc

[42] Richard F. Paige, Dimitrios S. Kolovos, and Fiona A.C. Polack. 2014. A tutorial on metamodelling for grammar researchers. *Science of Computer Programming* 96 (2014), 396–416. https://doi.org/10.1016/j.scico.2014.05.007 Selected Papers from the Fifth Intl. Conf. on Software Language Engineering (SLE 2012).

[43] Dennis Priefer, Wolf Rost, Daniel Strüber, Gabriele Taentzer, and Peter Kneisel. 2021. Applying MDD in the content management system domain: Scenarios, tooling, and a mixed-method empirical assessment. *Software and Systems Modeling* 20 (2021), 1919–1943.

[44] Louis M Rose, Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. 2010. Model migration with Epsilon Flock. In *Theory and Practice of Model Transformations: Third International Conference, ICMT 2010, Malaga, Spain, June 28-July 2, 2010. Proceedings 3*. Springer, 184–198.

[45] Louis M Rose, Richard F Paige, Dimitrios S Kolovos, and Fiona A Polack. 2009. An analysis of approaches to model migration. In *Proc. Joint MoDSE-MCCM Workshop*. 6–15.

[46] Thomas Stahl and Markus Völter. 2006. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, Inc.

[47] Daniel Strüber. 2017. Generating efficient mutation operators for search-based model-driven engineering. In *Theory and Practice of Model Transformation: 10th International Conference, ICMT 2017, Held as Part of STAF 2017, Marburg, Germany, July 17-18, 2017, Proceedings 10*. Springer, 121–137.

[48] Daniel Strüber, Kristopher Born, Kanwal Daud Gill, Raffaela Groner, Timo Kehrer, Manuel Ohrndorf, and Matthias Tichy. 2017. Henshin: A usability-focused framework for EMF model transformation development. In *ICGT*. Springer, 196–208.

[49] Daniel Strüber, Julia Rubin, Thorsten Arendt, Marsha Chechik, Gabriele Taentzer, and Jennifer Plöger. 2018. Variability-based model transformation: formal foundation and application. *Formal Aspects of Computing* 30 (2018), 133–162.

[50] Daniel Strüber and Stefan Schulz. 2016. A tool environment for managing families of model transformation rules. In *Graph Transformation: 9th International Conference, ICGT 2016, in Memory of Hartmut Ehrig, Held as Part of STAF 2016, Vienna, Austria, July 5-6, 2016, Proceedings 9*. Springer, 89–101.

[51] Dániel Varró. 2006. Model transformation by example. In *Model Driven Engineering Languages and Systems: 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006. Proceedings 9*. Springer, 410–424.

[52] Steffen Vaupel, Daniel Strüber, Felix Rieger, and Gabriele Taentzer. 2015. Agile bottom-up development of domain-specific IDEs for model-driven development. In *FlexMDE'15: Workshop on Flexible Model Driven Engineering*. CEUR-WS.org, 12–21.

[53] Manuel Wimmer, Michael Strommer, Horst Kargl, and Gerhard Kramler. 2007. Towards model transformation generation by-example. In *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*. IEEE, 285b–285b.

[54] Xenia Authors. 2019. Xenia Metmodel. Retrieved May 2022 from https://github.com/rodchenk/xenia/blob/master/com.foliage.xenia/model/generated/Xenia.ecore Accessed February, 2023.

[55] Xenia Authors. 2019. Xenia Xtext. Retrieved May 2022 from https://github.com/rodchenk/xenia/blob/master/com.foliage.xenia/src/com/foliage/xenia/Xenia.xtext Accessed February, 2023.

[56] Vadim Zaytsev. 2012. Language Evolution, Metasyntactically. *Electronic Communications of the EASST* 49 (2012), 17 pages.

[57] Weixing Zhang, Regina Hebig, Jan-Philipp Steghöfer, and Jörg Holtmann. 2023. Creating Python-Style Domain Specific Languages: A Semi-Automated Approach and Intermediate Results.. In *MODELSWARD*. 210–217.

[58] Weixing Zhang, Jörg Holtmann, Regina Hebig, and Jan-Philipp Steghöfer. 2023. Meta-model-based Language Evolution and Rapid Prototyping with Automate Grammar Optimization. Preprint available at SSRN: https://ssrn.com/abstract=4379232.

Received 2023-07-07; accepted 2023-09-01

Reuse and Automated Integration of Recommenders for Modelling Languages

Lissette Almonte
Universidad Autónoma de Madrid
Madrid, Spain

Antonio Garmendia
Universidad Autónoma de Madrid
Madrid, Spain

Esther Guerra
Universidad Autónoma de Madrid
Madrid, Spain

Juan de Lara
Universidad Autónoma de Madrid
Madrid, Spain

Abstract

Many recommenders for modelling tasks have recently appeared. They use a variety of recommendation methods, tailored to concrete modelling languages. Typically, recommenders are created as independent programs, and subsequently need to be integrated within a modelling tool, incurring in high development effort. Moreover, it is currently not possible to reuse a recommender created for a modelling language with a different notation, even if they are similar.

To attack these problems, we propose a methodology to reuse and integrate recommenders into modelling tools. It considers four orthogonal dimensions: the target modelling language, the tool, the recommendation source, and the recommended items. To make homogeneous the access to arbitrary recommenders, we propose a reference recommendation service that enables indexing recommenders, investigating their properties, and obtaining recommendations likely coming from several sources. Our methodology is supported by IronMan, an Eclipse plugin that automates the integration of recommenders within Sirius and tree-based editors, and can bridge recommenders created for a modelling language for their reuse with a different one. We evaluate the power of the tool by reusing 2 recommenders for 4 different languages, and integrating them into 6 modelling tools.

CCS Concepts: **• Software and its engineering → Integrated and visual development environments**; **• Information systems → Recommender systems**.

Keywords: Model-driven engineering, recommender systems, language engineering, modelling tools

ACM Reference Format:
Lissette Almonte, Antonio Garmendia, Esther Guerra, and Juan de Lara. 2023. Reuse and Automated Integration of Recommenders for Modelling Languages. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3623476.3623523

1 Introduction

Recommender systems (RSs) are increasingly being used to assist developers in all sorts of software engineering tasks [51]. Modelling is no exception, as we are recently witnessing the proposal of numerous recommenders for modelling languages [3]. Most of them help in creating models or meta-models by recommending, e.g., new attributes or references for classes, or new classes related to existing ones [2, 11, 17, 21, 55, 61]. They use a variety of methods – each with their own strengths and weaknesses – ranging from classical recommendation algorithms like collaborative filtering [2, 17] or content-based recommendations [2], to knowledge graphs [55], natural language processing [11], pre-trained language models [61] or graph kernels [21].

Given this growing plethora of modelling recommenders, the natural question is "*Can I reuse these RSs for my modelling notation, and integrate them within my modelling tool?*". However, the reuse and integration of RSs pose a number of practical challenges. Firstly, existing RSs may have been developed for a different (albeit perhaps similar) modelling language, such as an existing RS for Ecore models that one may like to reuse for UML class diagrams. Moreover, it can be useful to combine several RSs because they suggest different types of items (e.g., attributes, operations) for different target elements (e.g., classes, interfaces). Further, even if they suggest the same type of items, combining RSs might be useful to retain their best recommendations. Finally, from a technical point of view, RSs may be deployed in numerous ways (e.g., a stand-alone program, a service, within a modelling tool), and need to be integrated within heterogeneous modelling tools (e.g., graphical, textual, tree-based).

In this paper, we address the challenges of integrating and reusing RSs for modelling languages. To accomplish this goal, we propose deploying the RSs as services, on the basis of a standard API and a recommender server protocol. This also facilitates the recommenders' integration within arbitrary modelling tools. In addition, we enable the reuse of RSs tailored to a modelling language for other notations via a structural mapping. The combination of recommenders of the same type of item (e.g., attributes) relies on mechanisms for the *aggregation* of their recommendation lists [46], and our approach is flexible to accommodate several aggregation methods. Technically, we provide support for the automated integration of the assembled RSs within Eclipse modelling editors based on Sirius [57], and EMF tree editors [59].

Our approach is realised as an Eclipse plugin called IronMan (Integrating RecOmmeNders for Modelling lANguages), which guides in all steps of the integration task, including RS discovery and selection, adaptation to the modelling language (if needed), configuration of the aggregation method, and integration of the recommender within the (Sirius- or tree-based) modelling tool. To evaluate its usefulness, we assess the reuse and integration of two existing RSs into six third-party tools of the Eclipse ecosystem.

Paper organisation. Sec. 2 provides background on RSs for modelling languages and analyses the relevant dimensions for their reuse and integration. Sec. 3 presents the components of our approach: the reference recommendation service and its protocol, the adaptation of the RSs to the modelling language, the recommendation aggregation mechanism, and the integration of the RSs into modelling tools. Sec. 4 describes our tool and Sec. 5 reports on its evaluation. Sec. 6 compares with related work and Sec. 7 concludes.

2 Background and Integration Dimensions

Next, we overview RSs for modelling languages (Sec. 2.1) and present the dimensions for their integration (Sec. 2.2).

2.1 Background on Recommender Systems

RSs have become ubiquitous software tools that assist in decision-making tasks in situations of information overload. They are key components of a wide range of applications, including e-commerce sites (e.g., Amazon), social networks (e.g., Facebook), and music (e.g., Spotify), video (e.g., Netflix) and streaming platforms (e.g., Twitch) [50].

RSs suggest items that align with the preferences of a particular user. The term *item* refers to what is suggested to the user. RSs usually focus on a particular type of item (e.g., videos), using filtering and ranking algorithms to provide valuable recommendations for that item type. The recommendations are computed based on data about three entities: *target users*, *items*, and *user-item interactions* (often unary or numeric ratings) that express personal preferences [50]. When applied to modelling tasks, these entities are sometimes reinterpreted. As an example, in a RS suggesting attributes for classes, the recommended items are the *attributes*, the target users are the *classes*, and the user-item interactions

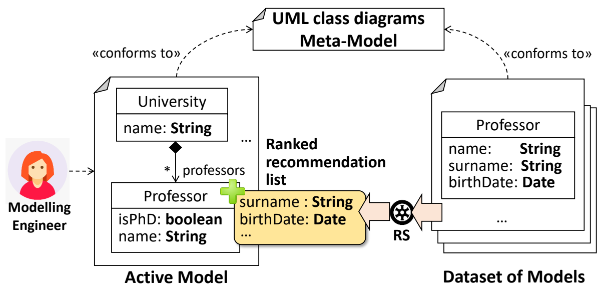


Figure 1. Working scheme of a modelling recommender.

are given by the inclusion of the attributes in each class and its superclasses. To avoid confusion, we use the term *target* to refer to the target users (classes in this example).

RSs can be classified into three main categories based on how they compute the recommendations: *content-based* systems recommend items similar to the ones that the user preferred in the past; *collaborative filtering* systems recommend items preferred by like-minded users; and *hybrid* systems combine the previous two techniques to overcome their limitations. The three approaches return a list of recommended items, which is often *ranked*. In addition, some recommendation methods provide a *rating* for each item, which quantifies the likelihood of the item to be relevant for the user.

Fig. 1 shows the working scheme of a RS for UML class diagrams. A RS for a modelling language is typically built on the basis of a dataset of models conformant to the language meta-model. Then, when a modelling engineer is working on a model conformant to the same meta-model, the RS can provide sensible recommendations. In the figure, the RS suggests new attributes to incorporate to a given class.

2.2 Dimensions of Integration of Modelling RSs

The integration and reuse of RSs for modelling tasks requires the consideration of several dimensions, summarised in Fig. 2 as a feature model [31].

Target modelling language. A RS can be integrated in modelling environments developed for the same modelling language as the RS supports (*homogeneous*), or alternatively, it can be reused for a different – albeit similar – modelling language (*heterogeneous*). For example, a RS for meta-modelling languages like Ecore [59] may be reused for UML class diagrams, and vice versa [4]. For this purpose, a mapping between the target modelling language and the RS is needed. Having the possibility to set this bridge is useful in cases where there is not enough data (i.e., models) to train a RS for a (domain-specific) modelling language, but a RS for a similar notation exists. We describe our approach to adapt recommenders in Sec. 3.3.

Recommendation sources. The recommendations may be produced *locally*, if the RS is deployed on the computer where the modelling tool is running [11, 17, 21, 61]. In

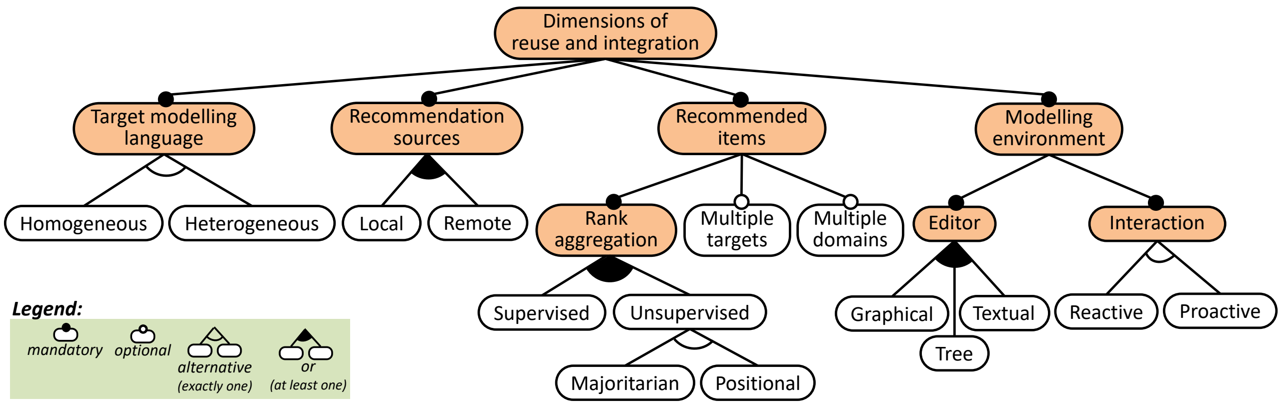


Figure 2. Dimensions of reuse and integration of RSs for modelling languages.

addition, recommendations may come from services deployed on a *remote* server [4, 55]. The latter option is more flexible, as it permits reusing recommenders within different tools, and aggregating recommendations from several sources. We propose a reference recommendation service in Sec. 3.2 for this purpose.

Recommended items. Integrating several RSs within a modelling environment enables the recommendation of items for *multiple targets* (e.g., for both classes and interfaces in class diagrams) and for *multiple domains* (e.g., RSs for medical, banking or transportation domains). When combining several RSs for the same type of target and item, their recommendation rankings need to be aggregated. According to [46], the approaches to recommendation *rank aggregation* are broadly classified into *unsupervised* and *supervised*. The former can be further divided into *majoritarian* and *positional*. When calculating a numerical score for each item in the aggregated recommendation list, positional methods use the absolute position of the item in the individual rankings, while majoritarian methods compare pairwise each item [46]. Unsupervised methods are generally simple, efficient and flexible. However, if ground truth data are available, supervised methods may be more effective. These methods may use a variety of techniques, like learning to rank [39] or genetic programming [60]. We focus on unsupervised methods, leaving supervised ones for future work. Sec. 3.4 exemplifies one unsupervised method recasted for modelling tasks.

Modelling environment. RSs need to be integrated into concrete modelling tools, typically offering *graphical*, *tree* and/or *textual editors*. The *interaction* with the RS may either be activated explicitly by the user (*reactive*) or be *proactive*, offering suggestions to the user when deemed appropriate (e.g., as in [40]). Sec. 3.5 explains our approach to integrate RSs into graphical and tree modelling editors with a reactive approach.

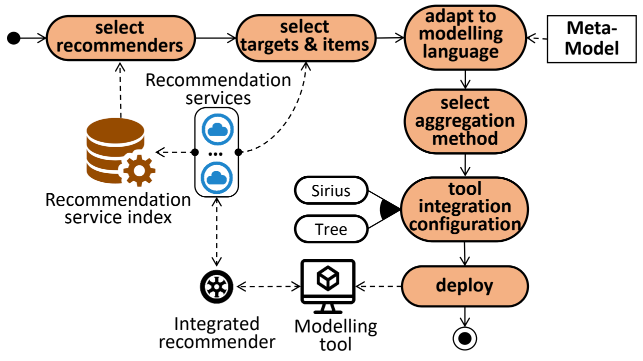


Figure 3. Overview of our methodology for RS integration.

3 Approach

This section presents our proposal to reuse and integrate RSs for modelling. First, Sec. 3.1 provides an overview. Then, Sec. 3.2 describes the recommendation service. Sec. 3.3 explains our approach to bridge RSs to modelling notations. Sec. 3.4 recasts existing aggregation methods for ranked recommendations to modelling RSs. Finally, Sec. 3.5 introduces our support for integrating RSs into modelling environments.

3.1 Overview

Fig. 3 shows the scheme of the methodology we have created for RS integration and reuse. It covers all dimensions of integration depicted in Fig. 2.

Our approach relies on deploying the RSs as services conformant to the reference REST API described in Sec. 3.2. This way, the first step in the integration consists in discovering the available RSs by means of a RS indexer. The indexer can filter the available services by diverse criteria, like the modelling language for which the RSs provide suggestions. In a second step, the user selects the recommendation targets and items (a subset of those provided by the RSs selected in the previous step). If several RSs of the same kind of items are chosen, the user will need to select an aggregation method for the recommendations. Moreover, if the modelling language where the RSs are to be integrated differ from the language supported by the RSs, then the user will have to adapt the RSs via a mapping. As the last step, the user configures the integration with the modelling tool. Currently, we support the adaptation of EMF-based modelling languages, and the integration with Sirius graphical editors [57] and tree editors. However, our extensible architecture facilitates future integration with other technologies, like Xtext [8, 62].

After performing these steps, our approach automatically integrates the assembled RSs within the modelling tool. The result is a plugin that communicates with the selected RS services to obtain recommendations, aggregating and adapting them to the modelling language.

3.2 Recommendation Service

We have developed a recommendation service consisting of two components: the recommendation service indexer API, and the recommendation service API. On the one hand, the *indexer* provides a standardised method to *register* and *update* recommendation services, and to *explore* the registered services, enabling clients to *discover* and access the recommendation services that best suit their needs. On the other hand, the *recommendation service* offers a uniform mechanism for *requesting recommendations* and accessing the *features* of the RSs registered in the indexer. This approach simplifies the integration of arbitrary RSs into modelling tools, avoiding the need to build custom, heterogeneous integrations.

Table 1 shows the REST endpoints of the indexer, which enable clients to register, update, explore, and discover services. The `/register` endpoint allows registering new recommendation services in the indexer. To register a service, clients need to provide its URL using the `/register?urlName=`⟨*url*⟩ endpoint, where ⟨*url*⟩ is the URL of the recommendation service. Several RSs can be placed within the same URL, and the registered URLs must define the endpoints defined in Table 2. In particular, upon registering a recommendation service, the indexer invokes its `/features` endpoint (explained below) to cache the characteristics of the RS.

The `/updateRegistration` endpoint allows clients to update a previously registered service. Similar to `/register`, clients only need to provide the URL of the service to be updated using the `/updateRegistration?urlName=`⟨*url*⟩ endpoint. As before, the indexer will then invoke the `/features` endpoint of the recommendation service.

The `/services` endpoint returns the list of all registered recommendation services and their metadata in JSON format, and `/discover` allows searching for deployed services using either the *name* or the *nsURI* of the RS. The *nsURI* is a unique identifier for meta-models, which is standard in modelling technologies like EMF [59]. This way, the API returns a JSON list with all recommenders with the given name or defined over a meta-model with the provided *nsURI*.

Table 1. Endpoints of the recommender indexer API.

Endpoint	`/register?urlName=`⟨*url*⟩
Desc.	Registers a new recommendation service.
Method	POST
Output	Ok/Error
Endpoint	`/updateRegistration?urlName=`⟨*url*⟩
Desc.	Updates a registered recommendation service.
Method	POST
Output	Ok/Error
Endpoint	`/services`
Desc.	Returns all registered recommendation services and their metadata. The optional query parameter `nsURI=true` groups services by *nsURI*.
Method	GET
Output	Available recommendation services (JSON).
Endpoint	`/discover?`(`name=`⟨*name*⟩ | `nsURI=`⟨*uri*⟩)
Desc.	Searches for registered recommendation services with the given RS name or meta-model *nsURI*.
Method	GET
Output	Registered recommendation services that match the search criteria (JSON).

Table 2. Endpoints of the recommendation service API.

Endpoint	`/features`
Desc.	Returns the features of all RSs within the recommendation service.
Method	GET
Output	Features of the recommendation service (JSON).
Endpoint	`/recommend/`⟨*name*⟩`?` (`newMaxRec=`⟨*maxRec*⟩)?, (`threshold=`⟨*threshold*⟩)?, (`itemType=`⟨*type*⟩)?
Desc.	Returns a list of recommendations from RS *name*, for a given target (within a context, if required). All parameters are optional: `newMaxRec` (integer) refers to the maximum number of recommendations to retrieve, `threshold` (double) to the threshold for the ranking, and `itemType` ([string]) to the type of recommended items.
Method	POST
Body	Target and its context, if required (JSON).
Output	List of recommendations (JSON).

Table 2 shows the endpoints of the recommendation service. They allow accessing the features of the registered services and requesting recommendations.

The `/features` endpoint allows clients to retrieve the metadata of all RSs within the service. Fig. 4 shows a conceptual model of the metadata. Class `RecommenderSystem` defines the name of the RS, the meta-model *nsURI*, and a description of the modelling *context* that the RS needs to compute the recommendations. The context can be `None` (only the target of the recommendation is needed), `Full` (requires the whole model containing the target element), or `Targets` (requires all objects with the same type as the target element).

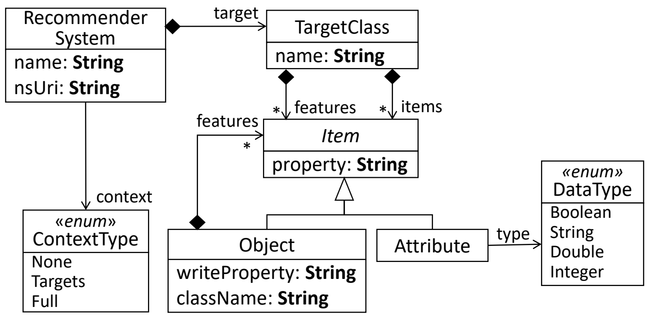
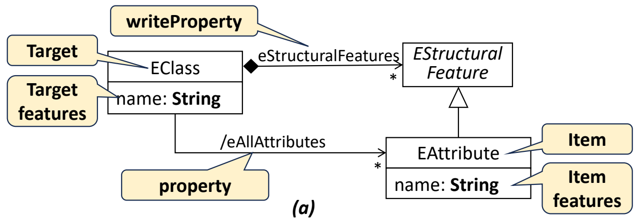


Figure 4. Conceptual model of RSs assumed by our approach.



```
"services": [
  { "name": "EcoreRecommender",
   "details": [
    { "nsURI"  : "http://www.eclipse.org/emf/2002/Ecore",
      "context": "Targets",
      "target": { "name"    : "EClass",
                  "features": [ {"property" : "name", "type" : "String" }]},
      "items":   [ { "name"          : "EAttribute"
                     "property"      : "eAllAttributes",
                     "writeProperty" : "eStructuralFeatures",
                     "features"      : [ { "property" : "name",
                                           "type" : "String" }] }]
    }]
  }]
```
(b)

Figure 5. (a) Excerpt of the Ecore meta-model, annotated with the role of the elements in the example RS. (b) Encoding excerpt of the RS returned by the /features endpoint.

The metadata also defines the target class of the recommendations, and its identifying features. For simplicity, our approach assumes that a RS only serves recommendations (e.g., attributes) for a target type (e.g., UML classes). If several target types are supported, then one RS for each target needs to be deployed. In addition, the metadata describe each item type to be recommended. For recommended attributes, it specifies their name and type, and for recommended objects, it specifies the reference name (connecting the object to the target class) and the features used to identify the object. We distinguish between the reference that provides access to an item (i.e., enabling to read the item, `property` in the figure), and the reference where to store an item (i.e., enabling writing the item, `writeProperty` in the figure). In EMF, the latter are containment references. Next, we use an example to illustrate the difference between both.

Example. Let's assume a RS for the Ecore meta-modelling language, which recommends attributes for classes. Fig. 5(a) shows the relevant parts of the Ecore meta-model for the RS[1]. The figure identifies that `EClass` is the target element, that the feature identifying `EClasses` is their `name`, that the recommended items are of type `EAttribute`, and that the feature provided when recommending an attribute is its `name`. In addition, `EAttributes` are *read* via the `eAllAttributes` derived reference, but written on the `eStructuralFeatures` composition reference. The former contains all attributes owned and inherited by the class, and the latter only the owned ones (and is a common container for both references and attributes). The rationale for distinguishing both is that modelling tools need to provide the items that any target object already has – owned and inherited attributes in our example, available via `eAllAttributes`. However, when a recommendation is accepted, the item needs to be created and assigned to the target – in our example, `eStructuralFeatures` is used.

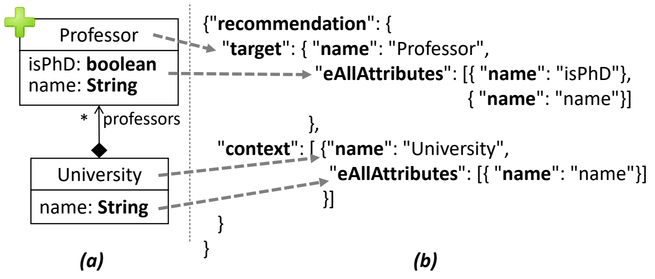
Fig. 5(b) shows the metadata (in JSON format) that would be returned when invoking the `/features` endpoint on the RS. In this case, the name of the RS is *EcoreRecommender*, and the RS needs to receive all other possible targets in the model (i.e., all `EClasses`) as context.

Some well-formedness criteria are required from the roles that meta-model elements can play in a RS. In particular, if the recommended items are of type c_i, then c_i should be the destination class of a write property p_w, or a subclass of such destination class: $c_i \leq dest(p_w)$. For instance, in Fig. 5(a), `EAttribute≤EStructuralFeature`, which is the destination of the write property `eStructuralFeatures`. This ensures compatibility of the items with the write property, so that newly created items can be inserted in it. Conversely, the destination of a read property p_r should be the type c_i of the recommended item, or a subclass: $dest(p_r) \leq c_i$. In the example, `EAttribute≤EAttribute`, which is the destination of the read property `eAllAttributes`. This ensures that the content of the read property is compatible with the item.

The last endpoint in Table 2 is `/recommend`, which allows clients to request recommendations by specifying the RS name as a path parameter, and providing a JSON file with the target of the recommendation, its current items, and its context (if needed). Clients can customise the recommendation by means of optional query parameters such as the maximum number of recommendations to retrieve (*newMaxRec*), the minimum ranking value threshold (*threshold*), and the desired item type when several are possible (*itemType*).

Example. Fig. 6 shows a recommendation request example for the RS in Fig 5. Part (a) shows an Ecore model being edited, where recommendation for class *Professor* is solicited. Part (b) shows the encoding of the request, where the target `EClass` is named *Professor*, and has two `EAttributes` called *isPhD* and *name*. They are encoded in the *read* feature `eAllAttributes`. As specified in Fig. 5, the only feature that identifies `EAttributes` is their *name*. Since the context of the

[1]The meta-model is slightly modified to ease understanding.



```
{"recommendation": {
  "target": { "name": "Professor",
              "eAllAttributes": [{ "name": "isPhD"},
                                 { "name": "name"}]
            },
  "context": [ {"name": "University",
                "eAllAttributes": [{ "name": "name"}]
              }]
  }
}
```

Figure 6. JSON representation for a `/recommend` request.

RS is set to `Targets`, the request needs to include the name and attributes of all other `EClasses` in the model. In this case, there is just one additional class named *University*.

3.3 Adaptation of RSs to the Modelling Notation

Our approach to reuse a RS for a modelling notation involves establishing a structure-preserving mapping $m : RS \longrightarrow MM$ between the classes and features used by the RS, and the elements of interest in the language meta-model MM.

Example. Fig. 7 exemplifies a mapping that adapts the RS for Ecore – which recommends `EAttributes` for `EClasses` – to UML. The adapted RS will then recommend properties for UML classes. *RS* on the left shows an excerpt of the Ecore meta-model containing the elements designated as targets, features and items. The mapping maps the Ecore meta-model elements playing a role in the RS to corresponding elements in the UML meta-model MM to the right.

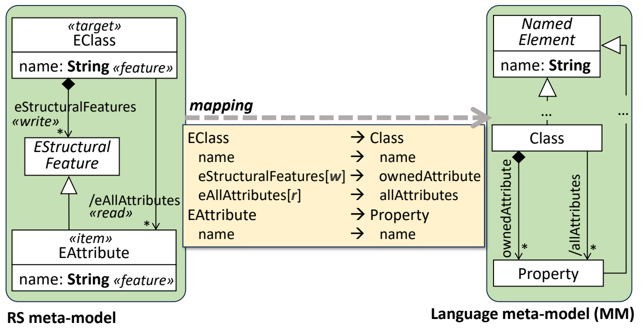


Figure 7. Adapting the RS to the modelling language.

Not any mapping is valid, but well-formed mappings need to preserve the structure of the source meta-model. For this purpose, we build on the notion of *binding*, which has been used to express generic model operations [15, 54]. Next, we use predicates $item(\_)$, $feature(\_)$, $property(\_)$, $writeProperty(\_)$, and $target(\_)$ to denote the role of the element in RS; predicate $relevant\ (e) = item(e) \lor feature(e) \lor property(e) \lor writeProperty(e) \lor target(e)$ to identify the elements that need to be mapped; $src(r)$ and $dest(r)$ for the source and destination class of reference r; and $c_i \leq c_j$ to denote that c_i is compatible with c_j (a subclass, or c_j itself).

This way, a mapping $m : RS \longrightarrow MM$ is well-formed iff it fulfils the following conditions:

Definition domain: m is defined exactly for each element e of RS s.t. $relevant(e)$.

Classes: If c is a class in RS s.t. $relevant(c)$, then $m(c)$ is also a class in MM.

Class subtyping is preserved and reflected: Given classes c_1 and c_2 of RS s.t. $relevant(c_1) \land relevant(c_2)$, then $c_1 \leq c_2 \iff m(c_1) \leq m(c_2)$.

Attributes: If a is a relevant attribute defined or inherited by a relevant class c in RS, then $m(a)$ is also an attribute inherited or defined in class $m(c)$. The type of the attribute must be preserved or generalised in the mapping: a.type $\leq m(a)$.type. For instance, an attribute of type *integer* can be mapped to a *double*.

References: If r is a relevant reference from class c_1 to c_2 in RS, then $m(r)$ is also a reference from class c'_1 to c'_2 in MM, with $m(c_1) \leq c'_1$. In addition, we need a further constraint for $dest(r)$, which depends on whether r is *read* ($property(r)$) or *write* ($writeProperty(r)$):

$property(r)$**:** Any relevant superclass of $dest(r)$ (including $dest(r)$, if it is relevant) is mapped to a superclass of c'_2, or to c'_2:
$\forall c_i \in RS \cdot dest(r) \leq c_i \land relevant(c_i) \implies c'_2 \leq m(c_i)$

$writeProperty(r)$**:** Any relevant class compatible with $dest(r)$ is mapped to a class compatible with c'_2:
$\forall c_i \in RS \cdot c_i \leq dest(r) \land relevant(c_i) \implies m(c_i) \leq c'_2$

Composition is preserved: If r is a relevant write composition in RS, then $m(r)$ is also a composition.

The condition for references permits a reference r to be declared exactly on the mapped class, or in a superclass (so that it is inherited). Similarly, the destination of the reference r can be the relevant class, a subclass (when $property(r)$), or a superclass (when $writeProperty(r)$), which then should be mapped preserving subtyping.

Typically, references that are *write* (allowing adding an item to a target) are composition references in RS, which needs to be preserved in the target by the last well-formedness condition. The mapping does not care about cardinalities, as they do not need to be preserved.

Example. The mapping of Fig. 7 is well-formed. This is so as both `EClass` and `EAttribute` are mapped to classes in the UML meta-model (`Class` and `Property`), and their attributes (`EClass.name` and `EAttribute.name`) are mapped to attributes of the target classes (actually inherited). Both references `eStructuralFeatures` and `eAllAttributes` are mapped according to the conditions, e.g.,: $m($`eStructuralFeatures`$)$ `=ownedAttribute`, the source of both references coincide ($m($`EClass`$)$ `=Class`), and for the destination, $m($`EAttribute`$)$ `=Property`, which is exactly $dest(m($`eStructuralFeature`$))$, but could be a subclass. Reference `eStructuralFeatures` is a *write* feature, and a composition, and so is `ownedAttribute`.

Our mapping enables consistent adaptations between structurally similar (but not identical) meta-models. For more complex mappings, our correspondences could be extended with an expression language – like OCL [29] – able to calculate derived elements in the target, or adapt attribute values.

3.4 Recommendation Aggregation

The RSs community has proposed different ranked item aggregation methods to combine recommendations from different RSs. A *rank aggregation method* aims to find the best permutation of recommendation lists based on an evaluation metric, such as precision. These methods can be used to provide more accurate and diverse suggestions by taking into account weaknesses or biases that specific recommenders may have, and to reduce the impact of items incorrectly ranked in high positions by an individual recommender [46].

When combining RSs for modelling languages, two scenarios can arise. In the first one, the RSs to combine tackle *different targets* (e.g., one RS provides recommendations for classes and another one for packages) or *different kinds of items* (e.g., one recommends attributes, and another operations). In such cases, no aggregation is needed, since the items are completely disjoint. This way, the composed RS would just use a different recommender for each kind of item, returning the lists of ranked items with no modification.

In the second scenario, multiple RSs recommend the *same kind of items* for a given target (e.g., two RSs of class attributes that use different algorithms, or different datasets). This scenario requires rank aggregation methods to obtain a consensus ranking containing a subset of their items.

Aggregation methods can be either supervised or unsupervised [46]. The former search for the aggregated ranking that optimises a given metric computed over ground-truth data. The latter lack ground truth data and rely on metrics computed using the available rankings of items. We focus on unsupervised methods, and consider score-based positional methods, as they are very popular due to their simplicity and efficiency. These methods sort the items based on their absolute position in the individual rankings. Positional methods receive as input a set of individual rankings, and use an aggregation function $f : U \times I \rightarrow R$ and a procedure to combine the item position-based scores, with U and I the sets of users and items in the system, respectively [46].

Unsupervised positional methods, such as Borda Count (BC) [7] and Median Rank Aggregation (MRA) [26], are popular for their simplicity and efficiency. Borda Count assigns points to items based on their rank, while MRA ranks items by their median position across the individual rankings.

Example. Fig. 8 illustrates the BC method for aggregating three hypothetical class attribute recommenders. ① shows the rankings of attributes that each RS suggests for a class named *Person*. ② depicts the scores assigned to the attributes

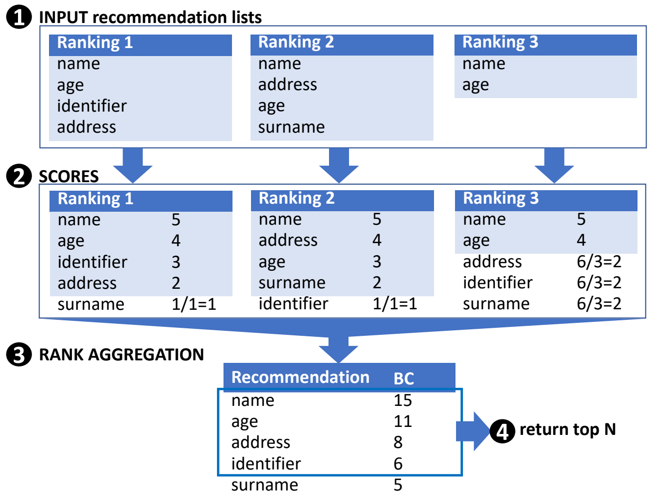


Figure 8. Rank aggregation example using Borda Count.

in each ranking. Since there are 5 unique (non-duplicate) attributes, the score of the first attribute in each ranking is 5, and this score is decreased for the subsequent positions of the ranking. The items that each RS do not recommend (e.g., *surname* in Ranking 1) receive equal portions of the remaining available points from the RS. ③ displays the aggregated score of each item, calculated as the sum of individual scores in the case of BC. Step ④ shows the returned top N list of recommendations. Incidentally, MRA would output the same aggregated rank, e.g., the rank of `name` is 1, which is the median of its positions in the three rankings.

3.5 Integration within Modelling Environments

The last step of the integration is to embed the RS into a modelling environment. This embedding will be different depending on the concrete syntax of the language. We currently consider two types: graphical syntaxes and tree-based ones. In both cases, we support a *reactive* approach by now, where the user needs to invoke the RS explicitly.

For graphical syntaxes, the integration adds an additional graphical layer in the modelling editor, which enables the option to invoke the RS when a shape corresponding to an instance object of the target class is selected. For tree-based syntaxes, a menu option becomes available when an instance object of the target class is right-clicked. In both cases, recommendations are requested to the recommender API of the selected RSs. Then, the recommended items can be applied to the model, assigned to the selected target object.

Sec. 4.2 will provide more details of this integration for the technologies we support (Sirius and EMF).

4 Architecture and Tool Support

We have realised the previous concepts on an extensible Eclipse plugin called IronMan. Its source code is available at: https://github.com/antoniogarmendia/integrate-reco

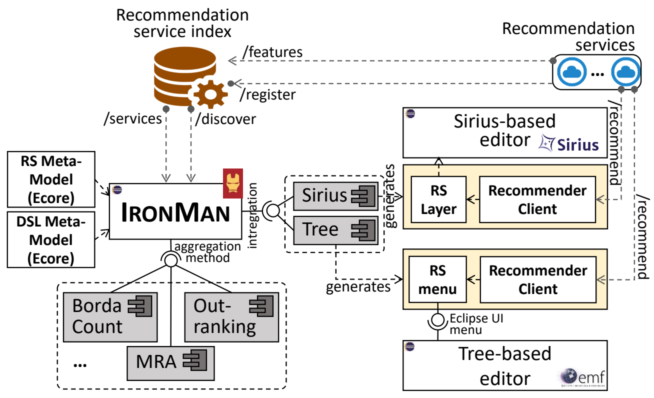


Figure 9. Architecture of IronMan.

mmenders-ironman. Next, Sec. 4.1 describes its architecture, and Sec. 4.2 reports on the tool itself.

4.1 Architecture

Fig. 9 shows the architecture of IronMan. The plugin uses the `/services` and `/discover` endpoints of the recommendation indexer to obtain the available RSs and filter them by meta-model name. IronMan supports the adaptation of the RS to modelling languages by the definition of a structural mapping between both meta-models, as Sec. 3.3 described.

IronMan provides two extension points. The first one is to define rank aggregation methods. IronMan currently supports Borda Count, MRA and Outranking [27], but the extension point permits adding more. The second one is to define code generators that can integrate the RSs with modelling tools. Currently, two implementations can generate integrations of RSs with Sirius and tree editors. For this purpose, the generated code makes use of the `/recommend` endpoint of the chosen recommendation service, and the selected aggregation method (if needed). In the case of Sirius editors, IronMan generates a recommendation layer that enables requesting the recommendations. In the case of tree editors, IronMan generates a menu that is activated when appropriate objects are selected in the tree.

4.2 Tool Support

Next, we describe the parts of our solution: the plugin, the services and the generated RS clients.

4.2.1 IronMan Plugin. Our tool provides a wizard to adapt RSs to a modelling language, configure the aggregation of recommendations for the same target (if needed), and integrate RSs into modelling workbenches. Fig. 10 shows 5 pages of the wizard. The first one permits selecting the available RSs from a set of recommendation service indexers. New indexers can be added using the IronMan preference page within the Eclipse IDE. Users can select any combination of RSs, as long as all of them are for the same language. In the figure, the indexers contain several RSs for UML and Ecore.

In page 2 of the wizard, the user can filter the recommended items of each selected RS. For example, in the figure, the page contains recommenders of attributes and operations for classes. The user might be interested in obtaining only attribute recommendations, which can be selected within this page. It is possible to select several RSs for the same target and items, or for the same target and different items.

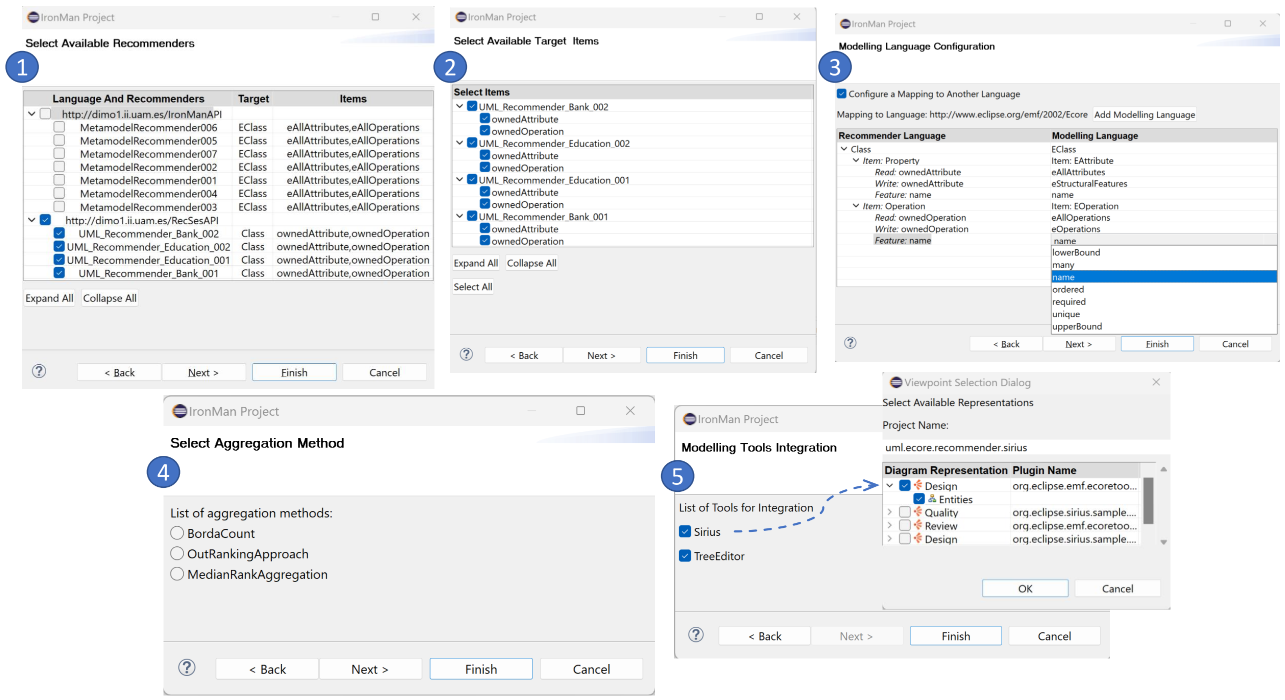
In page 3, the user can adapt the RS to a modelling language, in case the RS targets a different language. For this purpose, the user first selects the meta-model of the modelling language, and then, a tree-table enables mapping the relevant elements of the RS and the modelling language. In the figure, the user maps elements from UML to Ecore. For instance, in UML, the reference to obtain all attributes is `ownedAttribute`, but in Ecore is `eAllAttributes`. Similarly, the composition reference to add `Properties` to `Classes` in UML is `ownedAttribute` as well, but in Ecore is `eStructural-Features`. The identifier of attributes in both UML and Ecore is `name`. Since the API provides the RS metadata, there is no need to store the RS meta-model locally.

Page 4 is enabled only when the user selects RSs providing recommendations for the same target and item, which need to be aggregated. The figure shows the three aggregation methods implemented using the extension point.

In page 5, the user selects the environments – Sirius and/or tree editor – where the RS will be integrated. In case of Sirius, the user needs to select the viewpoint where the recommendation layer is to be inserted. The figure shows the selection dialog, where the user can select several views. The code generator produces plugin projects with the RS clients, which extend the modelling environments externally, without the need to have available their source code.

4.2.2 Recommendation Services. The IronMan service indexer is implemented as a Java-based REST service using Jersey [24], a framework for building RESTful web services and APIs. It is deployed on Tomcat [6], an open-source web server and Servlet container. Four core classes are responsible for handling requests from clients. *ServiceRegistration* handles registration-related requests, such as service registration, registration updates, deletion, or queries of registered services. *ServiceFeatures* handles requests for deployed and registered services, as well as their metadata. *ServiceRecommend* is responsible for generating recommendations. Finally, *ServiceDiscovery* enables service discovery. The response time for any request is generally less than a second.

4.2.3 Integration with Client Modelling Tools. IronMan synthesises code that extends externally existing (Sirius and tree) modelling editors. The generated code considers the defined mapping. It uses the EMF reflective API to query the relevant features of the target of the recommendations, and to create objects corresponding to recommended items when the user applies a recommendation.


IronMan Project
1
Select Available Recommenders
Expand All
Collapse All
< Back
Next >
Finish
Cancel
2
Select Available Target Items
Select All
3
Modelling Language Configuration
Add Modelling Language
4
Select Aggregation Method
5
Modelling Tools Integration
Viewpoint Selection Dialog
Select Available Representations
OK

Language And Recommenders	Target	Items
http://dimo1.ii.uam.es/IronManAPI		
MetamodelRecommender006	EClass	eAllAttributes,eAllOperations
MetamodelRecommender005	EClass	eAllAttributes,eAllOperations
MetamodelRecommender007	EClass	eAllAttributes,eAllOperations
MetamodelRecommender002	EClass	eAllAttributes,eAllOperations
MetamodelRecommender001	EClass	eAllAttributes,eAllOperations
MetamodelRecommender004	EClass	eAllAttributes,eAllOperations
MetamodelRecommender003	EClass	eAllAttributes,eAllOperations
✓ http://dimo1.ii.uam.es/RecSesAPI		
✓ UML_Recommender_Bank_002	Class	ownedAttribute,ownedOperation
✓ UML_Recommender_Education_002	Class	ownedAttribute,ownedOperation
✓ UML_Recommender_Education_001	Class	ownedAttribute,ownedOperation
✓ UML_Recommender_Bank_001	Class	ownedAttribute,ownedOperation

Select Items
- ✓ UML_Recommender_Bank_002
  - ✓ ownedAttribute
  - ✓ ownedOperation
- ✓ UML_Recommender_Education_002
  - ✓ ownedAttribute
  - ✓ ownedOperation
- ✓ UML_Recommender_Education_001
  - ✓ ownedAttribute
  - ✓ ownedOperation
- ✓ UML_Recommender_Bank_001
  - ✓ ownedAttribute
  - ✓ ownedOperation

✓ Configure a Mapping to Another Language

Mapping to Language: http://www.eclipse.org/emf/2002/Ecore

Recommender Language	Modelling Language
Class	EClass
Item: Property	Item: EAttribute
Read: ownedAttribute	eAllAttributes
Write: ownedAttribute	eStructuralFeatures
Feature: name	name
Item: Operation	Item: EOperation
Read: ownedOperation	eAllOperations
Write: ownedOperation	eOperations
Feature: name	name

lowerBound
many
name
ordered
required
unique
upperBound

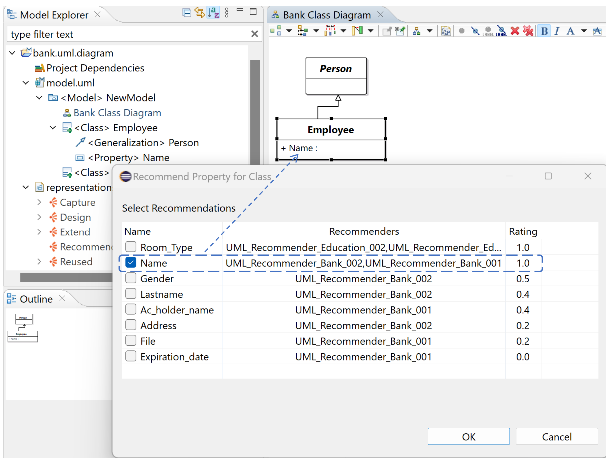
List of aggregation methods:
- ○ BordaCount
- ○ OutRankingApproach
- ○ MedianRankAggregation

List of Tools for Integration
- ✓ Sirius
- ✓ TreeEditor

Project Name:
uml.ecore.recommender.sirius

Diagram Representation	Plugin Name
✓ Design	org.eclipse.emf.ecoretoo...
✓ Entities	
Quality	org.eclipse.sirius.sample....
Review	org.eclipse.emf.ecoretoo...
Design	org.eclipse.sirius.sample....

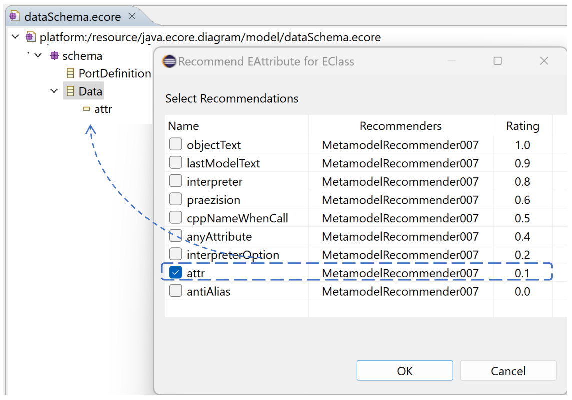
Figure 10. IronMan wizard: (1) Selecting recommender services, (2) filtering the items to be recommended, (3) mapping the RS to the modelling language (optional), (4) selecting the aggregation method, (5) configuring integration with modelling tool.



Select Recommendations

Name	Recommenders	Rating
Room_Type	UML_Recommender_Education_002,UML_Recommender_Ed...	1.0
✓ Name	UML_Recommender_Bank_002,UML_Recommender_Bank_001	1.0
Gender	UML_Recommender_Bank_002	0.5
Lastname	UML_Recommender_Bank_002	0.4
Ac_holder_name	UML_Recommender_Bank_001	0.4
Address	UML_Recommender_Bank_002	0.2
File	UML_Recommender_Bank_001	0.2
Expiration_date	UML_Recommender_Bank_001	0.0

Figure 11. Integration of the RS within a Sirius editor.


dataSchema.ecore
platform:/resource/java.ecore.diagram/model/dataSchema.ecore
schema
PortDefinition
Data
attr
Recommend EAttribute for EClass
OK
Cancel

Select Recommendations

Name	Recommenders	Rating
objectText	MetamodelRecommender007	1.0
lastModelText	MetamodelRecommender007	0.9
interpreter	MetamodelRecommender007	0.8
praezision	MetamodelRecommender007	0.6
cppNameWhenCall	MetamodelRecommender007	0.5
anyAttribute	MetamodelRecommender007	0.4
interpreterOption	MetamodelRecommender007	0.2
✓ attr	MetamodelRecommender007	0.1
antiAlias	MetamodelRecommender007	0.0

Figure 12. Integration of the RS within a tree editor.

Fig. 11 shows a screenshot of the result of the integration of a RS within the Eclipse plugin of Obeo's UML Designer [43]. Once the recommendation layer is active, the RS can be invoked over objects of the target element type (UML classes in this case). The recommender dialog shows a ranked list of recommendations, and the RSs these come from. In this case, two recommenders provide the recommendations, and the list is aggregated using the selected rank aggregation method. When the user selects an item, the corresponding object is created and added to the target.

Fig. 12 shows the integration of a RS within the standard Ecore tree editor [59]. The RS can be triggered upon selecting an `EClass`. When an `EAttribute` is selected in the recommendation list, the corresponding object is created and assigned to the selected `EClass`.

5 Evaluation

Next, we report on an evaluation to assess the usefulness of our approach in terms of its capacity to reuse RSs and to integrate them with existing modelling tools. Hence, we aim to answer the following research questions (RQs):

Table 3. Experiment set-up.

RS	Modelling lang.	Modelling tool
Ecore meta-models	Ecore	UML designer (Sirius)
UML class diagrams	UML class diagrams	UML tree editor
	ER diagrams	Ecore tools (Sirius)
	IFML	Ecore tools (tree)
		ISD designer (Sirius)
		IFML editor (Sirius)

RQ1. Can IronMan be used to adapt existing RSs to different languages?

RQ2. Can IronMan be used to integrate existing RSs into third-party modelling tools?

Next, Sec. 5.1 describes the experiment set-up, Sec. 5.2 reports the results, Sec. 5.3 answers the RQs, and Sec. 5.4 discusses threats to the validity of the experiment.

5.1 Experiment Set-Up

To answer the RQs, we used two RSs for Ecore and UML reported in [2] and [4]. Both recommend attributes and operations for classes. Since both were already deployed as services, their adaptation to make them conformant to the API described in Sec. 3.2 was very light.

The aim of our evaluation is twofold. On the one hand, we aim to assess if the RSs can be adapted to other modelling languages. In particular, we check if the RSs can be adapted to UML, Ecore, Entity relationship (ER) diagrams, and the Interaction Flow Modelling Language (IFML) [30]. The three first languages are widely-used structural notations to define software systems, modelling languages, and databases. IFML is an OMG standard to define the content, user interaction and behaviour of the front-end of software applications. On the other hand, we want to assess the integration of the RSs into existing tree and Sirius editors built by third parties.

Table 3 summarises the experiment set-up. In the experiment, each RS (e.g., for Ecore) was adapted to the other three languages (e.g., UML, ER and IFML) and integrated in all the six tools. This resulted in twelve integrations, covering environments based both on Sirius and the tree editor.

5.2 Experiment Results

Table 4 summarises the results of the integrations, including the number of mappings needed to adapt the RS to the modelling language, and the synthesised lines of code (LoC) by the code generator. We did not need any mapping when using a RS for the same language (e.g., Ecore for Ecore), while for the other cases, we required from 5 to 9 mappings. Since the original RSs recommend both attributes and operations, the number of mappings depended on whether the language had a notion akin to operations (absent both in ER and IFML).

The generated plugins use the EMF reflective API [59], and in average, 464 LoC were generated per plugin. This number does not include the implementation code to communicate with the recommender services. Additionally, in the case of Sirius, IronMan generates an *odesign* model automatically, which is the file that contains the description of the modelling environment, including the recommendation layer.

Table 4. Summary of the experiment results.

Integr.	RS	Language	Editor	Maps	LoC
1	Ecore	Ecore	Ecore-tree	0	455
2	Ecore	Ecore	Ecore-Sirius	0	528
3	Ecore	UML	UML-tree	9	457
4	Ecore	UML	UML-Sirius	9	530
5	Ecore	ER	ISD-Sirius	5	414
6	Ecore	IFML	IFML-Sirius	5	414
7	UML	Ecore	Ecore-tree	9	451
8	UML	Ecore	Ecore-Sirius	9	525
9	UML	UML	UML-tree	0	452
10	UML	UML	UML-Sirius	0	525
11	UML	ER	ISD-Sirius	5	411
12	UML	IFML	IFML-Sirius	5	411

Fig. 13 contains some screenshots of the resulting integrations. Labels 1–3 show the integration of the Ecore RS with the Sirius editor provided by Ecore tools. Label 1 shows the menu to activate the recommendation layer, label 2 the menu contribution of the recommender, and label 3 the dialog from which to choose the recommendations. Label 4 displays the integration of the UML RS with the UML tree editor. Finally, label 5 displays the integration of the UML RS within the Information System Designer (ISD) [42]. The resulting plugins are available at: https://github.com/antoniogarmendia/integrate-recommenders-ironman.

5.3 Answering the RQs

Overall, we can answer both RQs positively.

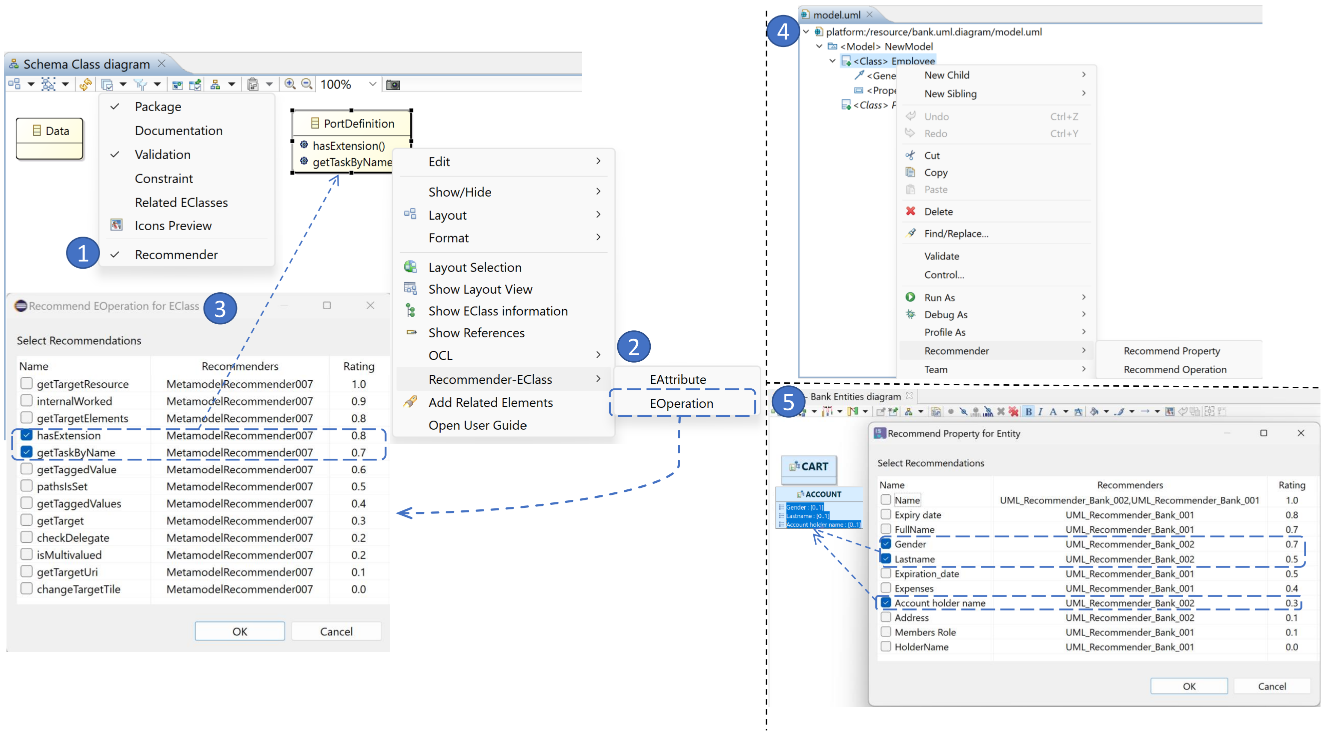
For RQ1, we could reuse RSs defined for Ecore or UML, and adapt them to other three languages (Ecore, UML, ER, IFML). The only requirement for this reuse was to map (subsets of) the meta-model of the RS and the meta-model of the target language, by defining between 0 and 9 declarative mappings.

For RQ2, we could automatically integrate each RS into six existing modelling tools based on Sirius and EMF tree editors. Remarkably, all the tools were built by third parties, and we did not need their source code.

5.4 Discussion and Threats to Validity

Our experiment shows evidence that IronMan is able to reuse existing RSs for other modelling languages, and integrate them automatically into existing tools.

However, regarding *external validity threats*, the number of RSs reused was limited, as the experiment only considered RSs for Ecore and UML. Similarly, we reused these RSs just for four other modelling languages. A stronger evaluation would be obtained by considering more RSs and more languages. In particular, the languages in the experiment were somewhat similar, and we are aware that considering more structurally different notations would require from a



Recommend EOperation for EClass

Select Recommendations

Name	Recommenders	Rating
☐ getTargetResource	MetamodelRecommender007	1.0
☐ internalWorked	MetamodelRecommender007	0.9
☐ getTargetElements	MetamodelRecommender007	0.8
✓ hasExtension	MetamodelRecommender007	0.8
✓ getTaskByName	MetamodelRecommender007	0.7
☐ getTaggedValue	MetamodelRecommender007	0.6
☐ pathsIsSet	MetamodelRecommender007	0.5
☐ getTaggedValues	MetamodelRecommender007	0.4
☐ getTarget	MetamodelRecommender007	0.3
☐ checkDelegate	MetamodelRecommender007	0.2
☐ isMultivalued	MetamodelRecommender007	0.2
☐ getTargetUri	MetamodelRecommender007	0.1
☐ changeTargetTile	MetamodelRecommender007	0.0

OK Cancel

Recommend Property for Entity

Select Recommendations

Name	Recommenders	Rating
☐ Name	UML_Recommender_Bank_002,UML_Recommender_Bank_001	1.0
☐ Expiry date	UML_Recommender_Bank_001	0.8
☐ FullName	UML_Recommender_Bank_001	0.7
✓ Gender	UML_Recommender_Bank_002	0.7
✓ Lastname	UML_Recommender_Bank_002	0.5
☐ Expiration_date	UML_Recommender_Bank_001	0.5
☐ Expenses	UML_Recommender_Bank_001	0.4
✓ Account holder name	UML_Recommender_Bank_002	0.3
☐ Address	UML_Recommender_Bank_002	0.1
☐ Members Role	UML_Recommender_Bank_001	0.1
☐ HolderName	UML_Recommender_Bank_001	0.0

OK Cancel

Figure 13. Screenshots of the integration results. (1–3) Ecore tools (Sirius), (4) UML tree editor, (5) ISD-Designer.

more powerful mapping mechanism (e.g., based on OCL or Java), able to bridge the structural dissimilarities between the meta-models. This is future work. However, to mitigate this threat, we reused the RSs for well-known modelling languages developed by third parties, which are representative of structural modelling. Moreover, we also considered IFML, which is related to interaction modelling.

Regarding the integration with tools, we chose existing tools built by third parties to avoid any bias. Still, integration with further tools would result in a stronger evaluation.

6 Related Work

Next, we review three lines of works related to our proposal: RSs for modelling tasks (Sec. 6.1), servitisation of RSs (Sec. 6.2), and aggregation of recommendations (Sec. 6.3).

6.1 RSs for Modelling

As reported in [3], the modelling community is showing a growing interest in RSs for modelling tasks. According to [3], the most common purposes of RSs in model-driven engineering (MDE) are the completion, finding, repair, reuse and, to a lesser extent, creation of modelling artefacts.

RSs for modelling have been normally developed ad-hoc for a specific modelling language, most frequently UML (e.g., class diagrams [10, 25, 28, 37, 40, 55] and sequence diagrams [13]), process modelling notations [16, 32, 33, 36, 38], or meta-models [1, 17]. Only a few language-independent approaches [1, 2, 23, 34, 47, 58] enable the definition of RSs for any language defined by a meta-model. Among them, Almonte et al. [2] propose the DSL *Droid* to facilitate the construction and subsequent evaluation of RSs for any modelling language. This DSL supports the selection and configuration of the recommendation algorithm. In this paper, we have evaluated our proposal using some RSs created with *Droid*. Given that RSs are typically fixed for a particular language, a tool like IronMan, which can adapt a RS for different notations, becomes useful in practice.

Outside the MDE community, tools have been proposed that, like Droid, simplify the creation and evaluation of RSs. For instance, *LEV4REC* [20] relies on MDE to configure the parts of a RS and generate the RS code; and the framework *Elliot* [5] executes a complete experimental pipeline for RSs by processing a simple configuration file. However, the RSs produced by these tools are neither specific for modelling nor automatically integrated into existing editors/tools.

Regarding the techniques to generate recommendations in MDE, the most popular ones stem from classical recommendation methods, most prominently knowledge-based techniques (i.e., systems that exploit the domain knowledge to produce recommendations, such as *AMOR* [10], *Baya* [14], *IPSE* [28], *RapMOD* [37], *Refacola* [58], *ReVision* [44] and Savary-Leblanc's recommender [55]), followed by content-based (e.g., *DoMoRe* [1] and *Refactory* [49]), hybrid (e.g., SBPR [32] and the approaches by Kögel et al. [34] and Koschmider et al. [36]), and based on collaborative filtering (e.g., *MemoRec* [17] and *ModBud* [53]). Some recent approaches apply machine learning to build the RSs. For example, Burgueño et al. [11] propose a RS for class diagrams based on natural language processing; Weyssow et al. [61] apply a deep learning model to recommend meta-model concepts; Di Rocco et al. [19] use an encoder-decoder network to aid modellers in executing model editing operations, and graph neural networks (GNNs) to assist in the specification of (meta-)models [18]; and Shilov et al. [56] use GNNs to assist in enterprise modelling processes.

Altogether, there is a wide variety of RSs for diverse modelling notations and methods. Our contribution in this line of works is a proposal to homogenise the access to all existing approaches behind a common API. This enables the combination of approaches by aggregating their results, and facilitates the access from arbitrary clients. Finally, our mappings permit adapting existing RSs to other modelling notations.

6.2 Deployment of RSs via Web Services

The idea of deploying RSs as web services is not new, but it has been adopted by both researchers and companies due to its benefits. A recommendation API is a (REST) service with recommendation functionalities akin to those of recommendation software libraries, but hosting the data in the cloud. The RSs community has coined the term *Recommendations-As-a-Service* (RaaS) [52] to refer to cloud platforms that enable the creation of RSs using a few clicks or LoC, by automating the steps of the recommendation generation process, from data indexing to recommendation generation and display. As an example, the engine *Recombee* [48] allows building recommendation services for any domain that has a catalogue of items and is interacted by users. The engine only supports content-based recommendation, but its recommendation model is customisable and permits defining business rules to filter or boost items based on their properties. The engine provides API endpoints to manage the (JSON-based) items, users and their interactions, and to get recommendations. While *Recombee* is generic, some RaaS are domain-specific, like *bX* [12], *BibTip* [9] and *Mr. DLib* [41] for digital libraries. All these approaches expose recommendation APIs as web services; however, the APIs are not modelling-specific, so their fine-tuning for modelling tasks is cumbersome.

Regarding RSs for modelling, most proposals are deployed locally and integrated ad-hoc in a specific modelling tool or IDE. One of the few exceptions are *Droid* [2] and Savary-Leblanc's recommender for UML [55]. The latter is a recommender for UML class diagrams, deployed as a service, and integrated within Papyrus. *Droid* permits the creation of RSs for modelling languages and their deployment as REST services. In [4], the authors illustrate the integration of this service in both EMF tree editors and a modelling chatbot. Instead, our approach aims to be more general by unifying any modelling recommendation service under a common API. This will facilitate the reuse and aggregation of existing RSs by the community, hence contributing to better modelling tools augmented with recommendation capabilities.

6.3 Aggregation of RSs

Rank aggregation has been used in a wide number of fields, such as meta-search engines [22], biology [35], criticality analysis [45], or spam detection [63], to name a few. A few proposals exist in the RSs literature as well. For example, based on the observation that many top-N recommenders disagree in their returned rankings, Oliveira et al. [46] studied 19 rank aggregation methods and identified the recommendation scenarios where they performed best or worst. They concluded that rank aggregation achieves the biggest improvements in scenarios with high-quality input rankings and high diversity; unsupervised methods should be avoided in case of poor-quality input rankings; and the results of both supervised and unsupervised methods is similar in case of input rankings with high-quality but low diversity.

To our knowledge, our proposal is the first one enabling the aggregation of recommendations for modelling. We currently support unsupervised methods, and leave supervised ones as future work.

7 Conclusions and Future Work

The increasing number of RSs for modelling calls for mechanisms to facilitate their reuse, combination and integration into modelling tools. We have proposed an approach towards this goal, based on a common recommendation API, mappings bridging the RSs and the modelling notations, and rank aggregation algorithms. The approach has been realised in IronMan, which is able to adapt and integrate RSs within Eclipse modelling tools based on Sirius and tree editors.

In the future, we would like to integrate other existing RSs, investigate the scenarios where aggregating RSs are beneficial, and extend our proposal with supervised rank aggregation methods [46]. We would also like to include more flexible means for adapting the RSs to the modelling language (e.g., using OCL or Java snippets).

Implementation-wise, the wizard requires that all selected RSs are defined for the same modelling language, and then adapted if needed. In the short term, we will support the selection of RSs for different languages, and provide assistants to adapt them to the modelling language (once the first mapping is defined). We also plan to support integration with textual notations, e.g., defined using Xtext [8, 62]. Finally, we are interested in exploring other types of integration of the RSs within modelling tools, e.g., a proactive approach where the RS monitors the modelling session and triggers recommendations when an opportunity is found.

Acknowledgments

Work supported by the Spanish MICINN (PID2021-122270OB-I00, TED2021-129381B-C21).

References

[1] Henning Agt-Rickauer, Ralf-Detlef Kutsche, and Harald Sack. 2018. DoMoRe - A Recommender System for Domain Modeling. In *6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. SciTePress, 71–82.

[2] Lissette Almonte, Esther Guerra, Iván Cantador, and Juan de Lara. 2022. Building recommenders for modelling languages with Droid. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 1–4.

[3] Lissette Almonte, Esther Guerra, Iván Cantador, and Juan de Lara. 2022. Recommender systems in model-driven engineering. *Softw. Syst. Model.* 21, 1 (2022), 249–280.

[4] Lissette Almonte, Sara Pérez-Soler, Esther Guerra, Iván Cantador, and Juan de Lara. 2021. Automating the Synthesis of Recommender Systems for Modelling Languages. In *14th ACM SIGPLAN International Conference on Software Language Engineering (SLE)*. ACM, 22–35.

[5] Vito Walter Anelli, Alejandro Bellogín, Antonio Ferrara, Daniele Malitesta, Felice Antonio Merra, Claudio Pomo, Francesco Maria Donini, and Tommaso Di Noia. 2021. Elliot: A Comprehensive and Rigorous Framework for Reproducible Recommender Systems Evaluation. In *44th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*. ACM, 2405–2414.

[6] Apache. (last accessed in July 2023). Tomcat. http://tomcat.apache.org/.

[7] Javed A. Aslam and Mark H. Montague. 2001. Models for Metasearch. In *International Conference on Research and Development in Information Retrieval (SIGIR)*. ACM, 275–284.

[8] Lorenzo Bettini. 2016. *Implementing domain-specific languages with Xtext and Xtend.* Packt Publishing Ltd.

[9] BibTip. (last accessed in July 2023). http://www.bibtip.com/en.

[10] Petra Brosch, Martina Seidl, and Gerti Kappel. 2010. A recommender for conflict resolution support in optimistic model versioning. In *25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*. ACM, 43–50.

[11] Loli Burgueño, Robert Clarisó, Sébastien Gérard, Shuai Li, and Jordi Cabot. 2021. An NLP-Based Architecture for the Autocompletion of Partial Domain Models. In *33rd International Conference on Advanced Information Systems Engineering (CAiSE) (LNCS, Vol. 12751)*. Springer, 91–106.

[12] bX. (last accessed in July 2023). https://www.exlibrisgroup.com/products/bx-recommender/.

[13] Thaciana Cerqueira, Franklin Ramalho, and Leandro Balby Marinho. 2016. A Content-Based Approach for Recommending UML Sequence Diagrams. In *28th International Conference on Software Engineering and Knowledge Engineering (SEKE)*. KSI Research Inc. and Knowledge Systems Institute Graduate School, 644–649.

[14] Soudip Roy Chowdhury, Florian Daniel, and Fabio Casati. 2014. Recommendation and Weaving of Reusable Mashup Model Patterns for Assisted Development. *ACM Transactions on Internet Technology* 14, 2-3 (2014), 21:1–21:23.

[15] Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. 2013. Reusable abstractions for modeling languages. *Inf. Syst.* 38, 8 (2013), 1128–1149.

[16] ShuiGuang Deng, Dongjing Wang, Ying Li, Bin Cao, Jianwei Yin, Zhaohui Wu, and Mengchu Zhou. 2017. A Recommendation System to Facilitate Business Process Modeling. *IEEE Trans. Cybern.* 47, 6 (2017), 1380–1394.

[17] Juri Di Rocco, Davide Di Ruscio, Claudio Di Sipio, Phuong T. Nguyen, and Alfonso Pierantonio. 2023. MemoRec: A recommender system for assisting modelers in specifying metamodels. *Softw. Syst. Model.* to appear (2023).

[18] Juri Di Rocco, Claudio Di Sipio, Davide Di Ruscio, and Phuong T. Nguyen. 2021. A GNN-based Recommender System to Assist the Specification of Metamodels and Models. In *24th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*. IEEE, 70–81. https://doi.org/10.1109/MODELS50736.2021.00016

[19] Juri Di Rocco, Claudio Di Sipio, Phuong T. Nguyen, Davide Di Ruscio, and Alfonso Pierantonio. 2022. Finding with NEMO: A Recommender System to Forecast the next Modeling Operations. In *25th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*. ACM, 154–164.

[20] Claudio Di Sipio, Juri Di Rocco, Davide Di Ruscio, and Phuong Thanh Nguyen. 2021. A Low-Code Tool Supporting the Development of Recommender Systems. In *15th Conference on Recommender Systems (RecSys)*. ACM, 741–744.

[21] Claudio Di Sipio, Juri Di Rocco, Davide Di Ruscio, and Phuong T. Nguyen. 2023. MORGAN: a modeling recommender system based on graph kernel. *Softw. Syst. Model.* to appear (2023), 70–81.

[22] Cynthia Dwork, Ravi Kumar, Moni Naor, and D. Sivakumar. 2001. Rank aggregation methods for the Web. In *International World Wide Web Conference, WWW*. ACM, 613–622.

[23] Andrej Dyck, Andreas Ganser, and Horst Lichter. 2014. A framework for model recommenders - Requirements, architecture and tool support. In *MODELSWARD*. IEEE, 282–290.

[24] Eclipse. (last accessed in July 2023). Jersey. https://eclipse-ee4j.github.io/jersey/.

[25] Akil Elkamel, Mariem Gzara, and Hanêne Ben-Abdallah. 2016. An UML class recommender system for software design. In *13th IEEE/ACS International Conference of Computer Systems and Applications (AICCSA)*. IEEE Computer Society, 1–8.

[26] Ronald Fagin, Ravi Kumar, and D. Sivakumar. 2003. Efficient similarity search and classification via rank aggregation. In *International Conference on Management of Data ()*. ACM, 301–312.

[27] Mohamed Farah and Daniel Vanderpooten. 2007. An outranking approach for rank aggregation in information retrieval. In *SIGIR 2007: Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 591–598.

[28] Hilke Garbe. 2012. Intelligent Assistance in a Problem Solving Environment for UML Class Diagrams by Combining a Generative System with Constraints. In *eLearning*. IADIS.

[29] Object Management Group. 2014. OCL Specification. http://www.omg.org/spec/OCL/.

[30] Object Management Group. 2015. IFML Specification. https://www.ifml.org/.

[31] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical Report CMU/SEI-90-TR-021. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

[32] Hadjer Khider, Slimane Hammoudi, and Abdelkrim Meziane. 2020. Business Process Model Recommendation as a Transformation Process in MDE: Conceptualization and First Experiments. In *8th International Conference on Model-Driven Engineering and Software Development, MODELSWARD*. SCITEPRESS, 65–75.

[33] Krzysztof Kluza, Mateusz Baran, Szymon Bobek, and Grzegorz J. Nalepa. 2013. Overview of Recommendation Techniques in Business Process Modeling. In *9th Workshop on Knowledge Engineering and Software Engineering, KESE (CEUR Workshop Proceedings, Vol. 1070)*. CEUR-WS.org.

[34] Stefan Kögel. 2017. Recommender system for model driven software development. In *11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 1026–1029.

[35] Raivo Kolde, Sven Laur, Priit Adler, and Jaak Vilo. 2012. Robust rank aggregation for gene list integration and meta-analysis. *Bioinform.* 28, 4 (2012), 573–580.

[36] Agnes Koschmider, Thomas Hornung, and Andreas Oberweis. 2011. Recommendation-based editor for business process modeling. *Data Knowl. Eng.* 70, 6 (2011), 483–503.

[37] Tobias Kuschke, Patrick Mäder, and Patrick Rempel. 2013. Recommending Auto-completions for Software Modeling Activities. In *16th International Conference on Model-Driven Engineering Languages and Systems (MoDELS) (LNCS, Vol. 8107)*. Springer, 170–186.
[38] Ying Li, Bin Cao, Lida Xu, Jianwei Yin, ShuiGuang Deng, Yuyu Yin, and Zhaohui Wu. 2014. An Efficient Recommendation Method for Improving Business Process Modeling. *IEEE Trans. Ind. Informatics* 10, 1 (2014), 502–513.
[39] Tie-Yan Liu. 2011. *Learning to Rank for Information Retrieval*. Springer.
[40] Patrick Mäder, Tobias Kuschke, and Mario Janke. 2021. Reactive Auto-Completion of Modeling Activities. *IEEE Trans. Software Eng.* 47, 7 (2021), 1431–1451.
[41] Mr-DLib. (last accessed in July 2023). http://mr-dlib.org/.
[42] Obeo. (last accessed in July 2023). IS-designer. https://www.obeosoft.com/en/products/is-designer.
[43] Obeo. (last accessed in July 2023). UML Designer. https://marketplace.eclipse.org/content/uml-designer.
[44] Manuel Ohrndorf, Christopher Pietsch, Udo Kelter, Lars Grunske, and Timo Kehrer. 2021. History-Based Model Repair Recommendations. *ACM Trans. Softw. Eng. Methodol.* 30, 2, Article 15 (2021), 46 pages. https://doi.org/10.1145/3419017
[45] Gabriele Oliva, Annunziata Esposito Amideo, Stefano Starita, Roberto Setola, and Maria Paola Scaparra. 2019. Aggregating Centrality Rankings: A Novel Approach to Detect Critical Infrastructure Vulnerabilities. In *International Conference on Critical Information Infrastructures Security, CRITIS (LNCS, Vol. 11777)*. Springer, 57–68.
[46] Samuel E. L. Oliveira, Victor Diniz, Anísio Lacerda, Luiz H. C. Merschmann, and Gisele L. Pappa. 2020. Is rank aggregation effective in recommender systems? An experimental analysis. *ACM Trans. Intell. Syst. Technol.* 11, 2 (2020), 16:1–16:26.
[47] Tanumoy Pati, Sowmya Kolli, and James H. Hill. 2017. Proactive modeling: a new model intelligence technique. *Softw. Syst. Model.* 16, 2 (2017), 499–521.
[48] Recombee. (last accessed in July 2023). https://docs.recombee.com/.
[49] Jan Reimann, Mirko Seifert, and Uwe Aßmann. 2013. On the reuse and recommendation of model refactoring specifications. *Softw. Syst. Model.* 12, 3 (2013), 579–596.
[50] Francesco Ricci, Lior Rokach, and Bracha Shapira. 2022. *Recommender Systems Handbook* (3 ed.). Springer US.
[51] Martin P. Robillard and Robert J. Walker. 2014. An Introduction to Recommendation Systems in Software Engineering. In *Recommendation Systems in Software Engineering*, Martin P. Robillard, Walid Maalej, Robert J. Walker, and Thomas Zimmermann (Eds.). Springer, 1–11.
[52] RS_c. (last accessed in July 2023). Recommendation-as-a-service. https://recommender-systems.com/resources/recommendations-as-a-service-raas/.
[53] Rijul Saini, Gunter Mussbacher, Jin L. C. Guo, and Jörg Kienzle. 2019. Teaching Modelling Literacy: An Artificial Intelligence Approach. In *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS*. IEEE, 714–719.
[54] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. 2012. Flexible Model-to-Model Transformation Templates: An Application to ATL. *J. Object Technol.* 11, 2 (2012), 4: 1–28.
[55] Maxime Savary-Leblanc, Xavier Le-Pallec, and Sébastien Gérard. 2021. A modeling assistant for cognifying MBSE tools. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 630–634. https://doi.org/10.1109/MODELS-C53483.2021.00097
[56] Nikolay Shilov, Walaa Othman, Michael Fellmann, and Kurt Sandkuhl. 2023. Machine learning for enterprise modeling assistance: An investigation of the potential and proof of concept. *Softw. Syst. Model.* to appear (2023), 1619–1374. https://doi.org/10.1007/s10270-022-01077-y
[57] Sirius. (last accessed in April 2023). https://www.eclipse.org/sirius/.
[58] Friedrich Steimann and Bastian Ulke. 2013. Generic Model Assist. In *16th International Conference on Model-Driven Engineering Languages and Systems, MODELS (LNCS, Vol. 8107)*. Springer, 18–34.
[59] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modeling Framework, 2^{nd} Edition*. Addison-Wesley Professional. See also http://www.eclipse.org/modeling/emf/.
[60] Javier Alvaro Vargas Muñoz, Ricardo da Silva Torres, and Marcos André Gonçalves. 2015. A Soft Computing Approach for Learning to Aggregate Rankings. In *Proceedings of the 24th ACM International Conference on Information and Knowledge Management, CIKM*. ACM, 83–92.
[61] Martin Weyssow, Houari Sahraoui, and Eugene Syriani. 2022. Recommending Metamodel Concepts during Modeling Activities with Pre-Trained Language Models. *Softw. Syst. Model.* 21, 3 (2022), 1071–1089.
[62] Xtext. (last accessed in July 2023). http://www.eclipse.org/Xtext/.
[63] Zheng Zhang, Ming-Yang Zhou, Jun Wan, Kezhong Lu, Guoliang Chen, and Hao Liao. 2023. Spammer detection via ranking aggregation of group behavior. *Expert Syst. Appl.* 216 (2023), 119454.

Received 2023-07-07; accepted 2023-09-01

GPT-3-Powered Type Error Debugging: Investigating the Use of Large Language Models for Code Repair

Francisco Ribeiro
francisco.j.ribeiro@inesctec.pt
HASLab/INESC TEC
University of Minho
Braga, Portugal

José Nuno Castro de Macedo
jose.n.macedo@inesctec.pt
HASLab/INESC TEC
University of Minho
Braga, Portugal

Kanae Tsushima
k_tsushima@nii.ac.jp
National Institute of Informatics
Sokendai University
Tokyo, Japan

Rui Abreu
rui@computer.org
INESC-ID
University of Porto
Porto, Portugal

João Saraiva
saraiva@di.uminho.pt
HASLab/INESC TEC
University of Minho
Braga, Portugal

Abstract

Type systems are responsible for assigning types to terms in programs. That way, they enforce the actions that can be taken and can, consequently, detect type errors during compilation. However, while they are able to flag the existence of an error, they often fail to pinpoint its cause or provide a helpful error message. Thus, without adequate support, debugging this kind of errors can take a considerable amount of effort. Recently, neural network models have been developed that are able to understand programming languages and perform several downstream tasks. We argue that type error debugging can be enhanced by taking advantage of this deeper understanding of the language's structure. In this paper, we present a technique that leverages GPT-3's capabilities to automatically fix type errors in *OCaml* programs. We perform multiple source code analysis tasks to produce useful prompts that are then provided to GPT-3 to generate potential patches. Our publicly available tool, Mentat, supports multiple modes and was validated on an existing public dataset with thousands of *OCaml* programs. We automatically validate successful repairs by using *Quickcheck* to verify which generated patches produce the same output as the user-intended fixed version, achieving a 39% repair rate. In a comparative study, Mentat outperformed two other techniques in automatically fixing ill-typed *OCaml* programs.

CCS Concepts: • **Software and its engineering → Software evolution**; *Software creation and management*; Software post-development issues.

Keywords: Automated Program Repair, GPT-3, Fault Localization, Code Generation

ACM Reference Format:
Francisco Ribeiro, José Nuno Castro de Macedo, Kanae Tsushima, Rui Abreu, and João Saraiva. 2023. GPT-3-Powered Type Error Debugging: Investigating the Use of Large Language Models for Code Repair. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3623476.3623522

1 Introduction

Programming languages usually have an associated type system responsible for determining whether some operation can be applied to some program term. This system ensures a program's correctness in terms of type safety. That is, if a program does not *typecheck*, it signals a logical error related to the inherent type constraints. However, even after knowing there is some type inconsistency, we still need to understand where and why that error occurred. In other words, a type system may be unable to provide the location of the error and the explanation as to why the error arose.

Undeniably, programmers are not completely left in the dark in this regard. Several programming languages provide *type inference* systems, which compute the expected type of expressions in the code. Despite considerable effort [9, 21, 24, 43, 47, 49] to provide helpful information for type error detection, compilers often fail to pinpoint the true cause of an error. Consider the following ill-typed *OCaml* program:

```
let rec add_list lst = match lst with
  | [] -> []
  | fst :: rest -> fst + (add_list rest)
```

Program 1. Ill-typed function: patterns differ on returned types

Program 1 consists of a recursive function `add_list` that takes a list of integers and should calculate the sum of all numbers. `ocamlc` would yield the following message[1]:

```
3 |       | fst :: rest -> fst + (add_list rest)
                                 ^^^^^^^^^^^^^^^
Error: This expression has type 'a list
       but an expression was expected of type int
```

The type system successfully detects a type error in the program and the compiler provides a message reporting the problem. If we replace the use of the *plus* operator (+) in line 3 by the *cons* operator (`::`), the whole program is well-typed. However, the expression highlighted as being problematic by the compiler is not the true origin of the error. As a consequence, the information about the mismatch of the expected type (`int`) and the inferred type (`'a list`) does not provide meaningful advice into how to approach the problem. Another way to fix this program, which corresponds to the programmer's intended fix, is to have it return zero (0) instead of the empty list (`[]`) in line 2. Hence, it often happens that the user's intended modification differs from what the compiler points out. That is because reporting type inconsistencies is influenced by the order in which expressions in a program show up. As long as no inconsistencies are detected, the inferred type for an expression is considered to be correct. As a result, the type system will have a *left-to-right* bias and errors tend to show up towards the end of a program [20]. Now consider that we swap the two patterns:

```
let rec add_list lst = match lst with
  | fst :: rest -> fst + (add_list rest)
  | [] -> []
```

This time, we get a different error message:

```
3 |       | [] -> []
                 ^^
Error: This expression has type 'a list
       but an expression was expected of type int
```

This means that the type error we are dealing with can have multiple causes. Depending on the order of the patterns, the cause that is reported changes.

However, even after recognizing the inherent limitations of type systems in accurately locating and explaining type inconsistencies, we are still left with fixing them. Automated program repair (APR) aims to generate patches for incorrect programs (either syntactically or semantically) with minimal human intervention [18]. Many approaches have emerged based on the competent programmer hypothesis or, put in other words, programmers "create programs that are close to being correct!" [13]. We argue that automatically finding repairs that eliminate type inconsistencies is one effective way of locating and understanding the root of a type error.

In this paper, we present an approach that leverages the code understanding and generation capabilities of models based on GPT-3 to automatically fix type errors in *OCaml* programs. Our focus is on analyzing the source code of ill-typed programs and generating prompts that are then provided to the model. By doing this, we aim to produce programs that are free from type errors and, thus, can be used to find and understand what was causing them. Our contributions are:

1. a source code analysis and manipulation technique that produces different kinds of prompts intended for GPT-3-based models (Section 3);
2. a publicly available tool, named Mentat, implementing this technique (Section 4);
3. an initial validation on a small set of programs, followed by a large-scale evaluation on an independent repository, with an analysis of the results obtained (Section 5);
4. a comparative study between our tool, Mentat, and two other techniques, namely Rite [41] and Seminal [25] on a common dataset, alongside the obtained insights.

Even though this work is concerned with type error debugging, it differs from previous approaches, which aim to improve the quality of type error messages [12, 24, 51], provide interactive type debugging [6, 7, 9, 48], and narrow down the area for type error debugging [19, 37, 42–44]. Instead, our work focuses on the automatic repair of type errors. We achieve this by analyzing and transforming source code and outputting it in a form that can be understood and processed by GPT-3, a large language model trained by *OpenAI*.

For the initial validation, we find that our tool presents at least one valid solution for each test program, with the *Fill* operation mode obtaining success rates varying from 53% to 60% for simple programs and from 83% to 100% for Dijkstra algorithm implementations. Regarding the large scale evaluation, we analyzed 1,318 buggy programs and were able to fix 516 of them, reaching a 39% repair rate. To automate this process we used two key features of property-based testing [10]: firstly, we automatically generate a very large number of random inputs, and secondly we define a property that tests whether the user-fixed program outputs the same result as the automatically repaired one. The program-specific property is also automatically generated, thus having a fully automated large scale validation process without relying on human intervention to inspect the generated patches. Also, we showed the potential for partial fixes by considering the results for programs that do not pass 100% of test cases. While the other operation modes perform worse overall when compared to *Fill*, they are still capable of generating successful results and, in some cases, succeed where *Fill* fails. Moreover, we performed a comparative study of our technique with two type repairing approaches, namely Rite [41] and Seminal [25]. Our first results show that Mentat gives the best program repair results with a 37.5% repair rate versus 33.4% from Rite and 7.8% from Seminal.

2 Background

A *type system* is a set of rules governing how data is represented and used in a program [34]. It is lightweight and does

[1]In *OCaml*, polymorphic type names are prefixed with a backquote.

not require special knowledge from the user. *Type inference* is a static algorithm to find the type of each part of a program without the programmer's annotation. The advantages of type systems include not only type-safety of programs but also efficient computation by enabling the generation of optimized code. For this reason, *Ruby*, a dynamically typed language, has recently introduced type inference, and *Python* has also introduced type-related features.

Originally, research in natural language processing (NLP) focused on ways of processing, analyzing, and manipulating natural language through statistical and rule-based modeling. More recently, the use of artificial intelligence has allowed the development of techniques that make NLP one of the most prominent fields in computer science. Simply put, NLP is responsible for encoding text into more appropriate machine level representations and also for processing and transforming these lower level descriptions into other forms of text. More specifically, neural networks have been very impactful in the development of NLP models. Some of the most important ones are BERT [14] and GPT [36] which have seen their utility displayed in an overwhelming amount of more specific downstream tasks. Encouraged by the achievements conducted in this area, the software engineering community has successfully applied some of NLP's fundamentals to build tools that improve software development workflow. Code completion is one of the most popular features and many code editors implement it one way or another. With the intent of going beyond basic level completions like more pertinent suggestions for API calls for a given context, the research community has also directed its efforts into developing versions of the BERT and GPT models that are specifically tailored towards programming languages. New models, such as CodeBERT [17] and CodeGPT [27], were created based on the original architectures. GPT-based code models are able to generate long and relatively complex code sequences by analyzing and inferring the context of the source code provided as input. One of the most recent iterations of such models is GPT-3 [4], which presents a high degree of success when employed in different scenarios such as cloze and completion tasks.

3 Technique

With our contribution, we intend to, for a given faulty *OCaml* program, extract as much information from it as possible, and then format it in a way that GPT-3 can understand and process it. For this, we first check for type inconsistencies. If they are present, we employ three different tasks: *type error location*, *inlining*, and *type unification*, with each being described in detail in the following sections. Figure 1 illustrates how the tasks generally interconnect and highlights them with the corresponding label. Nodes with dashed borders represent steps in which we make use of existing components and are not directly part of our contribution. Grey nodes and white nodes represent elements and actions, respectively. Depending on the way we wish to interact with GPT-3, the tasks may be combined in slightly different ways.

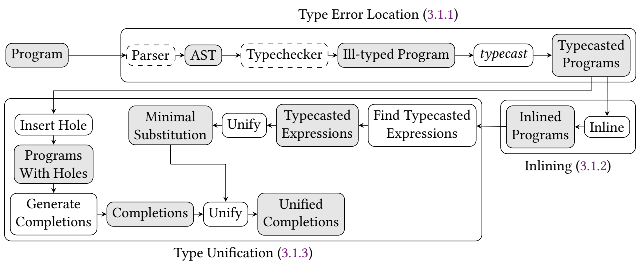


Figure 1. Interconnection of source code manipulation tasks

3.1 Source Code Analysis

3.1.1 Type Error Location. The compilers of strongly typed programming languages tend to check for source code errors in two separate steps when building an executable file: parsing and type checking. The parser checks whether the input is syntactically correct and if so it produces an *AST (Abstract Syntax Tree)*. The type checker traverses such a tree to check whether the underlying program obeys the type rules. If it does parse but it does not typecheck, then there is a type error, which is the focus of this work. If it does both parse and typecheck then, for our purposes, the program is considered correct.

Some programming languages offer us type conversion and manipulation tools; we focus on type conversion tools from strongly typed languages. Let us consider a function from now on referred to as *typecast*, which forcefully converts a value of any type into a value of any type. Of course, such function will not be able to actually do these conversions during a program's runtime, but it will be able to trick the compiler into interpreting an expression of one type as if it had a different type. As such, the *typecast* function will only be used when typechecking a program. In *OCaml*, this operation can be performed by using `Obj.magic`[2], which we will use in our tool. Any program referred to as *typecasted* from here onwards is a program in which part of it was transformed with the *typecast* function.

Recall the introductory example in Program 1. Because any type can be converted into any type with this function, we can, for example, apply *typecast* to the empty list (`[]`) to transform it into a different type *'b*, which the typechecker will deduce to be *int*. We could also apply *typecast* to the *plus* operator (+) thus transforming it into a function of type *'b* which the typechecker will deduce to be *'a ⟶ 'a list ⟶ 'a list*. Finally, we can also *typecast* the expression on the right-hand side and have the typechecker infer the type *'a list*. After parsing the *OCaml* program, we create multiple program

[2] `Obj.magic` has the type *'a ⟶ 'b*

variants, each with the *typecast* function applied to a single expression, and then type check each application. Every variation of the original program that typechecks correctly is stored; if changing the type of one value / expression with *typecast* fixes the program type-wise, then we consider that the replaced value can be the error that needs to be fixed. Finally, we replace the usage of this function with a mask, signaling a hole in the program that needs to be filled. The following tasks will focus on analysing and transforming the program variations (which we call the *typecasted* programs) produced in this task.

In this work, the focus is on type errors with a single location. Nonetheless, our approach is still flexible to some instances of errors with multiple locations if all of them are contained within a single function call expression.

3.1.2 Inlining. We make use of *inlining* not for the usual purposes of compiler optimisations, but to be able to make information available from some parts of the program in other places. That is, the *inlining* step we describe here is done on the actual source code to improve its analysis further ahead, and not to produce more efficient machine code. This step is particularly useful as it allows us to extract better results from later type unification and type inference tasks. Consider the following *typecasted* program:

```
let f x = (Obj.magic (&&)) x (x + x)
```

Program 2. Obj.magic hides the error from the type system

If we ask for the type of the `Obj.magic (&&)` expression, the type system will infer it to be $int \rightarrow int \rightarrow {}'a$. However, let us now extend the program with a test case:

```
let f x = (Obj.magic (&&)) x (x + x)
let t = (f 1) = 3
```

The second line specifies the usage of function `f`. By inlining function `f`, we associate it to a context in which the type system can take advantage of the extra information provided by the *int* parameter. As a result, the inferred type of the *typecasted* expression would be $int \rightarrow int \rightarrow int$.

To accomplish this *inlining* step, an environment is maintained throughout the underlying *AST* traversal. When a new definition is found, its identifier is stored and associated to the corresponding expression. As such, when the usage of an element stored in the environment is detected, the usage of the identifier is replaced by the expression's body, effectively *inlining* that piece of code. Special care needs to be taken for two scenarios: recursion and function arguments. For the first one, we need to avoid repeatedly *inlining* the same element as that could potentially lead to a non-terminating procedure. Nonetheless, there is still interest in performing this step once for recursive definitions. Thus, we allow *inlining* to happen exactly once in such cases. For the second scenario, most programming languages allow re-definition of variables in different scope levels and *OCaml* is no exception. It is possible to have a variable `x` already defined, and still define a new `x` in an inner scope. When inlining, in this case, we take care to inline the correct definition for the correct `x` variable. The inlined source code is only stored in memory and the original program is not modified, with GPT-3 never seeing the inlined version.

3.1.3 Type Unification. We make use of type unification to filter elements from a list of completion suggestions. Almost every code editor provides the ability to have completion suggestions on request from the user by specifying a place in the source code. Prior to requesting a list of completion suggestions, we replace *typecasted* expressions (obtained as described in Section 3.1.1) with typed holes. Then, we make use of an *OCaml language server (LSP)*[3] to automatically locate the introduced typed hole and to obtain a list of suggestions containing code elements that may fit. Let us consider Program 2 and its equivalent version with a typed hole represented by the underscore:

```
let f = fun x -> _ x (x + x)
```

Having introduced the typed hole, we can request a list of suggestions for the typed hole's location from the *OCaml LSP* and obtain 318 candidates. This list is not curated according to the type context and, as such, the suggested completions may present a type mismatch. In order to filter the list accordingly, type unification is performed between each element and the expression that was flagged as problematic according to the application of *typecast*. If unification succeeds, the suggestion will take part in the resulting list which, in this situation, will consist of 23 candidates.

3.2 Strategies

We make use of the available GPT-3 operation modes to implement three of the four repair strategies supported by our tool. Depending on which strategy we intend to use, we have to prepare and format the data accordingly.

3.2.1 Fill. GPT-3 provides an operation mode named *Insert*, in which, given some input text from the user which contains a hole denoted by the `[insert]` tag, a generation is produced by the model and placed where the tag was located. Thus, this operation mode is perfect for our use case, by filling programs in which a part is missing. There are several models available for this operation mode, notably `text-davinci-003` and `code-davinci-002`, the former being a general model and the latter being optimized for handling code. Next, we show an example of an input prompt for this operation mode:

```
let rec add_list lst = match lst with
  | [] -> [insert]
  | fst :: rest -> fst + (add_list rest)
```

For this prompt, using the web interface and with the default model parameters, both `text-davinci-003` and `code-davinci-002` models will output the following:

[3]https://github.com/ocaml/ocaml-lsp

```
let rec add_list lst = match lst with
  | [] -> 0
  | fst :: rest -> fst + (add_list rest)
```

For this program, we obtain the correct patch. Of course, to do so we need to first locate the error and replace it with the `[insert]` tag. We do this through the technique described in Subsection 3.1.1, by replacing the code that did not typecheck without the usage of the *typecast* function with said tag. This is the strategy in which we provide the least information to the GPT-3 model.

3.2.2 Choose. GPT-3 provides an operation mode named *Complete*, in which, after being fed input text from the user, it will attempt to generate more text based on it, that is, complete it. It can be used for non-code tasks such as writing stories or classifying tweets, as well as code tasks like translating plain text to an SQL query. We experimented with several approaches for usage of the *Complete* mode, because, unlike with the other operation modes, there is no intuitive way to use this mode to correct programs. Failed approaches include asking the model to rewrite the entire program replacing the missing hole (similar to the `[insert]` tag mentioned in Subsection 3.2.1) with the correct solution, or asking the model to just give us the code expected in that hole. Variations of this approach, by providing more clues, such as the expected type of the result, or by providing possible solutions to consider, were also unsuccessful. Ultimately, we were able to obtain favourable results by formatting the input as an exercise, similar to what would be found in a student exam. To do this, we present the source code with a missing hole denoted by the `<mask>` identifier, and a list of possible solutions. This list is produced as described in Subsection 3.1.3 and presented as numbered options, and the model is asked to select the most appropriate. We guide the model into selecting one option through *prompt engineering*. Specifically, every produced prompt is preceded with two example exercises that share this template but have the correct option selected (omitted in listings for brevity). The following program is an example of a prompt formatted for the *Complete* operation mode, as described.

```
Consider the following OCaml program:

let rec add_list lst = match lst with
 | [] -> <mask>
 | fst :: rest -> fst + (add_list rest)

Which of the following options should replace <mask>?
1) ( __LINE__ )
2) ( max_int )
3) ( min_int )

Correct option:
```

Notice that all presented options are incorrect - this is a limitation from using this kind of prompt. Because we are using the *OCaml LSP* to generate suggestions to be then presented here, we are limited in which suggestions can be included. In fact, the *OCaml LSP* will not generate common constant values such as `0`, which is the correct response here. All the listed suggestions are integer constants suggested by the *OCaml LSP*, where `__LINE__` is a compiler macro representing the code line number where it is written, and `max_int` and `min_int` are constants representing the maximum and minimum values possible to represent as integers in *OCaml*. The following is another prompt (re-formatted for brevity) we produce for the same program, but assuming an error in a different place.

```
Consider the following OCaml program:

let rec add_list lst = match lst with
 | [] -> []
 | fst::rest -> <mask> fst (add_list rest)

Which of the following options should replace <mask>?
1) ( fst )              8)  ( raise_notrace )
2) ( ! )                9)  ( snd )
3) ( exit )             10) ( @@ )
4) ( failwith )         11) ( max )
5) ( input_value )      12) ( min )
6) ( invalid_arg )      13) ( List.cons )
7) ( raise )            14) ( @ )

Correct option:
```

Notice that the list of suggestions grew — some of them, such as `exit` and `raise`, will match a lot of types, due to the polymorphic nature of these suggestions. However, some interesting suggestions are now listed, and in this case, the model suggests option **14** - the `@` operator. This operator concatenates two lists, and placing it into the hole in the source code will transform this function into a correct implementation of the `List.concat` function for joining a list of lists of values into a single list of values. It is, however, not what we tend to expect out of a function named *add_list*.

Communication with GPT-3 for this operation mode is fairly similar to other modes, but we have to limit the number of generated tokens, as the model tends to try to generate an explanation for its answer. It is also possible to specify a *stop sequence*, which is a sequence of tokens that, when generated by GPT-3, stops the whole generation process.

3.2.3 Instruct. Another way of interacting with GPT-3 is through its *Edit* mode which expects two inputs: a prompt and instructions describing how to edit the prompt. Similarly to the other modes, there is a more general textual model and a code specific variant. However, for this mode, there are specialized versions to handle text editing, namely `text-davinci-edit-001` and `code-davinci-edit-001`. Our approach uses a simplified form of the *Instruct* mode, which is applied when the step in Section 3.1.1 fails to produce a program that typechecks. In that case, the prompt consists of the original program, and the instruction will hold the message "*Fix the bug*". Alternatively, in case the previous step is able to produce a well-typed program[4], our approach

[4]Recall that whenever bypassing the type system by using the *typecast* function eliminates the type error, we explore that program variant.

performs inlining and type unification on the *typecasted program* in order to compute the minimal substitution holding the expected type with as much information as possible from the whole program. If we consider Program 1, the inputs sent to GPT-3 would be:

```
Prompt:
let rec add_list lst = match lst with
  | [] -> _
  | fst::rest -> fst + (add_list rest)
Instruction:
Replace the underscore with something of type int
```

The hole represented by the underscore is the place we wish to see filled in. Although the underscore character can appear in an *OCaml* program, we did not notice any interference in the ability of GPT-3 to apply the transformation in the intended place. The template we use for the edit instructions is "*Replace the underscore with something of type* `<inferred>`". For this program, GPT-3 responds with 0, which is the desired fix. Indeed, it may look like GPT-3 simply understands the program in question is missing the most adequate stop criteria and just answers with a corrected version, perhaps disregarding our instructions. However, the unification and inference step we perform in order to complete the message template with the expected type plays a crucial role. For the same example, fabricating messages referring illogical types such as *string* or *('a → 'b) → 'a list → 'b list* would see GPT-3 answer with the *empty string* and `List.map` respectively.

Because our approach explores every application of *typecast* that typechecks a program, we also produce another alternative:

```
Prompt:
let rec add_list lst = match lst with
 | [] -> []
 | fst::rest -> _ fst (add_list rest)
Instruction:
Replace the underscore with something of type 'a -> 'b
     list -> 'b list
```

Even though this alternative prompt will not generate the intended fixed program, it shows that the creation of adequate prompts is essential for GPT-3 to perform well. In this case, GPT-3 will respond with `(fun x y → x::y)`. Surely, integrating that piece of code into the original program produces a correct one from a typechecking perspective, although it does not fulfill the programmer's intention.

3.2.4 Without GPT-3. One interesting outcome from the implementation of the *Choose* strategy described in Section 3.2.2 is that we can make use of the work done to construct the prompt and skip the interaction with GPT-3. Thus, we provide a way to work completely offline. After coming up with an alternative program that typechecks and a list of suggestions (according to sections 3.1.1 and 3.1.3, respectively), we integrate each one into the original program. If no test cases have been provided, the tool simply displays which options fit the expected type. If there are test cases, the tool tests each suggestion and displays the resulting programs according to whether they satisfy the tests or not. Consider the following ill-typed program and the associated test case:

```
let f = fun x -> x && (x + x)
Test case: f 3 = 9
```

According to the test case, the intended fix consists of replacing the logical-and (&&) with the plus operator (+). For this case, our tool is able to filter 15 suggestions out of the 318 provided by the *OCaml LSP*, with one of them being the desired one. Each of the 15 suggestions is checked against the test case and the tool outputs the only program that satisfies the criteria:

```
let f = fun x -> (+) x (x + x)
```

Note that the type unification step requires the use of functions. In that sense, we convert the usage of operators such as '&&' to their equivalent prefix notation functions '`(&&)`', resulting in the generated patches also being written in this form, demonstrated by the use of the function '`(+)`'.

Indeed, the focus of our work is to evaluate GPT-3's performance regarding the automatic repair of type errors, and presenting a method in which the usage of the model is non-existing may seem counter-intuitive. However, we find this to be a validation of our approach, showing that the effort to assemble the prompt can guide the whole process towards the intended result as the correct patch may be found by further checking each plausible option.

3.3 Model Bias

We now experiment with providing more information, trying to guide the models into more relevant results. We do this by using the *bias* parameter which lets us guide the model's output by specifying the importance of certain tokens[5] through weights. A token represents a unit of text, like a character or a word. We use a tokenizer tool for this purpose.

We create a database of the most common tokens in the top 10 *OCaml* repositories on *GitHub* programmatically. To achieve this, we utilize GPT-3's tokenizer, which converts text into numerical sequences that the model processes. We analyze the tokens in source code files from these repositories and collect frequency data to construct a database of commonly used tokens in real-world programs.

We create a list of suggestions as per Section 3.1.3, convert them into token sequences, and assign positive weightings to these tokens. Then, we use the *bias* parameter in GPT-3 to guide the model toward these suggestions. The weightings are determined based on a database of token frequencies from real-world programs. We heuristically set minimum and maximum bias values at 1 and 3, respectively, to ensure effective guidance without extreme behavior. Figure 2 provides an overview of this process with sample values.

[5]Tokenizer available at https://beta.openai.com/tokenizer

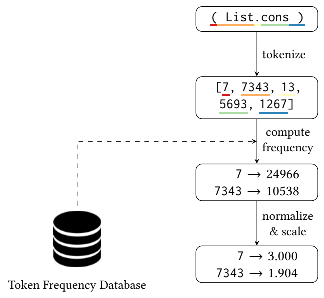


Figure 2. Bias computation for one *OCaml LSP* suggestion.

We experiment with *bias* values by comparing the *Choose* strategy with and without *bias*. In the *Choose* strategy with *bias*, we exclude suggestions from the textual prompt since their influence is already provided through *bias* values, as we show in the following example:

```
Consider the following OCaml program:

let rec add_list lst = match lst with
 | [] -> <mask>
 | fst :: rest -> fst + (add_list rest)

What should replace <mask>?

Answer:
```

3.4 Test Cases

Mentat allows including test cases when repairing a program. This additional information enhances the system's performance by narrowing the error search space and tightening the type constraints for the function under examination. To illustrate, recall Program 1, which contains two potential errors. Now, let's add a test case into this program.

```
let rec add_list lst = match lst with
  | [] -> []
  | fst :: rest -> fst + (add_list rest)

Test case: add_list [1;2;3] = 6
```

The newly added test case locks function `add_list` to specifically receive a list of integers and output a single integer. Because of this, the function now only has one possible source of error, which is the empty list (`[]`) in line 2. Previously, it was also considered that the *plus* operator (+) could be a source of error, but with the additional restrictions imposed by the test case, this is no longer possible.

Adding at least one test case to the framework also helps classify GPT-3's generations. After repairing a program, we can use the test case to check for type consistency and verify if it now passes the tests. For instance, in the case of this program, the correct fix would be to replace the empty list ('[]') with the number 0, but substituting it with any other integer would still pass the type-check, although it might produce incorrect results during testing.

4 Tool

To validate our approach, we implemented it as a publicly available tool called Mentat [6]. This tool, written in *OCaml*, can analyze *OCaml* programs and is accessible via the command line. Users can specify:

- the file containing the *OCaml* program to analyze;
- the repair strategy by issuing the corresponding flag;
- optionally, one or more test cases that should be satisfied.

Depending on the repair strategy selected by the user, Mentat interacts with GPT-3 by calling the relevant function and setting appropriate parameters. Interaction with *OpenAI's* GPT-3 like models requires an internet connection to use the API. Mentat handles these requests and processes the responses to generate potential fixes for type errors. The resulting programs are saved for further offline analysis, including whether they compile successfully and pass provided test cases if available. Detailed installation and usage instructions are provided in the tool's repository.

5 Experiments

We benchmark the effectiveness of our tool by running it against several *OCaml* programs containing type errors. For this, we run each strategy 3 times for each program, and record the results. All the examples and necessary resources to replicate the experiments are publicly available[6].

5.1 Simple Programs

This set includes 15 ill-typed programs sourced from an introductory *OCaml* class at a Japanese University and the type-error slicer *Skalpel* [37], and previously used in a type-error debugger [50]. These programs are simple, with issues like returning empty lists instead of sums, confusion between `Float` and `Int`, and using values when singleton lists were expected. They range from 29 to 117 tokens and consist of 2 to 8 lines of code. One could argue that simple programs are easier to fix because they are simple, or harder to fix due to the limited contextual information available.

For the text and code models used in the experiments, we use the default parameters (*temperature* of 0.7 for text and 0 for code, and *top_p* value of 1 for both). These settings were found to be the most suitable through extensive testing.

We present the experiment results in Table 1. Each test program was processed 3 times to measure successful patch generation, ensuring it passed at least one test case. We employed different repair strategies with models optimized for *text* (T columns) and *code* (C columns). The *C + Bias* column includes additional experiments detailed in Section 3.3. The rightmost column represents results without language models, measuring how many suggestions enabled program compilation and passed a test case. For example, program *S*2 was exclusively repaired by the code variant of the *Fill* strategy, with 10 successful repair suggestions that passed the test

[6]https://gitlab.com/FranciscoRibeiro/mentat

Table 1. Automatic repair results for 15 simple test programs.

Test Prog.	Fill		Choose			Instruct		No GPT-3
	T	C	T	C	C + Bias	T	C	
S1	3	3	3	3	3	3	3	3
S2	0	3	0	0	0	0	0	10
S3	3	3	0	0	3	3	3	3
S4	3	3	0	0	0	2	3	0
S5	3	3	0	0	3	3	3	2
S6	3	3	3	3	0	3	3	3
S7	3	3	0	3	3	0	0	0
S8	3	0	3	3	3	3	3	1
S9	3	3	3	3	0	0	0	1
S10	3	0	3	3	0	0	0	1
S11	3	3	0	0	3	3	3	0
S12	0	3	3	0	3	2	3	0
S13	3	3	3	3	3	0	0	1
S14	3	3	3	3	3	0	0	2
S15	3	3	3	3	3	3	3	1
%Repair	87%	87%	60%	60%	66%	56%	60%	–
%Test	53%	60%	47%	47%	40%	49%	53%	–

Table 2. Automatic repair results for 10 Dijkstra programs.

Test Prog.	Fill		Choose			Instruct		No GPT-3
	T	C	T	C	C + Bias	T	C	
D1	3	3	0	3	3	0	0	0
D2	1	3	0	0	0	0	0	0
D3	1	3	0	0	2	2	3	0
D4	3	3	3	3	0	1	3	1
D5	2	3	0	3	3	2	3	1
D6	3	3	0	0	0	0	0	0
D7	3	3	3	1	3	3	3	1
D8	3	3	0	0	3	3	3	0
D9	3	3	3	3	0	3	3	0
D10	3	3	0	0	3	0	0	0
%Repair	83%	100%	30%	43%	57%	47%	60%	–
%Test	83%	100%	20%	23%	20%	47%	60%	–

case. Further refinement may be possible by using different or additional test cases. In each column, we calculate two success rates: *%Repair*, indicating partial success (yellow or green), and *%Test*, indicating total success (treating yellow results as failures).

Each cell of the table is coloured red, yellow, or green. Red cells denote a total failure of patch generation, green cells denote the generation of the correct patch, and yellow cells denote partial success. Examples of patches that are categorized yellow include generating the incorrect arithmetic operator (such as generating a *minus* sign when a *plus* sign is expected, or not generating the correct constant value when one is expected) - in some cases, for example when a constant value is expected, it might be completely impossible to reasonably deduce which value the developer expects. Such results can be adjusted by using different test cases which favourably guide the GPT-3 model - for example, if, for a given test case, using a *plus* sign yields the same result as using a *minus* sign, perhaps changing the test case will make the usage of a different operator yield a different result. Nevertheless, we decided to not fine-tune the test cases to maximize result quality, as that is not always realistic.

The results showcased in Table 1 point towards the *Fill* strategy being the most efficient for automatic generation of patches. Most notably, all modes have a *%Repair* success rate above 50% and a *%Test* success rate above 40%, and all test programs were successfully repaired by at least one of the repair strategies. This fact points towards the combination of strategies being a robust approach to leverage the strengths of each other. We also denote that most cells contain the values 3 or 0, with rare occurrences of 2, which implies that the model tends towards the same results in different iterations. For this, we have experimented with different values of the parameters we supply to the model, focusing mainly on the *temperature* as it should change its randomness. Nevertheless, the results were not noticeably better, generally leading to lower overall success rate.

We observe that the usage of *bias* with the *Choose* operation mode yields relatively similar results in terms of success rates for these problems. The main difference when using *bias* lies in the fact that some programs that were not repaired with the previous approach are now able to be repaired and vice-versa. For this set of programs, we conclude that the usage of *bias* does not improve the results significantly, but it is capable of generating solutions complementary to the ones generated by the original *Choose* repair strategy.

5.2 Dijkstra Algorithm

In this set, we have longer and more complex programs for the *Dijkstra* algorithm, each with around 2,300 tokens and 170 lines of code. Deliberate errors were added to make the repairs more challenging. We followed the same methodology as in Section 5.1 for the results.

Table 2 summarizes the results for this program set. Like in the previous set (5.1), *Fill* remains the most effective strategy with an 83% repair rate for the *text* model and 100% for the *code* model. Despite the increased program complexity, *Fill* performed better, with a higher rate of programs passing the provided test cases. Conversely, the other strategies, *Choose*, *Instruct*, and *No GPT-3*, were less effective with this program set. Indeed, depending on the considered repair strategy, the discrepancies across the different sets of programs move in opposite ways. Increased program complexity may have improved *Fill*'s performance by providing more context for the model, while negatively affecting the other strategies, which seem more suited for shorter and simpler repairs.

Compared to the simpler programs in the previous section, the type errors in this set usually need more elaborate repairs. As an example, consider the ill-typed excerpt from a program contained in these experiments and its intended repair:

```
let rec search tree k = match tree with
  Empty -> raise Not_found
 | Node (left, key, value, right) ->
   if k = key then value
   else if k < key then left (* intended: search left k *)
   else search right k
```

Instead of `left`, the intended expression is `search left k`. These repairs need an aggregation of several terms, which is impossible to obtain with suggestions from the *language server*. Essentially, this severely hinders the *Choose* and *No GPT-3* modes, as they heavily rely on that list of code completions. The *Instruct* mode correctly infers the corresponding hole's type to be *(string * float) list*[7] but is unable to generate the call `search left k` and produces the empty list instead, which, nonetheless, produces a correctly typed program. From these experiments, we take that *Fill* works best for longer programs, as the pure context of the code seems to be enough and better allows GPT-3 to understand and reason about the program at hand. Extra analysis of the source code prior to providing the programs to GPT-3 is more helpful for smaller programs, in which naturally occurring context lacks. This is evidenced by programs *S*4, *S*5, *S*8, *S*9, *S*10 and *S*13, for which *Fill* presented incorrect or only partially correct results, while *Choose*, *Instruct* or *No GPT-3* were able to generate intended outcomes. This did not occur for the *Dijkstra* programs, as *Fill* showed that it could match the effectiveness of the other strategies for each case.

5.3 Large Scale Evaluation

We also conducted a large-scale evaluation of our approach. We analyzed a repository of 4,500 *OCaml* programs, which had already been created as part of Rite [41]. We provide detailed analysis of the results obtained from this evaluation, such as the total repair rate, the number of partially fixed programs and the distribution of effectiveness of the three repair strategies. Through this evaluation, we aim to demonstrate our tool's applicability in real-world scenarios and potential to improve the quality and reliability of large-scale software systems.

5.3.1 Pre-Processing the Data. To ensure a comprehensive and accurate evaluation of our tool, we applied a filtering process to the original dataset obtained from the Rite project. Specifically, we filtered out bugs that required modifications in multiple and disjointed places in the code, as the current version of our tool considers single expression bugs, only. Furthermore, we only considered bugs for which the original fixed version could properly execute for all test cases generated by the *OCaml* property-based testing tool *Quickcheck* [10]. Proper execution was defined as the absence of errors or timeouts for any given input. This was necessary to ensure that the bugs were genuine and that any improvements observed in our evaluation were a result of our tool's impact, rather than external factors such as faulty test cases or unreliable program behavior. After applying these filters, we evaluated a set of 1,318 bugs.

[7]Actually, the function is polymorphic, but the test case requires a more specialized type, which is what we get thanks to inlining.

5.3.2 Validating the Generated Patches. To validate the effectiveness of our tool in repairing bugs, we used *Quickcheck* to generate a random, large number of test cases. Moreover, we define properties to assert that the human-fixed program is "equivalent" to the repaired one. Thus, for each bug, we generated a set of patches and automatically instantiated a corresponding *Quickcheck* property. This is expressed according to the following template:

```
1 let%test_unit "testName" =
2   Quickcheck.test
3     [%quickcheck.generator: <input_signature>]
4    ~f:(fun args ->
5     [%test_eq: <output_signature>]
6     (Fix.functionToTest args) (Gen.functionToTest args))
```

To generate a property for the faulty program being repaired, we consider the faulty function's signature. The input part of the signature (line 3) is used to implement a generator for the input values that will be tested. The output part (line 5) is used to tell *Quickcheck* the type of the output values to compare. Line 6 represents the property that should be verified and means that the result of the original fixed program should be equal to the result of the patch being tested. The number of arguments needs to be adjusted according to the function being tested and, as such, `args` is modified to reflect that. By default, the generator produces 10,000 inputs. If the property holds, the patch is considered equivalent to the fixed version.

A bug is considered to be repaired if at least one of the generated patches produced the same output as the original fixed version for all input combinations generated by *Quickcheck*. By automating the instantiation of this property for every considered bug, we were able to accurately validate the effectiveness of our tool in repairing bugs. This approach also allowed us to provide quantitative metrics on the performance of our tool, such as the percentage of bugs repaired and the degree to which some bugs are partially fixed — Figures 3 and 4. This automated validation process is crucial given the amount of data at this stage. Furthermore, it also provides some insight into how it could be incorporated in real-world use cases. To the best of our knowledge, it is uncommon to provide a fully automatic validation process to verify whether generated patches successfully fix buggy programs. Our approach has the flexibility of allowing patches equivalent to the intended fix, without relying on human intervention to manually inspect the generated patches.

5.3.3 Results and Discussion. Our approach successfully repaired a substantial portion of the dataset. Among the 1,318 bugs evaluated, our tool repaired 516 of them, achieving a repair rate of 39.2%. We found that 441 of the programs were partially fixed, indicating that the generated patches were able to address some but not all of the identified issues in the program, representing a 33.5% partial repair rate. The consideration of partial fixes provides a more nuanced understanding of the capabilities of our technique. Rather than

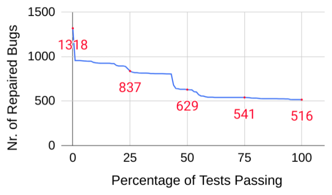


Figure 3. #Programs that pass at least a given percentage of tests. For example, 629 programs pass at least 50% of tests.

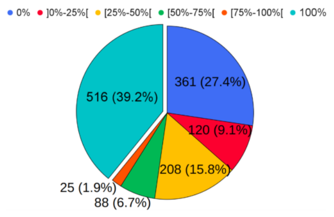


Figure 4. Distribution of test passing rate of programs. For example, 208 programs pass between 25% and 50% of tests.

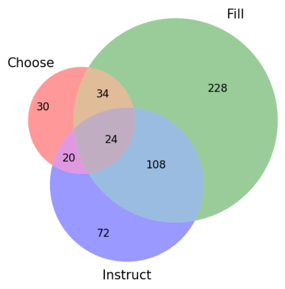


Figure 5. How many programs each mode successfully repairs. Intersections mean that a program is repaired correctly by both modes. There are 34 programs that can be repaired by both the Choose mode or the Fill mode.

simply categorizing a program as either fixed or not fixed, partial fixes enable us to explore the ground that separates a completely fixed program from a program that remains broken. Thus, we can form an idea of how the partial fixes are distributed along that spectrum. Out of the 361 programs that remained unfixed, we found that 247 of them produced some error during testing and the testing process did not finish. Additionally, 73 of the unfixed programs were due to our technique being unable to generate any patch for the identified bugs. Interestingly, we also found that 41 of the unfixed programs actually failed every test produced by *Quickcheck*, indicating that the bugs in these programs were particularly challenging to address.

Different tool modes exhibit varying degrees of repair effectiveness, as shown in Figure 5. The *Fill* strategy is the most effective, being responsible for fixing 394 out of the total 516 programs (76.4%). The *Instruct* strategy was also found to be effective, repairing 224 programs (43.4%). On the other hand, the *Choose* strategy is the least effective, with 108 fixed programs (20.9%). It is worth noting that some programs were repaired by multiple strategies, and in some cases, the same program was repaired by all three strategies. Specifically, there were 108 (20.9%) programs that were fixed by both *Fill* and *Instruct*, while 34 (6.6%) programs were fixed by both *Fill* and *Choose*, and 20 (3.9%) programs were fixed by both *Choose* and *Instruct*. Additionally, there were 24 (4.7%) programs that were repaired by all three strategies.

5.3.4 Limitations.

Our automated validation strategy excludes functions relying on user-defined data types, as it needs manually defined specific generators. This limitation reduces the number of programs we analyze, as discussed in Section 5.3.1. Moreover, we assume total functions, meaning that we consider every possible input for a given type, resulting in a more pessimistic repair validation. For instance, if a function has an integer as argument and is designed to work only with positive numbers, our fully automated approach will still test it with negative numbers (as produced by the predefined generator of integer numbers) reporting it as a non repaired function.[8]

Let us consider the *OCaml* implementation for *factorial*:

```
(* int -> int *)
let rec factorial n =
 if n = 0 then 1
 else n * factorial (n - 1)
```

The provided implementation is the usual recursive definition for *factorial*. Note this is a partial function as it is only defined for positive values of the input n. If n is a negative number, *factorial* will indefinitely call itself causing a stack overflow error. Now, let us consider that this implementation of *factorial* results from a repair process, either generated by Mentat or another tool. When we validate such repair with our automated validation approach, we use *Quickcheck* to automatically generate inputs for this function. In this case, the predefined generator for *int* will produce both positive and negative values. Although the repaired *factorial* function is correct, our validation will fail due to timeout as soon as it is called with a negative number.

5.4 Comparative Study

We performed a comparative study of our technique for automated program repair of ill-typed *OCaml* programs. We utilized the results provided in Rite's [41] repository for both their tool and Seminal to validate the efficacy of our fully automated validation strategy. In section 5.3.4, we acknowledge the demanding and pessimistic nature of our testing strategy by highlighting its consideration of total functions, encompassing every possible input for any given type. This ignores any restriction on the set of valid inputs. A manual validation process, similar to that employed by Rite, has the potential to increase the success rate for both

[8]Generators for positive integers and user-defined types can be implemented. However, this would break our goal of a fully automated process.

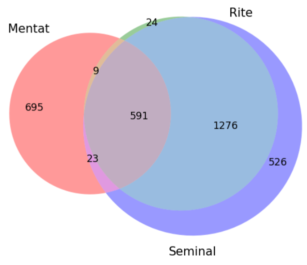


Figure 6. #Programs used in each repair technique and intersections.

our approach and the others. This kind of manual validation allows for a more sensitive consideration of program characteristics that may be overlooked by a more automated validation method. That is, some expected usage patterns may be better captured by a human evaluator with a more subjective evaluation criteria. An example is judging a function's implementation and considering it has been designed to only work with positive numbers, even though the type system may only reflect the function operates on type *int*. However, such manual validation would imply extensive manual effort and is infeasible for the size of this dataset.

We compare our approach with two other tools, namely Rite and Seminal, in terms of their repair capabilities on a common dataset. Although the three tools used a common dataset as an underlying basis, each work applied its own preprocessing criteria to prepare the dataset. As a consequence, in this comparative study, our original dataset of 1,318 bugs was filtered down to 591 bugs, which were common to all three approaches. Figure 6 shows the distribution of the bugs and how they intersect among Mentat, Rite and Seminal.

Our technique achieved a repair rate of 37.6% (222 out of 591 programs). It employs a fully automated analysis that considers a program fixed only if it becomes well-typed and passes all test cases. Our repair process leverages GPT-3, a powerful large language model, to generate patches for identified type errors. This eliminates the need for a comprehensive system and language-specific components due to GPT-3's extensive training on multiple languages.

Originally, Rite conducted a manual validation through a user study with 29 programmers in which a set of 21 buggy programs was selected and each participant was shown 10 randomly selected buggy programs alongside two candidate repairs, one generated by Rite and one by Seminal. A full validation of the entire dataset was not reported. To achieve this, we used our automated validation framework to verify which Rite and Seminal generated patches were able to pass all test cases produced by *Quickcheck*.

This way, we were able to evaluate the performance of Rite and Seminal on the same dataset. Rite repaired 198 programs out of 591 (33.5% repair rate), while Seminal repaired only 46 programs (7.8% repair rate). These results highlight the superior effectiveness of our technique over the existing

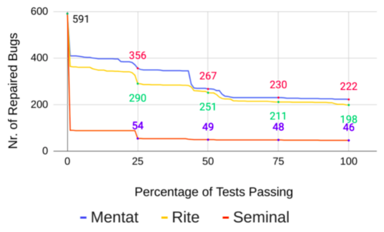


Figure 7. Number of programs that pass at least a given percentage of tests - comparative study.

state-of-the-art tools for automated program repair in the context of type errors in *OCaml* programs. Figure 7 shows the repair effectiveness of the three tools.

One noteworthy advantage of our approach is its language-agnostic nature. Our technique can be easily adapted to repair programs in other languages, as long as it is possible to statically determine the types of terms either through inference or annotations, and the ability to bypass the type system exists (e.g., `Obj.magic` for *OCaml* or `undefined` for *Haskell*). Furthermore, the reliance on LLM's, such as GPT-3, for generating patches liberates us from building language-specific generation systems for each case. By leveraging these prerequisites, our approach can be successfully applied to a wide range of programming languages.

We conclude that Mentat outperforms both Rite and Seminal in repairing type errors on a common dataset of *OCaml* programs. Our fully automated approach eliminates the need for user studies to validate patch relevance and ensures that the resulting programs are not only well-typed but also pass all the provided test cases.

Our results provide the following four insights: First, Mentat surpasses both Rite and Seminal in terms of effective program repair, i.e. patches are well-typed and are equivalent to the intended fixed version; Second, we thoroughly validated Rite's repairs, whereas their paper only validates 21 repairs with user involvement; Third, although Rite reports over 80% success in type repair, we show that the percentage of repairs passing the tests is 33.5%, which is significantly lower and highlights the potential for misleading results[9]; Fourth, our fully automated validation approach enabled us to validate other works that previously relied on manual analysis of a very limited subset of programs.

6 Related Work

Type error debugging research has a rich history spanning over 30 years, evolving from enhancing error messages [12, 24, 51] to interactive debugging tools [6, 7, 9, 48], and automated approaches that narrow down error causes [19, 37, 42–44]. These methods aim to pinpoint errors and require user intervention for correction. On the other hand, automatic correction of type errors is a nascent field; Seminal [25] is,

[9]This also contradicts the (informal) usual saying in functional programming: if it type checks, then it is correct.

to our knowledge, the first system for automatic correction of type errors in functional programming languages. It removes parts of the ill-typed program and attempts to make syntactic changes. This corresponds to *Fill* in our study: they used a syntactic modification to fill, and we used GPT-3. Rite [41] aims for program repair of ill-typed programs too. From a corpus of 4,500 ill-typed *OCaml* programs, it uses approximately half of the dataset to build a neural network that learns what modifications have been made to, ultimately, synthesize solutions for given ill-typed programs. Our tool, Mentat performs source code analysis to produce useful prompts that leverage GPT-3's language understanding and generation capabilities to generate potential patches. While Mentat, Rite and Seminal share a common objective of fixing ill-typed *OCaml* programs, they diverge in their validation methodologies. Rite relies on a manual analysis of 21 randomly selected programs from the repository by a limited number of programmers, whereas our technique employs a fully automated process to validate the generated repairs. This distinction allows our technique to perform validation on a larger scale, effectively addressing the challenges associated with manual validation processes. The definition of a fixed program in Rite is based on the ability of the generated program to typecheck correctly. In contrast, our work validates both typechecking and semantical equivalence of the generated repairs. To achieve this, our technique employs a methodology that generates and executes test cases for both the correct program and the generated repairs. It considers a program to be fully repaired only if the correct program and the repaired version produce identical outputs for all test cases. This crucial difference allowed us to verify that a pure type repair can fall short of being an effective repair. We demonstrated that Rite's reported +80% type repair rate is comparatively lower in terms of actual program repair, i.e. the generated repair satisfies the test cases 33.5% of times. As we mentioned in Section 5.3.4, this is based on a pessimistic view that a patch must pass all test cases. Indeed, a manual analysis may reveal that more of the generated patches are semantically equivalent to the intended program, potentially improving our results as well as those of Rite and Seminal.

DeepTyper [22] enhances type information for compilation using deep learning in *Python* and *JavaScript*. However, it lacks program repair capabilities. Our work utilizes *OCaml*'s type inference for source code analysis and prompt preparation. DeepTyper could be beneficial when extending our approach to other programming languages.

Fault localization [2, 32] is an initial debugging step [31]. Various methods, including execution trace analysis [5], mutation testing [29], qualitative reasoning [33], and semantic fault identification [38], help narrow down suspicious code elements. Models like *code2vec* [1] have been trained to specifically detect security vulnerabilities [11]. Our work concentrates on type errors and uses *OCaml*'s type inference to identify potentially responsible expressions by transforming them into different types.

APR is a prominent research field. Early approaches use genetic programming [3, 23], while others employ constraint-based methods [16, 30, 52]. Recent advancements incorporate machine learning and neural machine translation techniques [8, 26, 28]. However, translating buggy code to fixed code has limitations [15] and general-purpose models supporting code understanding and generation tasks [1, 17, 27, 45] started being considered. GPT-2's code completion effectively fixes *Java* bugs [39], and *Codex* has repaired *Python* and *Java* programs [35]. Our work stands out for targeting type errors in *OCaml*, which prevent program compilation, unlike other research focused on functional bugs.

7 Conclusion

This paper introduced a method to automatically fix type errors in *OCaml* programs using GPT-3. We achieve this by analyzing and modifying the faulty source code to create prompts for GPT-3-based models.

We developed the Mentat tool, initially validating it with simple programs and variations of the *Dijkstra* algorithm. In large-scale experiments involving 1,318 buggy programs, we achieved a 39% repair rate using a novel automated patch validation approach. In comparison with two other *OCaml* program repair tools, Mentat outperformed them, achieving a 37.6% repair rate on a shared dataset of 591 programs, while the other tools achieved rates of 33.5% and 7.8%, respectively.

This work used GPT-3, but future versions or other LLMs [46, 53] could be integrated. Moreover, fine-tuning a model for *OCaml* may enhance program correction.

Starting with single-location faulty programs enables us to assess the approach's effectiveness before addressing multiple locations. In future work, we plan to explore this possibility by strategically placing typecast operators to address program sections and incorporating multiple typecasts in suitable locations. This would facilitate the identification and repair of multiple bugs within a single program.

Replication Package

All the necessary resources to replicate this study are public:
- **Tool:** *https://gitlab.com/FranciscoRibeiro/mentat*
- **Artifact [40]:** *10.6084/m9.figshare.23646903.v2*

Acknowledgments

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project UIDP/50014/2020. Francisco Ribeiro and José Nuno Macedo acknowledge FCT PhD grants SFRH/BD/144938/2019 and 2021.08184.BD, respectively. Additional funding: JSPS KAKENHI-JP19K20248.

References

[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[2] Aaron Ang, Alexandre Perez, Arie Van Deursen, and Rui Abreu. 2017. Revisiting the practical use of automated software fault localization techniques. In *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 175–182.

[3] Andrea Arcuri. 2011. Evolutionary repair of faulty software. *Applied soft computing* 11, 4 (2011), 3494–3514.

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *CoRR* abs/2005.14165 (2020). arXiv:2005.14165 https://arxiv.org/abs/2005.14165

[5] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. 2012. Gzoltar: an eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM international conference on automated software engineering*. 378–381.

[6] Sheng Chen and Martin Erwig. 2014. Counter-factual typing for debugging type errors. In *Symposium on Principles of Programming Languages. Proceedings (POPL '14)*. ACM, 583–594. https://doi.org/10.1145/2535838.2535863

[7] Sheng Chen and Martin Erwig. 2014. Guided type debugging. In *Functional and Logic Programming. Proceedings (LNCS 8475)*, Michael Codish and Eijiro Sumii (Eds.). Springer, 35–51. https://doi.org/10.1007/978-3-319-07151-0_3

[8] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1943–1959.

[9] Olaf Chitil. 2001. Compositional explanation of types and algorithmic debugging of type errors. In *International Conference on Functional Programming. Proceedings (ICFP '01)*. ACM, 193–204. https://doi.org/10.1145/507635.507659

[10] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Not.* 35, 9 (sep 2000), 268–279. https://doi.org/10.1145/357766.351266

[11] David Coimbra, Sofia Reis, Rui Abreu, Corina Păsăreanu, and Hakan Erdogmus. 2021. On using distributed representations of source code for the detection of C security vulnerabilities. *arXiv preprint arXiv:2106.01367* (2021).

[12] Luis Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In *Symposium on Principles of Programming Languages. Proceedings (POPL '82)*. ACM, 207–212. https://doi.org/10.1145/582153.582176

[13] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *Computer* 11, 4 (1978), 34–41.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[15] Yangruibo Ding, Baishakhi Ray, Premkumar Devanbu, and Vincent J Hellendoorn. 2020. Patching as translation: the data and the metaphor. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 275–286.

[16] Thomas Durieux and Martin Monperrus. 2016. Dynamoth: dynamic code synthesis for automatic program repair. In *Proceedings of the 11th International Workshop on Automation of Software Test*. 85–91.

[17] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[18] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (nov 2019), 56–65. https://doi.org/10.1145/3318162

[19] Christian Haack and Joe B. Wells. 2004. Type error slicing in implicitly typed higher-order languages. *Science of Computer Programming* 50, 1-3 (2004), 189–224. https://doi.org/10.1016/j.scico.2004.01.004

[20] BJ Heeren, JT Jeuring, Doaitse Swierstra, and Pablo Azero Alcocer. 2002. Improving type-error messages in functional languages. (2002).

[21] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. 2003. Helium, for learning Haskell. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*. 62–71.

[22] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep Learning Type Inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 152–162. https://doi.org/10.1145/3236024.3236051

[23] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering* 38, 1 (2011), 54–72.

[24] Oukseh Lee and Kwangkeun Yi. 1998. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems* 20, 4 (1998), 707–723. https://doi.org/10.1145/291891.291892

[25] Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. 2007. Searching for Type-Error Messages. *SIGPLAN Not.* 42, 6 (jun 2007), 425–434. https://doi.org/10.1145/1273442.1250783

[26] Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 602–614.

[27] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[28] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 101–114.

[29] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 153–162.

[30] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 772–781.

[31] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 2011 international symposium on software testing and analysis*. 199–209.

[32] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 609–620.

[33] Alexandre Perez, Rui Abreu, and IT HASLab. 2018. Leveraging Qualitative Reasoning to Improve SFL.. In *IJCAI*. 1935–1941.

[34] Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

[35] Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can OpenAI's codex fix bugs? an evaluation on QuixBugs. In *Proceedings of the Third International Workshop on Automated Program Repair*. 69–75.

[36] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[37] Vincent Rahli, Joe B. Wells, John Pirie, and Fairouz Kamareddine. 2017. Skalpel: a constraint-based type error slicer for Standard ML. *Journal of Symbolic Computation* 80, 1 (2017), 164–208. https://doi.org/10.1016/j.jsc.2016.07.013

[38] Francisco Ribeiro, Rui Abreu, and João Saraiva. 2021. On Understanding Contextual Changes of Failures. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 1036–1047.

[39] Francisco Ribeiro, Rui Abreu, and João Saraiva. 2022. Framing Program Repair as Code Completion. In *Proceedings of the Third International Workshop on Automated Program Repair* (Pittsburgh, Pennsylvania) *(APR '22)*. Association for Computing Machinery, New York, NY, USA, 38–45. https://doi.org/10.1145/3524459.3527347

[40] Francisco Ribeiro, José Macedo, Kanae Tsushima, Rui Abreu, and João Saraiva. 2023. GPT-3-Powered Type Error Debugging: Investigating the Use of Large Language Models for Code Repair (SLE 2023). (10 2023). https://doi.org/10.6084/m9.figshare.23646903.v2

[41] Georgios Sakkas, Madeline Endres, Benjamin Cosman, Westley Weimer, and Ranjit Jhala. 2020. Type Error Feedback via Analytic Program Repair. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 16–30. https://doi.org/10.1145/3385412.3386005

[42] Thomas Schilling. 2012. Constraint-free type error slicing. In *Trends in Functional Programming. Proceedings (LNCS 7193)*, Ricardo Peña and Rex Page (Eds.). Springer, 1–16. https://doi.org/10.1007/978-3-642-32037-8_1

[43] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. 2003. Interactive type debugging in Haskell. In *Workshop on Haskell. Proceedings (Haskell '03)*. ACM, 72–83. https://doi.org/10.1145/871895.871903

[44] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. 2004. Improving type error diagnosis. In *Workshop on Haskell. Proceedings (Haskell '04)*. ACM, 80–91. https://doi.org/10.1145/1017472.1017486

[45] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1433–1443.

[46] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL]

[47] Kanae Tsushima and Kenichi Asai. 2012. An embedded type debugger. In *Symposium on Implementation and Application of Functional Languages*. Springer, 190–206.

[48] Kanae Tsushima and Kenichi Asai. 2013. An embedded type debugger. In *Implementation and Application of Functional Languages. Proceedings (LNCS 8241)*, Ralf Hinze (Ed.). Springer, 190–206. https://doi.org/10.1007/978-3-642-41582-1_12

[49] Kanae Tsushima and Kenichi Asai. 2014. A weighted type-error slicer. *Journal of Computer Software* 31, 4 (2014), 131–148.

[50] Kanae Tsushima, Olaf Chitil, and Joanna Sharrad. 2019. Type debugging with counter-factual type error messages using an existing type checker. In *Symposium on Implementation and Application of Functional Languages. Proceedings (IFL '19)*. ACM, Article 7, 12 pages. https://doi.org/10.1145/3412932.3412939

[51] Mitchell Wand. 1986. Finding the source of type errors. In *Symposium on Principles of Programming Languages. Proceedings (POPL '86)*. ACM, 38–43. https://doi.org/10.1145/512644.512648

[52] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering* 43, 1 (2016), 34–55.

[53] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. OPT: Open Pre-trained Transformer Language Models. arXiv:2205.01068 [cs.CL]

Received 2023-07-07; accepted 2023-09-01

Temporal Breakpoints for Multiverse Debugging

Matthias Pasquier
ERTOSGENER
Angers, France
matthias.pasquier@ertosgener.com

Ciprian Teodorov
Lab-STICC CNRS UMR 6285
ENSTA Bretagne
Brest, France
ciprian.teodorov@ensta-bretagne.fr

Frédéric Jouault
LERIA, University of Angers
ESEO
Angers, France
frederic.jouault@eseo.fr

Matthias Brun
LERIA, University of Angers
ESEO
Angers, France
matthias.brun@eseo.fr

Luka Le Roux
Lab-STICC CNRS UMR 6285
ENSTA Bretagne
Brest, France
luka.le_roux@ensta-bretagne.fr

Loïc Lagadec
Lab-STICC CNRS UMR 6285
ENSTA Bretagne
Brest, France
loic.lagadec@ensta-bretagne.fr

Abstract

Multiverse debugging extends classical and omniscient debugging to allow the exhaustive exploration of non-deterministic and concurrent systems during debug sessions. The introduction of user-defined reductions significantly improves the scalability of the approach. However, the literature fails to recognize the importance of using more expressive logics, besides local-state predicates, to express breakpoints. In this article, we address this problem by introducing temporal breakpoints for multiverse debugging. Temporal breakpoints greatly enhance the expressivity of conditional breakpoints, allowing users to reason about the past and future of computations in the multiverse. Moreover, we show that it is relatively straightforward to extend a language-agnostic multiverse debugger semantics with temporal breakpoints, while preserving its generality. To show the elegance and practicability of our approach, we have implemented a multiverse debugger for the AnimUML modeling environment that supports 3 different temporal breakpoint formalisms: regular-expressions, statecharts, and statechart-based Büchi automata.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Keywords: multiverse debugging, breakpoint, concurrency, temporal logic

ACM Reference Format:
Matthias Pasquier, Ciprian Teodorov, Frédéric Jouault, Matthias Brun, Luka Le Roux, and Loïc Lagadec. 2023. Temporal Breakpoints for Multiverse Debugging. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3623476.3623526

1 Introduction

Interactive debugging is an essential part of the development lifecycle. The increasing complexity of languages and projects leads to the need for associated tools to evolve. Omniscient debugging [13] allows to return back in time within one execution trace. Multiverse debugging [14] generalizes the breakpoint lookup function to a generic reachability query over the state-space of the program. Multiverse debugging has been introduced to enable the exploration of concurrent actor-based formalisms. This technique is especially useful for debugging concurrent systems (e.g., multithreaded, actor-based), where different execution schedules can hide bugs. For high-level specification languages, such as TLA+ [11] or UML [15], this feature is basically necessary due to the intrinsic non-determinism [16], which besides concurrency allows *a)* to capture entire families of implementations; and *b)* to model uncertainty (e.g., as under-specified decision points). Recently, the introduction of user-defined reductions [16] rendered multiverse debugging practical for large or even infinite state-spaces. The generalization of breakpoint lookup, and the introduction of state-space reductions open up promising research directions that bridge the gap between classical program debugging and explicit-state model-checking. This offers a strong formal basis to build powerful interactive debuggers. However, the expressive power of breakpoints is limited to local state-predicates, which require the instrumentation of the subject-program to capture causal stop conditions explicitly. More expressive breakpoints can be found in the literature [4, 6, 8], which emphasize the importance of using expressive logics for defining the target of breakpoint lookup. Nevertheless, these approaches only allow reasoning about the past of a computation. Furthermore, they rely either on language-specific features, such as reflexivity, or require *ad hoc* runtime support, which incurs a relatively high development cost.

The main research questions that we address in this study are:

1) How to improve the expressivity of the conditional breakpoints to allow reasoning on highly non-deterministic specifications?

2) How to isolate and modularly compose the conditional breakpoint semantics with the subject language semantics, so that the two can evolve independently?
From a methodological point of view we want to enforce an integrity constraint on the system-under-debug, which precludes us from applying system instrumentation.

In this paper, we address the breakpoint expressivity issue by introducing *temporal breakpoints*, which enable reasoning both about the past and the future of a computation, without the need for subject-program instrumentation. This allows us to describe a new taxonomy of breakpoints, as the possibilities offered by this approach allows us to describe three types of breakpoints:

- **Step Breakpoints** corresponds to stopping conditions expressed on a single execution step. The conditions can depend on any value present in the configurations, their evolution triggered by the execution step, as well as the executed action.
- **Safety Breakpoints** corresponds to expressions based on complex sequences of events, to create breakpoints describing a necessary path to follow before stopping.
- **Liveness breakpoints** allows us to create breakpoints from liveness properties, to express conditions on the future of execution of the system.

Moreover, we introduce the notion of a *dependent Semantic-Language Interface (SLI)* to fully isolate the breakpoint semantics from the subject-language, and thus achieve a highly-modular language-agnostic formalization of a temporal multiverse debugger. The contributions presented in this paper have been fully formalized using the L∃∀N theorem prover.

Besides the formalization, we have implemented a prototype debugger in JavaScript, which uses, without modification, the state-of-the-art AnimUML [10] execution engine as UML subject-language. To evaluate and emphasize the modularity of our approach, we have implemented 3 temporal breakpoint languages with different expressivity/concision profiles:

- **Regular expression breakpoints (reB)**, similar to stateful breakpoints [4], restricted to reasoning about the past (able to express safety properties);
- **Statechart-based breakpoints (scB)**, also restricted to past reasoning, however syntactically more concise due to use of variables besides the control-states; and
- **Statechart-based Büchi automata breakpoints ($\mathbf{scB}_{ba}$)**, allowing both past and future reasoning (able to express both safety and liveness properties).

Both statechart-based breakpoint languages (scB & scB_{ba}) were implemented by simply exposing the existing AnimUML SLI as a dependent SLI, which illustrates that our proposal fosters reuse.

The article is structured as follows. Section 2 overviews the related works. Section 3 discusses the core of our contribution. Section 4 illustrates the process of creating a temporal breakpoint language, using reB as an example. Section 5 discusses the integration with AnimUML and gives some examples of temporal breakpoints. We discuss the limits of this approach as well as some future research directions in Section 6 before concluding in Section 7.

2 Related Work

Multiverse debugging [14] extends omniscient debugging [13] by embracing the non-deterministic nature of programs. Initially proposed for actor-based subject languages [14], the approach has also been adopted for debugging specification languages [10, 16]. The AnimUML execution environment [10] proposes a unified approach for UML execution and verification, which recently integrated user-reduction strategies to improve scalability [16]. The generic multiverse debugger formalization proposed in [16] only supports local-state breakpoint definitions (i.e., predicates on current execution states), which leads to complex code instrumentation for expressing complex stop conditions. In this paper, we treat the breakpoint definition problem as a language problem, with its semantics dependent on the subject-language. Moreover, we show how to leverage the powerful languages used for temporal verification [19] to improve breakpoint expressivity.

Control-flow breakpoints debugging [6] allows the user to specify dynamic breakpoints which are conditional on the control-flow through which they were reached. Stateful breakpoints [4] expand on this idea by proposing a regular expression-like syntax for defining breakpoints. These tools allow both **step** and **safety breakpoints**, however our approach increases the breakpoint condition expressivity through **liveness breakpoints**, a new class of breakpoint conditions. Furthermore, this paper goes one step further by allowing arbitrary breakpoint languages, which can be more expressive, or concise when compared to existing approaches (based on regular expressions for instance).

Scripted debugging [8] automates repetitive debugging steps. Expositor [18] proposes an integration of scripted debugging with time-travel. Our contributions, in this paper, complement these efforts by offering a formal basis for defining similar APIs externally and independently from the subject language.

Analysis of program independent from the subject language have already been proposed in [12], which proposes a way to do runtime monitoring and logging in a language agnostic way. This paper shows the need for such independence, and is thus complementary to ours, as it does not allow multiverse debugging.

Paper [3] shows that any liveness verification (reasoning about the future) problem can be transformed to a safety verification problem (reasoning about the past) through instrumentation. By instrumentation [21] the subject-program is modified (transformed) to integrate auxiliary logging variables.

However, none of the existing solutions can be directly adapted to multiverse debugging. To understand the reason why, let's take the situation where we want a breakpoint to stop after a certain number of a repeating event, like on a precise iteration of a loop as a motivating example. In a traditional debugging context, there is two ways to achieve this: *instrumentation*, similar to [12], or *breakpoint-hit counters* as in [6].

In the first case **instrumentation** introduces a modification of the system-under-debug. In this case, we can add a logging variable to count our occurrences, and set a breakpoint to stop when this variable reach a certain value. The first problem with this method is that we are changing the system-under-debug and we are at risk of modifying the original behavior. There is also a risk of forgetting this logging elements in the model even when they are no longer necessary, leading to security risks, for instance logging confidential information.

To alleviate these problems, some approaches [6] use breakpoint hit counters, which stores the auxiliary state in the debugging context. This feature, present in many languages, is typically implemented as a counter allowing the user to specify the number of times the breakpoint condition is hit before triggering the breakpoint. Implementing this approach in the context of multiverse debugging is more challenging due to the non-determinism. The key observation is that the breakpoint state cannot be global to the execution, but it has to be local to each configuration of the system-under-debug. Rendering the breakpoint state local allows it to follow all linear execution paths through the multiverse. The approach presented here achieves this by encapsulating the breakpoint state in the configuration of the breakpoint language, which is manipulated by a specialized semantics operating synchronously with the semantics of the system-under-debug.

3 Temporal Multiverse Debugging

This section presents an extension to the formal semantics of multiverse debugging to allow temporal breakpoints. Temporal breakpoints greatly increase expressivity, allowing to capture complex causal conditions that require temporal logic operators. Section 3.1 provides a high-level overview of the debugger architecture. Section 3.2 introduces the extension to the Semantic Language Interface (SLI) that allows us to capture the inherently-dependent semantics of temporal breakpoint expressions. Section 3.3 discusses the core of our contribution: *the internal architecture of the temporal finder*, which bridges the gap between the subject language and breakpoint expressions during the execution of the *run_to_breakpoint* debug action. In the interest of brevity, this article only provides an overview of the formalization[1].

3.1 Overview

Figure 1 overviews our proposal showing the user-facing interfaces at the top, and emphasizing the internal debugger architecture, which treats breakpoint expressions as a language component (right-side of the figure) that is used by the *Temporal Finder* during breakpoint lookup. To obtain a *temporal multiverse debugger instance*, the tool-builder needs to provide both the semantics of the subject language (captured through an SLI) and the semantics of the breakpoint expressions (captured through an iSLI).

To start the debugging process, the user supplies its **specification** to the debugger, which will instantiate the appropriate subject language semantics, by the way of the *STR* interface which is described in Section 3.2.1.

The debugger exposes multiple **debug actions**, whose effect are described in Figure 2: *1) step* to move to the next configuration, *2) jump* to return to a previously explored one, *3) select* to solve cases of multiple parallel configurations, and *4) run_to_breakpoint* to lookup a particular point in the system's execution. During a debug session, the user can inspect the current **debug configuration**, which contains the set of execution traces along with the memory configuration at each execution point. When using the *run_to_breakpoint* action two more interfaces are available. The first one offers the ability to define a conditional **breakpoint expression**, the main subject of this article. The second interface, not presented here, is the reduction expressions as defined in [16][2]. The *Multiverse Debugger Core Semantics* can be seen as a bridge between the user and the debugged model. This semantics is dependent on the *subject language* runtime, and on a `finder` function. In our case, the subject language runtime is encapsulated in an SLI that provides language-agnostic observation and control. Following the approach in [16], the `finder` function is defined externally from the core debugger semantics. As we will see in Section 3.3, the `finder` function plays a central role in the context of our contribution, since it allows us to bridge the gap between the subject-language

[1]The full machine-parsable formalization is available at https://github.com/teodorov/reduced-multiverse-debugging

[2]Reduction expressions define state-space pruning strategies for improved scalability. However, since they are not necessary to understand temporal breakpoints, they are not presented here. The interested reader can check the full formalization (see previous footnote).

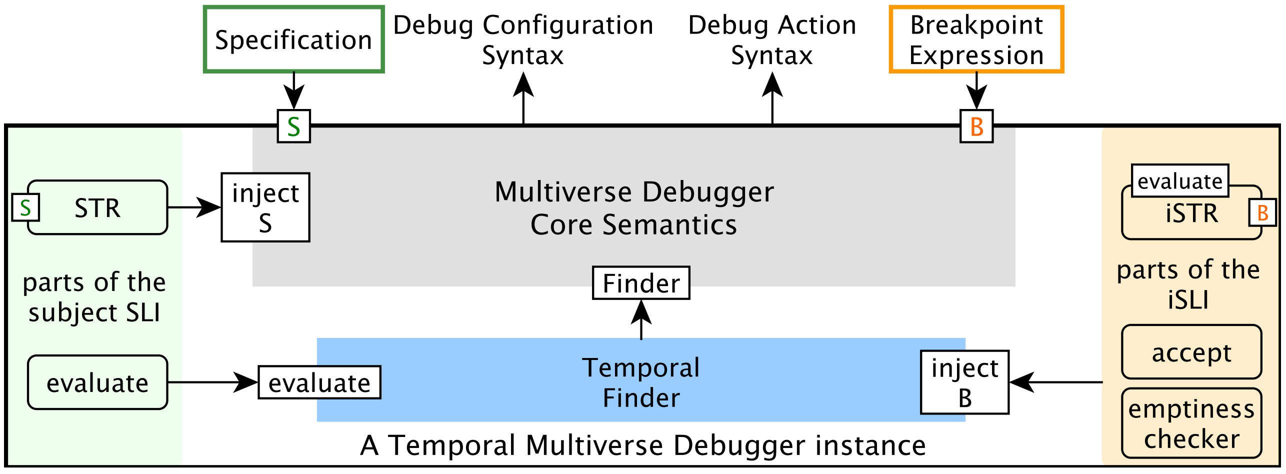


Figure 1. Overview of multiverse debugging with temporal breakpoints

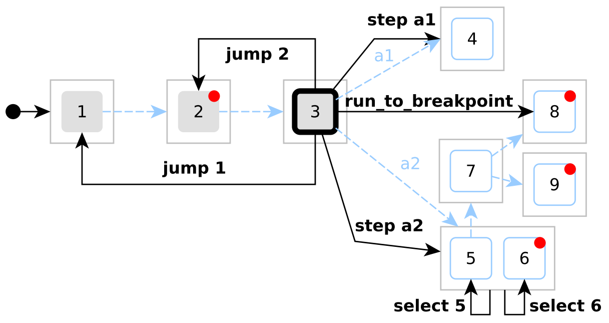


Figure 2. The Debug actions overlayed over a transition system

semantics and the breakpoint semantics during breakpoint lookup.

3.2 A Dependent Semantic Language Interface for Breakpoints

Like in [16], interactions with the subject language are mediated by a Semantic Language Interface (SLI), partially defined in Listing 1. This interface ensures an independence relation between the language and the debugger, while offering flexible observation and control. However, the SLI interface of [16] lacks some important features needed for high-fidelity observation, which we address by extending the scope of the `evaluate` function. Moreover, the SLI interface is not generic enough to capture the semantics of temporal breakpoints, which depends on the subject-language. This section overviews the basic SLI features (Section 3.2.1), before (1) discussing the need for an extended `evaluate` function (Section 3.2.2), and (2) generalizing the SLI to allow the definition of the inherently dependent breakpoint semantics (Section 3.2.3).

```
structure STR(C A):=
    (initial:          set C)
    (actions:      C →set A)
    (execute: A →C →set C)
def Step (C A):=
  (C × MaybeStutter A × C)
class Evaluate(C A E V):=
  (E →Step C A →V)
```

Listing 1. The Semantic Language Interface (SLI) definition

```
structure iSTR (C A E I V)
  (eval: E →I →V) :=
    (initial:              set C)
    (actions:      I →C →set A)
    (execute: A →I →C →set C)
def Step (C A I):=
  (C × I × MaybeStutter A × C)
class Evaluate(C A E' I V) :=
  (E' →Step C A I →V)
```

Listing 2. The Input Semantic Language Interface (iSLI) definition

3.2.1 Semantics Language Interface.

The main components of the SLI (Listing 1) are the *Semantic Transition Relation* (STR) and the `evaluate` function. The STR offers 3 functions to control the execution. The `initial` function returns the set of all initial configurations *set C*. For a given configuration *C*, the `actions` function returns the actions *set A* that can be executed next. Finally, the `execute` function executes an action *A* from a given configuration *C*, and returns the possible resulting configurations *set C*. These functions return sets to capture the non-determinism in the language. In [16], the `evaluate: E → C → V` function allowed the evaluation of an expression *E* over an arbitrary configuration *C* to obtain a value *V*, which resembles the `eval` function exposed reflexively in some languages. Here we propose to generalize the `evaluate` function, as discussed in the next paragraphs.

3.2.2 Evaluate on Execution Steps. A set of behaviors fully characterizes the execution of a specification. Each behavior (trace) is the linear ordering of a set of memory configurations connected by the actions that allowed to move from one configuration to the next. An *Execution Step* of a behavior is a triple (source configuration, action, target configuration). The SLI uses the `evaluate` function for observing (and reasoning) over the behaviors of the subject-language specification, but hides the *actions* and the links between different configurations. This problem has been identified in the literature and leads to the Temporal Logic of Actions [11], where the *actions* are defined as predicates over execution steps, and to state-event model-checking [5], where it gave rise to an extension of linear temporal logic that allows reasoning over the execution steps directly. In this paper, the domain of the `evaluate` function is extended to evaluate an expression in E over an execution step, while still returning a value in V. To allow for stuttering execution steps [1] (steps during which no action is fired) the action is wrapped in a *MaybeStutter* monad. This change allows setting breakpoints on the action that has been fired during an execution step, and on the one-step changes between two consecutive configurations.

3.2.3 Input Semantic Language Interface. To handle breakpoints as an independent language problem we need to be able to isolate their semantics from the semantics of the subject language. To understand the main issue, consider a simple conditional breakpoint as • [condition] ◦. The semantics of this breakpoint depends on the valuation of the condition by the subject-language, if and only if the condition is true as the control point (• – controlled by the breakpoint semantics) passes over the expression, marking it as (◦ [condition] •). This simple illustration emphasizes the nature of conditional breakpoint semantics, which can be seen as a layered semantics with one predicative layer solved by the subject-language (the condition evaluation), and another one proper to the breakpoint (the evolution of control points). The main problem here is that, during evaluation, the breakpoint control-layer needs to dispatch some expressions to the subject-language semantics.

In this paper, we address this problem by generalizing the SLI to obtain an *Input Semantic Language Interface* (iSLI), presented in Listing 2, which is parameterized by three arbitrary types E, I, V and an `eval` function (`eval: E → I → V`). Given an arbitrary expression in E and an input scope in I, `eval` will produce a value in V. The input scope (in I) is dynamically passed to the `actions` and `execute` functions of iSTR, enlarging their evaluation context. During the evolution of the execution, if an iSTR semantics encounters an expression E (either while computing the enabled actions or while executing an action), that comes from the subject language and thus cannot be interpreted, it can dispatch its evaluation to the `eval` function, passing in the current input scope (an inhabitant of I). Note that each execution *Step* of a dependent semantics, captured through an iSLI, also includes the input, which enables the iSLI `evaluate` function to distinguish between steps with different inputs that might otherwise be considered identical. Furthermore, note that the expression type exposed by the iSLI (E') `evaluate` is different from the E type parameter. E' captures the type of expressions exposed by the iSLI for its observation.

For encoding the semantics of a breakpoint language the iSLI parameters are instantiated as follows: *a)* the E type is mapped to the expression type exposed by the `evaluate` function of the subject language SLI; *b)* the I type is mapped to the execution *Step* of the subject-language SLI (`C × MaybeStutter A × C`); *c)* the V type of the iSTR is mapped to the return type of the subject-language `evaluate`[3]; *d)* the `eval` parameter is mapped to the step-aware `evaluate` function of the subject-language.

To encode temporal breakpoints, in this paper, we rely on a model-checking approach, which solves a language intersection problem to decide if the subject-language execution includes behaviors from the temporal property language [7], which is the temporal breakpoint language in our case. To achieve this, we choose to rely on an approach using state-based acceptance conditions, which allows us to encode all regular and ω-regular properties. To this end, we complement the iSLI interface with *(1)* an `accept: C → bool` predicate defining the set of accepting states of the breakpoint language; and *(2)* an `emptiness_checker` algorithm that completes the semantic interpretation of the breakpoint language. For a regular language, limited to past reasoning, the `emptiness_checker` can use any reachability algorithm. For ω-regular properties the `emptiness_checker` should check for the specific acceptance conditions, which here are limited to Büchi acceptance – detecting cycles in the behavior that contain accepting configurations.

3.3 The Temporal Breakpoint Finder

Up to here we have been focused on capturing the semantics of the subject language (shown left in Figure 1) and of the breakpoint language (shown right in Figure 1) in a general and modular fashion, only hinting at links with multiverse debugging. In this section, we discuss the `Temporal Finder` component that binds everything. The `Temporal Finder` does the actual breakpoint lookup upon the invocation of the *run_to_breakpoint* debug action. The core semantics [16] imposes the following high-level signature for the `Finder`

[3]Please note that, for simplicity, our formalization only requires that the subject-language and the breakpoint-language agree on the representation of booleans, since the subject-language `evaluate` is only used for predicates.

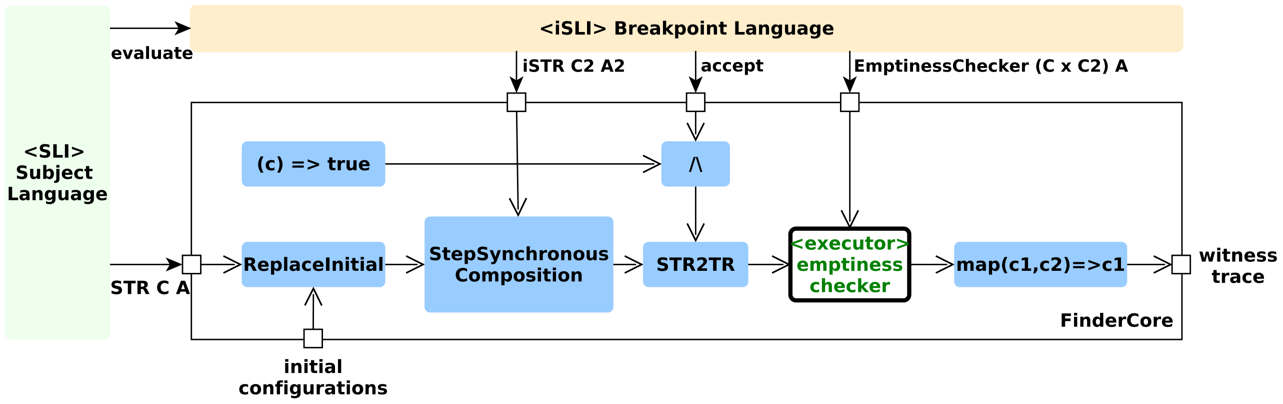


Figure 3. The architecture of the temporal finder function.

component: `def Finder(C A B R) := STR C A → set C → B → R → list C`, where the STR is the subject-language STR, set C is the set of configuration from which the lookup should start, and an inhabitant of B encodes the breakpoint syntax, a reduction in R encodes the state-reduction syntax used for state-space pruning. The `Finder` returns a list of subject-language configurations encoding a witness trace, i.e. an accepting behavior (according to the breakpoint semantics) starting from one of the start configurations.

Figure 3 overviews the architecture of a `Temporal Finder` that exploits a breakpoint semantics isolated behind an iSLI interface[4]. The architecture is split in 3 components, the parts needed from subject-language SLI (left in Figure 3), the parts needed from the breakpoint-language iSLI (top in Figure 3) and the *FinderCore* (bottom-right in Figure 3). The subject-language `evaluate` function is provided to the breakpoint-language iSLI, to obtain the iSTR instance. The *FinderCore* prepares and composes the two semantics, through a synchronous composition approach. The *ReplaceInitial* operator wraps the subject-language STR replacing the initial configurations, since the composition should start from the initial configurations provided as parameter. The resulting STR and the breakpoint iSTR are synchronized using the *StepSynchronousComposition* operator, which itself exposes an STR with composite configurations ($C_1 \times C_2$) and actions (`MaybeStutter` $A_1 \times A_2$) – note that deadlocks in the subject-language STR induce stuttering execution steps. The composed STR is lowered to a Transition Relation (TR) interface (*STR2TR* operator), as in [16], which reifies the relations between the composite configurations and the accepting predicate. Note that all states of the subject-language are considered as accepting (`(c) => true`), thus the *STR2TR* operator exhibits only the accepting conditions provided by the breakpoint language iSLI. This is typical in model-checking, since usually we do not want the subject language to prohibit the property (the breakpoint in our case) from marking a composite configuration as accepting. The resulting TR is fed to the *emptiness checker*, which searches for accepting behaviors. If a witness trace is found, it should be filtered by mapping each entry to the subject-language configuration. This last step removes the breakpoint configuration component from the trace allowing the core debugger semantics to patch the execution history with the witness.

[4]The interested reader should refer to the formalization repository for the full definition of the *Temporal Finder* component

3.4 The Step Synchronous Composition Operator

The StepSynchronousComposition, as described in Listing 3 is the operator used to combine the subject STR with the breakpoint iSTR for the breakpoint search phase. It takes as inputs the subject STR as lhs (for left hand side), instantiated with configuration type C_1 and action type A_1, the breakpoint iSTR as rhs (for right hand side), instantiated with types C_2 and A_2. It returns a new STR instantiated with two new types, the configurations composed from the left and right sides ($C_1 \times C_2$) and the action ($MaybeStutter\ A_1 \times A_2$), where the MaybeStutter offers the possibility for the lhs side of the action to stutter, meaning not taking a step while the rhs takes one. This is necessary for the correct implementation of liveness breakpoint semantics in the context of finite system behaviors, which arrives for terminating behaviors - wanted (normal termination) or unwanted (deadlocks).

This newly instantiated STR has the three expected functions: The `initial` function returns the set of configurations composed of all the combinations of initial configurations from the lhs and rhs. The `actions` function will first create the set of all possible steps from the lhs, with the source configuration taken from the left side of the function parameter. For all actions possible from this configuration, it takes the resulting configurations and creates all those steps. Then, the second part of the function begins depending on if this set of possible steps is empty or not. If at least a step is possible, for all those present in the set we check which actions are possible from the rhs. Then we return the action composed from the action taken from the step ($s_1.2.1$ is written this way because of the way the parenthesis are placed in the

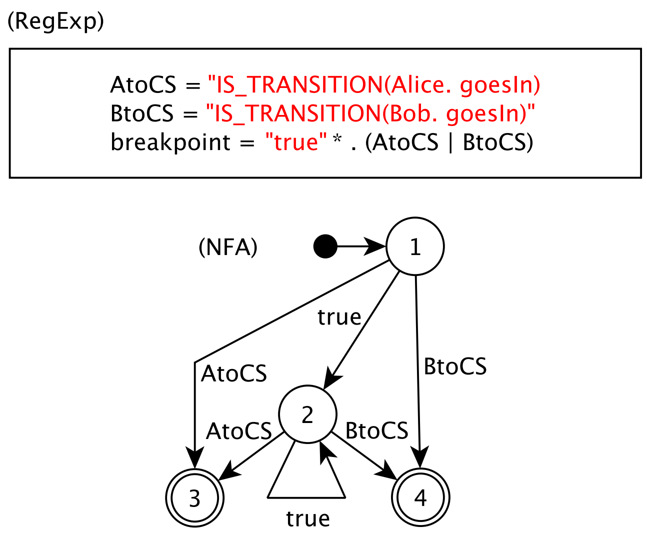
```
def StepSynchronousComposition
  {C₁ C₂ A₁ A₂ Ec Es : Type}
  [∀ actions : set (C₁ × MaybeStutter A₁ × C₁), decidable (actions = ∅)]
  [eval : Evaluate C₁ A₁ bool Ec Es]
  (lhs : STR C₁ A₁)
  (rhs : iSTR C₂ A₂ Es (Step C₁ A₁) bool eval.step)
  : STR (C₁ × C₂) (MaybeStutter A₁ × A₂) :=
  {
    initial := { c | ∀ (c₁ ∈ lhs.initial) (c₂ ∈ rhs.initial), c = (c₁, c₂) },
    actions := λ c,
      match c with
      | (c₁, c₂) := let S₁ := { s₁ |
        ∀ (a₁ ∈ lhs.actions c₁)
          (t₁ ∈ lhs.execute a₁ c₁), s₁ = (c₁, enabled a₁, t₁)}
        in if S₁ ≠ ∅
          then { a | ∀ (s₁ ∈ S₁) (a₂ ∈ rhs.actions s₁ c₂), a = (s₁.2.1, a₂)  }
          -- add stutter if lhs deadlock
          else { a | ∀ a₂ ∈ rhs.actions (c₁, stutter, c₁) c₂, a = (stutter, a₂)}
      end,
    execute := λ a c, { t |
      match a, c with
      | (stutter, a₂), (c₁, c₂) :=
        ∀ t₂ ∈ rhs.execute a₂ (c₁, stutter, c₁) c₂, t = (c₁, t₂)
      | (enabled a₁, a₂), (c₁, c₂) :=
        ∀ (t₁ ∈ lhs.execute a₁ c₁)
          (t₂ ∈ rhs.execute a₂ (c₁, enabled a₁, t₁) c₂), t = (t₁, t₂)
      end
    }
  }
```

Listing 3. The L∃∀N formalization of the Synchronous Composition Operator

cartesian product of the step: $(C \times (MaybeStutter\ A \times C))$, so the first field 2 access to the inner parenthesis, then the field 1 to the action) and the one allowed in the rhs. If no action is possible in the lhs, the system is in deadlock but we allow it to stutter so the rhs is not blocked. This way the breakpoint may trigger if the configuration is an accepting one, even if the system is blocked. To do this, we create a step where the action is stutter and the target configuration is identical to the source one. We check if an action is possible with this step as input, and if it is the case, we return it in an action composed with stutter. The `execute` function takes a composed action and composed configuration as an input. If the left side of the action is a stutter, we execute the rhs with a stuttering step as input like in the `actions` function, then return the configuration composed from the source lhs configuration and the target rhs configuration. If the action is enabled, we execute the lhs, then the rhs with resulting step and return the combination of both resulting target configurations.

4 Creating a Breakpoint Language

In the previous section, we showed how we built an architecture for our debugger that is independant from the breakpoint language, allowing to swap them depending on the need of the user. Before we give examples of how different languages can be useful, this section will cover the creation of one such language, to show which efforts are needed to define them and adapt them to our approach. We will create a temporal breakpoint language with a regular language expressivity. The language semantics is based on non-deterministic finite automata (NFA) with a binary vocabulary induced by subject-language predicates (valuated through the subjects `evaluate` function). The language outline is given using JavaScript in Listing 4. We will start by defining, through an example, a concrete JavaScript syntax (*breakpointAST*). Then we define a minimal semantics through an iSLI (*Semantics*). Finally, we briefly discuss how this language can be used as a compilation target for a more light-weight regular-expression based syntax.



(RegExp)

AtoCS = "IS_TRANSITION(Alice. goesIn)
BtoCS = "IS_TRANSITION(Bob. goesIn)"
breakpoint = "true" * . (AtoCS | BtoCS)

Figure 4. The breakpoint as expressed in Listing 4 in the form of a regular expression and a NFA

A JavaScript NFA Syntax. The example breakpoint that we use is based on a two-process mutual-exclusion specification. It looks-up a trace towards a configuration where one of the processes reaches the critical section. This condition can be captured using the (NFA) described in Listing 4. The graphical version of this NFA is also presented in the bottom of Figure 4. The `const breakpointAST` is associated to an object that encodes the NFA. The states are encoded using integers, the fanout of each state is represented as a list of objects with two fields: a guard – named *g*, and a target – named *t*. The automaton has three instance variables, encoding the *start* state (the integer 1 in our example), the *fanout* from each state given as a dictionary, and an *accepting* field which encodes the set of accepting states as a list of state ids. Note that, due to the space restrictions, some of the guards refer to the definition in the (RegExp) box, the substitution is considered implicit in the following.

A JavaScript iSLI for NFAs. The dependent semantics of the NFA language is defined by the class *NFASemantics*, which is instantiated with a model and a subject-language evaluate function (the two constructor parameters). The STR interface is satisfied by the 3 functions: *initial*, which simply returns a singleton array with the initial model state `this.model.start`; *actions*, which retrieves the fanout of its *configuration* argument and filters out the transitions for which the guard evaluation is false; and *execute*, which simply returns a singleton array with the target of the transition (line 23). Lastly, the *isAccepting* predicate allows checking the acceptance status of a given configuration by checking its inclusion in the *accepting* list of the syntax model. Note the semantic layering obtained through the call to the subject-language `evaluate` function performed with the *alien* guard

```
1  const breakpointAST = {
2      start: 1,
3      fanout: {
4        1: [{g: "true", t: 2},
5            {g: AtoCS, t: 3},
6            {g: BtoCS, t: 4}],
7        2: [{g: "true", t: 2},
8            {g: AtoCS, t: 3},
9            {g: BtoCS, t: 4}]},
10     accepting: [3, 4]
11 };
12
13 class NFASemantics{
14     constructor(model, evaluate){
15       this.model = model;
16       this.subjectEvaluate = evaluate; }
17     initial(){
18       return [this.model.start]; }
19     actions(input, configuration){
20       return this.model.fanout[configuration]
21                 .filter(a => this.subjectEvaluate
                      (a.g, input)); }
22     execute(action, input, configuration){
23       return [action.t]; }
24     isAccepting(configuration){
25       return this.model.accepting.includes(
             configuration); }
26 }
```

Listing 4. JavaScript semantics for our breakpoint language iSTR

syntax (a String) and the *input* in the `actions` function. To evaluate the guard, the subject-language `evaluate` function needs to compile the expression and to get its valuation on the *input*, which encodes a subject-language execution-step (computed by the *StepSynchronousComposition* operator).

Finally, we need to provide an *emptiness checker*, not shown in Listing 4, which performs the reachability query. Any search algorithm will work, however, we recommend using the *reduced reachability* procedure from [16], which can improve search scalability. Note that, syntactically, NFA are identical to Büchi automata, though the interpretation of the accepting configuration changes. Thus by replacing the *emptiness checker* component with a Büchi emptiness checking algorithm [9] we can obtain the more expressive ω-regular semantics, which subsumes linear temporal logic, for instance. The statechart-based Büchi automata breakpoints (scB$_{ba}$), presented in the following section, are based on this observation.

Regular-Expression Breakpoint Syntax. Given the NFA breakpoint semantics presented in the previous paragraphs, it is relatively simple to expose a regular-expression breakpoint syntax to the user, like the one illustrated in the (RegExp) box on the top-right of the Listing 4. In our implementation, discussed in the following section, we have relied on the Thompson construction [20] to convert the regular expressions to the equivalent NFA, followed by the removal of the ϵ transitions to reduce non-determinism.

5 A Temporal Multiverse Debugger for AnimUML

To illustrate the practicability of our approach, we have implemented a temporal multiverse debugger in JavaScript for the AnimUML modeling environment [10] [5]. A self-contained prototype is available at github.com/MatthiasPasquier97/temporal-breakpoints-for-multiverse-debugging. The following paragraphs discuss the implementation efforts before illustrating the three new classes of temporal breakpoints on a simple mutual exclusion example.

Figure 5 gives an overview of the implementation effort to create the prototype. As opposed to the previous work in [16], our implementation faithfully follows the formalization presented in the preceding section. The *Debugger Core* component semantics is implemented as a SLI, which encapsulates the debugger state in a specific configuration instance, and exposes the debug actions. The existing semantics of the AnimUML subject-language was used "as is" by wrapping it in an STR adapter. However we had to implement the step-evaluation function (`evaluateStep`), which was not available. We note however that the dynamic nature of the JavaScript implementation language eased the implementation effort. The *Finder* component was implemented from scratch, however we have isolated the synchronization and algorithmic components in an external standalone library, named *Z2MC*, which is available under a MIT License at https://github.com/teodorov/z2mc-javascript.

To demonstrate the independence between the debugger and the breakpoint language we have implemented three breakpoint languages natively, shown at the bottom of Figure 5: *1) Regular Expression* (reB), similar to stateful breakpoints [4], restricted to reasoning about the past (safety properties); *2) AnimUML statechart-based* (scB), also restricted to past reasoning, however syntactically more concise due to use of variables besides the control-states; *3) Buchi AnimUML statechart-based* (scB_{ba}), allowing both past and future reasoning (both safety and liveness properties).

The *Regular Expression* breakpoints implement the breakpoint language discussed in the previous section (Section 4). For the *AnimUML statechart-based* breakpoints, we have implemented the iSLI interface as an adapter over the existing AnimUML semantics. Here again the dynamic nature of the implementation language eased the task. The two advantages of the *scB* and scB_{ba} are that (1) statechart greatly improve the conciseness of the specifications compared to simple NFAs (2) in the context where the specifications themselves are AnimUML statecharts, we found it advantageous to provide a syntactically similar breakpoint language. Also note that a similar approach was previously proposed in [2], in the context of model-checking. However, in this paper, both the formalization and the implementation better isolate the subject-language from the dependent language (the breakpoint language). The emptiness checking algorithms paired with each of the breakpoint language is available in the *Z2MC* library.

The remainder of this section illustrates the three new classes of breakpoints, besides the configuration-based breakpoints, in the context of on a mutual exclusion specification given in Figure 6 (left-side). For simplicity here, the example breakpoints are all given textually in Listing 5 with one exception: the LTL breakpoint (line 14 - in Listing 5) that is was manually converted to the AnimUML büchi statechart shown in Figure 6 (right-side). This transformation, out of the scope of this contribution, can be easily automated [2].

To help the comprehension of this section, we will give a brief overview of the concept behind the algorithm we used as an example. Peterson's algorithm [17] is a mutual exclusion algorithm that allows two processes to share a single-use resource without interfering with each other. In this algorithm, the two process `alice` and bob have a `flag` that indicates whether it wants to enter the critical section. There is also a shared `turn` variable that indicates whose turn it is to enter the critical section. When a process wants to enter the critical section, it sets its `flag` to true and sets the `turn` variable to the other process attributed value. The process then waits until the other process sets its `flag` to false. Once the other process has released the resource and set its `flag` to false, the process can enter the critical section and perform its task. When the process is done, it sets its own `flag` to false, allowing the other process to enter the critical section. The algorithm ensures that only one process can enter the critical section at a time, and that each process takes turns entering the critical section.

Step Breakpoints: Looking at Execution Steps. Expressing conditions on execution steps expands the possibilities for debugging. First of all, the step breakpoints allow to reason on the action between the configurations. In their simplest form they can allow stopping the execution when a named action is reached, as illustrated with the first 3 step conditions (lines 7-9) in Listing 5. The model in Figure 6 contains two named transitions in the *alice* statechart (*wantsIn* from *Initial* to *Waiting* and *goesIn* from *Waiting* to *Critical*). The AnimUML iSLI a step-expression `IS_TRANSITION (<name>)` that evaluates to true when the transition named is fired on an execution step. The `AWantsIn` breakpoint will stop the execution when the model will execute a step with the *alice.wantIn* action. Furthermore, the step breakpoints allow reasoning on the delta changes between two consecutive configurations of a behavior (before and after an action). The `TurnChanged` breakpoint (line 10 in Listing 5 shows a

[5]The AnimUML modeling environment is available under an Eclipse Public License V2.0 at https://github.com/ESEO-Tech/AnimUML

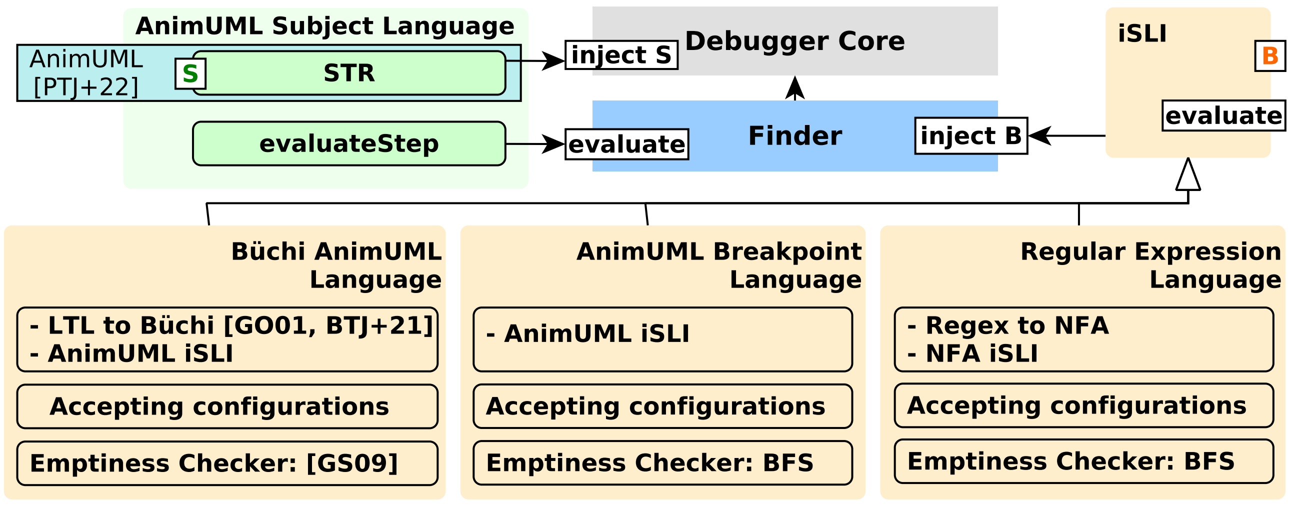


Figure 5. Overview of the implementation effort for obtaining temporal multiverse debugger for AnimUML

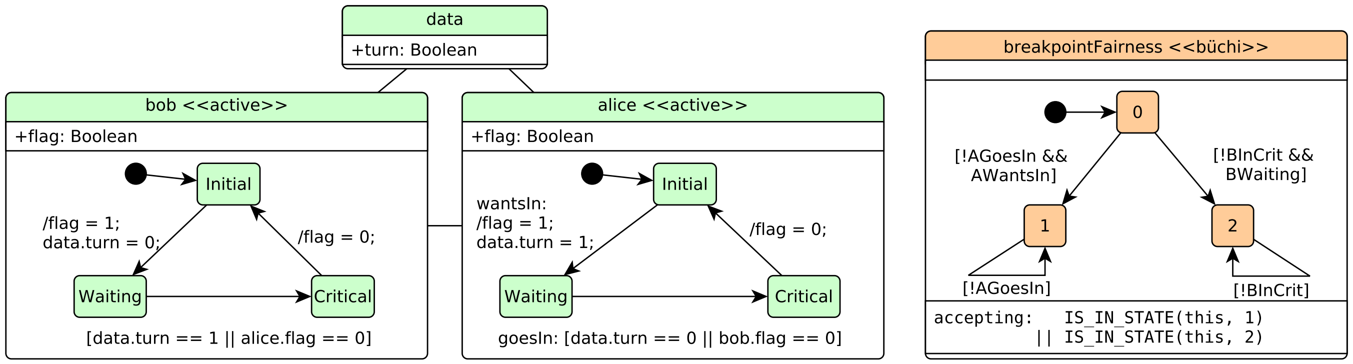


Figure 6. A mutual exclusion specification in UML statecharts along with an statechart-based büchi temporal breakpoint encoding a fairness condition.

breakpoint that will be triggered when the shared variable *turn* changes (note the @var-ref@ syntax for refering to the target configuration of a step. This feature can be used to implement the more classical halt-on-write breakpoint. Also it allows breaking on more specific differences between the variables.

Safety Breakpoints: Looking into the Past. The safety breakpoints, encoded here either using regular-expressions or using AnimUML statecharts, allow to capture complex sequences of events, in the spirit of stateful breakpoints [4] and scripted debugging [8], but adapted to the multiverse debugging context. The main point here is that the state of the breakpoint should be matched with the state of the subject, due to the need to navigate freely on all branches of the state-space. The *iteration* and *complexSeq* breakpoints (lines 13 and 14 in Listing 5) illustrate two scenarios. The *iteration* breakpoint stops the execution when *bob* reaches

```
1  //Configuration conditions
2  BWaiting      = | IS_IN_STATE(bob, bob.Waiting)|
3  BInCrit       = | IS_IN_STATE(bob, bob.Critical)|
4  NotBInCrit    = |!IS_IN_STATE(bob, bob.Critical)|
5  //Step conditions
6  AWantsIn      = | IS_TRANSITION(alice.wantsIn)|
7  AGoesIn       = | IS_TRANSITION(alice.goesIn)|
8  NotAGoesIn    = |!IS_TRANSITION(alice.goesIn)|
9  TurnChanged   = |data.turn !== @data.turn@|
10 //Regular expression breakpoints
11 iteration     = |true|*.(NotBInCrit*.BInCrit){10}
12 complexSeq    = (!AGoesIn*.BInCrit+){2}.(AgoesIn
        .(!AgoesIn && !BInCrit)*.BInCrit.(!AgoesIn &&
        !BInCrit)*).|true|*.TurnChanged
13 //LTL breakpoints
14 Fairness      = ! ( [] (  (AWantsIn -> <> AGoesIn)
15                        && (BWaiting -> <> BInCrit)))
```

Listing 5. Some breakpoints based on the regular expression langage

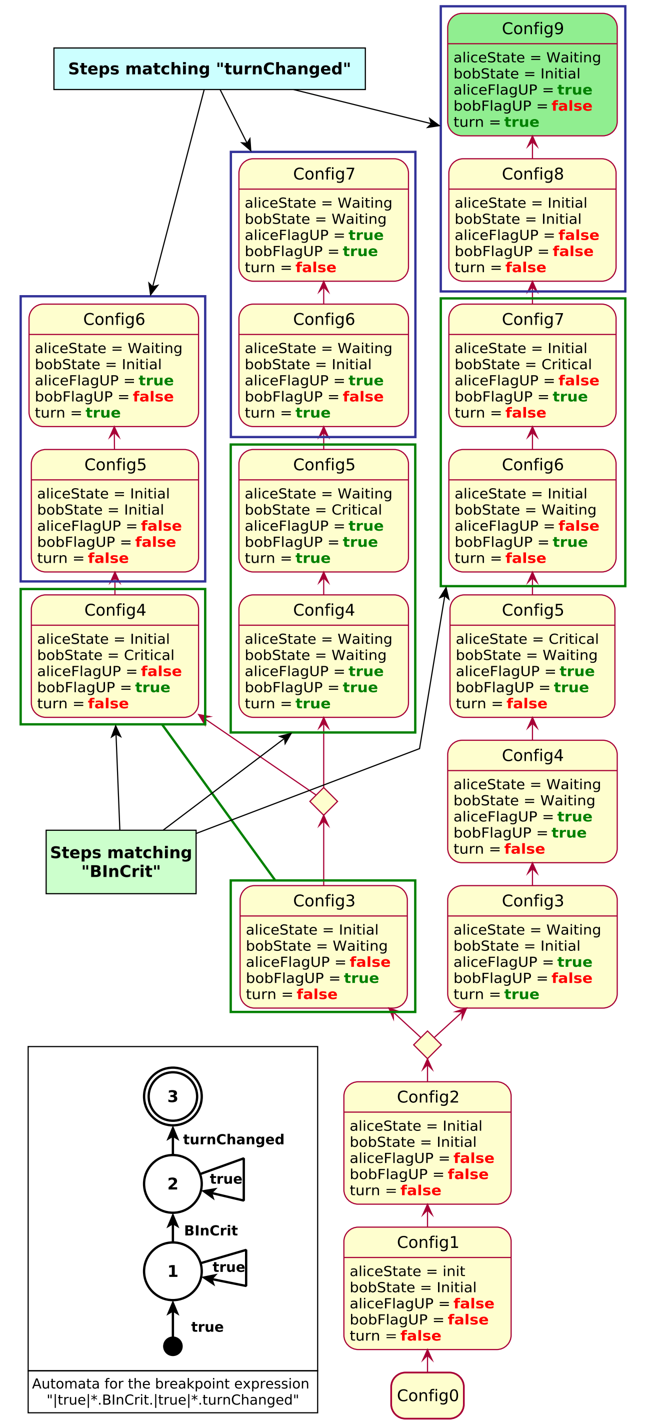


Figure 7. Some possible traces matching the breakpoint "|true|*.BInCrit.|true|*.turnChanged".

the critical section the tenth time. The *complexSeq* scenario looks up for a behavior where the *turn* variable changed after a sequence where *bob* passes twice through the critical section followed by the passage in the critical section of *alice* then of *bob*. While expressive this last example underlines one limitation of regular-expressions – they become large and difficult to understand fast. AnimUML statechart breakpoint can somewhat alleviate these difficulties, relying on the powerful syntactic sugar offered by the statechart formalism (variables and effects, hierarchical-states, orthogonal regions). Please also note that *complexSeq* scenario freely mixes configuration and step conditions. Figure 7 shows an example of the possible traces that can be obtained when running a breakpoint describing such sequences of events, overlayed with the execution steps matching *BInCrit* and *turnChanged* conditions. Note that the *turnChanged* condition observes the execution steps that change the turn variable.

Liveness Breakpoints: Talking about the Future. The expressivity of the regex and AnimUML statechart breakpoint is limited to regular languages, which encode only conditions on the past of the execution – something bad never happens. However, a complete system specification should also enable the desired behaviors to appear. In the literature, these conditions are known as liveness properties – something good is allowed to happen. Our prototype enables expressing breakpoint conditions like *Fairness* (line 14 in Listing 5) which will be triggered if there exists an infinite path (a loop in the state-space) where *alice* or *bob* want to get to the critical section, but they are not allowed in. It is expressed using LTL, which is a formal specification language used to reason about the temporal behavior of systems. LTL syntax consists of a set of logical operators, including temporal operators like "`X`" (next), "`[]`" (globally), "`<>`" (eventually), and "`U`" (until), combined with standard propositional logic operators such as "`!`", "`&&`" or "`||`".

As the Petteron algorithm respects the fairness property, we made a small demonstration on a faulty model in Figure 8. In the model, the turn variable mechanism is replaced by the possibility for alice to go from the Waiting state back to the Initial state to avoid a deadlock in the case where both actors want to enter the critical section at the same time. The Fairness Breakpoint then breaks when a loop is found where alice wants to get to the critical section (It enters the Waiting state), but never does.

We strongly believe that having different breakpoint languages will allow the community to find the balance between expressivity and ease of use. This section illustrates that our modular architecture can be implemented in practice and allows multiple formalisms for capturing temporal breakpoints during multiverse debugging. The next paragraphs discuss the current limitations of our proposal.

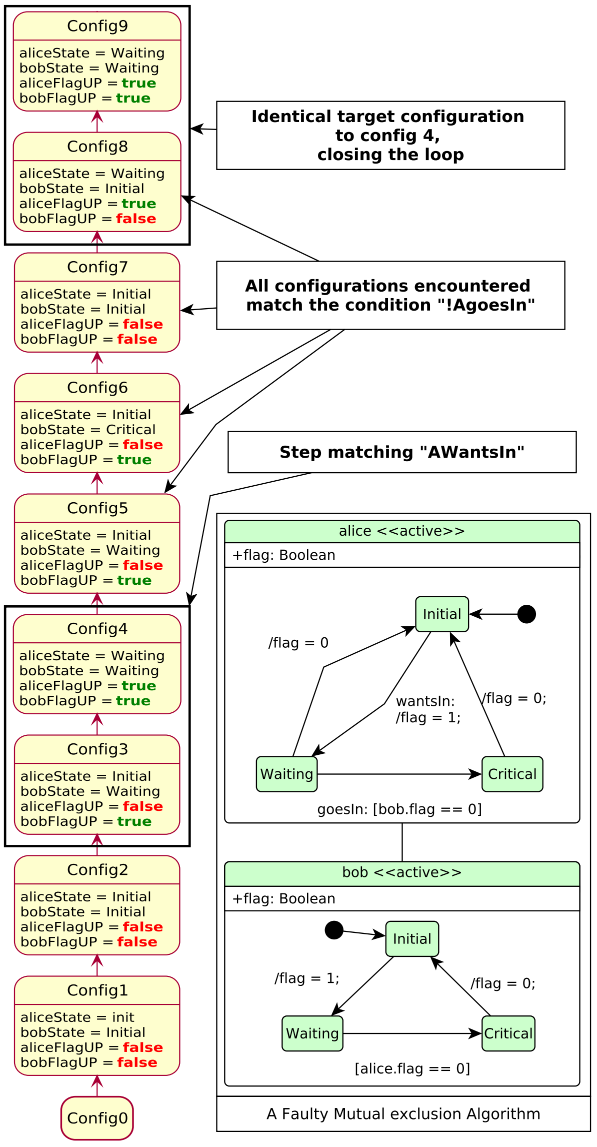


Figure 8. A trace obtained when running BreakpointFairness over a faulty mutual exclusion example

6 Discussion

We see some limits to this work. To keep the formalization simple, it only allows for one breakpoint lookup at a time. Allowing multiple breakpoints is possible, however care should be taken if the breakpoints interfere with each other in their composition with the system (for example, if one breakpoint blocks a behavior of the system which is not needed in the description of the path, but another breakpoint needs to observe it). Currently, our formalization does not allow step-in debug actions, thus not limiting the possible complexity of what is executed in a single step. To solve this problem, the action language could also implement the SLI interface, which would allow creating a step-in action that instantiates a new debugger on this language to explore the execution of this step, conceptually leading to a stack-like debugger hierarchy.

Among the improvements needed by the AnimUML prototype we can cite: the lack of a graphical interface, and experimenting with other languages (sequence diagrams for instance) for temporal breakpoints.

For now, the independence between the subject language and the debugger has only been tested on a guard action language as well as AnimUML, but we believe that our approach is applicable to other specification languages. Currently, we are working on instantiating it for the TLA+ specification language. In the future, we plan to investigate the cost of implementing the semantic language interface (SLI) proposed here for a highly-concurrent implementation language such as Erlang, which will bring new implementation challenges for multiverse debugging. In a more general way, the compatibility of our approach with existing debugging interfaces (like Java Debug Interface or Chrome DevTools Protocol Debugger) is dependent on the compatibility with our SLI interface. This need can be summarized by two main points. First, the possibility to capture the configurations encountered in the execution and compare them. Second, the presence of an evaluation function over the execution steps.

Finally, we need to investigate how temporal breakpoints work with reductions that are essential to explore bigger models. While they can make the exploration more efficient, they can also cut branches, preventing some breakpoints to be triggered. The goal the user is trying to reach must be taken into account when choosing a reduction, and not only try to counter the causes of state-space explosion.

7 Conclusion

The ever-increasing complexity of modern concurrent systems requires the use of highly non-deterministic specification languages, which challenge the limits of human understanding at all abstraction levels. This article introduces temporal breakpoints for multiverse debugging, which improve the expressivity of the stop conditions used during the diagnosis process. Our contribution is accompanied with a modular and compositional formal semantics, which eased the implementation of a prototype for the non-trivial AnimUML modeling environment.

References

[1] Martín Abadi and Leslie Lamport. 1991. The existence of refinement mappings. *Theoretical Computer Science* 82, 2 (1991), 253–284. https://doi.org/10.1016/0304-3975(91)90224-P

[2] Valentin Besnard, Ciprian Teodorov, Frédéric Jouault, Matthias Brun, and Philippe Dhaussy. 2021. Unified Verification and Monitoring of Executable UML Specifications: A Transformation-Free Approach. *Softw. Syst. Model.* 20, 6 (dec 2021), 1825–1855. https://doi.org/10.1007/s10270-021-00923-9

[3] Armin Biere, Cyrille Artho, and Viktor Schuppan. 2002. Liveness Checking as Safety Checking. *Electronic Notes in Theoretical Computer Science* 66, 2 (2002), 160–177. https://doi.org/10.1016/S1571-0661(04)80410-9 FMICS'02, 7th International ERCIM Workshop in Formal Methods for Industrial Critical Systems (ICALP 2002 Satellite Workshop).

[4] Eric Bodden. 2011. Stateful breakpoints: A practical approach to defining parameterized runtime monitors. *SIGSOFT/FSE 2011 - Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering*, 492–495. https://doi.org/10.1145/2025113.2025201

[5] Sagar Chaki, Edmund M. Clarke, Joël Ouaknine, Natasha Sharygina, and Nishant Sinha. 2004. State/Event-Based Software Model Checking. In *Integrated Formal Methods*, Eerke A. Boiten, John Derrick, and Graeme Smith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 128–147.

[6] Rick Chern and Kris De Volder. 2007. Debugging with Control-Flow Breakpoints. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development* (Vancouver, British Columbia, Canada) *(AOSD '07)*. Association for Computing Machinery, New York, NY, USA, 96–106. https://doi.org/10.1145/1218563.1218575

[7] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. 2018. *Handbook of Model Checking* (1st ed.). Springer Publishing Company, Incorporated.

[8] Thomas Dupriez, Guillermo Polito, Steven Costiou, Vincent Aranega, and Stéphane Ducasse. 2019. Sindarin: A Versatile Scripting API for the Pharo Debugger. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages*. Athens, Greece. https://doi.org/10.1145/3359619.3359745

[9] Andreas Gaiser and Stefan Schwoon. 2009. Comparison of Algorithms for Checking Emptiness on Büchi Automata. *ArXiv* abs/0910.3766 (2009).

[10] Frédéric Jouault, Valentin Besnard, Théo Le Calvar, Ciprian Teodorov, Matthias Brun, and Jerome Delatour. 2020. Designing, Animating, and Verifying Partial UML Models. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (Virtual Event, Canada) *(MODELS '20)*. Association for Computing Machinery, New York, NY, USA, 211–217. https://doi.org/10.1145/3365438.3410967

[11] Leslie Lamport. 1994. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.* 16, 3 (may 1994), 872–923. https://doi.org/10.1145/177492.177726

[12] Dorian Leroy, Benoît Lelandais, Marie-Pierre Oudot, and Benoit Combemale. 2021. Monilogging for Executable Domain-Specific Languages. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering* (Chicago, IL, USA) *(SLE 2021)*. Association for Computing Machinery, New York, NY, USA, 2–15. https://doi.org/10.1145/3486608.3486906

[13] Bil Lewis. 2003. Debugging Backwards in Time. *Computing Research Repository* cs.SE/0310016 (10 2003).

[14] Carmen Torres Lopez, Robbert Gurdeep Singh, Stefan Marr, Elisa Gonzalez Boix, and Christophe Scholliers. 2019. Multiverse Debugging: Non-deterministic Debugging for Non-deterministic Programs. In *33rd European Conference on Object-Oriented Programming*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. https://kar.kent.ac.uk/74328/

[15] OMG. 2017. Unified Modeling Language. https://www.omg.org/spec/UML/2.5.1/PDF

[16] Matthias Pasquier, Ciprian Teodorov, Frédéric Jouault, Matthias Brun, Luka Le Roux, and Loïc Lagadec. 2022. Practical Multiverse Debugging through User-Defined Reductions: Application to UML Models. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems* (Montreal, Quebec, Canada) *(MODELS '22)*. Association for Computing Machinery, New York, NY, USA, 87–97. https://doi.org/10.1145/3550355.3552447

[17] Gary L. Peterson. 1981. Myths About the Mutual Exclusion Problem. *Inf. Process. Lett.* 12 (1981), 115–116.

[18] Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. 2013. Expositor: Scriptable time-travel debugging with first-class traces. In *2013 35th International Conference on Software Engineering (ICSE)*. 352–361. https://doi.org/10.1109/ICSE.2013.6606581

[19] Philippe Schnoebelen. 2002. The Complexity of Temporal Logic Model Checking. *Advances in modal logic* 4, 393-436 (2002), 35.

[20] Ken Thompson. 1968. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM* 11, 6 (jun 1968), 419–422. https://doi.org/10.1145/363347.363387

[21] Qin Zhao, Rodric Rabbah, Saman Amarasinghe, Larry Rudolph, and Weng-Fai Wong. 2008. How to do a million watchpoints: Efficient debugging using dynamic instrumentation. In *Compiler Construction: 17th International Conference, CC 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 17*. Springer, 147–162.

Received 2023-07-07; accepted 2023-09-01

Cross-Level Debugging for Static Analysers

Mats Van Molle
mats.van.molle@vub.be
Vrije Universiteit Brussel
Belgium

Bram Vandenbogaerde
bram.vandenbogaerde@vub.Be
Vrije Universiteit Brussel
Belgium

Coen De Roover
coen.de.roover@vub.be
Vrije Universiteit Brussel
Belgium

Abstract

Static analyses provide the foundation for several tools that help developers find problems before executing the program under analysis. Common applications include warning about unused code, deprecated API calls, or about potential security vulnerabilities within an IDE. A static analysis distinguishes itself from a dynamic analysis in that it is supposed to terminate even if the program under analysis does not. In many cases it is also desired for the analysis to be *sound*, meaning that its answers account for all possible program behavior. Unfortunately, analysis developers may make mistakes that violate these properties resulting in hard-to-find bugs in the analysis code itself. Finding these bugs can be a difficult task, especially since analysis developers have to reason about two separate code-bases: the analyzed code and the analysis implementation. The former is usually where the bug manifests itself, while the latter contains the faulty implementation. A recent survey has found that analysis developers prefer to reason about the analyzed program, indicating that debugging would be easier if debugging features such as (conditional) breakpoints and stepping were also available in the analyzed program. In this paper, we therefore propose *cross-level* debugging for static analysis. This novel technique moves debugging features such as stepping and breakpoints to the base-layer (i.e., analyzed program), while still making interactions with the meta-layer (i.e., analysis implementation) possible. To this end, we introduce novel conditional breakpoints that express conditions, which we call *meta-predicates*, about the current analysis' state. We integrated this debugging technique in a framework for implementing modular abstract interpretation-based static analyses called *MAF*. Through a detailed case study on 4 real-world bugs taken from the repository of *MAF*, we demonstrate how cross-level debugging helps analysis developers in locating and solving bugs.

CCS Concepts: • **Software and its engineering** → **Automated static analysis**; **Software testing and debugging**.

Keywords: Static Program Analysis, Debugging

ACM Reference Format:
Mats Van Molle, Bram Vandenbogaerde, and Coen De Roover. 2023. Cross-Level Debugging for Static Analysers. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3623476.3623512

1 Introduction

Static analyses derive run-time properties of programs without actually running them. They provide the foundation for tools such as Integrated Development Environments, optimizing compilers, and quality assurance tooling. *Termination* is an important property for any static analysis, stating that the analysis always terminates even when the program under analysis does not. In application contexts such as compilers, analyses also ought to be *sound*, meaning that their results account for any possible execution of the program under analysis. For example, an analysis that determines whether integers can be stored in a unsigned variable, should only state that an expression will evaluate to a positive integer if this is the case for every possible program execution.

Unfortunately, analysis developers may make mistakes while trying to realise these properties. Such mistakes are often hard to locate and therefore fix. Debuggers have proven themselves as tools for locating the source of problems in an application. However, as Nguyen et al. [7] found in a survey conducted amongst 115 analysis developers, traditional debuggers are ill-suited for debugging a static analysis:

- *Debugging target mismatch:* a traditional source-level debugger targets the code of the analysis implementation. However, a bug usually manifests itself in an analyzed program. Therefore, for analysis developers, it can be easier to reason about the behavior of the analysis by looking at a specific analyzed program, rather than debugging the static analysis as a whole. Stepping features of the debugger should therefore also be able to target the analyzed program, rather than the analysis implementation itself.

- *Generic visualisation:* debuggers show generic information (e.g., the value of variables in the current call frame) about the implementation of a static analysis. As static analyses typically follow the same structure, *domain-specific* visualisations can be developed. Nguyen et al. find that these domain-specific visualisation help to understand the behaviour of the analysis, and help to locate bugs.

In this paper, we argue furthermore that the breakpoints from traditional debuggers are inadequate:

- *Shifting breakpoints to the base layer:* traditional debuggers that target the analysis implementation do not support placing breakpoints in the analyzed program. This makes debugging more difficult, since the analysis developer cannot easily suspend the analysis when a particular point in the analyzed code is reached.
- *Domain-specific conditional breakpoints:* conditional breakpoints enable developers to limit the number of times a debugged program is suspended at a breakpoint. Similar to regular breakpoints, we argue that the conditional ones ought to be placed in the analyzed code. However, they must be *cross-level*, meaning that they do not only express properties of the analyzed program (base level), such as the contents of an in-scope variable, but also properties about the global analysis state (meta level) at the point of its evaluation. Predicates for conditional breakpoints therefore become domain-specific, which renders expressing properties about the analysis state easier compared to expressing them in terms of implementation-specific data structures.

1.1 Contributions

In this paper, we propose *cross-level debugging* for static analysis. More specifically, we propose a debugger that moves stepping and breakpoints features to the analyzed program (base level), while still allowing for expressing properties about the analysis implementation (meta level) as conditional breakpoints. To this end, we propose domain-specific *meta-predicates* that can be used to formulate analysis-specific conditional breakpoints. Our debugger therefore crosses the boundary between the base level and meta level, and becomes cross-level. In summary our contributions are as follows:

- A novel debugging technique for static analysis called *cross-level* debugging, which includes domain-specific visualisations and stepping features that can step through each individual step of the analysis.
- Three categories of domain-specific *meta-predicates* that can be used as the conditions for our *cross-level conditional breakpoints.*
- An implementation for this debugger using *MAF*, a framework for implementing modular analyses for Scheme.

1.2 Motivating Example

We motivate the need for cross-level debugging of static analysis implementations through a hypothetical sign analysis that does not properly allocate the parameters of a function. The bug manifests itself during the analysis of the following Scheme program:

```scheme
1 ; define a function named "add"
2 (define (add x y)
3   (+ x y))
4 ; call the "add" function
5 (add 5 2)
```

When executed by a concrete interpreter, the program evaluates to 7. The corresponding analysis result for this program should be the + element of the sign lattice of abstract values (or its $\top$ element in case sound imprecisions are allowed). Imagine that the hypothetical sign analysis produces the $\bot$ lattice element instead, denoting the absence of sign information.

Without prior knowledge about this bug, it may be unclear what part of the analysis is to blame. Several analysis components may be at fault: the implementation of the abstraction of literals 5 and 2 to sign lattice elements, the implementation of the abstract + operation on these lattice elements, or the implementation of the abstract semantics of function calls and returns.

Inspecting the state of the analysis at the corresponding points in the analysed program would help to locate the bug in the analysis implementation. Unfortunately, regular debuggers are not well-equipped for this task. First, the analysis implementation does not exactly mirror the structure of the analysed program. Steps through the analysis implementation therefore do not necessarily correspond to steps in the analysed program. This motivates the need for moving stepping and breakpoint features to the analyzed program (base level) rather than the analysis implementation (meta level).

Second, regular debuggers do not understand the structure of the analysis state. For example, at line 3, it is expected that the analysis knows about variable y in the program under analysis. Although a regular debugger can visualize the state of the analysis implementation in terms of local and global variables, it does not provide an effective way to visualize the contents of, for example, the environments and stores that the analysis is manipulating. This motivates the need for *domain-specific visualisations* that show the analysis state on a more abstract level, rather than in terms of implementation-specific data structures.

Third, regular debuggers do not support analysis developers in formulating and testing hypotheses about the source of a bug in terms of program points from the program under analysis. For example, if the analysis developer suspects that the abstract semantics of function calls is to blame, it would be natural to place a breakpoint at the beginning of the add function. For the hypothesis that no addresses have been allocated in the store for the function's parameters, the following *domain-specific conditional breakpoint* can help reduce the number of steps required to test it:

```
(define (add x y)
   (break (not (store:contains "y")))
   (+ x y))
```

Note that this breakpoint has not been formulated in terms of the program under analysis, but in terms of the state of the analysis when it reaches the corresponding program point in the program under analysis. This motivates the need for *cross-level conditional breakpoints* that can be placed in the analyzed code itself.

2 Effect-Driven Modular Static Analysis

In this paper, we focus on static analyses defined as abstract definitional interpreters, which use global store widening and are effect-driven in their worklist algorithm. In the next few sections we introduce each of these parts of our target static analysis, and illustrate how bugs can arise in their implementation.

2.1 Abstract Definitional Interpreters

Abstract interpretation [3] is an approach to static analysis where an analyser is derived by starting from the concrete semantics of the language under analysis, and then abstracting parts of this concrete semantics. As an example, consider a language that consists of numeric literals, addition (+) and subtraction (−) or any combination thereof. In its concrete semantics, numeric literals evaluate to themselves and addition and negation are defined as usual.

An abstraction of this language could abstract each number to its *sign*. In this semantics, the abstraction for 5 would be +. We also write that $\alpha(5) = +$, where α is called the *abstraction function*. The abstract versions of the addition ($\hat{+}$) and subtraction ($\hat{-}$) operations have to be defined differently. For example, summing two positive numbers results in a positive number. However, summing a positive and negative number could result in either a positive or a negative number. To remain sound, an analysis has to account for both possibilities. Hence, a third value is introduced called *top* (denoted by the symbol $\top$) expressing that the sign of the number could be either negative or positive. A final value called bottom (denoted by $\bot$) is included to express the absence of sign information. The set of these values forms a *mathematical lattice*, meaning that a partial order ($\sqsubseteq$) and a join operation ($\sqcup$) can be defined. As illustrated above, abstract operations are often non-trivial to implement and could result in subtle issues with the result of the analysis.

Van Horn et al. [8] propose a recipe for deriving static analyses by systematically abstracting the small-step operational semantics of a programming language. More recently, this recipe has been transposed to the context of *definitional interpreters* [5, 10]. Definitional interpreters are a way of formally specifying programming language semantics through an interpreter implementation. These interpreters are usually formulated in a *recursive* way and proceed by case analysis on the type of expression that is being analyzed. For example, for evaluating a number literal, the following abstracted semantics can be used:

```
eval expr :=
   match expr with ℕ n → $\alpha(n)$ ; ... end
```

To satisfy our soundness requirement, this semantics needs to be *exhaustive*, meaning that it has to explore any possible program path that might occur at run time. For example, the analysis might be unable to compute the truth value of an `if` condition precisely (e.g., the value of `(> x 0)` is imprecise if x is $\top$). In those cases, the analysis has to explore both the consequent and the alternative branch, as either might be executed in a concrete execution. Such a semantics can be formulated as follows:

```
eval expr :=
   match expr with
      (if cnd csq alt) →
         let $v_{cnd}$ = eval(cnd)
             $v_{csq}$ = if isTrue ($v_{cnd}$) then eval(csq) else ⊥
             $v_{alt}$ = if isFalse($v_{cnd}$) then eval(alt) else ⊥
         in $v_{csq} \sqcup v_{alt}$
      ...
   end
```

Note that `isTrue` and `isFalse` may succeed simultaneously if the truth value of the condition is imprecise. The excerpt depicted above demonstrates that the implementation of an abstract definitional interpreter is non-trivial. Throughout the implementation, there is a need to account for all possible concrete executions —which leads to subtle bugs when implemented incorrectly.

2.2 Memory Abstraction

To analyze programs that include variables, some kind of memory abstraction is required. In the recipe by Van Horn et al. [8] the interpreter's memory is modelled as a combination of an environment, which represents the lexical scope in which a particular program state is executed, and a store which represents the program's memory. An environment is modelled as a mapping from variables to addresses, and a store is modelled as a mapping from these addresses to actual (abstract) values.

The original recipe includes the store in every abstract program state. In the worst case, this results in an exponential number of program states [9]. One solution to this problem is to widen these (per-state) local stores into a single global store. This global store is then used across all global states, reducing the state space from an exponential to a cubic one. The same approach is taken in the static analysis for which we propose a debugger in this paper. However, our ideas also translate to analyses that do not incorporate store widening.

2.3 Effect-Driven Modular Analyses

The analysis presented in this paper is *modular* [4]. In a modular analysis, the program under analysis is split into *components* that are analyzed separately from each other. Examples of such components are function calls [16], classes or spawned processes [18]. In practice, however, components might depend on each other through shared variables or return values. The result of analyzing some component B might therefore influence the analysis result for a component A, if the analysis of A depends on the analysis results of B.

Nicolay et al. [16] describe an algorithm for the modular analysis of higher-order dynamic programming languages. In higher-order dynamic programming languages, such as Scheme, the exact components of a program and their dependencies are not known before the program is executed. Each time a component is discovered, it is added to a *worklist* that keeps track of the components to analyze next. Whenever the results for a component change, its dependent components (e.g., through a shared variable) are added to this worklist and eventually reanalyzed to take the new results into account. The algorithm repeats itself until the workist is empty. This process results in a *dependency graph* that consists of components and the store addresses (representing shared variables and return values) through which they depend on each other.

3 Approach

In this section, we introduce the design of our analysis-tailored debugger. First we discuss its visualisation, then its stepping and breakpoint functionality, and finally we introduce our novel meta-predicates for cross-level conditional breakpoints.

3.1 Visualising the analysis state

The first feature of our debugger is its visualisation of the analysis state. This visualisation for the `factorial` program is depicted in fig. 1. The visualisation consists of four parts: the code that is being analyzed, a graph of the components and their dependencies, an overview of the global store, and a visualisation of the worklist algorithm.

The component graph (C) visualizes the components discovered so far and their dependencies. Colored in green are the components themselves, and colored in blue are the *store locations* (addresses) on which the components depend. The component currently under analysis is highlighted using a purple border. Each of the edges depicts dependencies on these store locations and their direction indicates the *flow of values*. For example, the call to the factorial function (depicted by the node labeled `Call...`) both reads (from its recursive call) and writes to its return value.

The global store visualisation (D) depicts the addresses and their corresponding values that are currently in the global store. Highlighted in yellow are addresses that are updated during the interval between the previous and current breakpoint, while highlighted in green are addresses that are added during that time frame.

Finally, the worklist visualisation (E) depicts the current contents of the worklist. Its order corresponds to the order in which components will be removed from the worklist and therefore shows their analysis order.

3.2 Stepping and regular breakpoints

Recall that analysis developers prefer to reason about a specific manifestation of a bug in an analyzed program, rather than debugging the analysis implementation as a whole. We achieve this in two ways. First the code is presented prominently in the interface of the debugger (area A). Second, the analysis developers step through and *break* in the analyzed program instead of through the analysis implementation itself. This is important since unsound results often occur in specific parts of the analyzed the program. Setting breakpoints and stepping through the analyzed program makes it easier to pin-point the problem, and reason about how the analysis proceeds for the analyzed program.

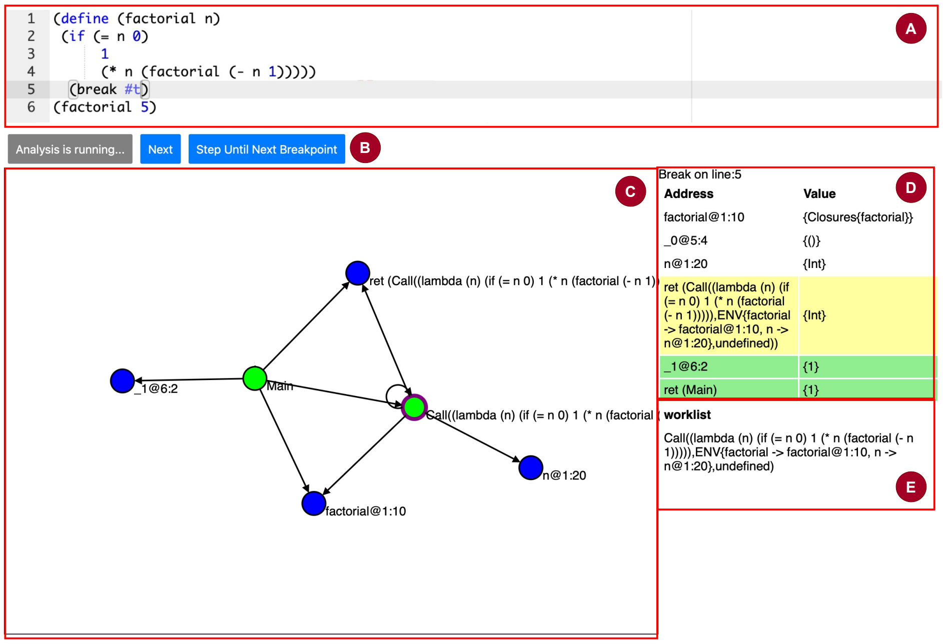
Similar to some debuggers for JavaScript, we choose to represent breakpoints as expressions in the analyzed program. This allows for more expressive freedom, since these expressions can be placed in arbitrary locations inside the program (i.e., in a specific subexpression, rather than on a specific line) and can reuse the same parsing facilities as the one available for the analyzed program.

Our debugger provides two types of stepping (as depicted in area B). The first type continues analysis until the next breakpoint is reached. The second type of stepping allows the developer to step over each expression in the analyzed program. Note that, because of our effect-driven worklist algorithm, this stepping feature never steps into function calls, since those are only analyzed once the component of the caller has been fully analyzed. However, once the analysis of a component is complete, this stepper continues stepping as before in the component that is analyzed next.

Recall that the branches of an if-expression need to be evaluated *non-deterministically*, in case the truth-value of the condition cannot be determined precisely. In that case, the stepper steps over these branches *sequentially* (evaluating the consequent branch first and then the alternative branch), and displays the state of the analysis accordingly.

3.3 Cross-Level Conditional Breakpoints

Conditional breakpoints are used in traditional debuggers to suspend the program once a particular condition is reached. These conditions are usually expressed in terms of program variables and predicates that act upon them. This type of breakpoint is especially important for static analyses where each program part can be analyzed more frequently than in their concrete execution. Hence, analysis developers need


```
1 (define (factorial n)
2  (if (= n 0)
3       1
4       (* n (factorial (- n 1)))))
5   (break #t)
6 (factorial 5)
```

A
Analysis is running...
Next
Step Until Next Breakpoint
B
C
ret (Call((lambda (n) (if (= n 0) 1 (* n (factorial (- n 1))
_1@6:2
Main
Call((lambda (n) (if (= n 0) 1 (* n (factorial (
n@1:20
factorial@1:10
D
E

Break on line:5

Address	Value
factorial@1:10	{Closures{factorial}}
_0@5:4	{()}
n@1:20	{Int}
ret (Call((lambda (n) (if (= n 0) 1 (* n (factorial (- n 1))))),ENV{factorial -> factorial@1:10, n -> n@1:20},undefined))	{Int}
_1@6:2	{1}
ret (Main)	{1}

worklist

Call((lambda (n) (if (= n 0) 1 (* n (factorial (- n 1))))),ENV{factorial -> factorial@1:10, n -> n@1:20},undefined)

Figure 1. Debugger visualisation, which features the following components: the code (A), debugger controls (B), component graph (C), store visualisation (D) and worklist visualisation (E).

.

conditional breakpoints that can express conditions on the current state of the analysis. We call these kind of predicates *meta-predicates* since they do not express constraints on the program containing the breakpoints (i.e., the analyzed program) but rather on the meta layer above it (i.e., the analysis implementation).

Based on the parts of the analysis’ state, we split our meta-predicates in three categories: store-based, worklist-based and lattice-based meta-predicates. A full list of predicates, split according to these categories is depicted in table 1.

3.3.1 Categories of Meta-Predicates.

Store predicates. Our store-based meta-predicates express conditions on the state of the global store. We propose two meta-predicates: `store:lookup` and `store:changed?`. The first predicate enables looking up a value on a specific address in the current global store. The argument of this meta-predicate must correspond to the string representation of the store address as displayed in the visualisation. If the address is absent, the meta-predicate returns `false`. This allows expressing conditional breakpoints that break on the absence of a particular store address. Note that the values returned by this predicate are *abstract* rather than concrete. Therefore, operations on these values can only be applied using the `lattice` meta-predicates.

The second predicate, called `store:changed?`, returns `true` whenever a particular address in the store has changed since the last break. It returns `false` whenever the value on that address has not changed or whenever the address could not be found.

The latter predicate is especially useful when components are (re-)analyzed frequently without actually changing any address of interest. Those re-analyses can simply be executed without breaking, therefore saving the analysis developer’s time.

Table 1. Overview of the meta-predicates in our debugger.

Store predicates	**Lattice predicates**	
`store:lookup`	`lattice:integer?`	`lattice:vector?`
`store:changed?`	`lattice:pair?`	`lattice:car`
Worklist predicates	`lattice:real?`	`lattice:cdr`
`wl:length`	`lattice:real?`	
`wl:prev-length`	`lattice:char?`	
`wl:component`	`lattice:bool?`	
`wl:prev-component`	`lattice:string?`	

Lattice predicates. To interact with the values returned from `store:lookup`, we provide an interface to the abstract lattice operations as `lattice` meta-predicates. We divide these predicates into two sub-categories: type-checking predicates, and reified abstract operations.

For the former category, we provide type-checking predicates for Scheme’s primitive values: integers, reals, characters, strings, booleans, pairs and vectors. The latter category provides operations on these datatypes, such as `lattice:car` and `lattice:cdr` to retrieve the first and second element of a pair respectively.

While the type-checking predicates return a boolean value that can be used for deciding whether to break, the abstract domain operations always return an abstract value. Therefore, a type-checking predicate is always needed to use an abstract operation in the condition of a conditional breakpoint.

Worklist predicates. Finally, we introduce predicates concerning the current state of the worklist: `wl:length` and `wl:component`. The former predicate returns the current length (as a concrete number) of the worklist. This length corresponds to the number of components that are scheduled for analysis. The latter returns the name of the component that is currently being analysed. This predicate is rather redundant for context-insensitive analyses, since the location of the breakpoint already implies which component is being analyzed. However, when the analysis is configured with a form of context sensitivity [17] (e.g., the last-k callers on the stack), multiple components might be created for the same function.

In addition to the aforementioned predicates, we also propose *history-aware* variants of them. These variants correspond to the `wl:prev-length` which returns the length of the worklist at the previous breakpoint, and `wl:prev-component` which returns the previously analyzed component. These predicates can be used to detect unusual behavior of the worklist algorithm. For example, a worklist that does not shrink in size could be indicative of a bug in the worklist algorithm itself. Furthermore, frequent re-analyses of the same component could hint that the analysis is not terminating. Using these meta-predicates, analysis developers can express invariants and expectations about the behavior of the worklist algorithm.

3.3.2 Examples.

In this section, we briefly show some examples of conditional breakpoints to illustrate the synergy between the different categories of meta-predicates.

```
(1) (break #t)
(2) (break (> (abs (- (wl-length) (wl:prev-length))) 100))
(3) (break (and (sto:changed? "adr")
                (lattice:string? (sto:lookup "adr"))))
(4) (break (lattice:char?
               (lattice:car (sto:lookup "adr"))))
```

Some brief examples are depicted in the listing above. (1) depicts a conditional breakpoint that always breaks, thus behaving like a regular breakpoint. In (2), the difference in length of the worklist is computed and some threshold (i.e., 100) is used to break. This breakpoint can be used to detect rapidly growing or shrinking worklists. (3) combines multiple predicates together using a conjunction (i.e., and). In this case the breakpoint will be triggered when the address *adr* has changed and the abstract value at that address in the store can be a `string`. Finally, (4) depicts a combination of type-checking lattice predicates, and lattice operations. In this case, the operation `car` is used to obtain the first value of a pair, and `lattice:char?` is used to check whether the value is a character.

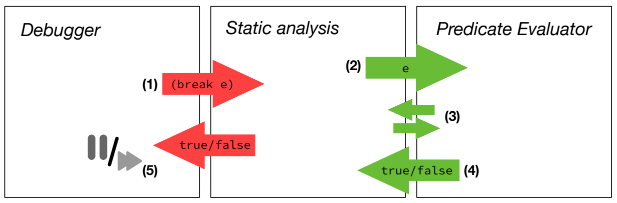


Figure 2. Interactions between the debugger and the meta-predicate evaluator.

Table 2. Summary of bugs from the Github repository.

Commit	Description
a2f43f6	Implemented `car` as `cdr`
1a3c6be	`vector-set!` ignores its own index
08bbe43	Variable arguments are ignored
8b98b9b	Unnecessary triggering of effects

3.3.3 Predicate evaluator.

Conditional breakpoints are evaluated in a separate evaluator which we call the *meta-predicate evaluator*. This evaluator has access to the current state of the analysis but cannot change it. Although the meta-predicate evaluator evaluates arbitrary Scheme expressions, these Scheme expressions cannot influence the results of the program under analysis. We argue that this is necessary for a clear separation between the debugging facilities and the analysis implementation to be maintained. The interactions between the debugger, static analyser, and meta-predicate evaluator are depicted in fig. 2.

The evaluation of a meta-predicate proceeds as follows. First, the `break` expression is analyzed by the static analysis (1). Then, since the static analyzer does not include semantics for evaluating predicates of those break expressions, the predicate expression e is passed to the predicate evaluator (2). Third, the predicate evaluator computes the truth value of the predicate e by querying the state of the static analysis (3). Finally, the computation results in a boolean value (true or false) which is returned to the static analysis (4). Based on this value the debugger decides whether to pause the analysis and show intermediate analysis results in its interface (5).

4 Evaluation

In this section we evaluate our approach through a case study. We first discuss the details of this evaluation method, and then demonstrate how our debugger supports locating 4 real-world bugs in the implementation of *MAF*.

4.1 Evaluation Method

We evaluated our approach by querying the *MAF* repository on Github[1] to find soundness related bugs. To this end, we queried for keywords such as: "bug" and "fix", From the results of this query we selected 4 real-world soundness bugs which are summarized in table 2. Additionally, to illustrate how termination issues can be debugged, we introduced a synthetic bug that affects the *worklist algorithm*.

Then, based on the fixes introduced in the aforementioned commits, we reintroduced the bugs back into the analysis itself, adapting the bug to the current state of the framework if necessary. Each case corresponds to one re-introduced bug in isolation, in order to avoid the effects of multiple bugs influencing each other and to replicate the precise environment in which the bug was originally found and fixed.

4.2 Studied Cases

The following cases correspond to re-introduced bugs found in the *MAF* repository, and to one synthetic bug introduced specifically to study the effectiveness of our worklist meta-predicates. For each case, we first detail the corresponding real-world or synthetic bug, then we show how it was resolved, before describing a scenario of successive interactions with our debugger that will lead to the bug being located. In the remainder of this section, we use ● to indicate the location of a breakpoint.

4.2.1 Implemented `car` as `cdr`.

In Scheme, pairs are constructed using the primitive `cons`. For example, the expression `(cons 1 2)` denotes a pair that consist of 1 as its first element (also called the `car`) and 2 as its second element (also called the `cdr`). In the bug studied in this first case, the `car` value was used for both the `car` and `cdr` of the pair allocated in the store.

We illustrate this bug in the program depicted below. This program provides an implementation for a *bank account*. The account is represented by a pair consisting of the account name and the current balance of that account (line 1-2). The functions `add-to-balance` (line 3-4) and `balance` (line 5-6) change and retrieve the balance of the bank account respectively.

```
  1   (define bankAccount
● 2     (cons "Lisa" 1983))
  3   (define (add-to-balance account amount)
  4      (set-cdr! account (+ amount(cdr account))))
  5   (define (balance account)
  6      (cdr account))
● 7   (add-to-balance bankAccount 10)
● 8   (balance bankAccount)
```

The analysis result for this program, produced by the buggy analysis implementation, is the pointer to `Lisa` instead of the expected value `integer`, rendering the analysis unsound.

Instrumenting the abstract definitional interpreter to output the analysis state at each evaluation step results in a large amount of unstructured information. Instead, we are interested in the analysis state for specific locations in the analyzed program. Recall that the store shows that the value of balance is `string`. To find the origin of the bug we start by checking whether `bankAccount` is still a `pair` consisting of a `string` and an `integer` after changing its balance. To this end, we place the following breakpoint before '`(balance bankAccount)`' (line 8), which breaks whenever the contents of the store has the expected structure.

```
(break (and
   (lattice:pair? (store:lookup "bankAccount@1:9"))
   (lattice:string? (lattice:car (store:lookup "bankAccount@1:9")))
   (lattice:integer? (lattice:cdr (store:lookup "bankAccount@1:9")))))
```

The inserted breakpoint does not suspend the analysis, meaning that the address does not point to a `cons` cell of the expected structure. Therefore, the bug has already occurred in the previous part of the program. A possible culprit could be the `set-cdr!` primitive, which mutates the `cdr` component of a pair. To test this hypothesis, we place the same breakpoint before `add-to-balance` (line 7). Again our analysis does not suspend, meaning that the structure of the pair is not affected by `set-cdr!`. Therefore the primitive `cons` itself could be the source of the bug. We test this by using the same breakpoint, but placing it right after the allocation (line 2). Again, this breakpoint does not result in a suspension of the analysis. We have now located that the implementation of `cons` itself is most likely to blame. To test this hypothesis, we reduce our conditional breakpoint to break whenever the `cdr` contains a value of an unexpected type (i.e., a value other than an `integer`).

```
(break (not (lattice:integer?
   (lattice:cdr (store:lookup "bankAccount@1:9")))))
```

Finally, the analysis suspends, which means that the bug resides in the implementation of the abstract allocation of the pair.

4.2.2 `vector-set!` ignores its own index.

In Scheme, *vectors* represent collections of a fixed size whose elements can be accessed in constant time. A vector can be allocated using the `make-vector` primitive which needs the length of the vector and an initial value for each position in the vector. Lookup and mutation are provided using primitives `vector-ref` and `vector-set!` respectively. In this example, we investigate a bug in the latter primitive.

The bug is located in the implementation of `vector-set!`. Recall that in order for an analysis to be sound, it must account for all possible program behavior. To this end, the implementation of `vector-set!` must join the previous values of the changed cell with the new value. Unfortunately, in this bug, only the old value was taken into account and the new value was simply ignored.

We demonstrate this bug with the program depicted below:

```
1   (define (change-age user age)
2      (vector-set! user 0 age))
3
```

[1] https://github.com/softwarelanguageslab/maf

```
  4   (define (paid user)
● 5     (vector-set! user #f))
  6
  7   (define (set-name user name)
● 8     (vector-set! user 1 name))
  9
 10   (define (get-name user)
 11     (vector-ref user 1))
 12
 13   (define new-user (make-vector 3 #f))
 14
 15   (change-age new-user 21)
 16   (paid new-user)
● 17  (set-name new-user "Steve")
 18   (define name (get-name new-user))
● 19  name
```

We expect that the result of the analysis will be the value of the final expression (i.e., the value of the variable name). Since the name of the user is supposed to be a string, the abstract value associated with the address corresponding to name in the store should at least contain a string. However, the analysis results in `false` only. To debug this problem, we start by placing a breakpoint after name (line 19) .

```
(break (store:lookup "name"))
```

This breakpoint suspends the analysis whenever the variable name is added to the store. We observe that the analysis suspends at this breakpoint, meaning that the analysis reaches the final expression and the variable has been correctly allocated.

We shift our attention to functions `paid` (line 16) and `set-name` (line 17), which both change the contents of the vector. We test whether calling these functions has an unexpected effect on the allocation of the vector. To this end we add the following breakpoint after the execution of these functions (line 17).

```
(break (store:lookup "PtrAddr((make-vector 3 #f))"))
```

Since the breakpoint suspends the analysis, the vector is still properly allocated after the calls to these functions have been analysed. We also note that 'Steve'   '21' have been added to the store.

Our `set-name` and `paid` functions are both implemented using a `vector-set!`. The expected semantics for this primitive is that it reads the current contents of the vector and updates the value at the specified index. Therefore, the value at the store address of this vector is supposed to change after the primitive has been executed. To verify whether this is the case, we place a breakpoint on line 5 and on line 8 to suspend the analysis whenever the store *has not* changed.

```
(break (not (store:changed? "PtrAddr((make-vector 3 #f)))"))
```

This results in the analysis suspending at both line 5 and 8, meaning that the vector operations did not have the desired effect. We can conclude that the bug is therefore situated in the implementation of `vector-set!`.

4.2.3 Variable arguments are ignored.

. Functions in Scheme can be defined to accept a variable number of arguments. This is expressed using a '.' in the function definition, followed by the variable which will collect any excess arguments.

The program depicted below illustrates this feature. The program defines two functions: `sum` and `compute`, and calls the `compute` function as its last expression.

```
  1   (define (sum . vs)
  2     (define (aux l)
● 3       (if (null? l)
  4           0
  5           (+ (car l) (aux (cdr l)))))
  6
● 7     (aux vs))
  8
  9   (define (compute initial)
 10     (+ initial (sum 1 2 3 4)))
 11
● 12  (compute 0)
```

The expected result of the analysis is + (in case of a sign analysis). However, for this bug, the analysis result is ⊥. An analysis result of ⊥ may indicate that the program under analysis does not terminate or that the analysis is incomplete. As the program depicted above clearly terminates with value 10 when executed by a concrete interpreter, this analysis result is unsound.

We add a normal breakpoint to each component (i.e., on lines 3, 7, and 12) of the program to determine which components can be analyzed. The analysis suspends for the `main` and `compute` components but does not for the `sum` component. We conclude that the call to the `sum` function must have failed, which could be related to its use of a variable number of arguments. However, our debugger cannot determine a more precise cause for the bug, and further debugging in the analysis implementation is required.

4.2.4 Ever-growing worklist.

Although the bug studied in this case is synthetic, it could easily manifest itself while implementing a worklist algorithm. The bug we introduce precludes the worklist from reducing in size as the component under analysis is taken but not removed from the worklist. As a consequence, the analysis never terminates. We illustrate this problem with the `factorial` depicted below:

```
  1   (define (factorial n)
  2     (if (= n 0)
  3         1
  4         (* n (factorial (- n 1)))))
● 5   (factorial 5)
```

Since the analysis does not terminate, our debugger never displays a visualisation of its final state. To suspend the analysis we use regular breakpoints (i.e., `(break #t)`), and place them after line 5. We can now step through the analysis state. We notice that each time 'Step Until Next Breakpoint' is pressed, the contents of the worklist remains the same and the analysis' state does not change. To test whether the analysis *makes progress*, we replace our regular breakpoint by a conditional one. This breakpoint suspends the analysis whenever the current component is the same as the previous component, and when the length of the worklist does not change.

```
(break (and (eq? (wl:component) (wl:prev-component))
            (= (wl:length) (wl:prev-length))))
```

Again, the breakpoint suspends on every analysis step, meaning that the same component is re-analyzed in each iteration of the worklist algorithm. This makes it clear that the current component is never removed from the worklist.

4.2.5 Unnecessary triggering of dependencies.

As explained in section 2.3, the analysis is backed by an effect-driven worklist algorithm. Components are re-analyzed when one of their dependencies changes. We say that a dependency *triggers* the reanalysis of a component, meaning that the component is added to the worklist for reanalysis. For example, the analysis result of a function *A* depends on its arguments, which are represented by store addresses in our global store. Whenever the abstract values for one of these store addresses changes, it triggers the reanalysis of function *A*. In this bug, dependencies are triggered even though the (abstract) value of their referenced store address no longer changes. This results in a non-terminating analysis, since components continue to be added to the worklist even if no new information can be derived.

We illustrate this bug by reusing the example program from section 4.2.3. The analysis of this program is infinite in the buggy analysis implementation. We add breakpoints to the body of each component of this program to make sure that no component *in particular* is analyzed continuously. Stepping through this program a number of times reveals that a single component *is* being reanalyzed continuously: the `aux` component.

To reduce the number of times the analysis is suspended, we remove all other breakpoints except the breakpoint in the `aux` function. Since a component is only reanalyzed when one of its dependencies changes, we are interested in the argument of `aux`. Therefore, we adapt this breakpoint to suspend the analysis only when the `l` no longer changes:

```
(break (not (store:changed? "l@2:17")))
```

As a result, the analysis suspends less frequently and we can step directly to the problematic infinite behavior of the analysis. Additionally, this debugger interaction gives us an indication of which store address is to blame, and which type of value is associated with it. This makes it easier to find the root cause of the bug in the analysis implementation by focussing on that specific address or looking into the implementation of lists. However, additional logging in the analysis implementation is required to learn more about the dependency triggering mechanism.

4.3 Discussion

Table 3 depicts on overview of the features of our debugger (columns) and the re-introduced bugs considered in the case studies (rows). The table indicates which debugging features were used to understand and locate each bug in the analysis implementation. In the case studies, predicates concerning the worklist are primarily used for solving bugs related to the termination of the analysis (bug 4). The store predicates are used in most of the case studies. The reason for their frequent usage is two-fold. First, the lattice predicates operate on abstract values from the store. Thus, each time a lattice predicate is used, at least one store predicate is required. Second, many bugs involve the store in some capacity. For example, bug 1 occurs because of a mistake in the *allocation* of pairs. Bug 2 is similar, in that the resulting value in the store is incorrectly updated, or not updated at all. The usage of the store meta-predicates in bug 5 is more subtle, here it is used to detect the absence of changes to the store in order to break when dependencies are triggered for addresses that no longer change.

Table 3. Overview of all the meta-predicate categories used for solving the bug

	Regular Break	Store	Worklist	Lattice
Bug 1		✓		✓
Bug 2		✓		
Bug 3	✓			
Bug 4	✓		✓	
Bug 5	✓	✓		

Bug 3 is interesting since it precludes components from being analyzed. Even worse, by preventing a component from being analyzed, its return value is always ⊥ which causes the analysis to halt early. In the program used for illustrating bug 3, this problem was rather obvious (i.e., the analysis results were empty). However, for larger programs, finding which component was prevented from being analyzed might be more difficult. Breakpoints related to the set of analyzed components (i.e., the seen set) might help to locate these components. However, such breakpoints are not included in our debugger and require further investigation. Therefore, only non-conditional breakpoints were used for debugging bug 3 in the case study.

5 Limitations & Future Work

As illustrated in the debugging scenario for bug 3, our current approach lacks meta-predicates to deal with components that fail to be analyzed. We argue that additional breakpoints that express properties on the dependency graph and the set of seen components can partially solve this problem, but leave this as future work.

Furthermore, our conditional breakpoints are *stateless*, meaning that they cannot keep any state between evaluations of the conditional breakpoints. We solve this problem in our current approach by introducing *history-aware* breakpoints such as `wl:prev-length`. However, as future work, *stateful* predicates can be considered to allow developers to keep track of an arbitrary state between the evaluation of breakpoints. To this end, language extensions and extensions to the predicate evaluator are needed.

Finally, whereas we establish a link between the analyzed code and the analysis implementation through fine-grained meta-predicates and visualisations of the analysis' state, Nguyen et al. [7] establish a clear correspondence between the analyzed code and the code of the analysis implementation by pairing them together visually in the debugger interface itself. Our debugger already keeps track of this information internally, but does not visualize it. However, we acknowledge that this pairing could be beneficial for understanding the analysis implementation as well as for finding the bug in the analysis implementation itself. We consider the integration of our debugger with an *Integrated Development Environment* as future work.

6 Related Work

Charguéraud et al. [2] propose a *double-debugger* for debugging Javascript interpreters using *domain-specific* breakpoints. These domain-specific breakpoints are about the internal interpreter state, and can be anchored within the interpreted program through predicates about line numbers and the contents of local variables. Similarly, Kruck et al. [11, 12] recognize that interpreter developers want to reason about the structure of the interpreted program and propose *multi-level* debugging. Their approach mainly focusses on the representation of call stack frames in a debugging environment, and represents them both from the perspective of the interpreted program as well as from the perspective of the interpreter itself. Similar approaches have been proposed for tailoring debuggers to specific applications or frameworks [14]. This allows developers to reason about the behavior of the interpreter more easily.

Both approaches are, however, not *cross-level*. They either provide domain-specific breakpoints on the meta level (e.g., breakpoints about the current line number of the interpreter), or do not provide them at all. Our breakpoints are placed in the analyzed code (base level) allowing the developer to specify the location where they are evaluated. Furthermore, the conditions in these breakpoints express properties of the analysis state (meta level) rather than the analyzed program (base level). These breakpoints therefore interact with each other and cross the boundaries between the base and the meta level.

Nguyen et al. [7] propose a tool called VisuFlow which is tailored to the visualisation of data flow analyses implemented in the Soot framework [13]. However, they do not propose cross-level domain-specific breakpoints and their approach is only applicable to data flow analyses. Static analyses based on the abstract definitional interpreter approach, however, have been shown to be applicable in many use-cases, including control flow analysis [1, 16, 20], data flow analysis [6] and soft contract verification [15, 19].

7 Conclusion

We proposed *cross-level debugging* for static analysis implementations, which moves stepping and breakpoints from the base level to the meta level. More specifically, we proposed domain-specific visualisations for visually depicting the current state of the analysis. We argue that this visualisation makes it easier to understand the behavior of the analysis and thus to locate the root cause of bugs.

Furthermore, we proposed *domain-specific conditional breakpoints* which enable breaking when a specific analysis state is reached. We divided these meta-predicates into three categories: store-related, worklist-related, and lattice-based predicates.

We implemented our debugger in a framework called *MAF*, and showed the applicability of our debugger on one synthetic and four real-world bugs lifted from the repository of the framework. In this case study, the debugger is highly effective for most bugs that relate to changes of store addresses and their contents, but less so for bugs that prevent analysis progress, or dependency-triggering related bugs. However, we argued that our approach is sufficienlty flexible to support these classes of bugs in future work through additional meta-predicates.

8 Acknowledgements

This work has been partially funded by FWO (Research Foundations Flanders) under grant number 1187122N.

References

[1] Martin Bodin, Philippa Gardner, Thomas P. Jensen, and Alan Schmitt. 2019. Skeletal semantics and their interpretations. *Proc. ACM Program. Lang.* 3, POPL (2019), 44:1–44:31. https://doi.org/10.1145/3290357

[2] Arthur Charguéraud, Alan Schmitt, and Thomas Wood. 2018. JSExplain: A Double Debugger for JavaScript. In *Companion of the The Web Conference 2018 on The Web Conference 2018, WWW 2018, Lyon , France, April 23-27, 2018*, Pierre-Antoine Champin, Fabien Gandon, Mounia Lalmas, and Panagiotis G. Ipeirotis (Eds.). ACM, 691–699. https://doi.org/10.1145/3184558.3185969

[3] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252.

[4] Patrick Cousot and Radhia Cousot. 2002. Modular Static Program Analysis. In *Compiler Construction, 11th International Conference (CC 2002) (Lecture Notes in Computer Science, Vol. 2304)*, R. Nigel Horspool (Ed.). Springer, 159–178.

[5] David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. 2017. Abstracting definitional interpreters (functional pearl). *Proceedings of the ACM on Programming Languages* 1, International Conference on Functional Programming (2017), 12:1–12:25. https://doi.org/10.1145/3110256

[6] Jens Van der Plas, Jens Nicolay, Wolfgang De Meuter, and Coen De Roover. 2023. MODINF: Exploiting Reified Computational Dependencies for Information Flow Analysis. In *Proceedings of the 18th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2023, Prague, Czech Republic, April 24-25, 2023*, Hermann Kaindl, Mike Mannion, and Leszek A. Maciaszek (Eds.). SCITEPRESS, 420–427. https://doi.org/10.5220/0011849900003464

[7] Lisa Nguyen Quang Do, Stefan Krüger, Patrick Hill, Karim Ali, and Eric Bodden. 2020. Debugging Static Analysis. *IEEE Transaction on Software Engineering* 46, 7 (2020), 697–709. https://doi.org/10.1109/TSE.2018.2868349

[8] David Van Horn and Matthew Might. 2010. Abstracting abstract machines. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, September 27-29, 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM, 51–62. https://doi.org/10.1145/1863543.1863553

[9] J. Ian Johnson, Nicholas Labich, Matthew Might, and David Van Horn. 2013. Optimizing abstract abstract machines. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 443–454. https://doi.org/10.1145/2500365.2500604

[10] Sven Keidel and Sebastian Erdweg. 2019. Sound and reusable components for abstract interpretation. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 176:1–176:28. https://doi.org/10.1145/3360602

[11] Bastian Kruck, Stefan Lehmann, Christoph Keßler, Jakob Reschke, Tim Felgentreff, Jens Lincke, and Robert Hirschfeld. 2016. Multi-level debugging for interpreter developers. In *Companion Proceedings of the 15th International Conference on Modularity, Málaga, Spain, March 14 - 18, 2016*, Lidia Fuentes, Don S. Batory, and Krzysztof Czarnecki (Eds.). ACM, 91–93. https://doi.org/10.1145/2892664.2892679

[12] Bastian Kruck, Tobias Pape, Tim Felgentreff, and Robert Hirschfeld. 2017. Crossing abstraction barriers when debugging in dynamic languages. In *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*, Ahmed Seffah, Birgit Penzenstadler, Carina Alves, and Xin Peng (Eds.). ACM, 1498–1504. https://doi.org/10.1145/3019612.3019734

[13] Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*. https://www.bodden.de/pubs/lblh11soot.pdf

[14] Matteo Marra, Guillermo Polito, and Elisa Gonzalez Boix. 2020. Framework-aware debugging with stack tailoring. In *DLS 2020: Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages, Virtual Event, USA, November 17, 2020*, Matthew Flat (Ed.). ACM, 71–84. https://doi.org/10.1145/3426422.3426982

[15] Cameron Moy, Phuc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. 2021. Corpse reviver: sound and efficient gradual typing via contract verification. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434334

[16] Jens Nicolay, Quentin Stiévenart, Wolfgang De Meuter, and Coen De Roover. 2019. Effect-Driven Flow Analysis. In *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, January 13-15, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11388)*, Constantin Enea and Ruzica Piskac (Eds.). Springer, 247–274. https://doi.org/10.1007/978-3-030-11245-5_12

[17] Olin Grigsby Shivers. 1991. *Control-flow analysis of higher-order languages or taming lambda*. Carnegie Mellon University.

[18] Quentin Stiévenart, Jens Nicolay, Wolfgang De Meuter, and Coen De Roover. 2019. A general method for rendering static analyses for diverse concurrency models modular. *Journal of Systems and Software* 147 (jan 2019), 17–45. https://doi.org/10.1016/j.jss.2018.10.001

[19] Bram Vandenbogaerde, Quentin Stiévenart, and Coen De Roover. 2022. Summary-Based Compositional Analysis for Soft Contract Verification. In *22nd IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2021, Limassol, Cyprus, October 3, 2022*. IEEE, 186–196. https://doi.org/10.1109/SCAM55253.2022.00028

[20] Guannan Wei, Yuxuan Chen, and Tiark Rompf. 2019. Staged abstract interpreters: fast and modular whole-program analysis via meta-programming. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 126:1–126:32. https://doi.org/10.1145/3360552

Received 2023-07-07; accepted 2023-09-01

Cascade

A Meta-language for Change, Cause and Effect

Riemer van Rozen
rozen@cwi.nl
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands

Abstract

Live programming brings code to life with immediate and continuous feedback. To enjoy its benefits, programmers need powerful languages and live programming environments for understanding the effects of code modifications on running programs. Unfortunately, the enabling technology that powers these languages, is missing. Change, a crucial enabler for explorative coding, omniscient debugging and version control, is a potential solution.

We aim to deliver generic solutions for creating these languages, in particular Domain-Specific Languages (DSLs). We present Cascade, a meta-language for expressing DSLs with interface- and feedback-mechanisms that drive live programming. We demonstrate run-time migrations, ripple effects and live desugaring of three existing DSLs. Our results show that an explicit representation of change is instrumental for how these languages are built, and that cause-and-effect relationships are vital for delivering precise feedback.

CCS Concepts: • **Software and its engineering** → *Visual languages*; *Domain specific languages*; *Integrated and visual development environments*; *Interpreters.*

Keywords: live programming, metamodels, domain-specific languages, bidirectional transformations, model migration

ACM Reference Format:
Riemer van Rozen. 2023. Cascade: A Meta-language for Change, Cause and Effect. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3623476.3623515

1 Introduction

Live programming caters to the needs of programmers by providing immediate feedback about the effect of changes to the code. Figure 1 illustrates a typical coding cycle [13].

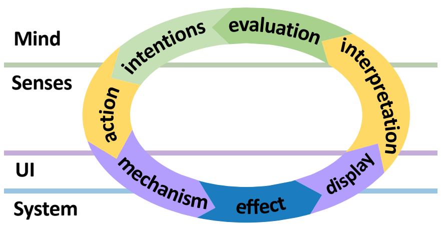


Figure 1. Live Programming speeds up coding cycles

Each iteration, programmers make improvements by performing *coding actions*, events that result in the construction, modification and deletion of objects over time. To help programmers realize their intentions, live programming environments offer suitable user interface mechanisms that enable performing the effects of coding actions. In addition, these environments offer feedback mechanisms that display changes for perceiving those effects, and evaluating if the action has been successful. Good feedback supports forming mental models and learning cause-and-effect relationships that help programmers predict effects of coding actions and make targeted improvements.

Despite the compelling advantages of live programming, its adoption remains sporadic due to a lack of enabling technology for creating the necessary programming languages. Unfortunately, creating languages whose users enjoy the advantages of live programming is incredibly complex, time-consuming and error-prone. Language engineers lack reusable abstractions and techniques to account for run-time scenarios with eventualities such as run-time state migrations, e.g., removing the current state of a state machine.

Several Domain-Specific Languages (DSLs) support a form of live programming that modifies running programs, e.g., the State Machine Language (SML) [24], Questionnaire Language (QL) [20] and Machinations [23]. However, these are one-off solutions with hand-crafted interpreters that are difficult to extend and maintain.

We study how to create such DSLs in a principled manner, how to express their liveness, and how to add this liveness to existing ones. We hypothesize that an explicit representation of change, a crucial enabler for exploratory coding, omniscient debugging and version control, is the missing factor in the currently available language technology. Our main objective is to deliver language-parametric solutions for creating change-driven DSLs that foreground cause-and-effect relationships, and let programmers perceive effects.

SLE '23, October 23–24, 2023, Cascais, Portugal
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0396-6/23/10.
https://doi.org/10.1145/3623476.3623515

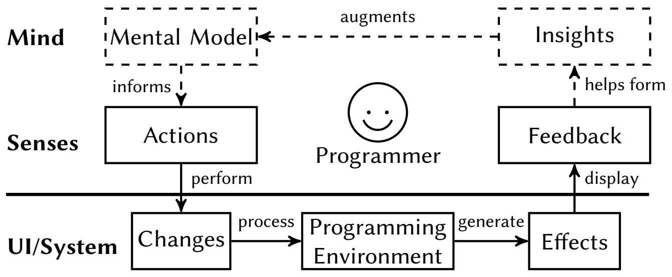
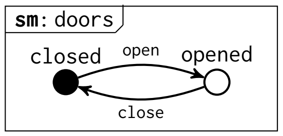


Figure 2. Relating actions to events, feedback and insights

```
machine doors
  state closed
    open => opened
  state opened
    close => closed
```

(a) Textual doors program



(b) Visual program

Figure 3. State Machine Language: doors example

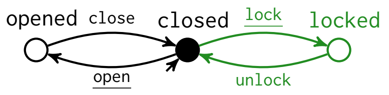
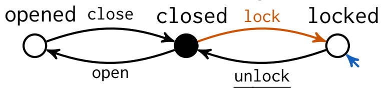
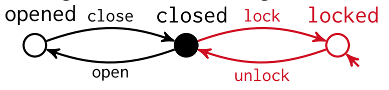
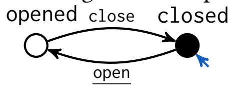
#	Origin	Effect	Run-time state
1.	Coding actions.	Adding state locked. 	closed: 1 * opened: 0 $\boxed{\text{locked: 0}}$
2.	User interaction.	Locking the door. 	closed: 1 opened: 0 locked: $\boxed{1\ *}$
3.	Coding action.	Deleting the locked state. 	closed: 1 opened: 0 ~~locked: 1 *~~
4.	State migration.	Updating the current state. 	closed: $\boxed{2\ *}$ opened: 0

Figure 4. Live programming scenario of a running doors program that demonstrates run-time state migration

We propose a novel meta-modeling approach that leverages an explicit representation of change. We present Cascade, a meta-language for expressing DSLs with input and feedback mechanisms that drive live programming. Cascade expresses "cascading changes" that introduce liveness using bi-directional model transformations with side-effects.

Using its compiler, language engineers generate interpreters that integrate with Delta, Cascade's runtime. Delta offers a built-in Read-Eval-Print-Loop (REPL) for simulating live programming scenarios that bring the code to life. By executing sequential commands on the REPL, engineers can simulate coding actions, user interaction and feedback.

When Delta executes events, it generates transactions as cause-and-effect chains. These transactions update a live program's syntax and run-time state. We investigate how Cascade can help express the interpreters of SML, QL and Machinations. Our results show Cascade is instrumental for rapidly creating executable DSL prototypes with concise and maintainable designs. Our contributions are: 1) Cascade: a meta-language for change, cause and effect; 2) Delta: a runtime for creating live programming environments; and 3) three case studies that reproduce liveness[1].

2 Problem Overview

We study a form of live programming that works on running programs. We relate the needs of programmers, illustrated by Figure 2, to changes and feedback in Section 2.2. We formulate hypotheses and objectives in Sections 2.3 and 2.4. First, we introduce a scenario that motivates this work.

2.1 Scenario: Modifying a running machine

The Live State Machine Language (LiveSML) is a DSL for simultaneously creating and running state machines.

[1]An earlier version of this paper has been presented as: R. van Rozen. 2022. Cascade: A Meta-Language for Change, Cause and Effect: Enabling Technology for Live Programming. In *Workshop on Live Programming, LIVE 2022.*

We use this DSL as a illustrative example because it is easy to comprehend, and also appears in related work [19, 24]. Figure 3 shows an SML program called doors. When executed, it can be either in the opened or closed state.

As a concise example, Figure 4 describes a live programming scenario of a running doors program. After starting the program of Figure 3, each step shows the origin of a change and the effects on the program and its run-time state.

First, the programmer adds a locked state, and two transitions for locking and unlocking the door. These additions are marked in green. In response, the interpreter introduces a locked count of zero (shown in a box). Initially, the current state (marked *), is closed, and the lock and open transitions (underlined) can be activated. In step two, the programmer triggers the lock transition in the user interface. The interpreter performs the transition (feedback shown in orange), updates the current state to locked, and raises its count (shown in a box). Updates are shown in blue.

Finally, in step three, the programmer deletes the locked state. In response, the interpreter also deletes the lock and unlock transitions (shown in red). Because the current state is removed (indicated with strikethrough), this state has become invalid. In step four, which follows immediately, the interpreter migrates the program to the initial state closed.

2.2 The need for Live Programming

Though limited in its complexity, the scenario illustrates key requirements that programming environments must fulfil to cater to the programmers' needs. Every coding cycle, the challenge is relating feedback about the effects of changes to insights about improvements, as illustrated by Figure 2.

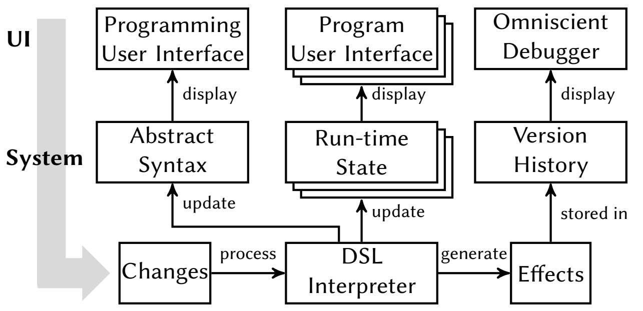


Figure 5. Change-based Live Programming Environment

R1. Gradual change. As software evolves, programmers constantly need to *make changes* to the code. Each iteration, they realize intentions, improve behaviors and fix bugs.

R2. Immediate feedback. Feedback is essential for testing hypotheses and verifying behaviors. For making timely improvements, programmers need feedback with every change.

R3. Evolving behavior. Programmers run programs to evaluate behaviors. For assessing the impact of changes, they need to observe the difference in behavior.

R4. Learnable behavior. For making targeted improvements, programmers need to learn from successes and mistakes. Programmers have to learn to predict the outcomes of changes.

R5. Exploratory design. To support gradually improving insights, programmers need to freely explore design spaces through do, undo, redo, record, and playback functionality.

2.3 Change-based DSL environments

Using programming environments, programmers can perform *coding actions*, events that result in the construction, modification and deletion of program elements. Invalid or syntactically incorrect edits are not coding actions. We study the effects of coding actions, specifically, changes to visual programs that work directly on the abstract syntax and affect running programs. Two main hypotheses drive this study:

1. Live programming can make code come alive in the imagination of the programmer by keeping "test cases" running.
2. DSLs are especially suitable to support live programming and to deliver feedback that appeals to the imagination.

We aim to empower programmers with programming environments for exploring the run-time effects of coding actions.

2.4 Language-parametric enabling technology

We study how to create such DSLs in a principled manner, how to express their liveness, and how to add this liveness to existing ones. We envision change-based live programming environments, as illustrated by Figure 5, based on a set of reusable principles, formalisms and components. For providing live feedback, language designs must account for run-time eventualities, valid changes to programs and run-time states. These cannot easily all be linearly represented due to the multitude of valid executions and dependencies.

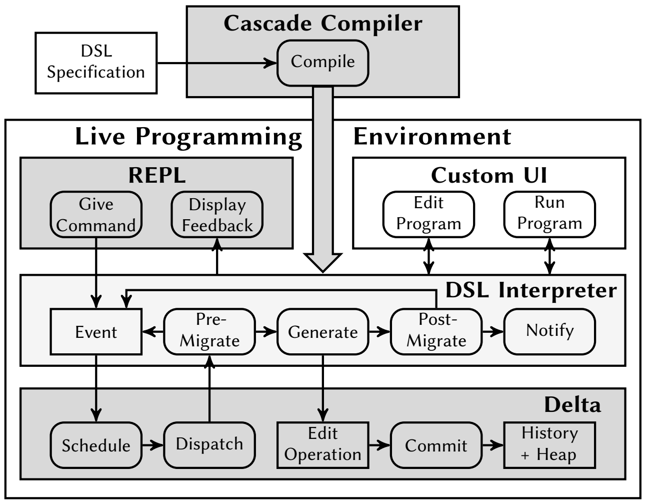


Figure 6. Cascade Framework: Generating DSL Interpreters

We address the need for enabling technology that powers these environments. We aim to simplify language design by abstracting from individual scenarios and the ordering of events. Next, we propose a generic approach that expresses change as modular and reusable model transformations. This solution can steer global run-time executions through local and conditional side-effects defined on the meta-level.

3 Cascade Framework

We present the Cascade framework, a language-parametric solution for developing change-based live programming environments. Cascade, illustrated by Figure 6, offers a meta-language and a set of generic reusable components (gray) that integrate domain-specific additions (white) for addressing the following technical challenges.

3.1 Cascade Meta-Language

Cascade is a meta-language for change, cause and effect. We introduce its features in Section 5. To create DSLs realizing the requirements of Section 2.2 language engineers can:

T1 Express the syntax and run-time states using meta-models (R1, R3). Section 5.1 introduces these concepts.
T2 Design actions, events and effects as bi-directional model-transformations that support exploratory coding (R5). Section 5.2 explains how to design interactions that gradually change the syntax and program behaviors (R1–3).
T3 Design side-effects as relationships between events with predicable outcomes to steer behaviors (R4). Section 5.3 explains how to express mutations of program elements.
T4 Design cascading changes that are central to live programming, e.g, for expressing run-time state migrations (R3, R4) that must account for many run-time eventualities (R5). Section 5.4 discusses design considerations.

In three case studies, we explore how Cascade helps to express the language designs and program execution of DSLs in a principled manner. In Sections 7, 8 and 9 we investigate:

```
o_new(|uuid://1|, "State");
o_set([|uuid://1|], "name", "opened", null);
o_new(|uuid://2|, "List<Trans>");
o_set([|uuid://1|], "out", [|uuid://2|], null);
o_new(|uuid://3|, "Trans");
o_set([|uuid://3|], "src", "opened", null);
o_set(|[uuid://3|], "evt", "close", null);
l_insert([|uuid://2|], 0, [|uuid://3|]);
```

(a) Example edit operations

```
s = new State();
s.name = "opened";
s.out = new List<Trans>();

t = new Trans();
t.src = s;
t.evt = "close";
s.out.push(t);
```

(b) Generic edit script

Figure 7. Contrasting edit operations from edit scripts

a) consistency and run-time state migrations of LiveSML;
b) trickle-effects and fixpoint computations of LiveQL; and
c) live desugaring and visual feedback of Machinations.

3.2 Live Programming Environments

Cascade provides reusable components for developing live programming environments, easing the authorial burden.

3.2.1 User Interface. Programmers need appropriate input and feedback mechanisms for changing code, obtaining feedback, and observing changes. Cascade offers a choice: 1) develop a user-friendly custom UI, based on its event APIs; or 2) use a generic Read-Eval-Print-Loop (REPL), e.g., before creating a custom UI. In this paper, we explore both. We use the REPL to simulate live programming scenarios textually. By executing sequential commands, we can simulate coding actions, user interaction and feedback on a line by line basis.

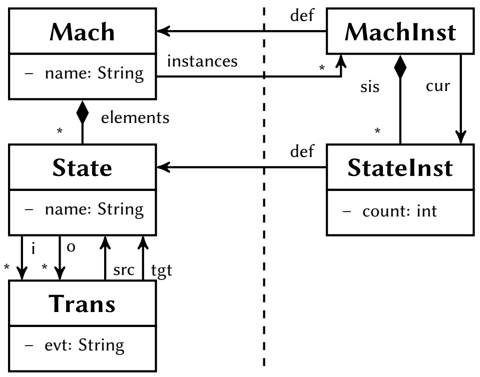
3.2.2 Interpreter. A powerful interpreter is the driving force behind live programming. We formulate technical challenges for creating interpreters that can support the requirements of Section 2.2.

T5 Integrate DSLs in a common run-time environment.
T6 Schedule and execute events that perform bi-directional model transformations and run-time state migrations.
T7 Maintain a version history and a heap, for updating syntax trees and run-time states with gradual changes.
T8 Generate the effects of transformations as historical transactions that: a) capture changes as edit operations; and b) preserve causal relationships in cause-and-effect chains.

To tackle these challenges, Cascade generates a DSL interpreter from its specification. Section 6 discusses the design of the compiler, the generated interpreters and Delta, Cascade's runtime. We begin by introducing edit operations.

4 Edit Operations

Cascade introduces an explicit representation of change for expressing behavioral effects of coding actions, user interactions and program executions. Cascade expresses change as model transformations that work on models (or programs) and run-time states, which each consist of objects. We base its representation on a language variant of the edit operations [2]. Originally introduced for differencing and merging, these operations have since also been used as a low-level storage format for expressing run-time effects [20, 24].



(a) Static meta-model **(b)** Run-time meta-model

Figure 8. Static and run-time meta-models of LiveSML

Cascade leverages edit operations to express transactional effects, maintain a version history, and support exploratory live programming. Figure 7a shows example operations that create new objects and replace attribute values. Appendix A describes a complete set of edit operations that work on commonly used data structures: objects, lists, sets and maps.

However, edit operations alone are not sufficient. An expressive meta-language for change requires variables, not just values. Cascade introduces a script notation, illustrated by Figures 7b, that resolves this issue. Next, we explain how Cascade's transformations encapsulate these edit scripts.

5 The Cascade Meta-Language

Cascade is a meta-programming language for expressing change, cause and effect. Using Cascade, language engineers can create interpreters (language back-ends) described as meta-models with bi-directional model transformations.

At run time, the interpreter executes these transformations in sequence and produces transactions consisting of edit operations. Upon completion, it commits the transactions to the version history as cause-and-effect chains. Next, we introduce the main language concepts and features.

5.1 Models and meta-models

Cascade expresses languages and changes using meta-models. Programs are models that conform to the meta-model of the language. In particular, these models are Abstract Syntax Graphs (ASGs) composed of objects. The language semantics steer the behavior of running programs. A program's run-time state, also a model, stores the results of program execution and user interactions.

For instance, Figure 8 shows the UML class diagram of LiveSML's meta-model. The static metamodel (on the left), defines the abstract syntax. A machine consists of a number of states with transitions between them. The run-time meta-model (on the right) expresses *running state machines.* A running machine has a current state, and registers how often it has resided in each state (the count).

```
class Mach {
  String name;
  Set<State> states;
  Set<MachInst> instances;
  ...
```

(a) Machine

```
class State {
  String name;
  Set<Trans> i;
  Set<Trans> o;
  ...
```

(b) State

```
class Trans {
  ref State src;
  String evt;
  ref State tgt;
  ...
```

(c) Transition

```
class MachInst {
  ref Mach def;
  ref StateInst cur;
  Map<State, StateInst> sis;
  ...
```

(d) Machine Instance

```
class StateInst {
  ref State def;
  int count;

  ...
```

(e) Machine Instance

Figure 9. Cascade definitions of LiveSML's meta-model

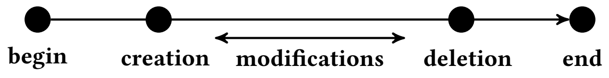


Figure 10. Object lifeline, events and modifications

The notation for meta-models resembles object-oriented programming, as shown in Figure 9. Aside from classes, it supports the base types **String**, **int**, **bool**, and **enum**, and the composite types List, Set and Map. Attribute ownership is explicit, and by default, a class owns its attributes. The **ref** keyword denotes an alias. We omit visibility because the aim is encapsulating change.

5.2 Actions, events and transformations

Cascade is designed to express run-time transformations with explicit effects. As a back-end language, it does not distinguish between coding actions and user interactions. Mechanisms for both can be expressed using three kinds of parameterized event declarations, called **effect**, **trigger** and **signal**. An *effect* describes how a specific object can be created, modified or deleted. A *trigger* is an input event that has no direct effect, but can schedule other events, side-effects that happen afterwards. A *signal* is an output event that flags an occurrence such as an exception or an error.

5.2.1 Objects. Objects have a limited life span. Instead of operating on objects directly, events work on object *lifelines*, as shown in Figure 10. The life span of an object begins before its creation and ends after its deletion. Any number of changes may happen in between. These life stages are called *issued*, *bound* and *retired*.

5.2.2 Effects. The basic unit of change, called **effect**, offers a parameterized abstraction for scripting and reuse. Effects are bi-directional model transformations whose body is an edit script. Each effect has parameters, type-value pairs separated by commas that determine its scope. Figure 11 shows an example.

5.2.3 Creation. Creation effects are used to create new objects of a certain class. For instance, Figure 11 shows a partial specification of the Machine class. We use the REPL to create a new state machine called "doors" as shown in

```
 1 class Mach {
 2   String name;
 3   Set<State> states;
 4   Set<MachInst> instances;
 5
 6   effect Create(future Mach m,
 7       String name) {
 8     m = new Mach();
 9     m.name = name;
10     m.states = new Set<State>();
11     m.instances =
12       new Set<MachInst>();
13   }
14   inverse effect Delete(past Mach m,
15       String name = m.name) {
16     delete m.instances;
17     delete m.states;
18     m.name = null;
19     delete m;
20   } pre {
21     foreach(State s in m.states) {
22       State.Delete(s, s.name, m); }
23     foreach(MachInst mi in m.instances) {
24       MachInst.Delete(mi, m); }
25   }
26   ...
```

Figure 11. Partial Cascade specification of the Mach class

```
var m; ↩
Mach.Create(m, "doors"); ↩
```

(a) Creating a machine

```
print m; ↩
machine doors
```

(b) Obtaining feedback

```
Mach.Create([|uuid://5|], "doors") {
  [|uuid://5|] = new Mach();
  [|uuid://5|].name = null ⟶ "doors";
  [|uuid://6|] = new Set<State>();
  [|uuid://5|].states = null ⟶ [|uuid://6|];
  [|uuid://7|] = new Set<MachInst>();
  [|uuid://5|].instances = null ⟶ [|uuid://7|];
}
```

(c) Generated transaction

Figure 12. Creating state machine from the REPL

Figure 12a. For conciseness, we will omit declaring variables from now on. This command calls the Create effect (lines 6–13). Note the ↩ symbol indicates REPL input (pressing the return key), and its absence indicates output the interpreter gives in response. We verify the results by reading the output from the REPL in Figure 12b. The interpeter also generates the changes that have occurred. The transaction shown in Figure 12c is a short-hand for encapsulated edit operations.

5.2.4 Subject. The first parameter of an effect, called *subject*, is always a reference to the object that is subject to change. The subject can optionally be preceded by an additional keyword that provides guarantees about its life before and after execution. The **future** keyword, used only in creations, denotes the subject must be issued and will be bound afterwards. The **past** keyword, used only in deletions denotes a bound subject will be retired afterwards. The lack of a keyword signifies it will continue to exist.

5.2.5 Parameters. There are two kinds of additional parameters that may follow the subject in the signature. Constant parameters are inputs that enable passing values such as an **int**, **bool**, **String**, **enum** or object reference. Change parameters enable updating the value of an object field from an old to a new value. In Figure 11, both parameters of the Create effect (lines 6–7) are constant parameters. The Delete effect (line 15) also has a change parameter. It indicates transactions will store the old and the new value of name field.

5.3 Side-effects and causal tranformations

5.3.1 Side-effects. The **pre** and **post** clauses enable scheduling effects before and after an event. Side effects can be used to create modular constructors and destructors that keep the syntax and the run-time states consistent. The **post** clause enables creating additional objects, booting up systems and defining effects of user interaction. These clauses can contain if-statements and while loops that read values and schedule events, but cannot modify values directly. Triggers only have a post-clause. Signals have no side-effects.

5.3.2 Begin statement. For weaving side-effects together, the `begin` and `end` statements issue objects references and revoke their validity. For instance, "`begin State s;`" is a statement that issues a new reference to a State object. Afterwards, we can schedule its creation.

5.3.3 Deletion. By design, every object that can be created can also be deleted. Unlike creations, which work on "blanks slates", deletions must account for ownership and consistency. The `pre` clause enables performing clean up tasks such as deleting owned data, removing aliases, and shutting down entire systems. For instance, calling `Mach.Delete(m)` from the REPL does not only delete a machine, but also cleans up the every state and any running instance it owns, as defined by the pre-clause on lines 20–25 of Figure 11. We demonstrate how LiveSML handles deletion in Section 7.

5.3.4 Inverse. Inverse effects, indicated by the `inverse` keyword, perform conceptually opposite operations to their preceding effects. Effects and their inverses must have compatible signatures. Create and Delete in Figure 11 are inverse effects. Create's `future` subject and constant parameters match Delete's `past` subject and change parameters. At run time, an inversion entails creating an opposite effect that can roll back a transaction, undoing its effects. An `invertible` effect, typically a setter, is its own inverse. For such an effect, an inversion matches the constant subject, and swaps the old and new values of the change parameter.

5.4 Design considerations

5.4.1 Root cause analysis. The design decision, that all change must be explicit, adds some verbosity but ensures events can always be related to their Cascade specification. We observe that default code for effects and inverses may be generated, and explicit ownership enables static analysis.

5.4.2 Consistency. To ensure bi-directionality, deletions and removals must also be explicit. Therefore objects cannot be garbage-collected. Creating a new object is, as one would expect, a sequence of operations that create new objects and *afterwards* assign initial values. However, deleting an object is more involved. Deletion requires a clean-up of every child object owned, and typically also erasure of references to the object (or aliases) *before* the object can be deleted itself. Fields must all have default values before deletion.

5.4.3 Liveness. Cascading changes can introduce liveness into DSLs. By adding relationships between coding actions and run-time effects, language engineers can improve input and feedback mechanisms that help programmers make gradual changes and observe differences in behavior. This has far reaching implications for the language designs of DSLs, as we will demonstrate in Sections 7, 8 and 9.

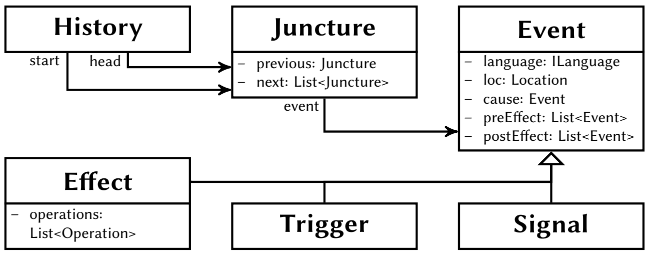


Figure 13. Delta's history consists of cause-and-effect chains

```
1 void schedule(Patcher p, Dispatcher d, Event e) {
2   d.resolve(e);
3   foreach(Event pre in d.preMigrate(e)){ schedule(p,d,pre); }
4   d.generate(e);
5   p.commit(e);
6   foreach(Event post in d.postMigrate(e)){ schedule(p,d,post); }
7 }
```

Figure 14. C# pseudo-code of Delta's event scheduler

6 Cascade Compiler and Runtime

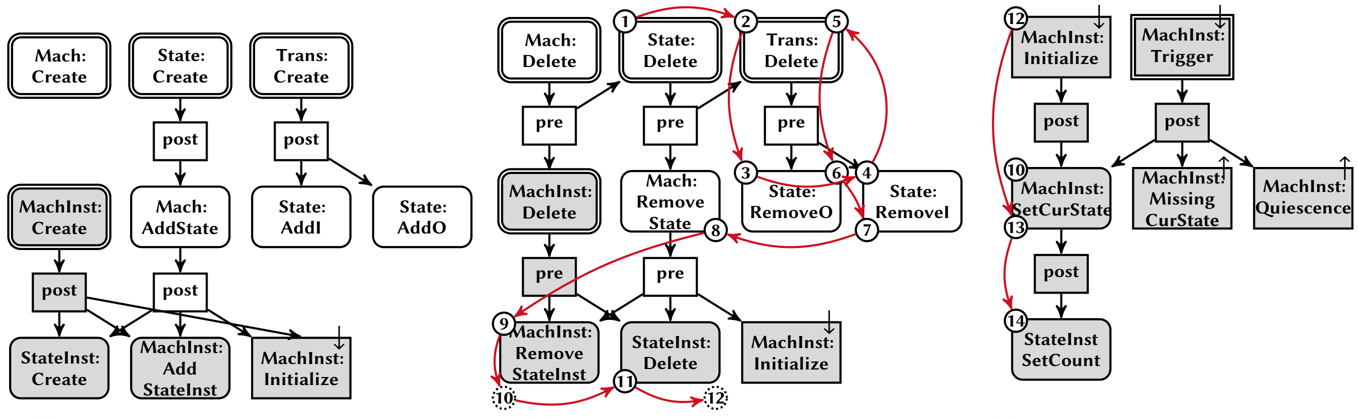
The Cascade framework consists of a compiler written in Rascal [8], and Delta, a runtime written in C#. The compiler translates Cascade specifications into language modules that integrate with Delta's extensible engine. Figure 6 gives an overview that illustrates how the main components process and transform events.

Each generated DSL interpreter (or language) consists of three sub-packages: 1) Model contains the classes of the meta-model; 2) Operation contains the classes representing events; and 3) Runtime contains components that process events and transform models. Key runtime components are the generator, and pre- and post-migrators, which generate edit operations and handle side-effects. Delta's engine has three main components. The dispatcher manages a set of languages, and determines which one handles an event. The patcher executes edit operations, maintains the heap and updates the version history. The scheduler determines the order in which events are scheduled, generated and migrated. Next, we explain how non-linear event scheduling works.

6.1 Scheduling events

The engine generates transactions in the form of cause-and-effect chains, as illustrated by Figure 13. Histories consists of junctures, branching points in time signifying events.

When called, the scheduler binds an event to a specific subject. We sketch the recursive algorithm that schedules each event in Figure 14. First, the dispatcher resolves the language that processes the event (line 2). Before processing the event itself, the *pre-migrator* of the language determines if any events need to happen before, and if so, those are scheduled first (line 3). Only when the recursive *pre-side-effects* have completed, the generator of the language generates the edit operations that perform the event's own effect (line 4).



(a) Dependencies between creation events **(b)** Dependencies between deletion events **(c)** Dependency of run-time triggers

Figure 15. LiveSML: Static dependency graphs of events, and an example run-time state migration

The patcher immediately commits the transaction to the history, before the operations go stale (line 5). Afterwards, post-migration schedules any events that need to happen afterwards (line 6). When each of those events completes, the event itself completes.

6.2 Implementation

The compiler consists of 3285 LOC of Rascal. Delta consists of 11.5 KLOC of C#. Delta's main parts are the edit operations interpreter (1745 LOC), the runtime (7288 LOC), and the REPL language (2603 LOC). Cascade is available under the 2-clause BSD license: https://github.com/vrozen/Cascade.

7 Live State Machine Language

We investigate how to express the design of LiveSML [24]. LiveSML exemplifies run-time state migrations with one-to-many relationships. Its semantics introduce dependencies between definitions of machines and states and their instances. Changes to definitions potentially have many side-effects on the run-time state. Because this state is not known a priori, run-time state migrations have to account for many eventualities. Using Cascade, we create an interactive prototype that reproduces the behavior of the original Java implementation. We demonstrate its interpreter accounts for run-time eventualities by reproducing the scenario of Section 2.1.

7.1 Event-based language design

We create an event-based language design. In addition to its meta-model, shown in Figure 8, we design its run-time transformations. Figure 15 schematically depicts the static dependencies between creation events, deletion events and run-time triggers. Events, shown as rounded rectangles, work on the syntax (white) and the run-time state (gray). Interactive events (double line) are coding actions (white) and user interactions (gray) with side-effects (single line). Arrows indicate if side-effects happen before (pre) or after (post) an event. Converging arrows indicate reuse in distinct scenarios.

Creation. The programmer begins with an empty state machine by creating one. At any moment, they can add states to a program and transitions between states by creating new states and transitions. They can also run a machine at any point in time. Each running machine separately keeps track of its visit counts. Therefore, creating a new machine also instantiates each state. Running machines update their bookkeeping when adding a new state to their machine definition. Afterwards, each running machine reinitializes (↓), since it may not have a current state yet.

Deletion. Of course, programmers can also delete a machine. Each machine cleans up its states and running instances. Deleting a state has side-effects that also remove and delete every transition from its inputs and outputs. In addition, removing a state also removes and deletes state instances from every running machine. Finally, removing the current state of a running machine migrates its to the first state in its definition.

Run-time triggers. Triggering (↓) a running machine can cause a transition that sets a new current state. However, it can also result in the signals (↑) missing state or quiescence. These signals do not cause any change, but do provide feedback to the user. When setting a new current state, its count is also increased by one.

Prototype. The Cascade implementation of LiveSML counts 213 LOC. Compiling the sources results in 2204 LOC of generated C#. Next, we will apply the fully generated prototype.

7.2 Live programming scenario

We reproduce the run-time state migration of Section 2.1. Instead of parsing an SML program, we simulate sequential coding actions, user interactions, and feedback directly from the REPL. Figure 16 shows the REPL commands and feedback that simulate the scenario.

We first create a new machine *doors* that contains a *closed* state (Figure 16a). Using the `print` command, we call the pretty printer. We obtain feedback and verify the syntax of

```
Mach.Create(m, "doors"); ↩
State.Create(s1, "closed", m); ↩
print m; ↩
machine doors
  state closed
```

(a) Creating the program

```
MachInst.Create(mi, m); ↩
print mi; ↩
machine doors
  closed : 1 *
```

(b) Runing the program

```
State.Create(s2, "opened", m); ↩
State.Create(s3, "locked", m); ↩
Trans.Create(t1, s1, "open", s2); ↩
Trans.Create(t2, s2, "close", s1); ↩
Trans.Create(t3, s1, "lock", s3); ↩
Trans.Create(t4, s3, "unlock", s1); ↩
```

(c) Completing the program

```
print mi; ↩
machine doors
  [open] [lock]
  closed : 1 *
  opened : 0
  locked : 0
```

(d) Updated run-time state

```
MachInst.Trigger(mi, "lock"); ↩
print mi; ↩
machine doors
  [unlock]
  closed : 1
  opened : 0
  locked : 1 *
```

(e) Locking the door

```
State.Delete(s3, "locked", m); ↩
print mi; ↩
machine doors
  [open]
  closed : 2 *
  opened : 0
```

(f) Deleting the locked state

Figure 16. LiveSML scenario simulated from the REPL

the DSL program is as expected. Although the program is not yet complete, we already run it. We create an instance, and use the **print** command to observe that, initially, its current state (*) is closed (Figure 16b).

We now complete the program (Figure 16c) by adding *opened* and *locked* states, and the transitions between them. Behind the scenes, several side-effect have occurred. We inspect the running program has also been updated (Figure 16d). The text between brackets denote "buttons" for the available actions. Users can now *open* or *lock* the closed door.

We simulate an action that locks the door (Figure 16e). Finally, we delete the locked state (Figure 16f). As expected, this causes a run-time state migration, setting the current state to closed, and increasing its count by one.

The resulting transaction, described in more detail in Appendix B, is super-imposed on the design on Figure 15. Its generated control flow traverses events that affect both the syntax and the run-time state. Note that, the edit operations of the deletion itself, actually happen last.

7.3 Analysis

Compared to the original LiveSML, which counts 1217 LOC of hand-written Java, our prototype is significantly smaller (213 LOC)[2]. Cascade addresses the main shortcoming of the Run-time Model Patching (RMPatch) approach that expresses run-time state migrations as hard-coding visitors on edit operations, which is time consuming and error-prone [24]. Instead, Cascade expresses them on the meta-level. As a result, LiveSML's modular design is more concise and maintainable.

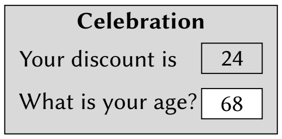
8 Live Questionnaire Language

The Questionnaire Language (QL) is a DSL for expressing interactive digital questionnaires. Originally designed for the

[2]https://github.com/vrozen/Cascade/tree/main/LiveSML

```
form Celebration {
  "Your discount is" :
    int discount = age/2-10
  "What is your age?" :
    int age
}
```

(a) Celebration form



Celebration

Your discount is	24
What is your age?	68

(b) Example filled-out form

Figure 17. Forms that calculates an age dependent discount

Dutch tax office, this DSL has since served as a benchmark for generic language technology, e.g., the Language Workbench Challenge [7]. We study LiveQL, a language variant that enables simultaneously designing and answering forms [20]. We focus on the liveness properties and trickle effects at the heart of its semantics. In particular, we aim to reproduce the behavior that propagates the effects of giving answers. Using Cascade, we express LiveQL and create a language prototype. We use its REPL to simulate a run-time scenario of an example that demonstrates a fixpoint computation.

8.1 Questionnaire Language

Forms consists of sequences (or blocks) of two kinds of statements: questions and if statements. Figures 17 shows a from that expresses an age-based discount, and an answered form.

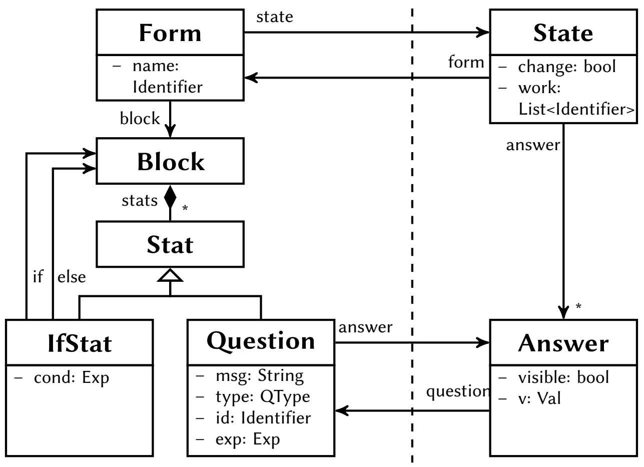
8.1.1 Questions. Each question consists of a textual message the user sees, a question type (int, str or bool), and a variable name that can be used to reference the question's answer. By default, questions are answerable. The question "what is your age?" is answerable. Users answer questions by supplying a value of the specified question type. In this case, age requires an int value, for instance 68.

However, when assigned with an expression, questions become *computed*. Instead of prompting the user to answer the question, the form computes the answer by evaluating the expression. In the example, discount is a computed answer. Its computed value is 24.

8.1.2 If statements. Conditional questionnaire sections can be designed using if statements. Each if consists of a condition (a boolean expression), an if-block and an optional else-block. The user sees the statements nested in the if-block if the condition is true, and those in the else-block otherwise. Statements that have an expression referencing a variable have *data dependency* on that variable. Statements nested inside an if-block have a *control dependency* on each variable referenced in the condition. Here, we omit an example.

8.1.3 LiveQL. Originally, QL required that users answer questions in a top-down manner [7]. Each statement could only refer back to variables whose value have been previously given or computed.

LiveQL relaxes this requirement by enabling forward references, and allowing changes to running forms [20]. These changes introduce two forms of liveness. First, when the programmer adds, removes or changes statements, this affects the running form. Second, when a user answers a question,



(a) Static metamodel **(b)** Run-time metamodel

Figure 18. Static and run-time meta-models of LiveQL

```
package LiveQL {
  class State {
    ref Form f;
    List<Answer> ans;
    bool change;
    List<Identifier> work;

    trigger TriggerID(State s, Identifier i) {
      SetChanged(s, false); //First, reset the changed flag.
      PushWork(s, i);       //Next, add the identifier to the work queue.
      DoWork(s);            //Schedule the work, which may cause re-evaluations.
      WhileChange(s, i); }   //Finally, check if there is more change.

    trigger WhileChange(State s, Identifier i) {
      if(s.change) {     //If a change has happened as a result of re-evaluation
        TriggerID(s, i); //continue the computation
      } else {           //otherwise
        Done(s, i);      //signal done.
      } }

    signal Done(State s, Identifier i);
```

Figure 19. LiveQL contains a fixpoint computation

the form re-evaluates dependent computed questions and if statements. As a result, answers may update and sections of the form can become visible or invisible.

8.2 Language design

We investigate how Cascade helps to express the behavior of LiveQL, in particular the trickle effects that result from answering questions. The meta-model of LiveQL, shown in Figure 18, is based on the original Java implementation [20]. The run-time meta-model extends the static meta-model with information about the current state of the form, such as answers to questions and visibility.

The key to expressing trickle effects is defining a fixpoint computation that schedules future events in sequence from the body of a trigger. When answering a question, dependent computed answers and if-statements recompute until no more changes can be observed. Figure 19 illustrates the main events. When the value of an identifier updates, `TriggerID` is called. After performing work, which potentially causes changes, the check of `WhileChange` determines if the computation completes or continues to propagate changes.

```
1 Form.Create(f, "Celebration"); ↩
2 Question.Create(q1, f, "Your discount is", QType.Int, "discount");↩
3 Question.SetExpression(q1, f, "age/2-10"); ↩
4 Question.Create(q2, f, "What is your age?", QType.Int, "age"); ↩
```

(a) Creating the Celebration form

```
Question.GiveAnswer(q2, f, "68"); ↩
```

(b) Answering the age question

```
print f; ↩
form Celebration {
  "Your discount is" :
    int discount = age/2-10
      ==> undefined
  "What is your age?" :
    int age ==> undefined
}
```

(c) Unanswered form

```
print f; ↩
form Celebration {
  "Your discount is" :
    int discount = age/2-10
      ==> 24
  "What is your age?" :
    int age ==> 68
}
```

(d) Answered form

Figure 20. LiveQL scenario simulated from the REPL

8.3 Prototype live programming environment

We create a textual DSL prototype, an interpreter with a built-in REPL. Its Cascade specification counts 1044 LOC[3]. Compiling the sources results in 8189 LOC of generated C#. Most components of the prototype are fully generated. We add the following components, which amounts to a total of 916 LOC hand-written C#.

The pretty-printer enables inspecting the syntax and run-time state from the REPL. We add an expression evaluator and two small helper classes for: 1) performing lookups for use-def relationships of variables; 2) collecting conditions of questions; and 3) evaluating the expressions of questions and if-statements. We use ANTLR 4 to create a QL parser that generates ASTs of programs and expressions. To bring these ASTs under management of Delta, we create a Builder that generates Cascade events for recreating the ASTs.

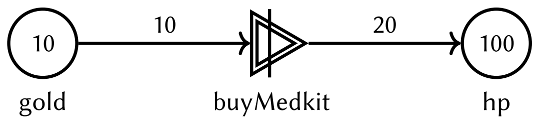
8.4 Live programming scenario

We demonstrate a trickle effect in a live programming scenario that reproduces the Celebration example of Figure 17. After creating the form, answering the question `age` with `68` should result in the `discount` becoming `24`.

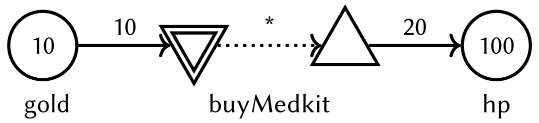
Using our prototype, we simulate coding actions and user interaction from the REPL, as shown in Figure 20. We begin with the coding actions shown in Figure 20a. First, we create a new form `f` called Celebration (line 1). We add a new question `q1` to form f with message "Your discount is", introducing the variable `discount` of type int (line 2). To make q1 a computed question, we set its expression to `age/2-10` (line 3). Finally we add the second question `q2` introducing int `age` (line 4).

LiveQL programs automatically run one instance. We verify the program runs, and observe `discount` and `age` are initially undefined, as shown in Figure 20c. Next, we answer question `q2` and give answer `68`, as shown in Figure 20b. Finally, we verify the value of `discount` has indeed become `24`. Figure 20d indicates the change has been correctly propagated. For conciseness, we omit the generated transaction.

[3] https://github.com/vrozen/Cascade/tree/main/LiveQL



(a) Mechanism for exchanging gold for health (what designers see)



(b) Desugared version of the diagram (what the engine runs)

Figure 21. Diagram showing an excerpt of the internal economy of Johnny Jetstream (adapted from van Rozen [21])

8.5 Analysis

Our results demonstrate Cascade helps express the trickle effects of LiveQL as a concise fixpoint computation. Creating the prototype, including its helper classes, cost approximately one working day, with only the experience of LiveSML. The original Java implementation, which measures 6179 LOC, also includes a visual front-end. The new prototype is significantly smaller, measuring 816 LOC.

9 Live Machinations

Machinations is a visual notation for game design that foregrounds elemental feedback loops associated with emergent gameplay [1, 1]. Micro-Machinations (MM) is a textual and visual programming language that addresses several technical shortcomings of its evolutionary predecessor. In particular, MM introduces a live programming approach for rapidly prototyping and fine-tuning a game's mechanics [23], and accelerating the game development process.

We study the design of the MM library (MM-Lib), including its run-time bahavior and state migrations. In particular, we explore how Cascade helps to express live desugaring. We create a visual prototype using Cascade and the Godot game engine. Vie is a tiny live game engine for simultaneously prototyping and playtesting a game's mechanisms. In live programming scenario of a simple game economy, we demonstrate Vie correctly desugars converters.

9.1 Micro-Machinations

Micro-Machinations programs, or diagrams, are directed graphs that can control the internal economy of running digital games. When set in motion through runtime and player interactions, the nodes act by pushing or pulling economic resources along its edges.

Figure 21a shows a mechanism in the internal economy of Johnny Jetstream, a 2D fly-by shooter [21]. Two *pool* nodes, shown as circles, abstract from the in-game resources, gold and health (hp). The integers inside represent current amounts. The edges are *resource connections* that define the rate at which resources can flow between source and target nodes. BuyMedkit is an interactive *converter* node, appearing

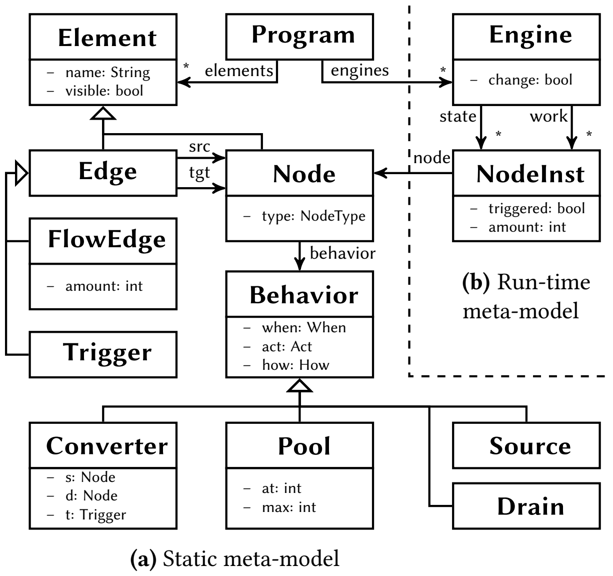


Figure 22. Partial meta-model of Micro-Machinations

as a triangle pointing right with a vertical line through the middle. This converter consumes 10 gold and produces 20 hp.

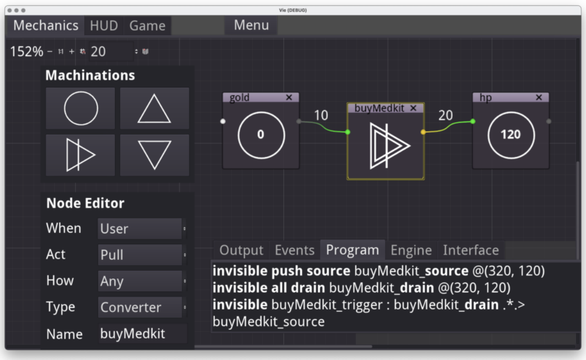
Converters are so-called syntactic sugar, a convenience notation, which translates into a simpler elements for efficient processing. Figure 21b shows that converters can be rewritten as a combination of a drain, a trigger and a source. When the drain node consumes the costs of the conversion, the trigger activates the source node, which then produces the benefits. During the translation, the inputs of the converter connect to the drain, and the outputs to the source.

Designers can simultaneously prototype and playtest running economies, e.g., by activating mechanisms or modifying node types. Therefore, desugarings must also happen live.

9.2 Language design

We investigate how Cascade helps to express the liveness of MM's core language features, focusing on live desugaring of converters in particular. As a starting point, we analyze the C++ implementation of MM-Lib, an embeddable script engine for MM [23]. Figure 22 shows a partial meta-model based on MM-Lib. As before, we express the dependencies between the abstract syntax and the run-time state. An engine, which instantiates a program, tracks the current amounts of pool nodes and which nodes are triggered for activation. Through a combination of effects and helper methods, it evaluates how the resources flow when nodes activate.

The solution for desugaring converter nodes introduces invisible elements that implement its behavior. When a node becomes a converter, a series of transformations immediately generate 1) a source, a trigger and a drain; 2) incoming edges to the drain; and 3) outgoing edges to the source. In addition, running engines obtain new node instances used for evaluating flow rates. Changes to a converter node are delegated its source and drain nodes. Changing the node's



Node Editor
When: User
Act: Pull
How: Any
Type: Converter
Name: buyMedkit

```
invisible push source buyMedkit_source @(320, 120)
invisible all drain buyMedkit_drain @(320, 120)
invisible buyMedkit_trigger : buyMedkit_drain .*.>
buyMedkit_source
```

Figure 23. Prototyping and playtesting a mechanism in Vie

behavior to another type, cleans up these invisible elements, and also removes node instances from running engines.

9.3 Vie: a tiny live game engine

We use Cascade and the Godot game engine to create a prototype that implements the design of MM. Vie is a tiny live game engine for simultaneously prototyping, playtesting and fine-tuning a game's design. MM's implementation in Cascade counts 1542 LOC[4]. Compiling its sources generates an interpreter, 11.7 KLOC of generated C#. We augment Engine with two helper classes, EvalContext (153 LOC) and Flow (29 LOC), for storing temporary run-time data.

Instead of using its built-in REPL, we create a visual front-end using Godot (v3.5.1). We leverage the GraphNode and GraphEdit framework, and program C# classes for connecting Cascade events to UI events. In another paper we further detail how we manually create this front-end [22].

9.4 Live programming scenario

We reproduce the behavior of the mechanism shown in Figure 11. Using Vie, we perform a sequence of prototyping and playtesting actions that demonstrate the behavior of MM, including live desugaring and run-time state migrations.

We first recreate the diagram using Vie's visual editor, shown in Figure 23. Next, we trigger the converter buyMedit by clicking on its center. The UI shows visual feedback. We observe the engine succeeds (yellow) in activating its drain, trigger and source. The nodes consume and produce resources at the expected rates (green). The textual view on the program, shows the invisible elements. Finally, we change the node type of the converter to pool. The resulting transaction is a long cause-effect-chain that cleans up the desugared converter and migrates the run-time state.

9.5 Analysis

MM-Lib measures 21.2 KLOC of C++. In comparison, Vie does not yet support every feature, e.g. modules. However, at a mere 1542 LOC, its Cascade specification is considerably more concise. Due to its representation of effects, Vie solves two limitations of MM-Lib. First, it adds traceability of cause-and-effect for all actions. Second, it expresses the effects of resource propagation and triggers as a fixpoint computation. Vie is more extensible and maintainable. Combining Godot with Cascade is straightforward. Since both have event APIs, and Godot support C#, they integrate well. Our effort went mainly into creating the UI. A benefit of Godot is that Vie is a portable app (Windows, Linux, MacOS, iOS, Android).

[4]https://github.com/vrozen/Cascade/tree/main/LiveMM

10 Discussion

Cascade has compelling benefits for creating live programming environments. Using its notations and abstractions, language engineers can concisely express DSL run-time behaviors, and account for many migration scenarios, on the meta-level. They can ensure transformations and side-effects are correct by design. However, no automated contextual analysis is provided yet. At no additional cost, Delta generates a history that traces how and why every event happens, while ensuring the run-time state is correctly updated. Cause-and-effect chains are instrumental for exploratory live programming, omniscient debugging and version control. Because the generated interpreters are event-driven, they combine well with visual UIs, e.g., bowsers or game engines.

Of course there are also costs. For language engineers, bidirectional thinking is not straightforward. Language designs do not normally include run-time state migrations. Learning how bi-directional designs work takes time and practice.

We have validated Cascade against a limited number of existing DSLs. Further validation will require introducing new liveness to DSLs. Additionally, the compiler is a complex meta-program that bridges a wide conceptual gap. As a result, it undoubtedly still contains bugs we have not yet identified. To address this, we plan to create a test harness that automates testing features and prototypes.

Of course, the proposed combination of transformations, migrators and feedback mechanisms generalizes beyond Cascade. These abstractions can also be programmed using General Purpose Languages (GPLs). Our compiler targets C#. Unlike Cascade, GPLs do not support bi-directional transformations or generate cause-and-effect chains out of the box. Adding support requires a considerable engineering effort.

We have not studied using Cascade to add liveness to GPLs. The underlying execution models, e.g., program counters and stack frames, do not directly support liveness. GPLs require additional mechanisms such as probes to introduce liveness.

Cascade's generic REPL has limitations. Its mechanisms are low-level, and not suitable for DSL users. Furthermore, Cascade still lacks a debugger for exploring histories, inspecting cause-and-effect chains and tracing source locations. Debugging a DSL involves stepping through the generated C# code. Debugging transactions involves inspecting the notation on the REPL (e.g., Figure 24 in Appendix B).

Cascade's integration in C# is helpful for extending it functionality with helper methods. The lack of a formal semantics complicates analyses. We see opportunities for checking cyclic dependencies and the correctness of inverse effects.

Live programming with run-time state migrations is inherently inconsistent. An open challenge is identifying formal properties of liveness. Cascade introduces a local dependencies between events that have a global consistency of effects.

11 Related Work

Live Programming is a research area that intersects Programming Languages (PL) and Human-Computer Interaction (HCI). The term refers to a wide array of user interface mechanisms, language features and debugging techniques that revolve around iterative changes and immediate and continuous feedback [16]. Tanimoto describes *levels of liveness* that help distinguish between forms of feedback in live programming environments [18]. Each level adds a property: 1) informative or descriptive; 2) executable; 3) responsive or edit triggered; and 4) live or stream driven. Many forms of live programming exist, each designed with different goals in mind. For instance, McDirmid describes how *probes*, a mechanism interwoven in the editor, helps diagnosing problems [11]. Ko describes *whyline*, a debugging mechanism for asking why-questions about Java program behavior [9].

Another approach is creating interpreters with a so-called Read-Eval-Print-Loop (REPL), a textual interface for executing commands sequentially [3] A REPL, by definition, lends itself naturally to exploration, incremental change and immediate feedback, each key ingredients to live programming. Interpreters created with Cascade have a built-in REPL and REPL-like APIs for designing DSLs with event-based input and feedback mechanisms, including visual ones.

Omniscient debugging, also called back-in-time debugging, is a form of debugging that allows exploring what-if scenarios by stepping forward and backward through the code [14, 15]. Such debuggers have been created for general purpose languages Java [15]. Retrofitting an omniscient debugger to an existing language can come at a considerable cost, redesign and implementation effort. Bousse et al. propose a meta-modeling approach for a generic debugger of executable DSLs that supports common debugging services for tracing the execution [4]. Cascade is also designed with omniscient debugging in mind. Cause-and-effect chains are a key data structure for creating omniscient debuggers.

The area of *modelware* is a technological space that revolves around the design, maintenance and reuse of models (or programs). Model transformations are a key technology for expressing change. In their seminal paper on "The Difference and Union of Models", Alanen and Porres describe a notation, originally intended for model versioning, known as edit operations, which expresses model deltas [2]. Van der Storm proposes creating live programming environments driven by "semantic deltas", based on this notation [20]. Van Rozen and van der Storm combine origin tracking and text differencing for textual model differencing [21].

Bi-directional Model Transformation (BX) is a well researched topic that intersects with several areas [6]. BX has impacted relational databases, model-driven software development [17], UIs, visualizations with direct manipulation, structure editors, and data serialization, to name a few. Cicchetti et al describe the Janus Transformation Language (JTL), a language for bi-directional change propagation [5].

The study of live modeling with run-time state migrations has initially focused on fine-grained patching with edit operations [24]. Constraint-based solutions instead focus on correct states with respect to a set of constraints [19], a course-grained approach that omits fine-tuning. Sanitization solutions regard run-time state migrations as a way to fix what is broken [25]. We instead take the position that they are an integral part of the language semantics.

Until now, no solution could explain why a particular migration happened. Cascade is a BX solution that addresses both *how* and *why*, for precise feedback about root causes and finger sensitive fine-tuning.

Cascade is the first language-parametric and generic approach for creating DSLs that leverages a bi-directional transformations for live programming. Compared to existing approaches, it adds a scheduling mechanism for defining complex deterministic side-effects. To the best of our knowledge, no other system exists that can generate run-time state migrations from meta-descriptions as cause-and-effect chains.

12 Conclusions and future work

We have addressed the lack of enabling technology for creating live programming environments. We have proposed Cascade, a meta-language for expressing DSLs with interface- and feedback-mechanisms that drive live programming. In three case studies, we have explored expressing the liveness features of LiveSML, LiveQL and Machinations. We have demonstrated how to express gradual change, run-time state migrations, ripple effects and live desugarings. Our results show that an explicit representation of change is instrumental for how these languages are built, and that cause-and-effect relationships are vital for delivering feedback.

In future work, we will investigate how to create a reusable omniscient debugger for change-based DSL environments.

Schema or program modifications require both instance migrations and view adaptations. We will further investigate how to express coupled transformations [10], e.g., for Vie.

Live programming requires interactive visual interfaces, and generic language technology to create them [12]. We will investigate how to automate the development of UIs that leverage game engine technology [22].

Acknowledgments

I thank the anonymous reviewers and the Agile Language Engineering (ALE) team (Inria & CWI) for their feedback.

Table 1. Edit operations supported by Delta

Operation	Inverse operation
o_new (ID id, StringVal class)	**o_delete** (ID id, StringVal class)
o_set (Path p, Field f, Val nv,Val ov)	**o_set** (Path p, Field f, Val ov,Val nv)
l_insert (Path p, IntVal i, Val v)	**l_remove** (Path p, IntVal i, Val v)
l_push (Path p, Val v)	**l_pop** (Path p, Val v)
l_set (Path p, IntVal i, Val nv,Val ov)	**l_set**(Path p, IntVal i, Val ov,Val nv)
s_add (Path p, Val v)	**s_remove** (Path p, Val v)
m_add (Path p, Val k)	**m_remove** (Path p, Val k)
m_set (Path p, Val k, Val nv, Val ov)	**m_set** (Path p, Val k, Val ov, Val nv)

A Edit Operations

This appendix describes on a complete set of edit operations for manipulating commonly used data structures: objects, lists, sets and maps. Cascade's runtime Delta uses these operations to execute bi-directional model transformations.

A.1 Objects, heap and qualified names

Edit operations work on objects. A global store, or heap, stores the current program state as a collection of objects with fields and references. Values can have a base type such as **`int`**, **`String`**, **`bool`**.

Enums and classes introduce custom objects and values. The built-in datatypes `Set`, `List` and `Map` help to create object hierarchies, such as trees and graphs. Edit operations can perform lookups using qualified names or paths. The notation uses dots and brackets for lookups in the global store, maps and lists. For instance, `[|uuid://1|]` performs a lookup of an object with Unique Universal Identifier (UUID) 1, and `[|uuid://1|].cur` retrieves its field `cur`.

A.2 Supported edit operations

Delta supports the edit operations shown in Table 1. We briefly describe these operations, which work on objects, lists sets and maps. Figure 7a shows an example sequence of operations that partially recreate the example SML program shown in Figure 3.

A.2.1 Objects. The **o_new** operation creates a new object with identifier id of a particular class. The new object will have all of its fields set to default values. Its inverse operation **o_delete** has the exact opposite effect, and deletes an object with identifier id. Deletions include the class parameter and they require that each fields of the object has default values. Without this, the operation cannot be reversed. Finally, to set the values of fields, the **o_set** operation, replaces the value ov of field f of the object denoted by path p by a new value nv. This operation is its own inverse, swapping the old and the new values. Figure 7a shows examples.

A.2.2 List. Specialized operations that only work on list objects of type List<X> are the following. The operations

```
root ① State.Delete([|uuid://26|], "locked", [|uuid://13|]) {
  pre ② Trans.Delete([|uuid://33|], [|uuid://26|], "unlock", [|uuid://16|]) {
    pre ③ State.RemoveOut([|uuid://26|], [|uuid://33]) {
      [|uuid://28|].remove([|uuid://33|]);
    }
    pre ④ State.RemoveIn([|uuid://16|], [|uuid://33|]) {
      [|uuid://17|].remove([|uuid://33|])
    }
    [|uuid://33|].src = [|uuid://uuid26|] ⟶ null;
    [|uuid://33|].evt = "unlock" ⟶ null;
    [|uuid://33|].tgt = [|uuid://uuid16|] ⟶ null;
    delete Trans [|uuid://33|];
  }
  pre ⑤ Trans.Delete([|uuid://32|], [|uuid://16|], "lock", [|uuid://26|]) {
    pre ⑥ State.RemoveOut([|uuid://16], [uuid://32]) {
      [|uuid://18|].remove([|uuid://32|]);
    }
    pre ⑦ State.RemoveIn([|uuid://26|], [uuid://32]) {
      [|uuid://27|].remove([|uuid://32|])
    }
    [|uuid://32|].src = [|uuid://uuid16|] ⟶ null;
    [|uuid://32|].evt = "lock" ⟶ null;
    [|uuid://32|].tgt = [|uuid://uuid26|] ⟶ null;
    delete Trans [|uuid://32|];
  }
  pre ⑧ Mach.RemoveState([|uuid://13|], [|uuid://26|]) {
    pre ⑨ MachInst.RemoveStateInst([|uuid://19|], [|uuid://29|], [|uuid://26|]){
      pre ⑩ MachInst.SetCurState([|uuid://19|], null, [|uuid://29|]) {
        [|uuid://19|].cur = [|uuid://uuid29|] ⟶ null;
      }
      [|uuid://20|][[|uuid://26|]] = [|uuid://29|] ⟶ null;
      [|uuid://20|].remove([|uuid://26|]);
    }
    pre ⑪ StateInst.Delete([|uuid://29|], [|uuid://26|]) {
      [|uuid://29|].def = [|uuid://26|] ⟶ null;
      [|uuid://29|].count = 1 ⟶ 0;
      delete StateInst [|uuid://29|];
    }
    pre ⑫ MachInst.Initialize([|uuid://19|]) {
      post ⑬ MachInst.SetCurState([|uuid://19|], [|uuid://21|]) {
        [|uuid://19|].cur = [|uuid://21|];
        post ⑭ StateInst.SetCount([|uuid://21|], 2, 1) {
          [|uuid://21|].count = 1 ⟶ 2;
        }
      }
    }
    [|uuid://14|].remove([|uuid://26|]);
  }
  [|uuid://26|].name = "locked" ⟶ null;
  delete Set<Trans> [|uuid://27|];
  delete Set<Trans> [|uuid://28|];
  delete State [|uuid://26|];
}
```

Figure 24. Cause-effect chain that deletes the locked state

l_insert and **l_remove** are each other's inverse. These operations respectively insert or remove a value v at an index i in the list denoted by path p.

To modify a value in a list, the invertible operation **l_set**, replaces an old value ov by a new value nv at index i in the list denoted by path p. For convenience, **l_push** inserts a value v at the tail of an existing list, and **l_pop** removes it, without specifying the index.

A.2.3 Set. Two operations work on set objects of type Set<X>. The inverse operations **s_add** and **s_remove** respectively add or remove a value v of type X in an existing set denoted by path p.

A.2.4 Map. Operations that work on map objects of type Map<K,V> are the following. The operations **m_add** and **m_remove** respectively add or remove a map record denoted by key k of type K in an existing map denoted by path p. To ensure correctness, the initial and final value in a record must be defaults. To update a map record, the operation **m_set** replaces an old value ov by new value nv in the record denoted by key k in the map denoted by path p.

B LiveSML: Cause-and-effect chain

We detail the results of Section 7, which reproduces the example LiveSML live programming scenario of Section 2.1. Figure 24 shows the cause-and-effect chain that results from deleting the locked state. The numbers appearing in the circles coincide with those in the generated control flow that is superimposed on the static dependency graph of Figure 15.

References

[1] Ernest Adams and Joris Dormans. 2012. *Game Mechanics: Advanced Game Design.* New Riders.
[2] Marcus Alanen and Ivan Porres. 2003. Difference and Union of Models. In *«UML» 2003 (LNCS, Vol. 2863).* Springer.
[3] L. Thomas van Binsbergen, Mauricio Verano Merino, Pierre Jeanjean, Tijs van der Storm, Benoît Combemale, and Olivier Barais. 2020. A Principled Approach to REPL Interpreters. In *Onward! 2020.* ACM.
[4] Erwan Bousse, Dorian Leroy, Benoît Combemale, Manuel Wimmer, and Benoit Baudry. 2018. Omniscient Debugging for Executable DSLs. *J. Syst. Softw.* 137 (2018).
[5] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. 2010. JTL: A Bidirectional and Change Propagating Transformation Language. In *Software Language Engineering, SLE2010 (LNCS, Vol. 6563).* Springer.
[6] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. 2009. Bidirectional Transformations: A Cross-Discipline Perspective. In *Theory and Practice of Model Transformations (LNCS, Vol. 5563).* Springer.
[7] Sebastian Erdweg, Tijs van der Storm, et al. 2013. The State of the Art in Language Workbenches. In *Software Language Engineering, SLE 2013 (LNCS, Vol. 8225).* Springer.
[8] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Code Analysis and Manipulation, SCAM 2009.* IEEE.
[9] Amy J. Ko and Brad A. Myers. 2004. Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior. In *Human Factors in Computing Systems, CHI 2004.* ACM.
[10] Ralf Lämmel. 2016. Coupled Software Transformations Revisited. In *Software Language Engineering, SLE 2016.* ACM.
[11] Sean McDirmid. 2013. Usable Live Programming. In *New Ideas in Programming and Reflections on Software, Onward! 2013.* ACM.
[12] Mauricio Verano Merino, Jurgen J. Vinju, and Tijs van der Storm. 2020. Bacatá: Notebooks for DSLs, Almost for Free. *Art Sci. Eng. Program.* 4, 3 (2020), 11.
[13] Donald A. Norman. 1986. Cognitive Engineering. *User centered system design* 31 (1986), 61.
[14] Guillaume Pothier and Éric Tanter. 2009. Back to the Future: Omniscient Debugging. *IEEE Softw.* 26, 6 (2009).
[15] Guillaume Pothier, Éric Tanter, and José M. Piquer. 2007. Scalable Omniscient Debugging. In *Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007.* ACM.
[16] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2019. Exploratory and Live, Programming and Coding - A Literature Study Comparing Perspectives on Liveness. *Art Sci. Eng. Program.* 3, 1 (2019), 1.
[17] Perdita Stevens. 2010. Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. *Softw. Syst. Model.* 9, 1 (2010).
[18] Steven L. Tanimoto. 2013. A Perspective on the Evolution of Live Programming. In *Workshop on Live Programming, LIVE 2013.* IEEE.
[19] Ulyana Tikhonova, Jouke Stoel, Tijs van der Storm, and Thomas Degueule. 2018. Constraint-based Run-time State Migration for Live Modeling. In *Software Language Engineering, SLE 2018.* ACM.
[20] Tijs van der Storm. 2013. Semantic Deltas for Live DSL Environments. In *Workshop on Live Programming, LIVE 2013.* IEEE.
[21] Riemer van Rozen. 2015. A Pattern-Based Game Mechanics Design Assistant. In *Proceedings of the 10th International Conference on the Foundations of Digital Games, FDG 2015, 2015.* SASDG.
[22] Riemer van Rozen. 2023. Game Engine Wizardry for Programming Mischief. In *Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments, PAINT 2023.* ACM.
[23] Riemer van Rozen and Joris Dormans. 2014. Adapting Game Mechanics with Micro-Machinations. In *Proceedings of the 9th International Conference on the Foundations of Digital Games, FDG 2014.* SASDG.
[24] Riemer van Rozen and Tijs van der Storm. 2019. Toward Live Domain-Specific Languages - From Text Differencing to Adapting Models at Run Time. *Softw. Syst. Model.* 18, 1 (2019).
[25] Yentl Van Tendeloo, Simon Van Mierlo, and Hans Vangheluwe. 2019. A Multi-Paradigm Modelling Approach to Live Modelling. *Softw. Syst. Model.* 18, 5 (2019).

Received 2023-07-07; accepted 2023-09-01

Seamless Code Generator Synchronization in the Composition of Heterogeneous Modeling Languages

Nico Jansen
jansen@se-rwth.de
Software Engineering, RWTH Aachen University
Germany

Bernhard Rumpe
rumpe@se-rwth.de
Software Engineering, RWTH Aachen University
Germany

Abstract

In Software Language Engineering, the composition of heterogeneous languages has become an increasingly relevant research area in recent years. Despite considerable advances in different composition techniques, they mainly focus on composing concrete and abstract syntax, while a thorough yet general concept for synchronizing code generators and their produced artifacts is still missing. Current solutions are either highly generic, typically increasing the complexity beyond their actual value, or strictly limited to specific applications. In this paper, we present a concept for lightweight generator composition, using the symbol tables of heterogeneous modeling languages to exchange generator-specific accessor and mutator information. The information is attached to the symbols of model elements via templates allowing code generators to communicate access routines at the code level without a further contract. Providing suitable synchronization techniques for code generation is essential to enable language composition in all aspects.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**.

Keywords: Software Language Composition, Code Generation, Generator Composition, CRUD

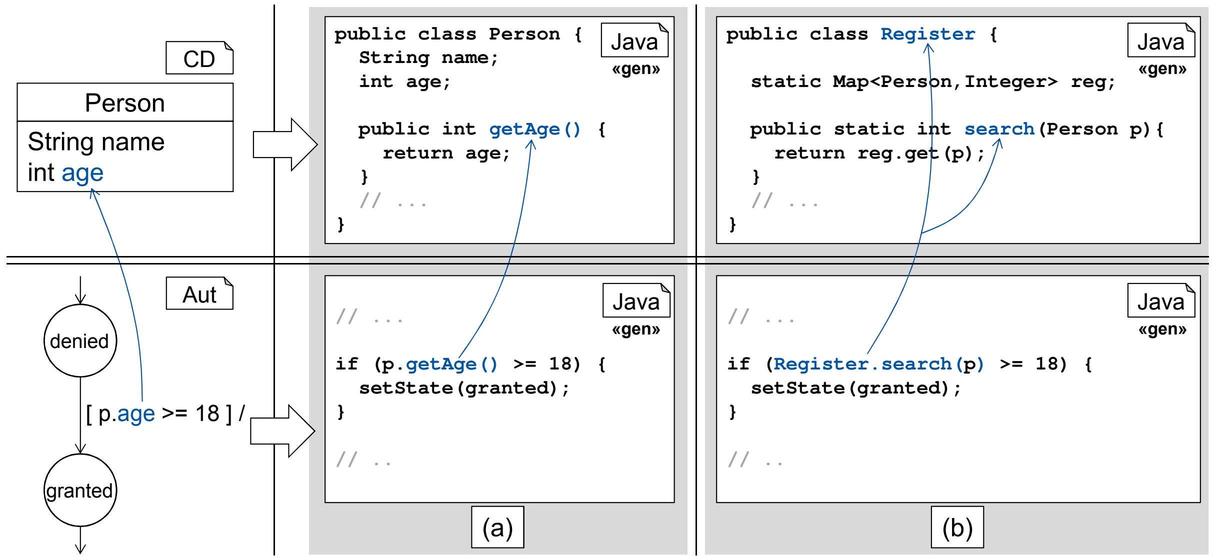
ACM Reference Format:
Nico Jansen and Bernhard Rumpe. 2023. Seamless Code Generator Synchronization in the Composition of Heterogeneous Modeling Languages. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3623476.3623530

1 Introduction

Model-driven engineering (MDE) [26] is a prominent research and application area in which models of various domains, such as automotive [2], robotics [30], and software development [29], represent the central development artifacts [12] in developing modern (often software-intensive) systems. These models conform to modeling languages, prescribing concrete and abstract syntax, additional well-formedness rules, and semantics [8] providing meaning [15]. The discipline of software language engineering (SLE) [19] investigates the efficient design, maintenance, and evolution of such languages (for both modeling and programming).

As software languages evolve and mature [11], and their constant support and development are time-consuming, reuse becomes increasingly critical in SLE [4]. This means not only reusability on a conceptual level but the actual reuse of the implementation, i.e., the language definition and its generated and hand-coded tooling. In this regard, the composition of software languages has been extensively investigated in the last decade [17], establishing libraries of reusable language components [5] and patterns for compositional language design [9].

While composition in a language's front end, i.e., its syntax and tooling, is already pretty sophisticated, composing the back end, usually code generators, is often neglected. Code generation is essential to modeling languages as it translates abstract models into executable program artifacts, thus bridging the gap between the problem and solution domain [22]. Access information must be distributed through the generation process so that generated artifacts can address each other correctly. Figure 1 sketches this challenge and serves as a running example. The left-hand side presents two models of different languages, a class diagram (top) with a class `Person` featuring a `name` and an `age` attribute and an automaton snippet (bottom), granting access once the `age` is at least 18. Thus, the expression at the automaton's transition refers to an attribute definition inside the class diagram. For simplicity reasons, we neglect the type-instance relation at the model level in this example. As the models refer to each other, it is apparent that their generated target artifacts do as well. Therefore, the code generator for the automaton language must adhere to the access information the class diagram generator provides. In scenario (a) (middle), the class is transformed in an intuitive fashion to a Java class



```
public class Person {
  String name;
  int age;

  public int getAge() {
    return age;
  }
  // ...
}
```

```
// ...
if (p.getAge() >= 18) {
  setState(granted);
}
// ..
```

```
public class Register {

  static Map<Person,Integer> reg;

  public static int search(Person p){
    return reg.get(p);
  }
  // ...
}
```

```
// ...
if (Register.search(p) >= 18) {
  setState(granted);
}
// ..
```

Figure 1. Model excerpts of a class diagram and an automaton (left) with an inter-model cross reference and two alternative generated code snippets (middle and right). In each scenario, these generated artifacts must match the respective access.

with attributes accessible via corresponding getters. Thus, the generated code for the automaton can utilize this getter. However, scenario (b) (right) depicts a rather different translation of the class diagram, resulting in a `Register` with a static access map requiring the corresponding person as key to retrieve the age. Therefore, assuming a provided `getAge` method, as in the first case, is not applicable anymore, the target code of the automaton must adapt. While scenario (b) is a contrived situation highlighting the underlying challenge, it is well within the realm of possibility. In reality, multiple cases of divergent access situations exist, such as employing builders or factories for object instantiation instead of native constructors or translating an automaton concerning different design patterns (e.g., the state pattern) [13].

Currently, there are no well-established solutions for composing generators or their generated target artifacts. This results in a gap when integrating models of distinct modeling languages, as their outputs must be synchronized, demanding additional manual effort. Some approaches exist but are either tied to integrating explicit generators of particular application domains [24], require strict compliance with generation rules, or are overly generic [23], raising the complexity beyond their practical usability. A general, lightweight solution is still missing.

In this paper, we envision a novel approach to synchronizing generated outputs via accessor template-enriched symbol tables. This approach harnesses the capabilities of already established composition techniques for languages' front ends and extends these to synchronize their generators as well. Our proposed solution is based on the symbol management infrastructure of a language enabling inter-model cross-referencing. Augmenting symbol tables with target access templates enables the adaptive generation of accessor code. An application programming interface (API) for synchronizing generated artifacts should be as lightweight as possible and ideally require down to no additional knowledge of another generator's intricacies. Therefore, we propose CRUD-like accessors for this API, as these operations are commonly known, language agnostic, and the notation is easily understandable. Our work focuses on template-based code generators that produce artifacts of a common yet arbitrary object-oriented, general-purpose programming language. Furthermore, we concentrate on harmonizing the target code for models of aggregated languages, as this composition technique preserves models as separated artifacts and, thus, only establishes a loose coupling [6]. Our main contributions are:

- An approach for enriching the symbol table with accessor templates enabling target code synchronization
- A conceptual API based on CRUD-like operations as a lightweight generator synchronization contract

The remainder of the paper is structured as follows: section 2 discusses the current state of the art comprising related approaches and preliminary work. section 3 presents our concept of integrating templates in symbol tables to synchronize generated artifacts. Finally, section 4 discusses our solution, states open challenges, and section 5 concludes.

2 State of the Art and Related Work

While language composition, in general, is a broad field of research in many language workbenches [10], there are currently only a few approaches to the integration of their generators. These attempts are often either tailored solutions for specific applications, too shallow to deliver a general concept, or too generic to be effortlessly applied in practice.

Most preliminary work in this field is often domain- or even application-specific. For example, there are approaches in robotics [24, 25] coupling generators specifically for component and connector systems via a separate generator configuration and an orchestrator. Other investigations follow a round-trip engineering approach defining framework-specific modeling languages [1]. However, these approaches are tightly coupled to the underlying application frameworks and do not aim for generalizable generator composition.

An interesting approach is based on reverse engineering existing target artifacts to extract corresponding accessor information [3, 14]. While this attempt can generally enable to generate syntactically well-formed and consistent code, it lacks the link to the original model elements. Therefore, it does not solve the issue of deriving target accessor information based on inter-model relationships.

A few framework solutions aim for the complete integration of languages and thus also incorporate their generators. CompoSE [20] provides for the integration of languages and generates glue code for the individual language components' target artifacts. Similarly, the SCOLAR framework also provides the ability to compose language components in the large [7, 23]. However, these works mainly concentrate on language embedding, i.e., a stronger coupling of the models. While such integrated framework solutions still address the problem of generator composition, they generally have the disadvantage that languages must be integrated into their respective ecosystem. Additionally, the defined communication interfaces are usually very generic and, therefore, uncomfortable to employ for arbitrary modeling languages.

Similarly, the Genesys project [18] provides for generator development conforming to predefined frameworks. It supports services for communication accessor information. However, the results are bound to the jABC ecosystem [27], and a more seamless composition of generators is considered future work.

Finally, a few lightweight approaches exist that provide for an exchange of information via the symbol tables of the models [21, 22]. The advantage of these attempts is that mainly already existing composition techniques are employed without creating over-complicated new infrastructures. So far, however, these approaches have only been weakly studied for predefined modeling and programming language combinations. A general solution that seamlessly composes generated artifacts of heterogeneous modeling languages still needs to be established.

3 Distributing Access Information Exchange via Symbol Tables

Our approach builds upon the concepts of [22], who propose enriching the symbol table with accessor and mutator code snippets of the model elements' corresponding target artifacts. While the basic idea is promising, their proposal only considers a fixed source (i.e., modeling) and target (i.e., general-purpose programming) language. Thus, the described symbol table extension and mapping are bound to these technological spaces. While further adaptions are possible, they require one mapping for each language combination, ultimately convoluting the symbol table infrastructure when incorporating more and more languages. Thus, our approach envisions a more generalized extension of the symbol table, which is as language-agnostic as possible and allows for arbitrary accessor and mutator mappings concerning different symbol kinds.

3.1 An Extended Symbol Table Infrastructure for Managing Target Access

For efficient cross-referencing, modern language workbenches support the concept of symbol tables, either directly (such as MontiCore [16]) or implicitly (such as MPS [28]). In a symbol table, symbols of language-specific kinds are ordered hierarchically inside scopes managing their visibility and accessibility. This principle is used, for instance, in language aggregation to compose models of different artifacts by resolving their respective inter-model references.

Utilizing the cross-referencing of symbols, our approach extends the symbol table infrastructure by enriching it with further generator information. Therefore, we augment the symbols further with access information templates of the target code. Figure 2 depicts our concept of the extended symbol table infrastructure. Similar to existing approaches, we foresee the extension of symbols with a `GeneratorInfo`, defining the overall API of the generator synchronization mechanism. As we strive for a general solution not bound to a definitive technological space, the signature is specified in a language-agnostic way.

In a first attempt, we propose CRUD operations for a generalized API to `create`, `read`, `update`, and `delete` constructs in an object-oriented sense. Thus, the `GeneratorInfo` attached to each symbol offers the corresponding methods independent of its kind. For proper access derivation, each operation expects a respective context in which the access occurs. For instance, in our running example, the context is the variable `p` of type `Person`, which is used quite differently in both scenarios. Next, as updating an object constitutes mutating access, it requires an additional value parameter to write, i.e., the new value to update with. This value, of course, is only provided at the model level. However, it is crucial to consider it here, as the generator needs to insert it in the mutator template. Finally, we conceive a collection of additional optional parameters that can be used to parametrize the access further. For instance, when updating only a single value inside a list, these parameters can provide the respective position.

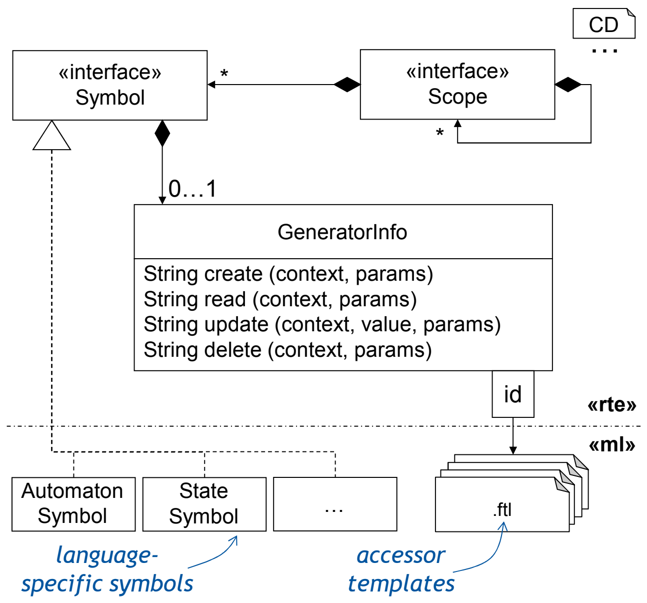
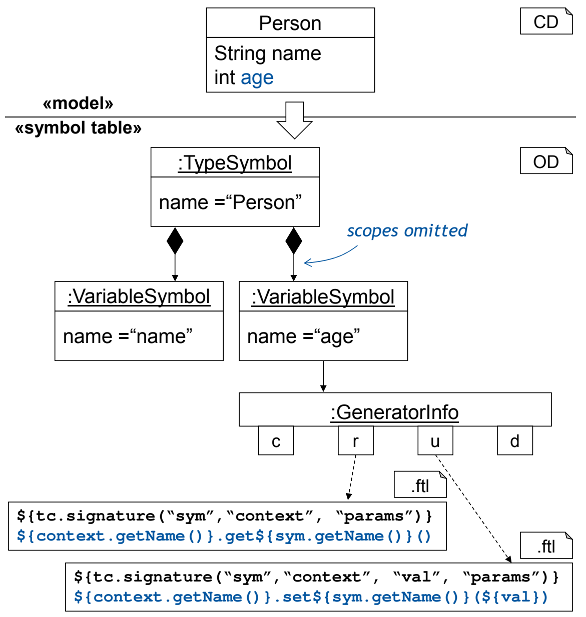

CD
…
«interface»
Symbol
*
«interface»
Scope
*
0…1
GeneratorInfo
String create (context, params)
String read (context, params)
String update (context, value, params)
String delete (context, params)
id
«rte»
«ml»
Automaton Symbol
State Symbol
…
.ftl
language-specific symbols
accessor templates

Figure 2. Extended symbol table infrastructure for storing generator-specific CRUD accessor templates for each symbol.

The proposed symbol table extension with generator information based on CRUD operations is independent of a particular modeling language. However, this only constitutes a general API via which generators can communicate the access data. The methods delegate to corresponding accessor templates which are, in contrast, attached to the symbols (resp. to their generator information) on the language level. Thus, each symbol kind comes with a particular set of templates carrying the accessor information for the respective generator's target code. In fact, it is even possible to provide different symbols of the same kind with distinct templates based on their use.

The extended symbol table infrastructure enables modular generators to look up the respective accessor code adaptively. Figure 3 shows a simplified view of a symbol table instance, providing accessor templates for the variable age of our running example. That is, the generator information of the variable symbol carries templates for `read` and `update` operations adhering to scenario (a). In this example, `create` and `delete` commands are not allocated. While the approach convinces the solution for arbitrary template engines, the presented templates are written in FreeMarker syntax (.ftl) for demonstration purposes.

Considering the template for the `read` operation, the first line defines the signature of the template, containing the variable symbol itself, the context, and optional parameters. The second line describes the derivation of the respective accessor (here, a getter method) by first referencing the corresponding context, followed by the static part `.get` and the dynamic name of the symbol itself, concluded by `()`. Please note that the presented template is a slight simplification as



```
${tc.signature("sym","context", "params")}
${context.getName()}.get${sym.getName()}()
```

```
${tc.signature("sym","context", "val", "params")}
${context.getName()}.set${sym.getName()}(${val})
```

Figure 3. Object diagram representation of the symbol table for a corresponding class diagram model with attached read and update templates for accessing the generated artifacts.

for a proper getter signature, the first letter of the attribute's name is capitalized, which is not reflected here. During generation, this template gets evaluated by the corresponding engine providing the required accessor code.

3.2 A Light-Weight Infrastructure for Composing Template-based Generators

Employing the extended symbol table infrastructure, we can further design compositional generator tools. For this purpose, we propose a generator architecture encapsulating language-specific printers that use the cross-language API of the generator information now delivered with the particular symbols. Figure 4 gives an overview of this architecture. Generally, such a tool comprises the standard components, such as a parser transforming the input models into an AST and a template engine for code generation. Additionally, we attach a composition printer for dealing with the incorporated symbol tables. Whenever an accessor code is required, this printer is incorporated, resolving the respective symbol and retrieving the corresponding accessor template.

It is important to note that such a printer only needs to know the constructs of its own language (usually but not limited to expressions) and not those of the referenced symbols. For instance, a printer for the automaton generator (cf. Figure 1) would know how to translate expressions, such as `x >= y`, in general, but fetches the concrete target code accessors from the templates of the loaded symbol table. This mechanism enables the seamless composition of target artifacts while simultaneously preserving the loose coupling of language aggregation.

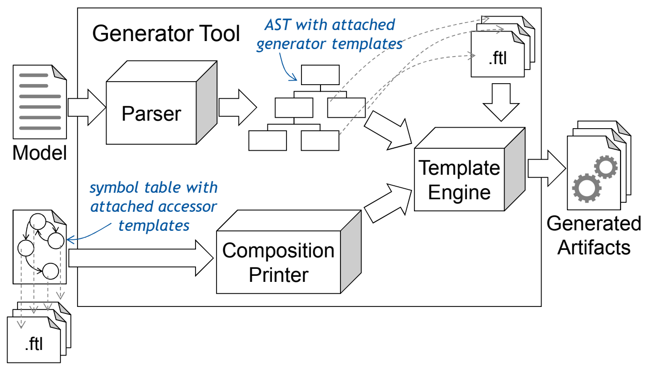


Figure 4. Conceptual generator tool with a built-in printer utilizing symbol-attached templates to supply the generator engine with accessor information on external artifacts.

4 Discussion and Open Challenges

The presented approach to seamlessly compose generated artifacts of heterogeneous modeling languages is based on template-enriched symbol tables and a lightweight interface using CRUD-like operations. Considering previous work, we envision extending existing concepts for accessor provisioning via symbols and establishing a more general solution for object-oriented target languages.

Previous approaches of related attempts require knowledge of the access signatures for each symbol type [22]. This hampers the loose coupling of some composition techniques, such as language aggregation, since generally, we cannot assume global knowledge of all loaded symbol kinds. Our proposal delegates access to CRUD-like operations, thus relaxing the tight coupling of generators. Moreover, previous approaches were limited to a fixed relation of the modeling and target programming languages, while our proposal aims for more generality.

Different approaches to composing generators or their generated artifacts on a more general level rely on highly generic, often bulky composition interfaces or require their explicit definition [7, 20]. Our technique avoids these drawbacks since we mainly build on existing composition mechanisms and rely only on a simple, standardized API. To provide for composability, a generator developer simply creates access templates and attaches these to symbols according to predefined CRUD operations. No further knowledge about employing generators is required. In turn, the developer of a generator, which requires access information, does not need any knowledge of the providing language despite its anyways provided symbols. A modeler using the languages does not need any knowledge of the generation process, the communication mechanism, or the attached templates.

However, our proposal for lightweight generator synchronization is a mere starting point with open challenges that need further investigation. For instance, the mentioned `context` in a generator template (cf. Figure 3) is often more complicated than in the highlighted example. In general, all symbols traversed during resolution must be considered, resulting in a collection of contextual symbols of which some might be relevant while others are omitted. While this is a task for the engineer providing the access templates (i.e., no issue that hampers the composability), it could turn out to be challenging to create the required templates for complicated situations in the first place. Thus, future investigations are required to evaluate whether the composition mechanism is as lightweight and applicable as envisioned.

Moreover, while our approach is generalized, it fits particularly well on infrastructures that provide an explicit symbol management system accessible to the language (or generator) developer. For frameworks that do not allow for customization or augmentation of symbols, concrete implementation, while generally possible, may turn out more difficult.

Furthermore, our approach does not yet consider that a model element can be mapped to several conflicting target elements. For instance, we can translate a single class of a class diagram into multiple classes on the code level. This makes the target access ambiguous. Generally, generators can always be modularized to minimize this issue. Other attempts propose an identifier, making the mapping unique [22]. However, both approaches have the disadvantage that knowledge about the generation process of the source generator is required.

Finally, the proposed API needs further investigation. While we envision a lightweight and straightforward generator synchronization interface, CRUD-like operations might not be sufficient. The requirements on the API strongly depend on which information of the target artifacts is relevant for synchronization. However, as a state is per se no type on the model level, this information does not directly emerge from the CRUD operations but requires additional consideration. Thus, while the CRUD-like API already covers a large set of necessary access patterns, it cannot be considered final. We need a detailed analysis of common symbol kinds and their potential target code information to extend further and refine the access interface.

5 Conclusion

As language engineering becomes increasingly sophisticated with composition techniques at concrete and abstract syntax level, so must it become for integrating generated artifacts. For this purpose, we presented an approach that envisions augmenting the symbol table, used in language aggregation, with additional templates to exchange access information of the target artifacts. We highlighted the need for a lightweight interface between generators and discussed a simple API based on CRUD-like operations. Furthermore, we have depicted challenges for further investigation regarding the portion of the information that must be exchanged. Seamlessly synchronizing generators is a crucial step for completely integrating languages.

References

[1] Michał Antkiewicz and Krzysztof Czarnecki. 2006. Framework-Specific Modeling Languages with Round-Trip Engineering. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 692–706. https://doi.org/10.1007/11880240_48

[2] Hans Blom, Henrik Lönn, Frank Hagl, Yiannis Papadopoulos, Mark-Oliver Reiser, Carl-Johan Sjöstedt, De-Jiu Chen, Fulvio Tagliabo, Sandra Torchiaro, Sara Tucci, et al. 2013. EAST-ADL: An architecture description language for Automotive Software-Intensive Systems. *Embedded Computing Systems: Applications, Optimization, and Advanced Design: Applications, Optimization, and Advanced Design* (2013), 456. https://doi.org/10.4018/978-1-4666-3922-5.ch023

[3] Hugo Bruneliere, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot. 2014. MoDisco: a Model Driven Reverse Engineering Framework. *Information and Software Technology* 56, 8 (2014), 1012–1032. https://doi.org/10.1016/j.infsof.2014.04.007

[4] Barrett Bryant, Jean-Marc Jézéquel, Ralf Lämmel, Marjan Mernik, Martin Schindler, Friedrich Steinmann, Juha-Pekka Tolvanen, Antonio Vallecillo, and Markus Völter. 2015. *Globalized Domain Specific Language Engineering*. Springer, 43–69. https://doi.org/10.1007/978-3-319-26172-0_4

[5] Arvid Butting, Robert Eikermann, Katrin Hölldobler, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. 2020. A Library of Literals, Expressions, Types, and Statements for Compositional Language Design. *Journal of Object Technology* 19, 3 (October 2020), 3:1–16. https://doi.org/10.5381/jot.2020.19.3.a4

[6] Arvid Butting, Judith Michael, and Bernhard Rumpe. 2022. Language Composition via Kind-Typed Symbol Tables. *Journal of Object Technology* 21 (October 2022), 4:1–13.

[7] Arvid Butting, Jerome Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. 2020. A Compositional Framework for Systematic Modeling Language Reuse. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. ACM, 35–46. https://doi.org/10.1145/3365438.3410934

[8] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. 2009. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09) (LNCS 5795)*. Springer, 670–684. https://doi.org/10.1007/978-3-642-04425-0_54

[9] Florian Drux, Nico Jansen, and Bernhard Rumpe. 2022. A Catalog of Design Patterns for Compositional Language Engineering. *Journal of Object Technology* 21, 4 (October 2022), 4:1–13. https://doi.org/10.5381/jot.2022.21.4.a4

[10] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. 2013. The State of the Art in Language Workbenches. In *International Conference on Software Language Engineering*. Springer, 197–217.

[11] J-M Favre. 2005. Languages evolve too! Changing the Software Time Scale. In *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*. IEEE, 33–42. https://doi.org/10.1109/IWPSE.2005.22

[12] Robert France and Bernhard Rumpe. 2007. Model-driven Development of Complex Software: A Research Roadmap. *Future of Software Engineering (FOSE '07)* (May 2007), 37–54.

[13] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Design Patterns. 1995. Elements of Reusable Object-Oriented Software. *Design Patterns. massachusetts: Addison-Wesley Publishing Company* (1995).

[14] I Garcıa, M Polo, and M Piattini. 2004. Metamodels and architecture of an automatic code generator. In *2nd Nordic Workshop on the Unified Modeling Language, NWUML*.

[15] David Harel and Bernhard Rumpe. 2004. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer Journal* 37, 10 (October 2004), 64–72. https://doi.org/10.1109/MC.2004.172

[16] Katrin Hölldobler, Oliver Kautz, and Bernhard Rumpe. 2021. *MontiCore Language Workbench and Library Handbook: Edition 2021*. Shaker Verlag, Aachen.

[17] Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. 2018. Software Language Engineering in the Large: Towards Composing and Deriving Languages. *Journal Computer Languages, Systems & Structures* 54 (2018), 386–405. https://doi.org/10.1016/j.cl.2018.08.002

[18] Sven Jörges, Tiziana Margaria, and Bernhard Steffen. 2008. Genesys: service-oriented construction of property conform code generators. *Innovations in Systems and Software Engineering* 4, 4 (2008), 361–384. https://doi.org/10.1007/s11334-008-0071-2

[19] Anneke Kleppe. 2008. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Pearson Education.

[20] Thomas Kuhn, Soeren Kemmann, Mario Trapp, and Christian Schäfer. 2009. Multi-Language Development of Embedded Systems. In *9th OOPSLA DSM Workshop, Orlando, USA*.

[21] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. 2015. Management of Guided and Unguided Code Generator Customizations by Using a Symbol Table. In *Domain-Specific Modeling Workshop (DSM'15)*. ACM, 37–42. https://doi.org/10.1145/2846696.2846702

[22] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. 2016. An Extended Symbol Table Infrastructure to Manage the Composition of Output-Specific Generator Information. In *Modellierung 2016 Conference (LNI, Vol. 254)*. Bonner Köllen Verlag, 133–140.

[23] Jérôme Pfeiffer and Andreas Wortmann. 2021. Towards the Black-Box Aggregation of Language Components. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 576–585. https://doi.org/10.1109/MODELS-C53483.2021.00088

[24] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. 2014. Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. In *Model-Driven Robot Software Engineering Workshop (MORSE'14) (CEUR Workshop Proceedings, Vol. 1319)*. 66 – 77.

[25] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. 2015. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)* 6, 1 (2015), 33–57.

[26] Bran Selic. 2003. The Pragmatics of Model-Driven Development. *IEEE software* 20, 5 (2003), 19–25. https://doi.org/10.1109/MS.2003.1231146

[27] Bernhard Steffen, Tiziana Margaria, Ralf Nagel, Sven Jörges, and Christian Kubczak. 2007. Model-Driven Development with the jABC. In *Hardware and Software, Verification and Testing: Second International Haifa Verification Conference, HVC 2006, Haifa, Israel, October 23-26, 2006. Revised Selected Papers 2*. Springer, 92–108.

[28] Markus Voelter and Vaclav Pech. 2012. Language Modularity with the MPS Language Workbench. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 1449–1450.

[29] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, and Simon Helsen. 2013. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons.

[30] Dennis Leroy Wigand, Arne Nordmann, Niels Dehio, Michael Mistry, and Sebastian Wrede. 2017. Domain-Specific Language Modularization Scheme Applied to a Multi-Arm Robotics Use-Case. *Journal of Software Engineering for Robotics* (2017).

Received 2023-07-07; accepted 2023-09-01

Enabling Blended Modelling of Timing and Variability in EAST-ADL

Muhammad Waseem Anwar
muhammad.waseem.anwar@mdu.se
School of Innovation, Design, and Engineering, Mälardalen University
Västerås, Sweden

Federico Ciccozzi
federico.ciccozzi@mdu.se
School of Innovation, Design, and Engineering, Mälardalen University
Västerås, Sweden

Alessio Bucaioni
alessio.bucaioni@mdu.se
School of Innovation, Design, and Engineering, Mälardalen University
Västerås, Sweden

Abstract

EAST-ADL is a domain-specific modelling language for the design and analysis of vehicular embedded systems. Seamless modelling through multiple concrete syntaxes for the same language, known as blended modelling, offers enhanced modelling flexibility to boost collaboration, lower modelling time, and maximise the productivity of multiple diverse stakeholders involved in the development of complex systems, such as those in the automotive domain. Together with our industrial partner, which is one of the leading contributors to the definition of EAST-ADL and one of its main end-users, we provided prototypical blended modelling features for EAST-ADL.

In this article, we report on our language engineering work towards the provision of blended modelling for EAST-ADL to support seamless graphical and textual notations. Notably, for selected portions of the EAST-ADL language (i.e., timing and variability packages), we introduce ad-hoc textual concrete syntaxes to represent the language's abstract syntax in alternative textual notations, preserving the language's semantics. Furthermore, we propose a full-fledged runtime synchronisation mechanism, based on the standard EAXML schema format, to achieve seamless change propagation across the two notations. As EAXML serves as a central synchronisation point, the proposed blended modelling approach is workable with most existing EAST-ADL tools. The feasibility of the proposed approach is demonstrated through a car wiper use case from our industrial partner – Volvo. Results indicate that the proposed blended modelling approach is effective and can be applied to other EAST-ADL packages and supporting tools.

CCS Concepts: • **Software and its engineering → System description languages**; *Domain specific languages.*

SLE '23, October 23–24, 2023, Cascais, Portugal
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0396-6/23/10.
https://doi.org/10.1145/3623476.3623518

Keywords: Blended modelling, EAST-ADL, Vehicular embedded systems

ACM Reference Format:
Muhammad Waseem Anwar, Federico Ciccozzi, and Alessio Bucaioni. 2023. Enabling Blended Modelling of Timing and Variability in EAST-ADL. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3623476.3623518

1 Introduction

The complexity of vehicular embedded systems is escalating exponentially due to modern technological innovations. Consequently, rapid design prototyping and early analysis are two critical aspects to attaining productivity and time-to-market. For this purpose, the Electronics Architecture and Software Technology – Architecture Description Language (EAST-ADL) was first introduced by the ITEA EAST-EEA project [1]. It is a domain-specific modelling language (DSML) that allows the specification of a complete vehicular system through four levels of abstraction [2] i.e., Vehicle, Analysis, Design, and Implementation. Furthermore, it comprises several other packages for different aspects like variability and timing. Consequently, EAST-ADL provides comprehensive modelling capabilities for vehicular embedded systems and it facilitates the early analysis of both hardware and software design. Furthermore, it speeds up overall system development by automatically generating required implementations from system models. Therefore, EAST-ADL has become a de-facto standard for developing vehicular embedded systems [3].

Domain-specific modelling demands a high level of customisation of modelling tools, typically involving combinations and extensions of DSMLs and tailoring of the modelling tools for their respective development domains and contexts. Furthermore, tools are expected to provide multiple modelling means, e.g., textual and graphical, to satisfy the requirements set by different development phases, stakeholder roles, and application domains.

EAST-ADL tools traditionally focus on diagrammatic notation rather than textual ones. This limits communication, especially across different stakeholders. A notation that is well understood by a software engineer may not be as easily understood by a tester. Moreover, different engineers may have different notation preferences; not supporting multiple notations negatively affects the throughput of engineers. Besides the limits to communication, choosing one particular notation also limits the pool of available tools to develop and manipulate models that may be needed. For instance, choosing a graphical notation limits the usability of text manipulation tools such as text-based diff/merge, which is essential for team collaboration. This mutual exclusion suffices the need to develop small-scale applications with only a few stakeholder roles.

For larger systems, with heterogeneous components and entailing different domain-specific aspects and different types of stakeholders, such as vehicular embedded systems, the provision of one notation only is restrictive and may void many of the benefits that model-driven engineering (MDE) can bring about. When applying MDE in large-scale industrial projects, efficient team support is crucial. Therefore, modelling tools need to allow different stakeholders to work on overlapping parts of models using different concrete syntaxes or simply notations; EAST-ADL is not an exception.

Blended modelling is defined as the activity of interacting seamlessly with a single model (i.e., abstract syntax) through multiple notations (i.e., concrete syntaxes) [4]. Blended modelling is expected to aid in keeping the cognitive flow of modelling effective and efficient, offering stakeholders a proper set of intertwined formalisms, notations, and supporting computer-aided mechanisms. This is important in the design of modern vehicular systems, as their complexity has been increasing exponentially over the past years [5].

Together with our industrial partner - Volvo, we provided prototypical blended modelling features for EAST-ADL and, in this paper, we report on the language engineering work towards the provision of these features.

The architecture of our proposed blended modelling framework is shown in Fig. 1.

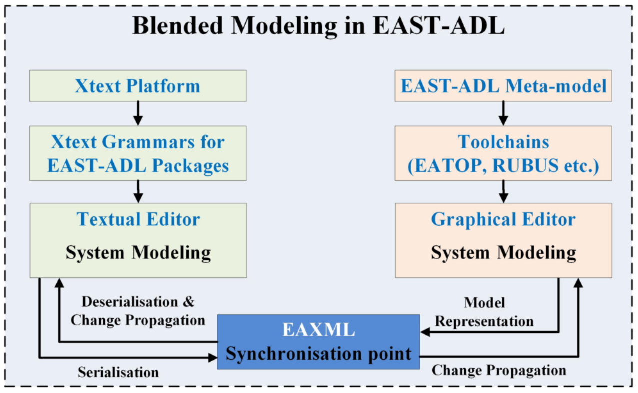

Blended Modeling in EAST-ADL
Xtext Platform
Xtext Grammars for EAST-ADL Packages
Textual Editor
System Modeling
EAST-ADL Meta-model
Toolchains (EATOP, RUBUS etc.)
Graphical Editor
System Modeling
Deserialisation & Change Propagation
Model Representation
EAXML Synchronisation point
Serialisation
Change Propagation

Figure 1. Architecture of our blended modelling framework for EAST-ADL

On top of the EAST-ADL meta-model, several tools and frameworks, like EATOP [6] and Rubus ICE [7], are developed to support system modelling via graphical notations. Textual notations for specific packages can be defined via any language workbench. We leveraged EATOP for graphical modelling and the Xtext platform [8] to specify textual notations in terms of EBNF grammars for the timing and variability packages. We propose the EAXML standard [9] as the central synchronisation point between graphical and textual notations. The transition from graphical to textual notations is achieved by de-serialising EAXML; serialisation and in-place modifications are instead used to transition from textual to graphical. Changes across notations are propagated through model transformations.

The viability of the proposed approach is assessed through a car wiper use case from Volvo. Notably, the design of a car wiper is modelled using EATOP's graphical editor, while timing constraints and variability are modelled using our textual notation, from where all design elements in the graphical notations are accessible. Subsequently, changes across the two notations are seamlessly propagated. The results indicate the feasibility of the approach and give indications of its applicability to other EAST-ADL packages at other levels of abstraction.

This article is organised as follows. Section 2 provides the summary of relevant state-of-the-art approaches regarding EAST-ADL architectural extensions and blended modelling in Section 2.1 and Section 2.2, respectively. Subsequently, the research motivation and goal are given in Section 2.3. The proposed blended modelling approach is presented in Section 3. Proof-of-concept is provided in Section 4, where implementation details about Xtext grammars and synchronisation mechanism are described in Section 4.1 and Section 4.2, respectively. Furthermore, the demonstration of blended modelling for the car wiper use case is presented in Section 4.3. The significant aspects in the given research context are discussed in Section 5. Finally, Section 6 concludes the article.

2 Related work

This section summarises the state-of-the-art approaches in our research scope. The studies dealing with EAST-ADL and blended modelling are given in Section 2.1 and Section 2.2, respectively. Furthermore, the identified research gaps as well as the motivation for our approach are highlighted in Section 2.3.

2.1 EAST-ADL

EAST-ADL is a domain-specific modelling language, particularly an adaptation of SysML, for automotive embedded systems [5]. It is based on four levels of abstraction, starting from the topmost vehicle level down to the implementation level. In addition to this, it provides other packages for the modelling of timing, variability and more. Consequently, it allows comprehensive system modelling with early analysis features for automotive embedded systems. It is managed by the EAST-ADL association which releases new EAST-ADL versions over time. Eclipse EATOP [6] is a well-known EAST-ADL tool grounded on EMF-based implementation of the EAST-ADL meta-model. It facilitates the tree-based system modelling supporting all four levels of abstraction. Furthermore, different plugins like a graphical viewer and Hip-HOPS bridge [10] are available to enhance the functionality of EATOP for particular purposes. Rubus ICE - Integrated Component model development Environment [7] is another EAST-ADL tool supporting advanced modelling and analysis features. Similarly, there exist several EAST-ADL tools from different vendors. The interoperability among various EAST-ADL tools is achieved through EAXML – an XML-based exchange format [5]. EAXML represents the serialised form of EAST-ADL meta model instances; therefore, its contents adhere to the XML schema of the complementary EAST-ADL version. Consequently, EAXML schema in XSD format is also released along with every EAST-ADL version [9].

With comprehensive specifications and rich tool support, EAST-ADL is frequently applied and researched from different perspectives for the development of automotive embedded systems. Etzel et al. [11] proposed an extension in EAST-ADL architecture to support the modelling and analysis of data dependency-based partitions. The proposed approach facilitates stakeholders to decide on correct design choices, from the analysis level down to the design and hardware level, based on the distribution of different system components. Architectural modelling of security aspects is not covered in EAST-ADL, and therefore, Zoppelt et al. [12] extended the EAST-ADL meta-model to include several security-related concepts like attack, adversary and so forth. Optimisation of Product Line Architectures (PLAs) becomes important in complex automotive embedded systems. Wägemann et al. [13] carried out a study to identify the characteristics, importance and challenges of automated design space exploration of product lines in industrial settings. In this context, Walker et al. [14] proposed an automated approach to optimize PLAs in EAST-ADL architecture. The proposed approach is mainly grounded on EAST-ADL variability semantics, and it automatically evaluates different objectives (e.g. cost and dependability) and performs PLAs optimisation automatically through genetic algorithms.

Energy-aware real-time automotive systems are designed to operate within strict timing constraints and consider the system's energy consumption. In this context, EAST-ADL architectural extensions are frequently explored. Kang et al. [15] proposed a probabilistic extension of EAST-ADL to support the modelling and analysis of energy constraints. The semantics of the expanded constraints were transformed into UPPAAL and SIMULINK to perform the formal analysis and simulation, respectively. Enoiu and Seceleanu [16] proposed EAST-ADL architectural mutations for the testing of design models. Marinescu et al. [17] proposed a verification framework to automatically generate test cases from EAST-ADL models.

Automotive embedded systems are usually developed on two platforms i.e., single and multicore. EAST-ADL architecture is also explored in the context of development platforms. Bucaioni et al. [18] proposed a cost-effective development methodology for vehicular embedded systems equally effective for single and multi-core platforms. In another study, Remko van Wagensveld et al. [19] proposed a supercore pattern comprising multiple processing cores for improving the performance of automotive embedded systems.

2.2 Blended modelling

Blended modelling is expected to aid in keeping the cognitive flow of modelling effective and efficient, offering stakeholders a proper set of intertwined formalisms, notations, and supporting computer-aided mechanisms. This is important in the design of vehicular systems, as their complexity has been increasing exponentially over the past years. At first sight, the notion of blended modelling may seem similar to or overlapping with multi-view modelling [20] (and even multi-paradigm modelling) that is based on the paradigm of viewpoint/view/model as formalised in the ISO/IEC 42010 standard. Multi-view modelling is commonly based on viewpoints (i.e. "conventions for the construction, interpretation, and use of architecture views to frame specific system concerns"[21]) that are materialised through views, which are composed of one or more models. The blended modelling paradigm focuses on the provision of multiple concrete syntaxes, or simply notations, for a non-empty set of abstract syntactic concepts.

Blended modelling aims at boosting the development of complex systems, such as vehicular embedded systems, by enabling seamless multi-notation modelling. This improvement is expected to improve both the throughput of engineers and the communication between different stakeholders.

The majority of existing tools supporting UML-based DSMLs only provide graphical notations. To achieve blended modelling in these cases, the development of textual notations and runtime synchronisation between graphical and textual notations are the two major concerns. In this context, several attempts have been made to facilitate blended modelling for UML-based DSMLs. Addazi et al. [22] proposed a blended modelling methodology for UML profiles. For demonstration purposes, an example based on a portion of the MARTE profile was considered. The authors developed and incorporated a textual notation defined in Xtext in Papyrus[1]. Both textual and graphical notations operate on a single UML resource, and therefore runtime synchronisation between notations

[1]https://eclipse.dev/papyrus/

was achieved through simplified transformations. Latifaj et al. [23] proposed a blended modelling approach for UML-RT state machines in the HCL RTist toolchain [24]. A textual editor was developed in Xtext and integrated in RTist to support textual modelling. Furthermore, model transformations were implemented first in QVTo[2] to achieve runtime synchronisation between graphical and textual notations; the generation of QVTo transformations via higher-order transformations was then proposed by the same authors too [25]. Maro et al. [26] presented a blended modelling approach for a proprietary UML-based DSML named Ericsson Hive. Firstly, an Ecore DSML was generated automatically from the Hive profile. Secondly, features of Xtext were exploited to provide textual notation. Finally, the transformations (from UML to Xtext and vice versa) were implemented in ATL to achieve runtime synchronisation between textual and graphical notations.

In addition to UML profiles, the application of blended modelling is also common in DSMLs based on meta-models described in other languages. Anwar et al. [27] proposed a blended modelling framework where, starting from an Ecore-based meta-model, the graphical and textual notations were generated automatically. An editor was presented to perform mappings between graphical and textual notations. Based on the mappings, transformations were implemented in the JAVACC platform for runtime synchronisation. Finally, both graphical and textual editors were developed in Sirius to achieve blended modelling. In another study [28], the same authors emphasized that the concept of blended modelling is not restricted to existing DSMLs. They proposed a blended meta-modelling framework for the development of DSMLs by integrating the techniques of Natural Language Processing (NLP) and MDE. The proposed framework implemented iterative transformations to automatically generate Ecore-based meta-models from restricted natural language and vice versa. Consequently, it allows the development of DSMLs through graphical (ECORE) and textual (natural language) notations seamlessly.

2.3 Research motivation and goal

EAST-ADL architectural extensions are frequently researched from different perspectives like security [12], energy consumption [17], and product lines [14] to improve automotive system development. Nevertheless, blended modelling support, regarded as a core needed feature by end-EAST-ADL users such as our industrial partner has not been targeted yet, to the best of our knowledge. As blended modelling offers enhanced modelling flexibility to boost collaboration, lower modelling time, and maximise the productivity of multiple diverse stakeholders involved in the development of such complex systems as those in the automotive domain, our goal together with our industrial partner was to provide prototypical blended modelling features for EAST-ADL for automotive embedded systems development leveraging the existing EATOP graphical modelling workbench. Also in accordance with our partner's internal investigations, we focused on a solution for the timing and variability packages in EAST-ADL. The reason behind this choice was that those packages were deemed to benefit the most from a textual specification according for multiple reasons. The effective management of variability enables our partner to meet the growing demand for customised products and produce vehicle variations at high volumes. Meanwhile, compliance with timing requirements is pivotal for safety certifications and the overall functional suitability of vehicular systems. More efficient modelling of these aspects is thereby a core aim for engineering teams at Volvo. We focus on providing a blended modelling approach that is compliant with the standard EAST-ADL architecture as defined in its meta-model [9]. This simplifies the applicability of our approach to a wide range of existing EAST-ADL tools.

3 Proposed approach

Our aim is to provide an approach that can be applied to existing tools supporting graphical EAST-ADL modelling to make them blended. Consequently, our first concern was selecting a platform for developing a textual notation that could be functional with the graphical notations of existing EAST-ADL tools. In this context, we investigated Eclipse-based solutions, i.e. Xtext, and JetBrains MPS. We chose Xtext since it integrates finely with the EAST-ADL's open-source modelling framework, EATOP since both are compatible with the Eclipse Modeling Framework (EMF).

Another important concern is the realization of runtime synchronisation between graphical and textual notation. One way to achieve this was to implement specific transformations for different EAST-ADL tools separately, which was clearly not viable. Therefore, we chose to use the EAXML format as a pivot language for synchronisation between notations. EAXML is the XML-based standard exchange format to attain interoperability among EAST-ADL tools. Consequently, a single EAXML-based synchronisation mechanism is workable theoretically with any EAST-ADL tool adhering to EAXML.

The architecture of the proposed blended modelling approach is shown in Fig. 2. Several tools supporting EAST-ADL like EATOP, Papyrus, and Rubus ICE provide sophisticated graphical editors. Therefore, in the proposed approach, we advocate the use of existing graphical notations/editors from any EAST-ADL tool adhering to the EAXML format, as shown in Fig. 2. For the introduction of a textual notation and for runtime synchronisation between notations, we propose the following five steps:

1. **Definition of Xtext grammar and textual editor:** We advocate the definition of an Xtext grammar for the

[2]https://projects.eclipse.org/projects/modeling.mmt.qvt-oml

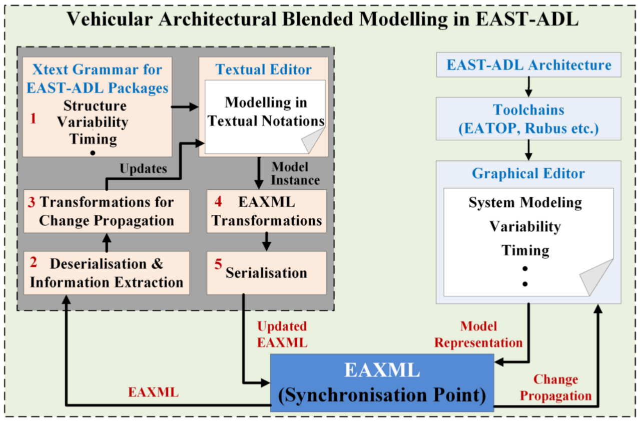

Vehicular Architectural Blended Modelling in EAST-ADL
Xtext Grammar for EAST-ADL Packages
1
Structure
Variability
Timing
Textual Editor
Modelling in Textual Notations
Updates
Model Instance
3 Transformations for Change Propagation
4 EAXML Transformations
2 Deserialisation & Information Extraction
5 Serialisation
EAST-ADL Architecture
Toolchains (EATOP, Rubus etc.)
Graphical Editor
System Modeling
Variability
Timing
Updated EAXML
Model Representation
EAXML (Synchronisation Point)
EAXML
Change Propagation

Figure 2. Architecture of proposed blended modelling approach

EAST-ADL packages of interest by adding the needed syntactic sugar for the textual notation to be comprehensive and effective while preserving the semantics of the entailed EAST-ADL meta-concepts. This is essential to achieve seamless runtime synchronisation through EAXML at later steps. Once the grammar is successfully specified, a basic textual editor is automatically generated in Xtext. Advanced editing features like content assistance and custom syntax highlighting can be included by exploiting built-in extension features in Xtext.

2. **Deserialisation and information extraction:** This step entails the deserialisation of an EAST-ADL model in EAXML from a graphical notation. Subsequently, the required information about various level entities in the model can be extracted via model transformations.
3. **Information/change propagation:** Information extracted in Step 2 cannot be propagated to the textual notation directly. This is due to likely syntactic differences between the extracted information and their representation in the textual notation, because of the syntactic sugar mentioned in Step 1. Therefore, we leverage specific model transformations to manipulate extracted information for conformance to the textual notation.
4. **EAXML transformations:** Steps 2 and 3 allow propagating changes from graphical to textual notation. To propagate changes from textual to graphical, we leverage specific model transformations to manipulate textual models and convert the information that they carry into EAXML.
5. **Serialisation:** Transformations in step 4 convert textual models into the basic EAXML structure without validating its conformance with standard XSD schema. This step entails the systematic merging of model concepts from textual and graphical notations in serialised EAXML format, compliant with standard XSD, for change propagation from textual to graphical notation.

It is important to mention that Step 1 is meant to be a one-time effort, only to be carried out once to establish a specific textual notation. The same goes for the implementation of the model transformations in the subsequent steps. On the other hand, Steps 2 to 5 are executed iteratively every time we need to perform synchronisation between graphical and textual notations (which may be either on-the-fly or on-demand, depending on the configuration of the modelling environment and the end-user needs). Although the approach was described as if we were starting from the graphical notation, the other way around, that is to say starting from the textual notation, is equally possible and the order of the steps will change accordingly.

4 Proof-of-concept implementation

The feasibility of the proposed blended modelling approach for EAST-ADL is demonstrated by implementing a proof-of-concept prototype and leveraging a car wiper use case from Volvo. We leverage the EATOP modelling tool where the car wiper can be graphically modelled at different levels of abstraction, as shown in Fig. 3. In line with Step 1

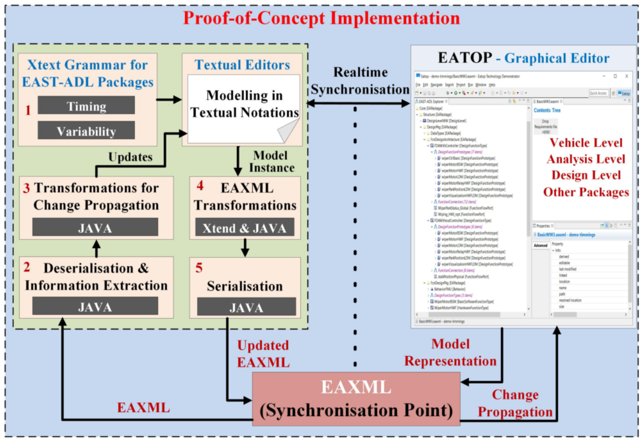


Figure 3. Overview of proof-of-concept implementation

of our approach, we developed a textual concrete syntax in terms of Xtext grammars and corresponding editors for EAST-ADL's timing and variability packages. Steps 2 to 5 are implemented in terms of model transformations in either plain Java or Xtend, to achieve runtime synchronisation between the two notations, as shown in Fig. 3. Eventually, we assessed the usability of our EATOP blended modelling environment through the car wiper use case. More specifically, the design model was created using the EATOP graphical notation while timing constraints were specified using the Xtext textual notation leveraging the information from design function prototypes expressed in the graphical one.

Subsequently, the timing constraints can be visualised and edited in the graphical editor as well, and changes propagated back and forth across notations thanks to the runtime synchronisation based on EAXML.

In the remainder of this section, we first explain how we defined and implemented Xtext grammars and thereby textual editors for EAST-ADL timing and variability packages. Then we describe how we engineered runtime synchronisation steps (i.e., 2 to 5) and, eventually, we reason on the effectiveness of the proposed approach in terms of the car wiper use case.

4.1 Definition and implementation of textual concrete syntaxes and editors with Xtext

When specifying the Xtext grammars, our aim was to provide syntactic sugar on top of EAST-ADL's abstract syntax for maximising the effectiveness of textual notations while preserving the language's semantics. We are interested in timing and variability meta-concepts as available in the EAST-ADL meta-model and depicted in Fig. 4 (a) and 4 (b).

EAST-ADL timing is based on TimingDescriptions and TimingConstraints elements. TimingDescriptions include several other related concepts like Events, FunctionEvents, EventChain and so forth. TimingConstraints overall offer seventeen different types of constraints, including Time Augmented Description Language (TADL 2) constraints [29]. EAST-ADL variability is mostly based on the concept of ConfigurationDecisionModel, which incorporates other core concepts like SelectionCriterion, VehicleLevelBinding, and so forth. A detailed description of timing and variability concepts is available in the EAST-ADL specification [9]. For demonstration purposes, we consider only a subset of EAST-ADL timing and variability concepts.

One of the most important tasks in the definition of a grammar is to decide upon the syntax of the resulting textual notation to be used by the end-user to edit textual models. In this regard, we discussed both natural and programming language-like syntax options with our industrial partner and, upon their recommendation, we eventually opted for a programming language-like syntax. Snapshots of the Xtext grammars for timing and variability packages are shown in Fig. 5 (a) and Fig. 5 (b), respectively. For the timing grammar, we defined 26 rules to provide the textual notation for EAST-ADL timing concepts, as shown in the outline in Fig. 5 (a). The timing rule (Fig. 5 a) defines the syntax and semantics of the timing elements in grammar. Particularly, it starts with the 'Timing ' keyword, contains five variables, and is organised into events and constraints. Similarly, the syntax and semantics of each rule are defined in timing grammar. For the variability grammar, overall, 29 rules were defined, as shown in the outline in Fig. 5 (b).

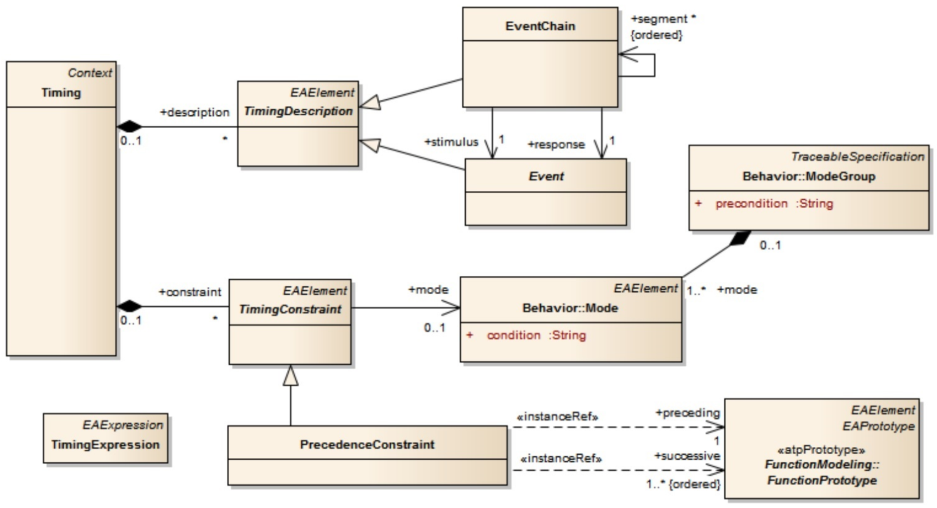
After the definition of Xtext grammars for the two packages, the corresponding textual editors with basic functionalities, like syntax highlighting, are generated with Xtext. We enhanced the editors with advanced editing features by leveraging the built-in Xtext extension features. Particularly, in Xtext, a content assistant (also known as code completion or auto-complete) makes suggestions on the language's syntax and the current context in the textual editor. We overrode the content proposal provider (AbstractJavaBasedContentProposalProvider method) to incorporate such a content assistant based on the information that is gathered in real-time from the graphical editor. This makes modelling in the textual editor more effective. The resulting textual editors for EAST-ADL timing and variability packages are shown in Fig. 6 (a) and Fig. 6 (b), respectively. Note that we developed two separate textual concrete syntaxes and editors for EAST-ADL timing and variability packages for demonstration purposes as well as for separation of concerns. Nevertheless, it is fully possible to merge them into a single grammar and textual editor, as well as to include additional packages for EAST-ADL.

4.2 Implementation of runtime synchronisation

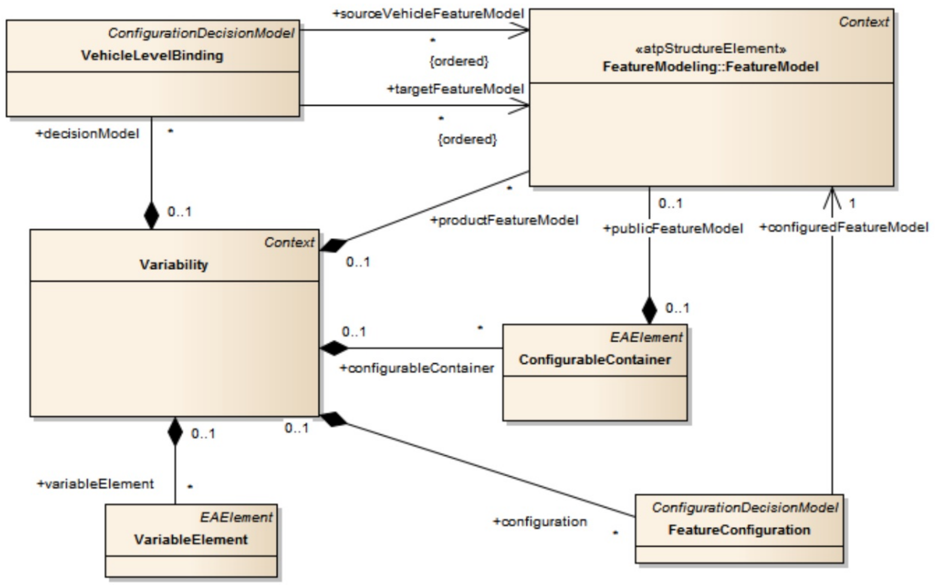
We propose four steps (i.e., Step 2 to 5) to achieve runtime synchronisation between textual and graphical notations, as depicted in Fig. 3. The implementation details for each synchronisation step are as follows.

4.2.1 Deserialisation and information extraction. We implemented an EAXML parser in Java using XML DOM (Document Object Model) for deserialisation and information extraction from a given EAXML. Particularly, our solution parses the EAXML nodes in the given model file following the EAXML schema, and it stores the required information about modelling entities through the transformer object. The definitions and structure of EAXML nodes compliant with EAST-ADL packages are available in the XSD that comes along with the EAST-ADL specification. Although we implemented an EAXML parser for structure, timing, and variability packages for demonstration purposes, our approach can be extended to support other EAST-ADL packages and concepts. For the interested reader, the source code of the EAXML parser is publicly available at [30].

4.2.2 Transformations for change propagation. We extract information like vehicle features, analysis functions, design function prototypes, timing constraints, and so forth after the deserialisation. However, this information cannot be directly propagated to the textual notation due to the syntactic differences between the two concrete syntaxes. Therefore, we implemented a set of transformations in Java to establish conformance between the extracted information and the textual notation. For instance, the names of analysis and design function prototype elements are transformed into a hierarchical string pattern for correct change propagation to the textual notation.



(a) Timing

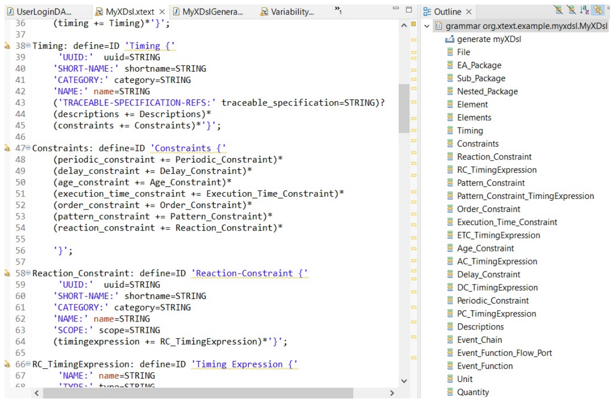


(b) Variability

Figure 4. Organization of entities in EAST-ADL meta-model [9]

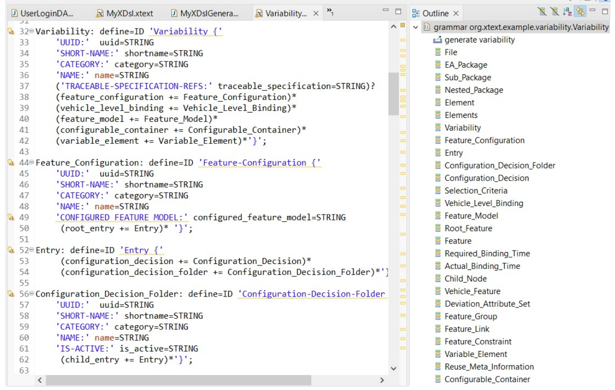
4.2.3 EAXML transformations. We defined and implemented transformations in Xtend to generate the desired EAXML target model from a source textual model. These transformations are executed on the fly, i.e. target EAXML model is generated side by side during the specification of the textual instance model. We defined an Xtend template, with the placeholders for the runtime values to be gathered from the textual model to generate the EAXML file. Snapshots of Xtend templates for timing and variability textual editors are shown in Fig. 7 (a) and Fig. 7 (b), respectively. The transformation rules to generate EAXML for event functions and event function flowports within timing are depicted in Fig. 7 (a), where placeholders (e.g., «timing.name») are replaced with instance values from the textual model when executing the transformation. Similarly, EAXML transformation rules are implemented for other timing and variability concepts.

4.2.4 Serialisation. EAXML transformations convert a textual instance model into a basic EAXML structure. Then, we implemented a serialiser, using Java and XML DOM, to encode all model information into EAXML's XSD. Furthermore, we provide a merger, which puts the serialised textual instance model into the EAXML representing the graphical model. To ensure the correctness of the merge operation, we validate the conformance of the resulting EAXML with the XSD. For this, we implemented a validator for the merged



```
    (timing += Timing)*'}';

Timing: define=ID 'Timing {'
     'UUID:'  uuid=STRING
    'SHORT-NAME:' shortname=STRING
    'CATEGORY:' category=STRING
    'NAME:' name=STRING
    ('TRACEABLE-SPECIFICATION-REFS:' traceable_specification=STRING)?
    (descriptions += Descriptions)*
    (constraints += Constraints)*'}';

Constraints: define=ID 'Constraints {'
    (periodic_constraint += Periodic_Constraint)*
    (delay_constraint += Delay_Constraint)*
    (age_constraint += Age_Constraint)*
    (execution_time_constraint += Execution_Time_Constraint)*
    (order_constraint += Order_Constraint)*
    (pattern_constraint += Pattern_Constraint)*
    (reaction_constraint += Reaction_Constraint)*

    '}';

Reaction_Constraint: define=ID 'Reaction-Constraint {'
     'UUID:'  uuid=STRING
    'SHORT-NAME:' shortname=STRING
    'CATEGORY:' category=STRING
    'NAME:' name=STRING
    'SCOPE:' scope=STRING
    (timingexpression += RC_TimingExpression)*'}';

RC_TimingExpression: define=ID 'Timing Expression {'
     'NAME:' name=STRING
```

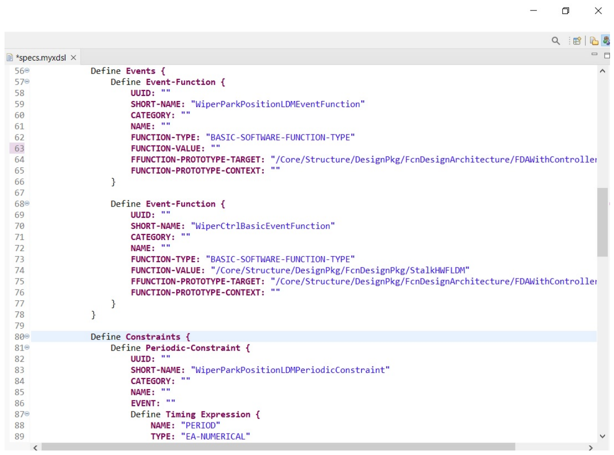
(a) Timing package



```
Variability: define=ID 'Variability {'
    'UUID:'  uuid=STRING
    'SHORT-NAME:' shortname=STRING
    'CATEGORY:' category=STRING
    'NAME:' name=STRING
    ('TRACEABLE-SPECIFICATION-REFS:' traceable_specification=STRING)?
    (feature_configuration += Feature_Configuration)*
    (vehicle_level_binding += Vehicle_Level_Binding)*
    (feature_model += Feature_Model)*
    (configurable_container += Configurable_Container)*
    (variable_element += Variable_Element)*'}';

Feature_Configuration: define=ID 'Feature-Configuration {'
    'UUID:'  uuid=STRING
    'SHORT-NAME:' shortname=STRING
    'CATEGORY:' category=STRING
    'NAME:' name=STRING
    'CONFIGURED FEATURE MODEL:' configured_feature_model=STRING
     (root_entry += Entry)* '}';

Entry: define=ID 'Entry {'
     (configuration_decision += Configuration_Decision)*
     (configuration_decision_folder += Configuration_Decision_Folder)*'}

Configuration_Decision_Folder: define=ID 'Configuration-Decision-Folder
    'UUID:'  uuid=STRING
    'SHORT-NAME:' shortname=STRING
    'CATEGORY:' category=STRING
    'NAME:' name=STRING
    'IS-ACTIVE:' is_active=STRING
     (child_entry += Entry)*'}';
```

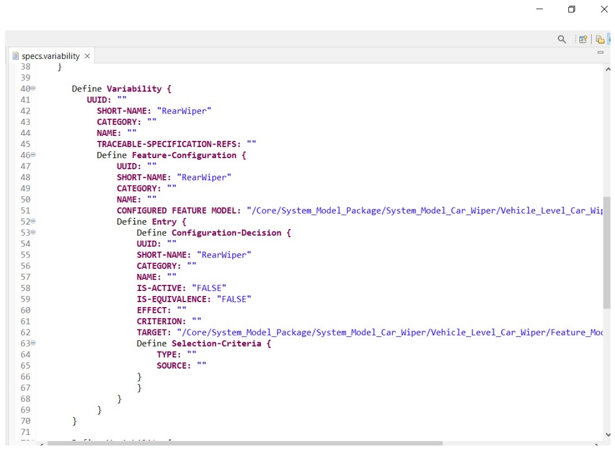
(b) Variability package

Figure 5. Implemented Xtext grammars



```
        Define Events {
            Define Event-Function {
                UUID: ""
                SHORT-NAME: "WiperParkPositionLDMEventFunction"
                CATEGORY: ""
                NAME: ""
                FUNCTION-TYPE: "BASIC-SOFTWARE-FUNCTION-TYPE"
                FUNCTION-VALUE: ""
                FFUNCTION-PROTOTYPE-TARGET: "/Core/Structure/DesignPkg/FcnDesignArchitecture/FDAWithController
                FUNCTION-PROTOTYPE-CONTEXT: ""
            }

            Define Event-Function {
                UUID: ""
                SHORT-NAME: "WiperCtrlBasicEventFunction"
                CATEGORY: ""
                NAME: ""
                FUNCTION-TYPE: "BASIC-SOFTWARE-FUNCTION-TYPE"
                FUNCTION-VALUE: "/Core/Structure/DesignPkg/FcnDesignPkg/StalkHWFLDM"
                FFUNCTION-PROTOTYPE-TARGET: "/Core/Structure/DesignPkg/FcnDesignArchitecture/FDAWithController
                FUNCTION-PROTOTYPE-CONTEXT: ""
            }
        }

        Define Constraints {
            Define Periodic-Constraint {
                UUID: ""
                SHORT-NAME: "WiperParkPositionLDMPeriodicConstraint"
                CATEGORY: ""
                NAME: ""
                EVENT: ""
                Define Timing Expression {
                    NAME: "PERIOD"
                    TYPE: "EA-NUMERICAL"
```

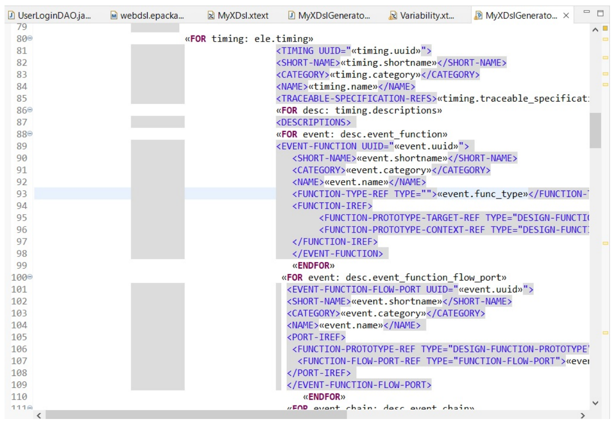
(a) Timing package



```
    }

        Define Variability {
           UUID: ""
             SHORT-NAME: "RearWiper"
             CATEGORY: ""
             NAME: ""
             TRACEABLE-SPECIFICATION-REFS: ""
             Define Feature-Configuration {
                 UUID: ""
                 SHORT-NAME: "RearWiper"
                 CATEGORY: ""
                 NAME: ""
                 CONFIGURED FEATURE MODEL: "/Core/System_Model_Package/System_Model_Car_Wiper/Vehicle_Level_Car_Wi
                 Define Entry {
                     Define Configuration-Decision {
                     UUID: ""
                     SHORT-NAME: "RearWiper"
                     CATEGORY: ""
                     NAME: ""
                     IS-ACTIVE: "FALSE"
                     IS-EQUIVALENCE: "FALSE"
                     EFFECT: ""
                     CRITERION: ""
                     TARGET: "/Core/System_Model_Package/System_Model_Car_Wiper/Vehicle_Level_Car_Wiper/Feature_Mo
                     Define Selection-Criteria {
                         TYPE: ""
                         SOURCE: ""
                     }
                     }
                 }
             }
        }
```

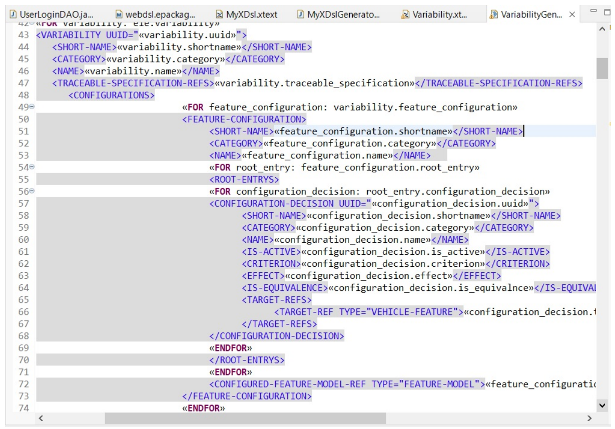
(b) Variability package

Figure 6. Textual editors



```
                   «FOR timing: ele.timing»
                               <TIMING UUID="«timing.uuid»">
                               <SHORT-NAME>«timing.shortname»</SHORT-NAME>
                               <CATEGORY>«timing.category»</CATEGORY>
                               <NAME>«timing.name»</NAME>
                               <TRACEABLE-SPECIFICATION-REFS>«timing.traceable_specificat
                               «FOR desc: timing.descriptions»
                               <DESCRIPTIONS>
                               «FOR event: desc.event_function»
                               <EVENT-FUNCTION UUID="«event.uuid»">
                                 <SHORT-NAME>«event.shortname»</SHORT-NAME>
                                 <CATEGORY>«event.category»</CATEGORY>
                                 <NAME>«event.name»</NAME>
                                 <FUNCTION-TYPE-REF TYPE="">«event.func_type»</FUNCTION-
                                 <FUNCTION-IREF>
                                     <FUNCTION-PROTOTYPE-TARGET-REF TYPE="DESIGN-FUNCTI
                                     <FUNCTION-PROTOTYPE-CONTEXT-REF TYPE="DESIGN-FUNCT
                                 </FUNCTION-IREF>
                                 </EVENT-FUNCTION>
                                 «ENDFOR»
                                «FOR event: desc.event_function_flow_port»
                                <EVENT-FUNCTION-FLOW-PORT UUID="«event.uuid»">
                                <SHORT-NAME>«event.shortname»</SHORT-NAME>
                                <CATEGORY>«event.category»</CATEGORY>
                                <NAME>«event.name»</NAME>
                                <PORT-IREF>
                                 <FUNCTION-PROTOTYPE-REF TYPE="DESIGN-FUNCTION-PROTOTYPE
                                  <FUNCTION-FLOW-PORT-REF TYPE="FUNCTION-FLOW-PORT">«eve
                                </PORT-IREF>
                                </EVENT-FUNCTION-FLOW-PORT>
                                   «ENDFOR»
```

(a) Timing package



```
<VARIABILITY UUID="«variability.uuid»">
   <SHORT-NAME>«variability.shortname»</SHORT-NAME>
   <CATEGORY>«variability.category»</CATEGORY>
   <NAME>«variability.name»</NAME>
   <TRACEABLE-SPECIFICATION-REFS>«variability.traceable_specification»</TRACEABLE-SPECIFICATION-REFS>
      <CONFIGURATIONS>
                       «FOR feature_configuration: variability.feature_configuration»
                       <FEATURE-CONFIGURATION>
                            <SHORT-NAME>«feature_configuration.shortname»</SHORT-NAME>
                            <CATEGORY>«feature_configuration.category»</CATEGORY>
                            <NAME>«feature_configuration.name»</NAME>
                            «FOR root_entry: feature_configuration.root_entry»
                            <ROOT-ENTRYS>
                            «FOR configuration_decision: root_entry.configuration_decision»
                            <CONFIGURATION-DECISION UUID="«configuration_decision.uuid»">
                                  <SHORT-NAME>«configuration_decision.shortname»</SHORT-NAME>
                                  <CATEGORY>«configuration_decision.category»</CATEGORY>
                                  <NAME>«configuration_decision.name»</NAME>
                                  <IS-ACTIVE>«configuration_decision.is_active»</IS-ACTIVE>
                                  <CRITERION>«configuration_decision.criterion»</CRITERION>
                                  <EFFECT>«configuration_decision.effect»</EFFECT>
                                  <IS-EQUIVALENCE>«configuration_decision.is_equivalnce»</IS-EQUIVA
                                  <TARGET-REFS>
                                        <TARGET-REF TYPE="VEHICLE-FEATURE">«configuration_decision.
                                  </TARGET-REFS>
                            </CONFIGURATION-DECISION>
                            «ENDFOR»
                            </ROOT-ENTRYS>
                            «ENDFOR»
                            <CONFIGURED-FEATURE-MODEL-REF TYPE="FEATURE-MODEL">«feature_configurati
                       </FEATURE-CONFIGURATION>
                       «ENDFOR»
```

(b) Variability package

Figure 7. EAXML transformations in Xtend

EAXML, by leveraging XML DOM. This validation is essential for seamless change propagation from textual to graphical notations, as most EAST-ADL tools operate on serialised EAXML only.

In this section, we provided details on our proof-of-concept implementation for validating our approach. Although we implemented grammars and textual editors only for the subset of EAST-ADL timing and variability packages, the same approach can be used to provide blended modelling support for other EAST-ADL packages as well. Similarly, the proposed synchronisation mechanisms can be extended to cover other EAST-ADL packages too. The proof-of-concept implementation (Xtext grammars and synchronisation mechanisms) is publicly available at [30].

4.3 Feasibility of our approach

In this section, the feasibility of our approach is shown through the car wiper use case using our proof-of-concept implementation. Notably, the design-level modelling of the car wiper is carried out in the EATOP graphical editor. The modelling of timing constraints and variability is done instead using our textual notations and related editors. When modelling in our textual editors, design-level elements defined in the graphical editor are available for auto-completion and syntax highlighting. On the other hand, although modelled in our textual editors, timing constraints and variability details are reflected in the graphical editor. Any alteration done to the timing and variability concepts in either notation is seamlessly propagated on the fly to the other as well.

With the evolution of vehicular technology, the wiping systems also evolves, and advanced features are incorporated in modern vehicles. In our use case, we consider the design of a basic car wiper with a stalk and a motor. The modelling is done using EAST-ADL design-level concepts in the EATOP graphical editor, as shown in Fig. 8 (a). Particularly, we define two DesignFunctionTypes of car wiper (i.e., with and without controller) through 13 DesignFunctionPrototypes, 20 FunctionConnectors and 3 FunctionFlowPorts. Moreover, the behaviour of the car wiper is modelled through 6 FunctionBehaviors. Furthermore, different basic software and hardware FunctionTypes are defined for the wiper motor. For brevity, we do not describe in detail the design level meta-concepts like DesignFuctionPrototype; for more details on them please refer to the EAST-ADL specification [9].

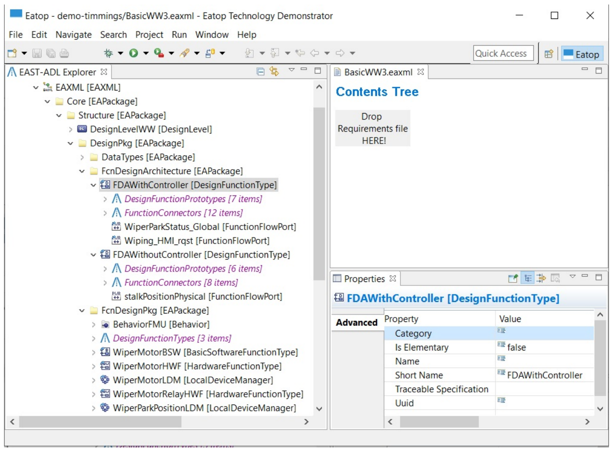
We model timing constraints in our textual editor. In particular, we model three data types (i.e., period, second and time) and two EventFunctions (WiperCtrlBasic and WiperParkPositionLDM). Moreover, we model four types of timing constraints i.e., ExecutionTime, periodic, delay and age constraints. In Fig. 8 (b), all DesignFunctionPrototypes are accessible in a dropdown list through content assistance (CTRL + Space) during the textual modelling of EventFunctions. Similarly, other design elements like DesignFunctionTypes, software and hardware FunctionTypes are accessible in the textual editor too. Initially, timing constraints are specified in the textual editor, but after that, they can be edited in both notations seamlessly. Changes to timing concepts across notations are propagated on the fly through Steps 4 and 5 of our approach via the synchronisation mechanisms. It can be seen in Fig. 9(a) that the data types, EventFunctions, and timing constraints modelled in the textual editor are reflected in the graphical editor. We define BLENDED EventFunction in the graphical editor for demonstration purposes. This EventFunction is immediately accessible in the textual editor during the modelling of periodic constraint, as shown in Fig. 9(b). Concerning variability, for demonstration purposes, we model an additional variant of car wiper, automatic with a rain sensor, as shown in Fig. 10. When textually modelling variants, graphical elements like FeatureModel and VehicleFeatures are accessible. Changes across notations are propagated by the same mechanisms as for the timing constraints. It can be seen in Fig. 10 that EAXML is also generated on the fly by the EAXML transformations when textually modelling timing and variability.

We showed the feasibility of the proposed approach by modelling a car wiper. We focused on showing the textual modelling capabilities and the seamless synchronisation mechanisms between textual and graphical editors.

5 Discussion

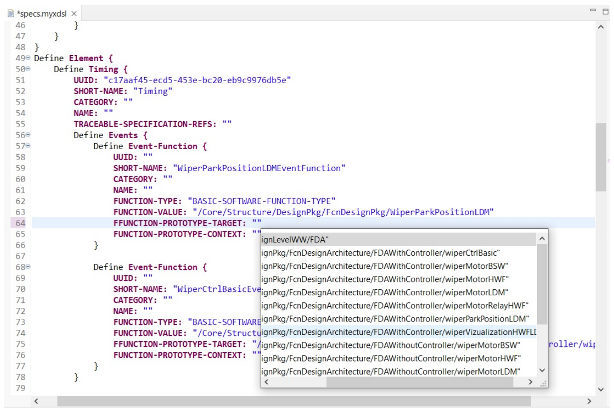
We provided a solution for blended modelling of timing and variability for EAST-ADL. These packages are deemed to benefit the most from a textual specification according to internal investigations at our industrial partner for multiple reasons. The effective management of variability enables our partner to meet the growing demand for customised products and produce vehicle variations at higher rates, while compliance with timing requirements is pivotal for safety certifications and the overall functional suitability of vehicular systems. More efficient modelling of these aspects, which is brought by blended modelling, is a core aim for engineering teams at Volvo and was the driving factor of this work.

The scalability of the solution, especially in terms of latency for change propagation across editors is a very important aspect, especially in the case of large system models. We assessed this aspect for our synchronisation mechanisms. We used an HP CORE i5 8th generation laptop with 8 GB RAM to perform the blended modelling of the car wiper use case. The initial size of the use case model in EAXML format is around 72 KB. We model two EventFunctions in graphical and textual editors separately for analysis purposes. We found that the changes from textual to graphical are propagated in <900 milliseconds while change propagation from graphical to textual takes <1800 milliseconds. The change propagation time from graphical to textual is larger because the deserialisation of EAXML for an entire system model is a compute-intensive


Eatop - demo-timmings/BasicWW3.eaxml - Eatop Technology Demonstrator
File Edit Navigate Search Project Run Window Help
Quick Access
Eatop
EAST-ADL Explorer
EAXML [EAXML]
Core [EAPackage]
Structure [EAPackage]
DesignLevelWW [DesignLevel]
DesignPkg [EAPackage]
DataTypes [EAPackage]
FcnDesignArchitecture [EAPackage]
FDAWithController [DesignFunctionType]
DesignFunctionPrototypes [7 items]
FunctionConnectors [12 items]
WiperParkStatus_Global [FunctionFlowPort]
Wiping_HMI_rqst [FunctionFlowPort]
FDAWithoutController [DesignFunctionType]
DesignFunctionPrototypes [6 items]
FunctionConnectors [8 items]
stalkPositionPhysical [FunctionFlowPort]
FcnDesignPkg [EAPackage]
BehaviorFMU [Behavior]
DesignFunctionTypes [3 items]
WiperMotorBSW [BasicSoftwareFunctionType]
WiperMotorHWF [HardwareFunctionType]
WiperMotorLDM [LocalDeviceManager]
WiperMotorRelayHWF [HardwareFunctionType]
WiperParkPositionLDM [LocalDeviceManager]
BasicWW3.eaxml
Contents Tree
Drop Requirements file HERE!
Properties
FDAWithController [DesignFunctionType]
Advanced

Property	Value
Category	
Is Elementary	false
Name	
Short Name	FDAWithController
Traceable Specification	
Uuid	

(a) Graphical modelling of car-wiper's architecture

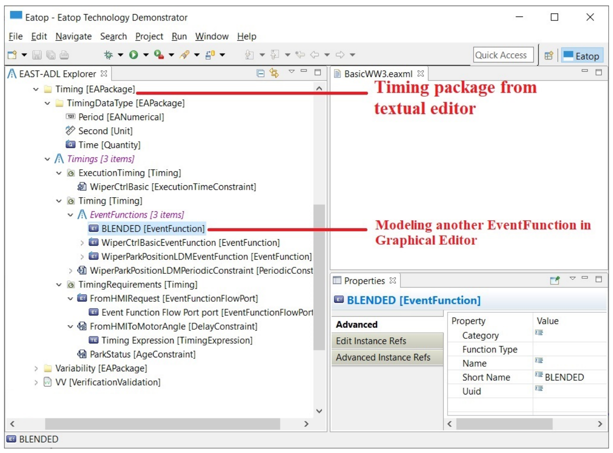


```
        }
    }
}
Define Element {
    Define Timing {
        UUID: "c17aaf45-ecd5-453e-bc20-eb9c9976db5e"
        SHORT-NAME: "Timing"
        CATEGORY: ""
        NAME: ""
        TRACEABLE-SPECIFICATION-REFS: ""
        Define Events {
            Define Event-Function {
                UUID: ""
                SHORT-NAME: "WiperParkPositionLDMEventFunction"
                CATEGORY: ""
                NAME: ""
                FUNCTION-TYPE: "BASIC-SOFTWARE-FUNCTION-TYPE"
                FUNCTION-VALUE: "/Core/Structure/DesignPkg/FcnDesignPkg/WiperParkPositionLDM"
                FFUNCTION-PROTOTYPE-TARGET: ""
                FUNCTION-PROTOTYPE-CONTEXT: ""
            }

            Define Event-Function {
                UUID: ""
                SHORT-NAME: "WiperCtrlBasicEve
                CATEGORY: ""
                NAME: ""
                FUNCTION-TYPE: "BASIC-SOFTWARE
                FUNCTION-VALUE: "/Core/Structu
                FFUNCTION-PROTOTYPE-TARGET: "/                                          roller/wi
                FUNCTION-PROTOTYPE-CONTEXT: ""
            }
        }

```

```
ignLevelWW/FDA"
ignPkg/FcnDesignArchitecture/FDAWithController/wiperCtrlBasic"
ignPkg/FcnDesignArchitecture/FDAWithController/wiperMotorBSW"
ignPkg/FcnDesignArchitecture/FDAWithController/wiperMotorHWF"
ignPkg/FcnDesignArchitecture/FDAWithController/wiperMotorLDM"
ignPkg/FcnDesignArchitecture/FDAWithController/wiperMotorRelayHWF"
ignPkg/FcnDesignArchitecture/FDAWithController/wiperParkPositionLDM"
ignPkg/FcnDesignArchitecture/FDAWithController/wiperVizualizationHWFLI
ignPkg/FcnDesignArchitecture/FDAWithoutController/wiperMotorBSW"
ignPkg/FcnDesignArchitecture/FDAWithoutController/wiperMotorHWF"
ignPkg/FcnDesignArchitecture/FDAWithoutController/wiperMotorLDM"
```

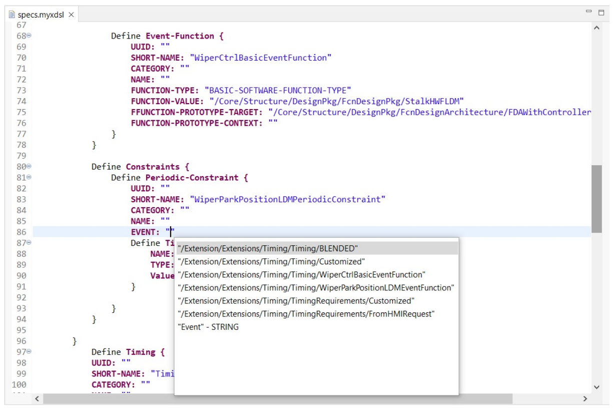
(b) Textual modelling of car-wiper's timing constraints

Figure 8. Blended modelling


Eatop - Eatop Technology Demonstrator
File Edit Navigate Search Project Run Window Help
Quick Access
Eatop
EAST-ADL Explorer
Timing [EAPackage]
TimingDataType [EAPackage]
Period [EANumerical]
Second [Unit]
Time [Quantity]
Timings [3 items]
ExecutionTiming [Timing]
WiperCtrlBasic [ExecutionTimeConstraint]
Timing [Timing]
EventFunctions [3 items]
BLENDED [EventFunction]
WiperCtrlBasicEventFunction [EventFunction]
WiperParkPositionLDMEventFunction [EventFunction]
WiperParkPositionLDMPeriodicConstraint [PeriodicConst
TimingRequirements [Timing]
FromHMIRequest [EventFunctionFlowPort]
Event Function Flow Port port [EventFunctionFlowPort
FromHMIToMotorAngle [DelayConstraint]
Timing Expression [TimingExpression]
ParkStatus [AgeConstraint]
Variability [EAPackage]
VV [VerificationValidation]
BasicWW3.eaxml
Timing package from textual editor
Modeling another EventFunction in Graphical Editor
Properties
BLENDED [EventFunction]
Advanced
Edit Instance Refs
Advanced Instance Refs
BLENDED

Property	Value
Category	
Function Type	
Name	
Short Name	BLENDED
Uuid	

(a) Real-time change propagation of timing package from textual to graphical editor



```

            Define Event-Function {
                UUID: ""
                SHORT-NAME: "WiperCtrlBasicEventFunction"
                CATEGORY: ""
                NAME: ""
                FUNCTION-TYPE: "BASIC-SOFTWARE-FUNCTION-TYPE"
                FUNCTION-VALUE: "/Core/Structure/DesignPkg/FcnDesignPkg/StalkHWFLDM"
                FFUNCTION-PROTOTYPE-TARGET: "/Core/Structure/DesignPkg/FcnDesignArchitecture/FDAWithController
                FUNCTION-PROTOTYPE-CONTEXT: ""
            }
        }

        Define Constraints {
            Define Periodic-Constraint {
                UUID: ""
                SHORT-NAME: "WiperParkPositionLDMPeriodicConstraint"
                CATEGORY: ""
                NAME: ""
                EVENT: ""
                Define Ti
                    NAME:
                    TYPE:
                    Value
                }

            }
        }

    }
        Define Timing {
        UUID: ""
        SHORT-NAME: "Timi
        CATEGORY: ""
```

```
"/Extension/Extensions/Timing/Timing/BLENDED"
"/Extension/Extensions/Timing/Timing/Customized"
"/Extension/Extensions/Timing/Timing/WiperCtrlBasicEventFunction"
"/Extension/Extensions/Timing/Timing/WiperParkPositionLDMEventFunction"
"/Extension/Extensions/Timing/TimingRequirements/Customized"
"/Extension/Extensions/Timing/TimingRequirements/FromHMIRequest"
"Event" - STRING
```

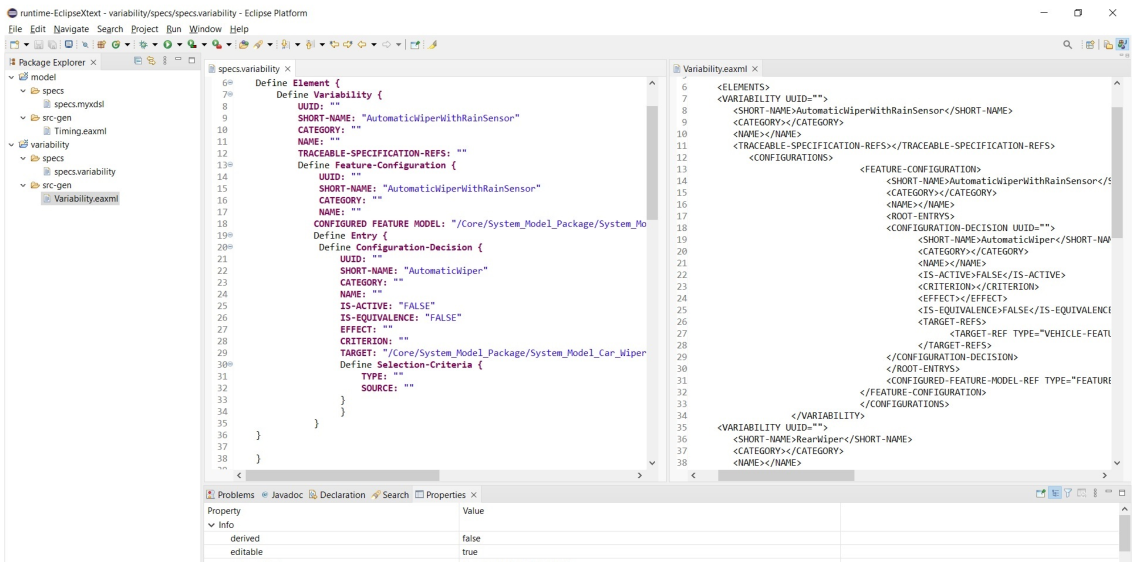
(b) Real-time "BLENDED" EventFunction change propagation from graphical to textual editor

Figure 9. Real-time synchronisation between editors

task. Furthermore, additional transformations need to be performed before propagating the changes to the textual editor. On the other hand, since the target EAXML is generated concurrently under the hood while modelling via our textual editors, related transformations are not required for change propagation from textual to graphical. Therefore, the change propagation time from textual to graphical editor is much lower. We are planning further investigations on larger use cases to realistically assess scalability and perform optimisations to the synchronisation mechanisms if needed.

Our approach encompasses the notations of the standard EAST-ADL meta-model, which is available in EMF [3]. In EATOP, which is EMF-based, the abstract syntax can be instantiated through both graphical and tree-based concrete syntaxes. We leverage EATOP's tree-based syntax for experimentation (and in the paper for brevity). However, the same models shown in tree-based syntax can be visualised diagrammatically using the Graphical Viewer plug-in [10]. Our approach does not affect the tree-based syntax, but the abstract syntax behind it, and thereby the same model changes can be appreciated in all concrete syntaxes provided out-of-the-box by the hosting tool, in this case, EATOP.

Blended modelling is based on runtime synchronisation between notations. In the proposed approach, we provide on-the-fly synchronisation, that is applied when models are saved in either notation. Note that this can be simply configured so that changes are propagated in real-time (upon editing, rather than saving). This was not a viable option in our industrial settings for reasons related to the editing rights of different stakeholders who may be working collaboratively on the same models.



```
    Define Element {
        Define Variability {
            UUID: ""
            SHORT-NAME: "AutomaticWiperWithRainSensor"
            CATEGORY: ""
            NAME: ""
            TRACEABLE-SPECIFICATION-REFS: ""
            Define Feature-Configuration {
                UUID: ""
                SHORT-NAME: "AutomaticWiperWithRainSensor"
                CATEGORY: ""
                NAME: ""
               CONFIGURED FEATURE MODEL: "/Core/System_Model_Package/System_Mo
               Define Entry {
                Define Configuration-Decision {
                    UUID: ""
                    SHORT-NAME: "AutomaticWiper"
                    CATEGORY: ""
                    NAME: ""
                    IS-ACTIVE: "FALSE"
                    IS-EQUIVALENCE: "FALSE"
                    EFFECT: ""
                    CRITERION: ""
                    TARGET: "/Core/System_Model_Package/System_Model_Car_Wiper
                    Define Selection-Criteria {
                        TYPE: ""
                        SOURCE: ""
                    }
                    }
               }
    }

    }
```

```
    <ELEMENTS>
    <VARIABILITY UUID="">
       <SHORT-NAME>AutomaticWiperWithRainSensor</SHORT-NAME>
       <CATEGORY></CATEGORY>
       <NAME></NAME>
       <TRACEABLE-SPECIFICATION-REFS></TRACEABLE-SPECIFICATION-REFS>
          <CONFIGURATIONS>
                           <FEATURE-CONFIGURATION>
                                <SHORT-NAME>AutomaticWiperWithRainSensor</S
                                <CATEGORY></CATEGORY>
                                <NAME></NAME>
                                <ROOT-ENTRYS>
                                <CONFIGURATION-DECISION UUID="">
                                      <SHORT-NAME>AutomaticWiper</SHORT-NAM
                                      <CATEGORY></CATEGORY>
                                      <NAME></NAME>
                                      <IS-ACTIVE>FALSE</IS-ACTIVE>
                                      <CRITERION></CRITERION>
                                      <EFFECT></EFFECT>
                                      <IS-EQUIVALENCE>FALSE</IS-EQUIVALENCE
                                      <TARGET-REFS>
                                            <TARGET-REF TYPE="VEHICLE-FEATU
                                      </TARGET-REFS>
                                </CONFIGURATION-DECISION>
                                </ROOT-ENTRYS>
                                <CONFIGURED-FEATURE-MODEL-REF TYPE="FEATURE
                           </FEATURE-CONFIGURATION>
                           </CONFIGURATIONS>
                    </VARIABILITY>
    <VARIABILITY UUID="">
       <SHORT-NAME>RearWiper</SHORT-NAME>
       <CATEGORY></CATEGORY>
       <NAME></NAME>
```

Property	Value
Info	
derived	false
editable	true

Figure 10. Textual modelling of variability with synchronisation

The proposed approach operates on two resources (i.e., Xtext and EAXML) to achieve runtime synchronisation between notations. Therefore, the implementation is accomplished through model transformations. This is a common solution in state-of-the-art blended modelling approaches, such as Latifaj et al.[25], which operates on two resources (Xtext and UML). An exception is the approach by Addazi et al. [22], where synchronisation between notations is achieved through a single resource (i.e., UML). We defined our textual notations according to the needs of our industrial partner, as in related approaches [23]. However, ours is the first documented approach towards effectively providing blended modelling for EAST-ADL via EAXML. This sets a solid platform to bring the benefits of blended modelling into the development of vehicular embedded systems.

One of the core requirements of this work was applicability. To maximise applicability, we selected EAXML as the synchronisation pivot for change propagation between editors. EAXML format is expected to be supported by the majority of EAST-ADL tools, and therefore, the proposed blended modelling approach can easily be included in toolsets supporting EAXML. In this work, we tested the proposed approach in EATOP, but we are planning to evaluate it with other EAST-ADL tools too. In this context, it is pertinent to mention that an updated EAXML schema is also released along with every new version of the EAST-ADL specification. In this work, we used EAXML schema version 2.1.12 [9] for our proof-of-concept implementation. Consequently, integration of the current solutions with EAST-ADL tools supporting version 2.1.12 is straightforward; for coming versions, a few adjustments may be required in the synchronisation mechanisms depending on how the EAST-ADL specification would evolve.

6 Conclusion and future work

This article presents a novel approach to the provision of blended modelling in terms of textual and graphical concrete syntaxes for EAST-ADL. Notably, for selected portions of the EAST-ADL language (i.e., timing and variability packages), we introduced ad-hoc textual concrete syntaxes to represent the language's abstract syntax using textual notations, while preserving the language's semantics. Furthermore, the approach offers runtime synchronisation between the notations using EAXML – a standard model exchange format for EAST-ADL tools – as a pivot language. The feasibility of the proposed approach was shown through a car wiper use case from our industrial partner - Volvo. In particular, textual concrete syntaxes representing EAST-ADL's timing and variability packages were implemented and the EATOP tool was used for graphical modelling. The car wiper use case was modelled through both textual and graphical concrete syntaxes simultaneously. This significantly boosts collaboration and lowers development time, eventually improving the overall productivity of vehicular embedded systems.

Important aspects like the scalability of the synchronisation mechanisms need further investigation. We are currently performing a more in-depth assessment of the proposed approach on larger use cases. Furthermore, we will extend blended modelling to other EAST-ADL packages at different levels of abstraction. This will provide room for further enhancements of the current approach. Lastly, we are in the process of assessing the applicability of our approach by leveraging it on other EAST-ADL tools.

Acknowledgments

This research work was supported by the ITEA3/Vinnova BUMBLE (rn. 18006), the KKS MoDev (rf. 20200234), and the VR ORPHEUS (rf. 2022-03408) projects.

References

[1] Kaijser H. Lönn H. Papadopoulos Y. Reiser M. Kolagari R. T. Tucci S Blom H. Chen, D. East-adl: An architecture description language for automotive software-intensive systems in the light of recent use and research. *International Journal of System Dynamics Applications (IJSDA)*, 5(3):1–20, 2016.

[2] Aon Safdar, Farooque Azam, Muhammad Waseem Anwar, Usman Akram, and Yawar Rasheed. Modlf: A model-driven deep learning framework for autonomous vehicle perception (avp). In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, MODELS '22, page 187–198, New York, NY, USA, 2022. Association for Computing Machinery.

[3] Jörg Holtmann, Jan-Philipp Steghöfer, and Henrik Lönn. Migrating from proprietary tools to open-source software for east-adl metamodel generation and evolution. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '22, page 7–11, New York, NY, USA, 2022. Association for Computing Machinery.

[4] F. Ciccozzi, M. Tichy, H. Vangheluwe, and D. Weyns. Blended modelling - what, why and how. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 425–430, Los Alamitos, CA, USA, sep 2019. IEEE Computer Society.

[5] Ali Shahrokni Jan Söderberg and Bashar Nassar. Using xpath to define design metrics. In *The Sixth International Conference on Performance, Safety and Robustness in Complex Systems and Applications*, PESARO, page 13–17, port, 2016. IARIA.

[6] Eclipse EATOP. East-adl toolchains. https://projects.eclipse.org/projects/modeling.eatop, 2020. Accessed: (May 2023).

[7] Alessio Bucaioni, Antonio Cicchetti, and Mikael Sjödin. Towards a metamodel for the rubus component model., 2014.

[8] Eclipse Xtext. Platform for language engineering. https://www.eclipse.org/Xtext/, 2022. Accessed: (June 2022).

[9] EAST-ADL association. East-adl specifications. https://www.east-adl.info/Specification.html, 2020. Accessed: (July 2023).

[10] Synligare Consortium. Synligare project tooling. http://synligare.eu/Tooling.html, 2020. Accessed: (March 2023).

[11] Christoph Etzel and Bernhard Bauer. Modeling and analysis of partitions on functional architectures using east-adl. In Slimane Hammoudi, Luís Ferreira Pires, and Bran Selić, editors, *Model-Driven Engineering and Software Development*, pages 298–319, Cham, 2020. Springer International Publishing.

[12] Markus Zoppelt and Ramin Tavakoli Kolagari. Sam: A security abstraction model for automotive software systems. In Brahim Hamid, Barbara Gallina, Asaf Shabtai, Yuval Elovici, and Joaquin Garcia-Alfaro, editors, *Security and Safety Interplay of Intelligent Software Systems*, pages 59–74, Cham, 2019. Springer International Publishing.

[13] Tobias Wägemann, Ramin Tavakoli Kolagari, and Klaus Schmid. Exploring automotive stakeholder requirements for architecture optimization support. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 37–44, 2019.

[14] Martin Walker, Mark-Oliver Reiser, Sara Tucci-Piergiovanni, Yiannis Papadopoulos, Henrik Lönn, Chokri Mraidha, David Parker, DeJiu Chen, and David Servat. Automatic optimisation of system architectures using east-adl. *Journal of Systems and Software*, 86(10):2467–2487, 2013.

[15] Eun-Young Kang, Dongrui Mu, Li Huang, and Qianqing Lan. Model-based analysis of timing and energy constraints in an autonomous vehicle system. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 525–532, 2017.

[16] Eduard Paul Enoiu and Cristina Seceleanu. Model testing of complex embedded systems using east-adl and energy-aware mutations. *Designs*, 4(1), 2020.

[17] Raluca Marinescu, Mehrdad Saadatmand, Alessio Bucaioni, Cristina Seceleanu, and Paul Pettersson. A model-based testing framework for automotive embedded systems. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 38–47, 2014.

[18] Alessio Bucaioni, Lorenzo Addazi, Antonio Cicchetti, Federico Ciccozzi, Romina Eramo, Saad Mubeen, and Mikael Sjödin. Moves: A model-driven methodology for vehicular embedded systems. *IEEE Access*, 6:6424–6445, 2018.

[19] Remko van Wagensveld, Tobias Wägemann, Ralph Mader, Ramin Tavakoli Kolagari, and Ulrich Margull. Evaluation and modeling of the supercore parallelization pattern in automotive real-time systems. *Parallel Computing*, 81:122–130, 2019.

[20] Antonio Cicchetti, Federico Ciccozzi, and Alfonso Pierantonio. Multi-view approaches for software and system modelling: A systematic literature review. *Softw. Syst. Model.*, 18(6):3207–3233, dec 2019.

[21] David Emery and Rich Hilliard. Every architecture description needs a framework: Expressing architecture frameworks using iso/iec 42010. In *2009 Joint Working IEEE/IFIP Conference on Software Architecture European Conference on Software Architecture*, pages 31–40, 2009.

[22] Lorenzo Addazi and Federico Ciccozzi. Blended graphical and textual modelling for uml profiles: A proof-of-concept implementation and experiment. *Journal of Systems and Software*, 175:110912, 2021.

[23] Malvina Latifaj, Federico Ciccozzi, Muhammad Waseem Anwar, and Mattias Mohlin. Blended graphical and textual modelling of uml-rt state-machines: An industrial experience. In Patrizia Scandurra, Matthias Galster, Raffaela Mirandola, and Danny Weyns, editors, *Software Architecture*, pages 22–44, Cham, 2022. Springer International Publishing.

[24] Eclipse HCL. Rtist environment. https://marketplace.eclipse.org/content/hcl-rtist, 2023. Accessed: (June 2023).

[25] Malvina Latifaj, Federico Ciccozzi, and Mattias Mohlin. Higher-order transformations for the generation of synchronization infrastructures in blended modeling. *Frontiers in Computer Science*, 4, 2023.

[26] Salome Maro, Jan-Philipp Steghöfer, Anthony Anjorin, Matthias Tichy, and Lars Gelin. On integrating graphical and textual editors for a uml profile based domain specific language: An industrial experience. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2015, page 1–12, New York, NY, USA, 2015. Association for Computing Machinery.

[27] Muhammad Waseem Anwar, Malvina Latifaj, and Federico Ciccozzi. Blended modeling applied to the portable test and stimulus standard. In Shahram Latifi, editor, *ITNG 2022 19th International Conference on Information Technology-New Generations*, pages 39–46, Cham, 2022. Springer International Publishing.

[28] Muhammad Waseem Anwar and Federico Ciccozzi. Blended metamodeling for seamless development of domain-specific modeling languages across multiple workbenches. In *2022 IEEE International Systems Conference (SysCon)*, pages 1–7, 2022.

[29] Marie-Agnès Peraldi-Frati, Arda Goknil, Julien DeAntoni, and Johan Nordlander. A timing model for specifying multi clock automotive systems: The timing augmented description language v2. In *2012 IEEE 17th International Conference on Engineering of Complex Computer Systems*, pages 230–239, 2012.

[30] EAST-ADL Blended Modelling Environment. Mälardalen university sweden. https://github.com/MDH-BUMBLE/EASTADL-Blended, 2023. Accessed: (September 2023).

Received 2023-07-07; accepted 2023-09-01

Towards Efficient Model Comparison using Automated Program Rewriting

Qurat ul ain Ali
quratulain.ali@york.ac.uk
University of York
UK

Dimitris Kolovos
dimitris.kolovos@york.ac.uk
University of York
UK

Konstantinos Barmpis
konstantinos.barmpis@york.ac.uk
University of York
UK

Abstract

Model comparison is a prerequisite task for several other model management tasks such as model merging, model differencing etc. We present a novel approach to efficiently compare models using programs written in a rule-based model comparison language. As the comparison is done at the model element level, and each element needs to be traversed and compared with its corresponding elements, the execution of these comparison algorithms can be computationally expensive for larger models. In this paper, we present an efficient comparison approach which provides an automated rewriting facility to compare (both homogeneous and heterogeneous) models, based on static program analysis. Using this analysis, we reduce the search space by pre-filtering/indexing model elements, before actually comparing them. Moreover, we reorder the comparison match rules according to the dependencies between these rules to reduce the cost of jumping between rules. Our experiments demonstrate that the proposed model comparison approach delivers significant performance benefits in terms of execution time compared to the default ECL execution engine.

***CCS Concepts:* • Software and its engineering → Domain specific languages**.

Keywords: Model-Driven Engineering, Scalability, Model Comparison, Static Analysis, Program Analysis

ACM Reference Format:
Qurat ul ain Ali, Dimitris Kolovos, and Konstantinos Barmpis. 2023. Towards Efficient Model Comparison using Automated Program Rewriting. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3623476.3623519

1 Introduction

While there is increased adoption of Model-Driven Engineering (MDE) principles, tools and technologies in industry [10], going forward scalability of these tools remains one of the key challenges [9]. To enable the use of MDE in large-scale applications it is essential to make MDE tools and technologies scalable. Model management languages are often interpreted and hence slower compared to general-purpose programming languages [22]. So, optimising these model management languages can deliver performance benefits on the top of already provided underlying dedicated task-specific support.

Model comparison is usually a prerequisite to various other key model management activities such as model differencing, model versioning, etc. It involves establishing matches/correspondences between elements of two models. There are different possible ways to compare models, such as traditional text-based comparison, comparison based on unique identifiers, model-to-model (M2M) transformation to establish comparison as in [11], or to use a dedicated comparison language, such as the Epsilon Comparison Language (ECL), which supports specifying matching criteria. Such model comparison can be computationally very expensive because each element of the first model needs to be traversed and compared to a corresponding element of the second model, which does not scale well.

In this paper, we introduce an efficient model comparison approach based on static program analysis and automated program rewriting. We have developed a prototype implementation of the proposed approach that can rewrite ECL programs, which operate on models with Ecore-based metamodels. According to the current ECL engine, all elements of one type are compared against the elements of their matching type based on the provided comparison logic by the developer. Using program analysis, we pre-filter the elements to be compared, index them and then compare the pre-filtered instances rather than all instances. These pre-filtered elements are automatically embedded into the original ECL program, using program rewriting, and executed using the traditional ECL engine. Also, we ensure that all the rules that are needed for the execution of a particular rule have already been executed, by reordering the rules, to avoid extra overhead of finding the appropriate rule to invoke. The output of an ECL program is a match trace that contains all the established results of matches between elements of two models. Our approach yields a reduced match trace, by omitting any unsuccessful matches, whenever it was possible to identify these beforehand through program analysis.

Our proposed approach has shown performance gains up to 95% in terms of execution time in the experiments we have conducted.

The rest of the paper is structured as follows: Section 2 presents the background concepts, tools and technologies used for the implementation of the proposed approach, followed by a running example. Section 3, presents the overview of the proposed comparison optimisation approach and then discusses each component step-by-step. Experiments, case studies and the obtained results are presented and analysed in Section 4. Section 5 discusses the relevant state-of-the-art in the field of model comparison optimisation and static analysis. Finally, Section 6 concludes the paper and presents direction for future work.

2 Background & Motivation

This section provides the background concepts and a brief overview of the technologies used to implement the proposed approach. It also presents a running example which will be used to motivate this work.

2.1 Model Comparison

Model comparison is one of the fundamental model management tasks, usually a prerequisite for other tasks such as versioning, model merging, model differencing and model transformation testing. Model comparison establishes correspondences between matching elements of two models[14]. Such comparison can be performed both on homogeneous and heterogeneous models. One example scenario could be to identify matching elements before merging two models. Such correspondences can also be used to test model-to-model transformation pairs (source and corresponding target elements). Moreover, model comparison can be used in order to establish matching elements before calculating the differences between two models.

2.2 Epsilon

Epsilon [4] is a family of task-specific languages for performing several model management tasks, such as model merging (Epsilon Merging Language - EML [17]), model validation (Epsilon Validation Language - EVL [1]), model-to-model transformation (Epsilon Transformation Language - ETL [18]) and pattern matching (Epsilon Pattern Language - EPL [15]). All these languages extend a core language, the Epsilon Object Language (EOL) [16], which provides imperative constructs such as loops, conditionals and operations (both built-in and user-defined). All languages of Epsilon support managing models from a number of modeling technologies (and their respective persistence formats), through a uniform interface, the Epsilon Model Connectivity (EMC) layer [5].

The reason for choosing Epsilon as the basis of this work is twofold. Firstly, Epsilon provides a dedicated language for model comparison. Secondly, the developed optimisation facilities can be leveraged by a wide range of modelling technologies, as Epsilon supports languages like EMF, Simulink and XML, and can be further extended to work with currently unsupported technologies using its EMC layer.

2.3 Epsilon Comparison Language

The Epsilon Comparison Language (ECL)[1] is a hybrid rule-based dedicated model comparison language, provided by the Epsilon framework. ECL lets developers specify custom comparison algorithms in a rule-based script to identify matching elements between homogeneous and heterogeneous models. An ECL program contains a number of *MatchRules* and optional pre and post-block(s) executing before and after the rules respectively. A *MatchRule* enables developers to specify comparison logic between model elements at a high level of abstraction. *MatchRules* consist of a declared name along with two parameters (left and right) to specify the types of elements they can compare. A *MatchRule* can also optionally extend a number of match rules and can be labelled as *abstract*, *lazy* and/or *greedy* using corresponding annotations.

- An **abstract match rule** must be extended by other *MatchRule*s. Abstract match rules cannot be invoked standalone, they get invoked only when the rules that extend them are invoked.
- A **lazy match rule** will get executed only when it is required by another *MatchRule*, using the *matches* operation.
- A **greedy match rule** is executed for *all* pairs that have a kind-of relationship with the types specified by the left and the right parameters of the *MatchRule*.

The execution engine automatically evaluates non-abstract, non-lazy match-rules in two passes, starting with the order in which they appear.

2.4 Motivating Example

In this paper, as a running example, we consider comparing class diagrams with sequence diagrams. Figure 1 is illustrates the metamodel of a class diagram language. The class diagram shows a structural view of the system containing the classes, their attributes and their operations.

Then we consider a metamodel of a sequence diagram, an excerpt of which is shown in Figure 2. Sequence diagrams show the interaction between objects of a system - its intended behaviour.

[1]https://www.eclipse.org/epsilon/doc/ecl/

```
model Left driver EMF{
nsuri = "sd"
};

model Right driver EMF{
nsuri = "cd"
};

rule Lifeline2Class
  match l : Left!Lifeline
  with r : Right!Class {
  compare : l.type = r.name
}

rule Message2Operation
  match l : Left!Message
  with r : Right!Operation {

  compare : l.`operation` = r.name
    and (l.`to`.matches(r.class) or l.`to`.
        matches(r.class.superTypes)) and l.
        parameters.matches(r.parameters)
}

rule Param2Param
  match l: Left!Parameter
  with r: Right!Parameter {

  compare : l.name = r.name and l.type = r.
      type.name
}

operation String matchOperation(others :
    Collection<Right!Operation>) : Boolean
    {
   return others.exists(o|o.name = self);
}
```

Listing 1. Example ECL script before optimisation

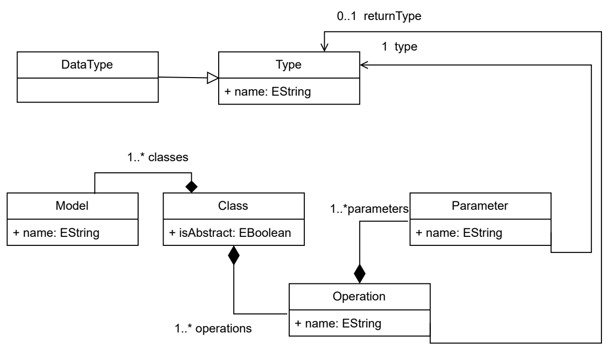


Figure 1. An excerpt of the Class Diagram metamodel

Now, as a sequence diagram depicts the interaction between objects and a class diagram represents the classes and their features, we can establish correspondences between the two, which can be used for downstream activities such as validation, model merging etc.

```
model Left driver EMF {
nsuri = "sd"
};

model Right driver EMF {
nsuri = "cd"
};

pre {
  var Lifeline2ClassMap = Right!Class.all.
      mapBy(param|param.name);
  var Message2OperationMap = Right!
      Operation.all.mapBy(param|param.name)
      ;
  var Param2ParamMap = Right!Parameter.all.
      mapBy(param|param.name);
}

rule Lifeline2Class
    match l : Left!Lifeline
    with r : Right!Class
    from : Lifeline2ClassMap.get(l.type) ?:
        Sequence{}{
   compare : true
}

rule Param2Param
    match l : Left!Parameter
    with r : Right!Parameter
    from : Param2ParamMap.get(l.name) ?:
       Sequence{}{
     compare : true and l.type = r.type.
         name
}

rule Message2Operation
    match l : Left!Message
    with r : Right!Operation
    from : Message2OperationMap.get(l.`
       operation`) ?: Sequence{} {
     compare : true and (l.`to`.matches(r.
         class) or l.`to`.matches(r.class.
         superTypes)) and l.parameters.
         matches(r.parameters)
}

operation String matchOperation(others :
    Collection(Right!Operation)) : Boolean
    {
  return others.exists(o : Right!Operation|
      o.name = self);
}
```

Listing 2. Example ECL script after optimisation

Table 1. Match trace produced from the execution of Listing 1 on the models in Figure 3 and Figure 4

S #	Left	Right	Matching
1	Lifeline (qa: User)	Class (User)	True
2	Lifeline (qa: User)	Class (ATM)	False
3	Lifeline (qa: User)	Class (Card)	False
4	Lifeline (hsbc: ATM)	Class (User)	False
5	Lifeline (hsbc: ATM)	Class (ATM)	True
6	Lifeline (hsbc: ATM)	Class (Card)	False
7	Message (enterPin)	Operation (verifyPin)	False
8	Message (enterPin)	Operation (dispenseCash)	False
9	Message (enterPin)	Operation (enterPin)	True
10	Message (enterPin)	Operation (depositCash)	False
11	Message (enterPin)	Operation (withdrawCash)	False
12	Message (enterPin)	Operation (activate)	False
13	Message (verifyPin)	Operation (verifyPin)	True
14	Message (verifyPin)	Operation (dispenseCash)	False
15	Message (verifyPin)	Operation (enterPin)	False
16	Message (verifyPin)	Operation (depositCash)	False
17	Message (verifyPin)	Operation (withdrawCash)	False
18	Message (verifyPin)	Operation (activate)	False

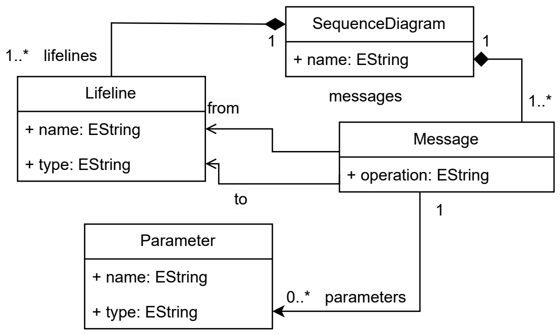


Figure 2. An excerpt of the Sequence Diagram metamodel

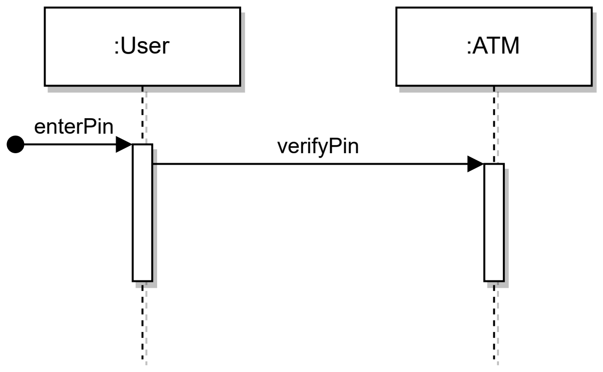


Figure 3. Sequence Diagram of ATM

A custom comparison algorithm written in ECL is shown in Listing 1. For this comparison, we have the following basic criteria:

- A lifeline matches a class when the type of the lifeline is the same as the name of the class in class diagram.
- A message matches an operation when the operation of the message is the same as the name of the operation. Also, the class corresponding to the “to" lifeline of the message or one of its supertypes should contain the operation.
- The parameters of the message need to be matched with the parameters of the operation.

We discuss some builtin operations supported by ECL and EOL that are used in the running example.

EOL supports a safe navigation operator ?., for making the null checks more concise. The use of the safe navigation operator is shown in the listing below, where we return *someProperty* if the *a* has a non-null value and returns *anotherProperty* if *a* contains a null value. `var result = a?.someProperty?.anotherProperty;`

Map (*mapBy(iterator : Type | expression)*) is a function that returns a map containing the results of the expression as keys and the respective items of the collection or collections of elements as values.

ECL provides a built-in operation *matches(right :Any)* for model elements and collections. When invoked, the *matches()* operation returns the cached result, if the elements have been already matched, otherwise, it finds rules that can compare the elements, executes them, and returns the result. In this ECL program, we have three match rules: *Lifeline2Class* (Line 9-13), which compares the type of lifeline to the class name (Line 12), *Message2Operation*, which compares the operation of *Message* with *Operation* name (Line 20). *Message2Operation* also compares whether the operation's owner class is the same as the *to* (Lifeline) of the *Message*. Finally, *Param2Param* compares the name and type of the parameters of both models (Line 27).

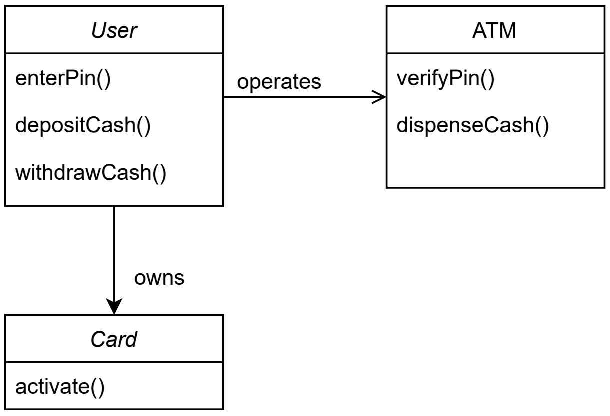


Figure 4. Class Diagram of ATM

As an example, let us consider matching a class diagram of an ATM system as shown in Figure 4) with its corresponding sequence diagram as shown in Figure 3. If we execute the ECL program (Listing 1) over these two models it would produce the match trace shown in Table 1. As we can see, it returns all matches of each element with its corresponding type and a boolean indicating if the element were matched or not.

The default execution engine of ECL will compare each instance of the left parameter (i.e., *Lifeline*) to all the instances of the right parameter (i.e., *Class*). The complexity of this rule here would be *O(M× N)*, if there are *M* number of *Lifelines* and *N* number of *Class*es. Using program analysis, we could index the instances by analysing these compare blocks as shown in Listing 2. Considering example sequence and class diagrams in Figure 3 and Figure 4, as there are 2 Lifelines and 3 Classes so there will be 6 matches for the rule *Lifeline2Class*. In the rule *Lifeline2Class*, we can filter the *Class* instances only keeping ones where the name of the class is equal to the *type* of the *Lifeline*. These indices can be pre-computed once, and then used as required. This could reduce the complexity to *O(M)*, considering the complexity of the hash function to be *O(1)*. Again considering the example models, if we execute Listing 2, the resultant match trace would be the same as shown in Table 2. This can be observed that there are only two matches for the same *Lifeline2Class* rule. Hence, the idea of this work is to analyse the ECL matching program and to automatically replace it with an efficiently rewritten program, to reduce the complexity of (some of) the comparisons.

3 Proposed Approach

In this section, we present proposed approach, an overview of which is illustrated in Figure 5. The idea is to optimise ECL matching programs automatically using program analysis. The developer writes the comparison algorithm in ECL to compare two models, say left and right. The expected outcome is a match trace resulting from computing the compare block of each match rule. The match trace contains a number of matches, each match contains the two objects that were matched and a boolean to indicate if the match was successful or not. So using the proposed optimisation approach we generate a match trace, which is a reduced or pre-filtered version, containing a significantly smaller number of unsuccessful matches. We have hence reduced the search space, making the comparison faster. This is because we do not compare all instances of left parameter to all instances of right parameter (which is done in existing ECL execution), rather we compare instances of left parameter to pre-filtered/pre-indexed instances of the right parameter.

The first step in our proposed approach is the (1) static analyser, a block used to populate the Abstract Syntax Tree (AST) of the ECL matching program, with the respective type information. This type resolved AST is then used for two purposes: i) For the (2) dependency graph extractor, a block that extracts the dependencies between different match rules of an ECL program by analysing compare blocks and *matches()* operations. Dependency here means that if a rule *MRx* invokes another rule *MRy* then the rule *MRx* would be dependent on *MRy*. The dependency graph is then used by the (4) rule scheduler to efficiently reorder the execution of rules. So that if a rule invokes another rule like in Line 20 of Listing 1, rule *Message2Operation* is dependent on rule *Lifeline2Class* and *Param2Param*. Both the rules on which *Message2Operation* is dependent should be executed before the execution of *Message2Operation*. ii) For the (3) optimisable rule detector, a block for program analysis to

Table 2. Match trace produced from the execution of Listing 2 on the models in Figure 3 and Figure 4

S #	Left	Right	Matching
1	Lifeline (qa: User)	Class (User)	True
2	Lifeline (hsbc: ATM)	Class (ATM)	True
3	Message (enterPin)	Operation (enterPin)	True
4	Message (verifyPin)	Operation (verifyPin)	True

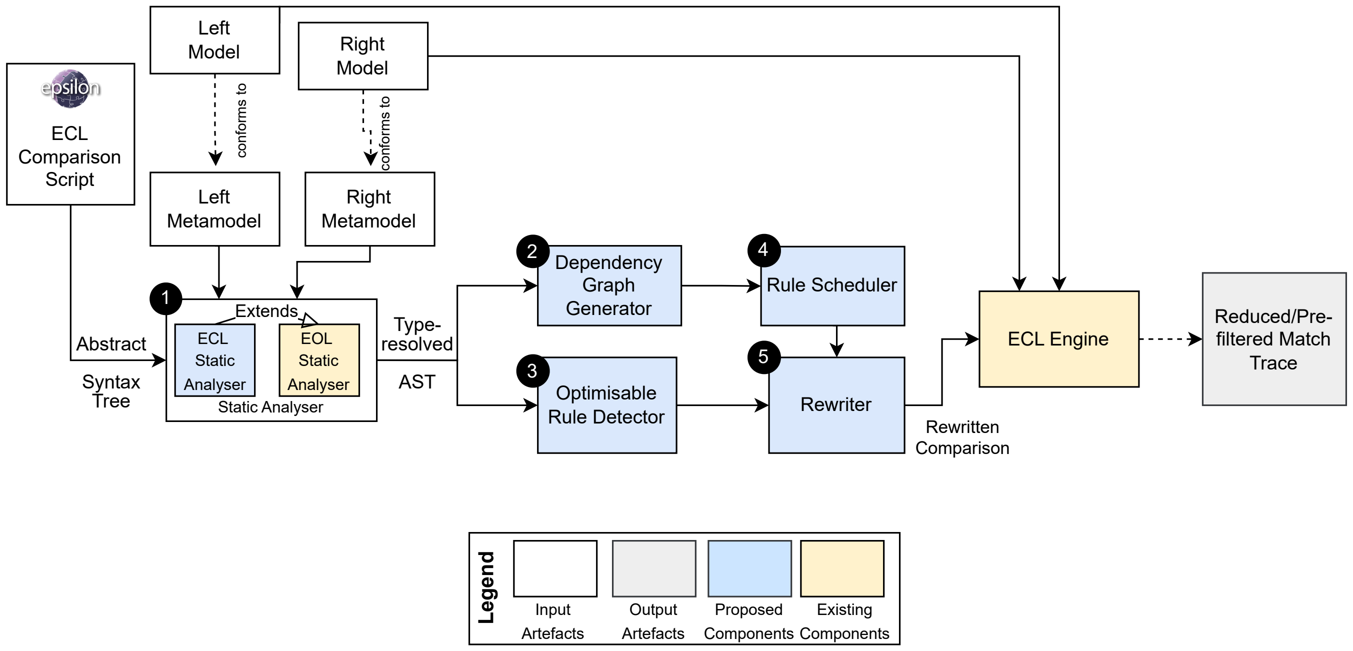


Figure 5. An overview of the proposed approach

identify the match rules which can be optimised based on the expressions in compare block. Here, optimisable rules mean the rules which are matching two elements on the basis of a specific property and can be indexed. So, this step identifies optimisable match rules along with the specific property name. Finally, (5) the rewriter block will replace the original program with a rewritten optimised program along with the new order of the match rules. This optimised comparison program will then be executed by the existing ECL engine. The resultant match trace would be a subset of the trace that would have been produced by the original comparison program. This subset trace would exclude the matches which would not satisfy the domain (an EOL expression to narrow the search space), while including all positive matches.

3.1 Static Analysis

Static analysis is the first step of our proposed approach workflow. It analyses the ECL program's abstract syntax tree (AST) and computes the types of all expressions in it. This type information is extracted using metamodel introspection, type resolution and type inference. *ModelDeclarationStatements* in Lines (1-3 and 5-7) in Listing 1 actually access the metamodel structure and help retrieve the types and their hierarchy available in the metamodel. To statically analyse ECL programs, we extended the already available EOL static analyser [2] by adding language specific support (e.g. analysing *MatchRules*, compare blocks etc.). The resolved types of various constructs in Listing 1 are shown in Table 3. The outcome of the static analyser block is a type-resolved AST, which is just the input AST with its nodes populated with their respective types.

3.2 Dependency Graph

When *matches()* operation is invoked, it returns the cached result, if the elements have been already matched. Otherwise, it finds the rule comparing the same two elements and

[2]https://github.com/epsilonlabs/static-analysis

Table 3. Resolved types of various constructs in Listing 1

Line#	Expression	Resolved Type
13	l	Left!Lifeline
14	r	Right!Class
15	l.type	String
15	r.name	String
19	l	Left!Message
20	r	Right!Operation
21	l.operation	String
21	r.operations	Collection<Right!Operation>
21	r.superTypes.operations	Collection<Right!Operation>
29	l.parameters	Collection<Left!Paramter>
29	r.parameters	Collection<Right!Paramter>
33	l	Left!Parameter
34	r	Right!Parameter
36	l.name	String
36	r.type.name	String

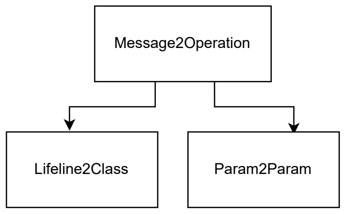


Figure 6. Dependency graph of Listing 1

then returns its results. Due to these rule invocations, rules can be dependent on one another. These dependencies can be extracted by the help of the type resolved AST, as done for model-to-model transformations in [8]. To construct a dependency graph, we create a vertex for each *MatchRule* declared in the ECL program. If a rule *MRx* has a statement in its compare block that calls a *matches()* operation which invokes another rule, say *MRy*. The resolution of which rule is invoked by the *matches()* operation is done by finding the rule where the type of the left and right parameters of rule is the same as the type of the target and parameter expressions of the *matches()* operation. Then, we create an edge from the vertex corresponding to *MRx*, to the vertex corresponding to *MRy*. If there are multiple rules invoked by the *matches()* operation, we create multiple edges from *MRx*. For example, as in rule *Message2Operation* Line 19 of Listing 1 there is a call to the *matches()* operation with *l.to* (resolved type: *Lifeline*) as the target expression and *r.class* (resolved type: *Class*). This means that this *matches* operation call will invoke a rule which is matching *Lifeline* with *Class* i.e., rule *Lifeline2Class*. So, we create an edge from *Message2Operation* to *Lifeline2Class* as shown in Figure 6. The reason for extracting the dependency graph is to reorder the rules in a way that if *MRx* is invoked by a rule *MRy* then *MRy* is scheduled before *MRx*. When *x.matches(y)* is called in ECL, if x and y have not already been matched, the ECL engine needs to find rule(s) that can match them, invoke these rules and return the result to *matches(...)*. This can have a non-negligible cost for large models and sets of match rules. By reordering rules to maximise the number of pairs of x and y that have been already matched before *x.matches(y)* is called, we reduce that cost of *jumping* between rules. This rescheduling can help improve performance, because it can reduce the number of attempts needed to find the appropriate rules to invoke. Any rule invocation using a *matches()* operation can use the cached results in the match trace, if the rules have been reordered properly. We do not create an edge when a rule invokes itself, as it does not affect the reordering for which we extract dependency graph. However, ECL provides a mechanism to avoid an infinite loop, in case of a cyclic invocation of a rule i.e., two rules implicitly invoking each other. ECL maintains a temporary trace along with the primary trace. In a primary trace the matching value is added after the execution of compare block, while the matching value is set to true in the temporary trace before the execution of the compare block. In case of another attempt to match elements from already invoked rules, these rules would not be re-invoked. Finally, the temporary trace is reset when a top-level rule returns.

3.3 Identifying Optimisable *MatchRules*

This is the third step of the approach that takes in a type-resolved AST as an input with the aim to identify the rules which can be optimised. By optimisable rules, we mean the rules which are comparing the elements of the two models based on a specific property. This is done by traversing the compare block of each *MatchRule* and finding expressions where two elements are compared on the basis of a specific property or attribute. Currently, the rewriting approach only considers equality operators, as checking for name/id-like attribute equality is very common in model matching in our experience, but it can be extended to support other operators in the future too. In this case, the elements can be indexed based on that property. The process for identifying such optimisable rules is specified in Algorithm 1. The algorithm traverses a set of Match rules and its compare block. Then, in a compare block all DOM elements are traversed to identify cases where a *PropertyCallExpression* is used within an *EqualsOperatorExpression* and it records the relevant Match rules and properties in a HashMap for later use in indexing. With one exception, if there is a logical operator between equals expression, we just record the index if it is an and operator. For instance, in Listing 1 Line 12 the rule *Lifeline2Class* is comparing the *Lifeline* from Sequence diagram to *Class* from Class diagram on the basis of the property *name*, in this class. So the Algorithm 1, would return the hashmap containing rule *Lifeline2Class* with the respective property "*name*".

Algorithm 1 Algorithm for Identifying optimisable rules

```
1: Let $op$ = HashMap<rule, NameExpression>
2: for all Matchrules $rule$ do
3:   Visit all DOM elements ($elem$) of compare block of $rule$
4:   if $elem$ instanceof PropertyCallExpression and !($op$.contain($rule$)) then
5:     $parent \leftarrow elem$.parent
6:     if $parent$ instanceof EqualsOperatorExpression then
7:       $parent \leftarrow parent$.parent
8:       if $parent$ instanceof OperatorExpression then
9:         if $parent$ instanceof AndOperatorExpression then
10:          op $\leftarrow rule$ and elem.NameExpression
11:        end if
12:      else
13:        op $\leftarrow rule$ and elem.NameExpression
14:      end if
15:    end if
16:  end if
17: end for
```

3.4 Program Rewriting

The final step is the rewriting phase illustrated in Algorithm 2, now that we have all the program analysis in place. As discussed in the previous step, we have identified the optimisable rules say *MR1, MR2.., MRn* along with the specific properties say *p1, p2.., pn* on the basis of which we are comparing the elements in the compare block. We index all instances of the right parameter of the identified rule *MRn* on the basis of the respective property *pn*. This is done using a built-in method called *mapBy* (Line 10 in Listing 2), which returns a map containing the results of the parameter expression as keys and the respective items of the target collection as values. The *mapBy* operation is called with all instances of the identified rule's right parameter and assigned to a newly declared variable (Line 4-10 of Algorithm 2). The naming convention of these variables is the rule name concatenated with the string "Map". So, a Map for the rule *Lifeline2Class* will be called as *Lifeline2ClassMap* (Line 11 of Algorithm 2). These variable statements are then added to the *pre* block of the ECL program (Line 13 of Algorithm 2). The *Pre* block is a set of EOL statements that are executed before the execution of match rules in ECL. This can be seen in Listing 2 (Line 10-12).

The next step is to utilise these pre-computed hashmaps (indices). For this, we have added the facility of specifying domains in ECL. Each parameter in an ECL rule can define a domain, which is an EOL expression that yields a set of model elements, allowing the developers to narrow down the search space. We support two types of domains in ECL. Static domains which are computed once for one match rule and are independent of bindings of the other parameter of the *MatchRule*. Static domains are denoted by the "*in*" keyword) and dynamic domains which are recomputed every time the other parameter value is changed. Dynamic domains are dependent on the other parameter values and are denoted by the "*from*" keyword. So we use these hashmap variables added in the pre block, as a dynamic domain for the right parameter of the corresponding *MatchRule*. For instance in Line 18 of Listing 2, we retrieve the value from the corresponding hashmap i.e., *Lifeline2ClassMap* using the left parameter's compared property (identified in the previous step) as a key.

Hashmaps return null values if they don't contain the mapping for a particular key, so to cater for possible null pointer exceptions, we use a safe navigation operator. The use of the safe navigation operator is shown in Line 18 of Listing 2, where we return an empty *Sequence* if the *get()* operation returns a null value. `var result = a?.someProperty?.anotherProperty;`

If *a* is not null, *someProperty* would be assigned to *result*, otherwise, *anotherProperty* would be assigned.

The last step of the rewriting phase is to rewrite the order of the rules as described in the Rule Scheduler step. The reordering is done on the basis of the dependency graph as in Figure 6, so that dependency-free rules can be executed first and then the ones dependent on them. Now, instead of the ECL engine executing the original program written by the developer, as listed in Listing 1, the automatically rewritten program as in Listing 2 will be executed. During execution, to minimize the storage of unnecessary unsuccessful match traces, only the unsuccessful traces that are required based on the dependency graph (i.e., if there are no corresponding *matches()* calls are saved.

4 Evaluation

In this section, we first present the experimental setup, including the case study and the models used for our benchmarks, and then we present the results of the conducted experiments. Finally we conclude the section by analysing and then stating any threats to the validity of the presented results.

4.1 Experimental Setup

To evaluate the proposed approach, we measured the execution time of the original ECL programs using the existing ECL engine with the rewritten ECL programs (also using the existing ECL engine). Since Epsilon already supports parallel execution of ECL programs, we conducted all these experiments with the parallel execution mode. Program rewriting

Algorithm 2 Algorithm for Program Rewriting

```
1:  Let $op$ = HashMap of rules with the corresponding properties as in Algorithm 1
2:  $DG$ = Dependency Graph
3:  for all $rule$ in $op$ do
4:      Construct property call expression ($pce$)
5:      target ← type of right parmeter of $rule$
6:      property ← all
7:      Construct operation call expression ($oce$
8:      target ← $pce$
9:      operation ← mapBy
10:     expression ← $op$.get($rule$)
11:     declare variable ($v$) with name $rule$.getName()+"Map"
12:     $v \leftarrow oce$
13:     add $v$ to pre block
14:     add domain block with expression $v$.get(leftParameter.property)
15: end for
16: reorder rules according to topological order of $DG$
```

with the help of dependency graph, identifies the independent rules that can be executed in parallel. First, we measured the execution time for running the comparison program with the existing ECL engine (without any optimisations) in parallel mode and we refer this as ECL in all the results tables and graphs. Second, we use the proposed approach to automatically rewrite the ECL program (as described in Section 3 and execute the rewritten program using the existing ECL engine in parallel mode. We refer to this as *Optimised ECL* in the results tables and graphs.

Table 4. Sizes of the models used for benchmarking

	No of model elements					
ID	**OO**	**DB**	**OO+DB**	**Seq**	**Class**	**Class+Seq**
1	287	184	471	305	356	661
2	357	229	586	417	356	773
3	427	274	701	417	469	886
4	497	319	816	342	356	698
5	567	364	931	342	356	698
6	637	409	1046	305	469	774
7	707	454	1161	342	469	811

4.1.1 Case Study & Models. For evaluating our approach, we used two case studies: one is the class and sequence diagram comparison as shown in Listing 1, the second is the comparison of object oriented (OO) models with database (DB) models. We have used the class and sequence diagram models of different sizes conforming to these metamodels publicly available on GitHub [13]. OO & DB are the synthetic models generated in [8]. The number of elements of different models are mentioned in Table 4. The point to note is that the sizes of the models that we are using are not very large but the comparison of these models still becomes computationally very expensive. Hence, a notable performance gain can be observed in these models.

```
rule Class2Table
  match l : OO!Class
  with r : DB!Table{

  compare : l.name = r.name
}

rule Attribute2Column
  match l : OO!Attribute
  with r : DB!Column
  {
  compare : l.name = r.name and l.owner.
      matches(r.table)
}
```

Listing 3. ECL comparison program for OO-DB models

To compare OO models with DB ones, we used a simple comparison algorithm (depicted in Listing 3) to establish matches between tables and classes, when their names are same. In the second rule, we compare attributes with columns on the basis of the property *name*, and also whether they belong to same class and table respectively. This example is quite simple but we have used this as a case study to show the substantial performance benefits observed even for simpler matching programs, with increasing model sizes. The comparison program in Listing 3 would be optimised and rewritten as represented in Listing 4.

4.1.2 Correctness. As the approach is based on automatic rewriting of the program, it is crucial that the rewritten program preserves the semantics of the original program. To ensure this, we use equivalence testing to compare the match trace for both the original and the rewritten programs. We used several ECL comparison programs mined from GitHub to compare models both conforming to same and different metamodels and then compared their output match traces. Mostly, comparison programs available on GitHub were comparing models from the same modelling language. We verified that the number of successful matches in both the optimised and the unoptimised version remained the same, as shown in Table 5 and 6. While the number of successful matches are the same, one can observe the difference in number of unsuccessful matches in the Table 5 and 6. This is because of the successful pre-filtering/pre-indexing in the proposed approach. We filter some of the instances which, using the static program analysis, can be categorised as unsuccessful, before actually running the comparison algorithms.

```
pre {
  var Class2TableMap = DB!Table.all.mapBy(
      param|param.name);
  var Attribute2ColumnMap = DB!Column.all.
      mapBy(param|param.name);
}

rule Class2Table
match l : OO!Class
with r : DB!Table
from : Class2TableMap.get(l.name) ?:
    Sequence{} {
  compare : true
}

rule Attribute2Column
match l : OO!Attribute
with r : DB!Column
from : Attribute2ColumnMap.get(l.name) ?:
    Sequence{} {
  compare : true and l.owner.matches(r.
      table)
}
```

Listing 4. ECL rewritten program for OO-DB models

4.1.3 Machine Specification.

The set of evaluation experiments presented in this paper were performed on a MacBookPro @ M2 Core i7, 24 GBs of RAM, Mac operating system Ventura version 13.0, and Java 17 on JDK 17.0.6 with JVM MaxHeapSize 6GBs.

4.2 Results

In this section, we present the results from the conducted experiments. Table 7 presents the execution time in milliseconds for the OO and the DB model comparison, and the Class and Sequence Diagram model comparison respectively. This execution time also includes the time taken for indexing. The rewriting and reordering of rules are done before the execution and takes negligible amount of time (≈2ms). The results can also be visualised for the OO & DB comparison in Figure 7 and the Class and Sequence diagram in Figure 8.

Table 7. Execution time of existing ECL and optimised ECL, in ms

	OO - DB		CL - SEQ	
ID	**ECL**	**Optimised**	**ECL**	**Optimised**
1	1962	535	3287	196
2	3488	781	3109	194
3	6745	1238	3894	205
4	14051	1735	4046	188
5	22044	1924	4286	287
6	31611	3705	4342	199
7	52159	4520	5050	250

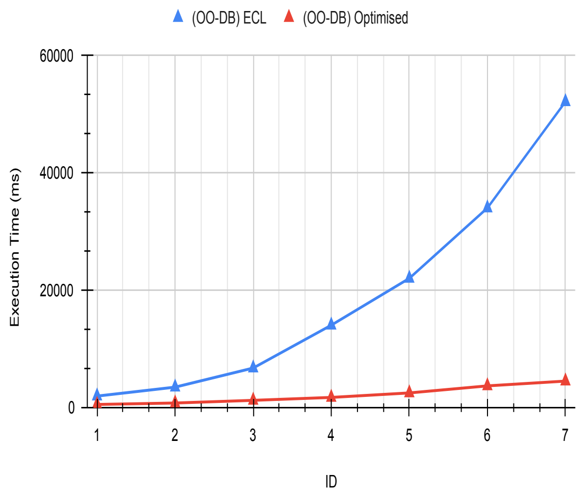


Figure 7. Comparison of Execution time in OO DB Comparison

As seen in Table 4, the OO and the DB models are of increasing sizes, while this is not the case with the Class and Sequence Diagram models. Keeping these sizes of models in mind, we can see a continuous rise in performance gain as the model size increases (Figure 7). While in Figure 8, we can see almost a constant performance gain compared to the existing ECL engine. This suggests that our performance benefits are proportional to model size.

This performance gain is achieved by reducing the search space needed for matching. We can clearly observe in the match traces produced for both case studies in the Tables Table 5 and Table 6 that the number of unsuccessful matches are significantly reduced in our proposed approach.

Another important factor to notice here is that this approach might not bring performance benefits when comparing very small models. As the proposed approach provides

Table 5. Match Trace of Class and Sequence Diagram Comparison

ID	ECL (All)	Optimised (All)	ECL (Successful)	Optimised(Successful)
1	29400	92	38	38
2	29400	92	38	38
3	33700	78	0	0
4	33284	54	0	0
5	33284	54	0	0
6	33700	78	0	0
7	38100	208	72	72

Table 6. Match Trace of OO and DB Comparison

ID	ECL (All)	Optimised (All)	ECL (Successful)	Optimised(Successful)
1	18060	4120	60	60
2	28200	6400	75	75
3	40590	9180	90	90
4	55230	12460	105	105
5	72120	16240	120	120
6	91260	20520	135	135
7	112650	25300	150	150

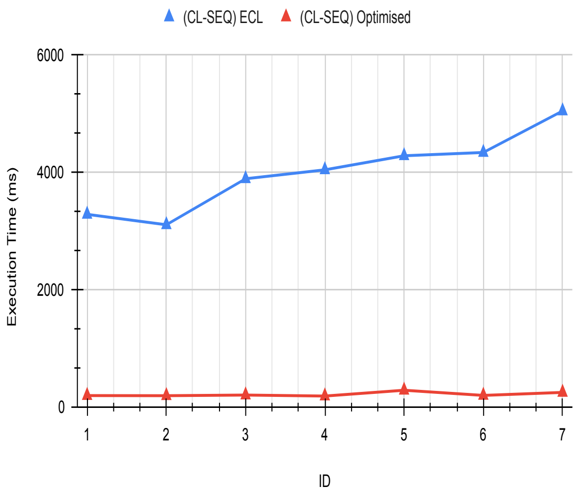


Figure 8. Comparison of Execution time in Class Sequence Diagram Comparison

a caching mechanism, the optimization indeed comes at the expense of increased memory footprint. As the size of the computed caches can be estimated from the number of rules/indexed properties in a straightforward manner. Precomputing the indices (as mentioned in the program rewriting section) has an overhead, which is paid off for larger models, and we expect to see a much clearer improvement in performance when it comes to larger models.

4.3 Threats to Validity

A primary threat to the validity of the results presented here, is that the measured performance may be particular to the models that were created for the tests, to the kind of model, or to the comparison programs that were proposed. A key challenge identified in MDE research is a lack of publicly accessible real-world models [20]. Although we used both synthetic models in the OO-DB case studies and publicly available models for the class and sequence diagrams one, this can still affect the measured performance benefits. To further generalise the results, we would need to perform experiments with different models and comparison programs as well as with different modeling technologies such as Simulink and CDO to demonstrate the scalability of our proposed approach, especially for larger models.

As the rewriting is based on static analysis, we recommend explicitly stating the types of the constructs wherever possible, to allow accurate type resolution and enable automated rule optimisation (as described in Section 3).

5 Related Work

Model comparison deals with finding similarities and differences between elements of different models. This comparison can be done on the basis of structure, semantics and metrics etc., [12]. In the context of this paper we will be stating the use of program rewriting in optimisation and also the state of the art that involves structural model comparison.

Program rewriting has proven to be beneficial for various optimization purposes, as demonstrated in [24] where it was utilized for optimizing type level model queries. Additionally, rewriting has played a crucial role in translating EOL expressions to Viatra for incremental evaluation [6], as well as converting them to MySQL queries for efficient execution on relational databases [7].

It has been demonstrated in [23] that conventional text-based comparison and differencing techniques are insufficient for model comparison due to the structured nature of models.

Model-to-model transformations have shown to be used for comparing models as in [11]. As M2M languages are not tailored for model comparison task and hence generally very verbose as M2M languages do not have constructs tailored for model comparison activities.

Change-based model comparison was presented in [25] where the comparison is done only for the model elements that have been changed since the previous version which is quite efficient compared to state-based comparison.

EMF Compare [2] & EMF Diff Merge[3] are two tools available to compare and then merge two models. EMF Compare uses built-in heuristics for model element references and attribute values while a tailored language like ECL lets you write custom matching rules for different model elements.

There are other comparison approaches as shown in [21] that demonstrates comparison between different UML models but the approach is only limited to models conforming to a single metamodel. Additionally as mentioned in [14], most similarity-based approaches such as SiDiff [23] and DSMDiff [19], have limited support when it comes to heterogeneous models which is supported by ECL, where one can specify complex matching algorithms for models conforming to different metamodels.

6 Conclusions & Future Work

We have presented an approach for efficiently comparing models using programs written in rule-based model comparison language. This efficient comparison approach incorporates an automatic rewriting facility to speed up the model comparison (both homogeneous and heterogeneous) based on static analysis. The rewriting automatically extracts dynamic domains to provide pre-filtering of model elements before actually comparing them. Additionally, static analysis also helps reorder the rules based on the dependencies identified between these match rules through the creation of a dependency graph. This enables us to execute independent rules before those dependent on them, optimizing the comparison process by reducing the cost of jumping between comparison rules. Through experiments, we demonstrate that our approach significantly improves execution time compared to the default ECL execution engine, providing substantial performance benefits.

In future work, the proposed approach can be potentially used to provide correspondence between models from heterogeneous modelling technologies. For instance, it can facilitate the comparison between Simulink models and EMF models. Moreover, this automatic domain rewriting facility can be integrated with other rule-based languages such as Epsilon's pattern matching language (EPL).

Acknowledgments

This research is supported by the Lowcomote project, funded by the European Union's H2020 Research and Innovation Programme under the Marie Skłodowska-Curie GA n°813884.

References

[1] 2022. Epsilon Validation Language. https://www.eclipse.org/epsilon/doc/evl/. [Online; accessed 29-April-2022].

[2] 2023. Eclipse EMF Compare. https://projects.eclipse.org/projects/modeling.emfcompare. [Online; accessed 10-April-2023].

[3] 2023. EMF DiffMerge. https://wiki.eclipse.org/EMF_DiffMerge. [Online; accessed 10-April-2023].

[4] 2023. Epsilon. https://www.eclipse.org/epsilon/. [Online; accessed 26-March-2023].

[5] 2023. Epsilon Model Connectivity Layer. https://www.eclipse.org/epsilon/doc/emc/. [Online; accessed 26-March-2023].

[6] Qurat Ul Ain Ali, Benedek Horváth, Dimitris Kolovos, Konstantinos Barmpis, and Ákos Horváth. 2021. Towards Scalable Validation of Low-Code System Models: Mapping EVL to VIATRA Patterns. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 83–87. https://doi.org/10.1109/MODELS-C53483.2021.00019

[7] Qurat ul ain Ali, Dimitris Kolovos, and Konstantinos Barmpis. 2020. Efficiently Querying Large-Scale Heterogeneous Models. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings* (Virtual Event, Canada) *(MODELS '20)*. Association for Computing Machinery, New York, NY, USA, Article 73, 5 pages. https://doi.org/10.1145/3417990.3420207

[8] Qurat ul ain Ali, Dimitris Kolovos, and Konstantinos Barmpis. 2022. Selective Traceability for Rule-Based Model-to-Model Transformations. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering* (Auckland, New Zealand) *(SLE 2022)*. Association for Computing Machinery, New York, NY, USA, 98–109. https://doi.org/10.1145/3567512.3567521

[9] Antonio Bucchiarone, Jordi Cabot, Richard F Paige, and Alfonso Pierantonio. 2020. Grand challenges in model-driven engineering: an analysis of the state of the research. *Software and Systems Modeling* 19, 1 (2020), 5–13.

[10] Justin Cooper, Alfonso De la Vega, Richard Paige, Dimitris Kolovos, Michael Bennett, Caroline Brown, Beatriz Sanchez Piña, and Horacio Hoyos Rodriguez. 2021. Model-Based Development of Engine Control Systems: Experiences and Lessons Learnt. In *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and*

Systems (MODELS). 308–319. https://doi.org/10.1109/MODELS50736.2021.00038

[11] Marcos Didonet Del Fabro and Patrick Valduriez. 2007. Semi-Automatic Model Integration Using Matching Transformations and Weaving Models. In *Proceedings of the 2007 ACM Symposium on Applied Computing* (Seoul, Korea) *(SAC '07)*. Association for Computing Machinery, New York, NY, USA, 963–970. https://doi.org/10.1145/1244002.1244215

[12] Lucian Gonçales, Kleinner Farias, Murilo Scholl, Toacy Oliveira, and Mauricio Veronez. 2015. Model Comparison: a Systematic Mapping Study. https://doi.org/10.18293/SEKE2015-116

[13] Faezeh Khorram, Masoumeh Taromirad, and Raman Ramsin. [n. d.]. SeGa4Biz: Model-Driven Framework for Developing Serious Games for Business Processes. ([n. d.]).

[14] Dimitrios S. Kolovos. 2009. Establishing Correspondences between Models with the Epsilon Comparison Language. In *Model Driven Architecture - Foundations and Applications*, Richard F. Paige, Alan Hartman, and Arend Rensink (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 146–157.

[15] Dimitris S Kolovos and Richard F Paige. 2017. The epsilon pattern language. In *2017 IEEE/ACM 9th International Workshop on Modelling in Software Engineering (MiSE)*. IEEE, 54–60.

[16] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. 2006. The epsilon object language (EOL). In *European conference on model driven architecture-foundations and applications*. Springer, 128–142.

[17] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. 2006. Merging models with the epsilon merging language (eml). In *Model Driven Engineering Languages and Systems: 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006. Proceedings 9*. Springer, 215–229.

[18] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. 2008. The epsilon transformation language. In *Theory and Practice of Model Transformations: First International Conference, ICMT 2008, Zürich, Switzerland, July 1-2, 2008 Proceedings 1*. Springer, 46–60.

[19] Yuehua Lin, Jeff Gray, and Frédéric Jouault. 2007. DSMDiff: a differentiation tool for domain-specific models. *European Journal of Information Systems* 16, 4 (2007), 349–361.

[20] José Antonio Hernández López and Jesús Sánchez Cuadrado. 2021. Towards the Characterization of Realistic Model Generators using Graph Neural Networks. In *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 58–69. https://doi.org/10.1109/MODELS50736.2021.00015

[21] Dirk Ohst, Michael Welle, and Udo Kelter. 2003. Differences between Versions of UML Diagrams. *SIGSOFT Softw. Eng. Notes* 28, 5 (sep 2003), 227–236. https://doi.org/10.1145/949952.940102

[22] Massimo Tisi, Salvador Martínez, and Hassene Choura. 2013. Parallel execution of ATL transformation rules. In *Model-Driven Engineering Languages and Systems: 16th International Conference, MODELS 2013, Miami, FL, USA, September 29–October 4, 2013. Proceedings 16*. Springer, 656–672.

[23] Christoph Treude, Stefan Berlik, Sven Wenzel, and Udo Kelter. 2007. Difference computation of large models. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 295–304.

[24] Qurat Ul Ain Ali, Dimitris Kolovos, and Konstantinos Barmpis. 2021. Identification and Optimisation of Type-Level Model Queries. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 751–760. https://doi.org/10.1109/MODELS-C53483.2021.00121

[25] Alfa Yohannis, Rodriguez Hoyos Rodriguez, Fiona Polack, and Dimitris Kolovos. 2019. Towards Efficient Comparison of Change-Based Models. *Journal of Object Technology* 18, 2 (July 2019), 7:1–21. https://doi.org/10.5381/jot.2019.18.2.a7 The 15th European Conference on Modelling Foundations and Applications.

Received 2023-07-07; accepted 2023-09-01

Deriving Integrated Multi-Viewpoint Modeling Languages from Heterogeneous Modeling Languages: An Experience Report

Malte Heithoff
heithoff@se-rwth.de
Software Engineering, RWTH Aachen University
Germany

Nico Jansen
jansen@se-rwth.de
Software Engineering, RWTH Aachen University
Germany

Jörg Christian Kirchhof
kirchhof@se-rwth.de
Software Engineering, RWTH Aachen University
Germany

Judith Michael
michael@se-rwth.de
Software Engineering, RWTH Aachen University
Germany

Florian Rademacher
rademacher@se-rwth.de
Software Engineering, RWTH Aachen University
Germany

Bernhard Rumpe
rumpe@se-rwth.de
Software Engineering, RWTH Aachen University
Germany

Abstract

In modern systems engineering, domain experts increasingly utilize models to define domain-specific viewpoints in a highly interdisciplinary context. Despite considerable advances in developing model composition techniques, their integration in a largely heterogeneous language landscape still poses a challenge. Until now, composition in practice mainly focuses on developing foundational language components or applying language composition in smaller scenarios, while the application to extensive, heterogeneous languages is still missing. In this paper, we report on our experiences of composing sophisticated modeling languages using different techniques simultaneously in the context of heterogeneous application areas such as assistive systems and cyber-physical systems in the Internet of Things. We apply state-of-the-art practices, show their realization, and discuss which techniques are suitable for particular modeling scenarios. Pushing model composition to the next level by integrating complex, heterogeneous languages is essential for establishing modeling languages for highly interdisciplinary development teams.

CCS Concepts: • **Software and its engineering → Model-driven software engineering**; **Domain specific languages**.

Keywords: Software Language Engineering, Language Composition, Domain-Specific Languages, Language Families, Reuse, Internet of Things, Assistive Systems

ACM Reference Format:
Malte Heithoff, Nico Jansen, Jörg Christian Kirchhof, Judith Michael, Florian Rademacher, and Bernhard Rumpe. 2023. Deriving Integrated Multi-Viewpoint Modeling Languages from Heterogeneous Modeling Languages: An Experience Report. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3623476.3623527

1 Introduction

Software and systems engineering faces an increasing level of complexity as we have to handle the increasing complexity of the world. Using modeling approaches has proven to be a suitable approach to handle this complexity [86]. To create models of reality for domains such as production [10, 32], automotive [87], and medicine [77], to be used in, e.g., digital twins [36], for explainable cyber-physical systems [9], or complex systems-of-systems, it is necessary to consider a range of perspectives and viewpoints. This requirement is commonly known as multi-viewpoint modeling, which entails addressing different properties of systems for the diverse disciplines involved in an accessible fashion.

One approach to meeting the specific needs of particular disciplines in their engineering efforts is to use Domain-Specific Languages (DSLs). Although such DSLs can be employed simultaneously for different use cases, in practice, they often cover only a single viewpoint if not further supported by tooling, such as projective approaches. As a result, also considering that a single DSL often cannot suit every use case alone, this requires combining several languages to achieve a more holistic view of a system. To address this issue, researchers have proposed various techniques, such as using multiple DSLs, developing a unified language that covers different viewpoints such as UML or SysML, or using language workbenches, which provide an integrated environment for designing, implementing, and using DSLs.

It can be argued that the number of modeling languages in different domains is steadily increasing [15, 20, 59, 66] which, next to maturity and evolution, raises the question on how to integrate these languages—not only for coping with the complexity of contemporary software systems that consist of many heterogeneous parts, but also for the sake of *reuse* [80]. Reuse in software engineering benefits, among others, (i) quality by gradually accumulating error fixes; (ii) productivity by decreasing the demand for new software; and (iii) reliability by increasing the chance to find errors through higher usage rates [78]. However, a key ingredient for software reuse is the establishment of interoperability between heterogeneous components, e.g., by means of mutually agreed interfaces. These reuse considerations also apply to the DSLs and their related tooling. When integrating heterogeneous modeling languages, we aim to reuse both, the languages or parts of the languages themselves as well as the already developed or generated tools such as parsers, pretty printers, or full generators. We study the integration of heterogeneous modeling languages by leveraging different mechanisms for language composition. This allows us to make independently developed languages reusable, achieving different viewpoints at the model level for the distinct application domains.

This experience report tackles the research question of *how to integrate different modeling languages via established language composition techniques achieving multi-viewpoint modeling languages.* In this paper, we elaborate on our experiences from two case studies of complex, real-world, software-intensive systems and show which language composition methods are applied there. One is a language family for model-driven development of IoT applications. The other language family is used to support the model-driven engineering of assistive systems and to use models at runtime of the system. Additionally, we discuss the different mechanisms for language composition used and detail our experiences on which techniques were suitable for which cases and whether they contribute to establishing multi-viewpoint modeling. For our studies, we use the MontiCore language workbench [45] as it comes with various composition techniques.

Structure. Sec. 2 provides background information for our approach. Sec. 3 discusses related work for the composition of modeling languages, language workbenches with composition support, and modeling languages. In Sec. 4, we introduce two use cases from complex, real-world, software-intensive systems, namely to develop IoT systems and assistive systems. Sec. 5 discusses the application of the different language composition approaches in our modeling scenarios and their contribution to achieving a multi-viewpoint modeling environment. The last section concludes.

2 Background

We provide relevant background on model-driven engineering as well as language composition mechanisms.

2.1 Model-Driven Engineering

Model-Driven Engineering (MDE) [18] is a software engineering paradigm that promotes the use of models as first-class citizens in all or selected phases of the software engineering process. In the sense of MDE, a model is a software artifact that abstracts from certain details of a software system, and can replace specific parts of the system for certain purposes such as implementation, testing, or simulation.

Next to model application, MDE also systematizes model construction, evolution, and maintenance. All of these activities require an unambiguous notion of model validity that is commonly defined by the language in which a model is expressed [18].

To this end, a modeling language consists of (i) an abstract syntax that specifies the essential information of models independent of their representation; (ii) a concrete syntax that specifies the user-facing representation of model elements; and (iii) a semantic associating each model element with a meaning [18, 44].

Language workbenches [29] denote IDEs that bundle tools for modeling language construction, e.g., meta-grammars, parser generators, and language composition facilities. Examples of contemporary language workbenches include MPS [70], Xtext [30], and MontiCore [45]. Due to its mature support for a variety of language composition mechanisms, we henceforth leverage MontiCore to study the derivation of multi-viewpoint modeling languages by language composition.

MontiCore is a language workbench [45] whose EBNF-like [88] meta-grammar allows the specification of grammars for modeling languages with textual concrete syntaxes. From a language grammar expressed in its meta-grammar, MontiCore is able to generate (i) the implementation of the corresponding abstract syntax in the form of a metamodel; (ii) the parser infrastructure to instantiate the metamodel from input files adhering to the grammar; and (iii) additional infrastructure for common concerns in modeling language implementation such as context condition checking, symbol table management, and template-based code generation. With its generative approach, MontiCore effectively reduces the effort in modeling language implementation. In addition, and by contrast to other language workbenches, MontiCore's meta-grammar also provides constructs for modeling language composition [13], thereby facilitating the integrated evolution of a modeling language from a single source artifact, i.e., the language's grammar.

Given its versatility, MontiCore is actively used to create and maintain DSLs targeting heterogeneous domains such as automotive [24], cloud services [27], Internet of Things [53], robotics [1], and systems engineering [19].

2.2 Language Composition Mechanisms

For the efficient engineering of modeling languages, MontiCore especially focuses on compositional language design. To this end, MontiCore supports multiple language composition techniques and corresponding design patterns, enabling combining multiple DSLs [26]. Furthermore, it provides an extensive library of language components [12] serving as a common foundation for building more sophisticated modeling languages. Overall, MontiCore supports four different types of language composition realized either directly via the language specification, i.e., the grammar, or indirectly via the symbol table infrastructure [14] of a language.

To explain the various composition techniques, we consider a number of languages developed in MontiCore's ecosystem. The following language definitions are simplified versions for clarity reasons and space limitations. The original sources are referenced accordingly.

First, we introduce a simple automata language. Such an automaton gradually processes letters of an input alphabet. A sequence of letters (i.e., a word) is accepted if there exists a path to a final state. Otherwise, the word is rejected. Overall, the automaton represents the set of words it accepts. Figure 1 contains the grammar of the automata language[1]. An `Automaton` (l. 02) starts with the respective keyword, has a name, and consists of multiple states and transitions (l. 03). A `State` (also indicated via a corresponding keyword) has a name (l. 05) and can be marked as «`initial`» or «`final`» (l. 06). Finally, a `Transition` (l. 08) describes the change from the source (`src`) to a target (`tgt`) state via an `input` letter enclosed in an arrow-like syntactical structure.

```
01 grammar Automata extends MCBasics {
02   symbol scope Automaton =
03     "automaton" Name "{" (State | Transition)* "}" ;
04
05   symbol State = "state" Name
06     (("<<" ["initial"] ">>") | ("<<" ["final"] ">>"))* ;
07
08   Transition = src:Name "-" input:Name ">" tgt:Name ";";
09 }
```

Figure 1. Simplified version of MontiCore's automaton language[1] for modeling non-hierarchical automata with states and transitions.

The production rules of the automaton language employ predefined constructs such as the `Name` token (cf. l. 03). This usage already indicates the first application of language extension as this respective token comes from the base grammar `MCBasics`, which is extended by the automaton language (l. 01), importing its productions.

The next DSL under consideration is the class diagram language `CD4Analysis` used to describe data structures consisting of classes and their attributes. Its context-free grammar is depicted in Figure 2. Please note that the original specification[2] is designed in a highly compositional fashion, further modularizing the different constituents. Thus, the Class Diagram (CD) languages in this paper are simplified versions that roughly sketch the structure and are tailored to explain the distinct composition techniques.

The root node of a class diagram model is the `CDCompilationUnit` (l. 02). It consists of a package declaration and a set of import statements, two inherited properties. It entails a `CDDefinition`, representing the actual diagram (l. 03). The `CDDefinition` (l. 05) depicts the start of the diagram via a corresponding keyword. It has a name and comprises multiple elements contained in curly brackets. These elements are specified by the interface nonterminal `CDElement` (l. 07). This interface can be implemented by other nonterminals, thus serving as an explicit extension point. In this grammar, the only element implementing it is the `CDClass` (ll. 09-12) that has a name and comprises multiple `CDAttributes`. In turn, these attributes (l. 14) consist of a type and a corresponding name.

```
01 grammar CD4Analysis extends MCBasics, MCBasicTypes {
02   CDCompilationUnit = MCPackageDeclaration
03      MCImportStatement* CDDefinition;
04
05   CDDefinition = "classdiagram" Name "{" CDElement* "}";
06
07   interface CDElement;
08
09   symbol scope CDClass implements CDElement =
10     "class" Name "{"
11       CDAttribute*
12     "}";
13
14   symbol CDAttribute = MCType Name;
```

Figure 2. Simplified version of MontiCore's class diagram language[2] for modeling the data structure of a system via classes and their attributes.

2.2.1 Language Inheritance.

The first composition technique of MontiCore is language inheritance. Here, the constructs of an original language are adopted and extended or modified for a new use case. While the original language remains unchanged, the new DSL incorporates concrete and abstract syntax, as well as the generated tooling and its handwritten extensions.

[1]Automata language definition available at: https://github.com/MontiCore/automaton

[2]Compositional class diagram language definition available at: https://github.com/MontiCore/cd4analysis

Figure 3 shows an example of language inheritance by an extended class diagram language. The inheritance relation is indicated by the `extends` keyword (l. 01) followed by the corresponding host language name that should be adopted. As the overall structure of a class diagram remains unchanged, we keep the starting nonterminal `CDCompilationUnit` (l. 02). In addition to the adopted constructs, we expand the language by modifying or adding production rules for modeling elements. Thus, the nonterminal `CDClass` is overwritten (ll. 04-08) to enable basic method signatures in the class body in addition to the already existing attributes. For these signatures, we introduce the corresponding nonterminal `CDMethod`, which describes the syntax of the element (ll. 10-11) and can be referenced in other production rules. Besides modifying the class contents, the extended language also introduces interface definitions (ll. 13-16) and enumerations (ll. 18-23) as new diagram elements.

```
01 grammar CD4Code extends CD4Analysis {
02   start CDCompilationUnit;
03
04   @Override
05   symbol scope CDClass implements CDElement =
06     "class" Name "{"
07       (CDAttribute | CDMethod)*
08     "}" ;
09
10   symbol CDMethod =
11     MCType Name "(" (argT:MCType argN:Name)* ")" ";";
12
13   symbol scope CDInterface implements CDElement =
14     "interface" Name "{"
15       CDMethod*
16     "}" ;
17
18   symbol scope CDEnum implements CDElement =
19     "enumeration" Name "{"
20       (EnumLiteral || ",")*
21     "}" ;
22
23   symbol EnumLiteral = Name;
24 }
```

Figure 3. Extended class diagram language[2] including methods, interfaces, and enumerations. Application of language inheritance with conservative extension.

2.2.2 Language Extension. Language extension is a specific, more restrictive form of language inheritance. This technique also takes over all constituents of a host language. The difference is that changes are only allowed in the form of conservative extension. This means that only new elements may be added to a language or existing elements may only be modified in an extending but non-restricting way. Thus, valid models of the original language still remain valid in the context of the extended variant.

In fact, the inheritance example in Figure 3 features only conservative extensions. Adding further elements such as `CDInterface` or `CDEnum` only extends the set of valid sentences. Also, despite being overwritten, the altering of `CDClass` remains conservative as it further introduces methods inside a class without impacting the use of their attributes (l. 07). Figure 3 also illustrates the benefit of languages being intentionally tailored for their extension (or the drawbacks if not). Thus, while adding methods to classes via overwriting the production is possible, it includes lots of duplication in the production rules. This is a considerable overhead for adding a single reference inside a production rule. In contrast, adding `CDInterface` and `CDEnum` to the overall diagram yields no overhead. The difference is that the newly introduced nonterminals implement the already existing interface nonterminal `CDElement` (cf. Figure 2, l. 07) of the original language. This element is an explicit extension point that supports inheriting languages to weave new constituents into existing production rules. In this case, the original `CDDefinition` (l. 05) references the interface, allowing all incarnations as valid CD elements. Thus, designing a language with extension in mind can significantly improve the engineering of further variants.

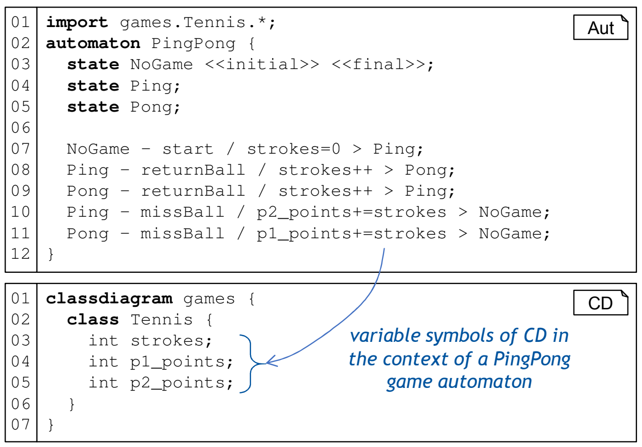
2.2.3 Language Embedding. Language embedding integrates multiple DSL definitions, combining their production rules in a single grammar, enabling integrated modeling via their combined constituents. Therefore, this technique not only collects the entirety of nonterminals of all included languages but automatically combines their usages concerning shared interface definitions and usages. This is especially useful for integrating default language components, such as expressions, into an existing DSL. A language component is a (possibly incomplete) definition comprising a grammar, corresponding generated and handwritten artifacts, as well as an integration interface established via predefined extension points. They constitute a decoupled set of reusable standard productions explicitly tailored for embedding. Technically in MontiCore, language embedding employs multiple inheritance. Thus, embedding a language into another is as simple as extending both in the grammar signature.

```
01 grammar MealyAutomata extends Automata,
02         CommonExpressions, AssignmentExpressions {
03
04   MealyAutomaton = MCImportStatement* Automaton;
05
06   @Override
07   Transition =
08     from:Name "-" input:Name "/"
09     output:Expression ">" to:Name ";" ;
10 }
```

Figure 4. Language for modeling mealy automata, non-conservatively inheriting from the automaton language and embedding expressions.

Figure 4 contains an example of embedding expressions into the already established automata language by extending both definitions (ll. 01-02). Simultaneously, the new language adds the possibility to import other artifacts (l. 04) and advances the automaton to mealy machines (i.e., processing an input and producing a corresponding response action). The latter is achieved via overwriting the production rule of the `Transition` (ll. 06-09) and further separating the `input` from a newly established `output` expression, separated by a slash. As this addition is not optional, the language is extended in a non-conservative way, i.e., original automaton models are not valid anymore in this variant. Realizing the output as an expression enables arbitrary terms of all embedded languages, such as `s1 - a / (x == 4+3) > s2`, indicating a state change from `s1` to `s2` on the input `a` and triggering the action evaluation for the boolean expression `x == 4+3`.

2.2.4 Language Aggregation. Language aggregation enables integrating models of multiple DSLs while simultaneously keeping them as separate artifacts. In contrast to embedding, the technique of aggregating languages only loosely couples DSL definitions and makes them operable in a common context. This inter-operationality is achieved via MontiCore's symbol table infrastructure, allowing cross-referencing, even over multiple artifacts.


Aut
CD
variable symbols of CD in the context of a PingPong game automaton

```
01 import games.Tennis.*;
02 automaton PingPong {
03   state NoGame <<initial>> <<final>>;
04   state Ping;
05   state Pong;
06
07   NoGame - start / strokes=0 > Ping;
08   Ping - returnBall / strokes++ > Pong;
09   Pong - returnBall / strokes++ > Ping;
10   Ping - missBall / p2_points+=strokes > NoGame;
11   Pong - missBall / p1_points+=strokes > NoGame;
12 }
```

```
01 classdiagram games {
02   class Tennis {
03     int strokes;
04     int p1_points;
05     int p2_points;
06   }
07 }
```

Figure 5. Models of the extended automaton and CD DSLs used in a shared context via language aggregation.

Considering again the extended automata language in Figure 4 introduced for language embedding, we can further extend its usage aggregating class diagrams. The main idea here is to combine a structural and a behavioral language. Figure 5 presents the composition of two exemplary models, preserving them as separate artifacts. At the top, we have a simplified automaton model of a customized `PingPong` game featuring three states (top, ll. 03-05) and five transitions for depicting the gameplay (top, ll. 07-11). The second model is a class diagram defining a set of variables for counting the `strokes` and the points of two respective players (bottom, ll. 03-05). Furthermore, the automaton imports the class diagram (top, l. 01), making all types and variables accessible. Thus, the embedded expressions at the transitions can be employed to reference the externally defined variables[3], e.g., for incrementing the number of strokes when the ball is returned (top, ll. 08-09) or assigning points to a player when the other misses the ball (top, ll. 10-12). That way, the embedded expressions are additionally employed to reference symbols of other models, enabling a seamless composition over multiple artifacts.

3 Related Work

This section presents work related to the composition of modeling languages (Sect. 3.1), language workbenches with composition support (Sect. 3.2), and families of modeling languages derived by language composition (Sect. 3.3). For a comprehensive overview of model view approaches, we refer the readers to [11].

3.1 Composition Approaches

A straightforward approach to modeling language composition is the exploitation of inter-model references [89]. To realize this approach, the referring modeling language must integrate modeling concepts by which it can establish links to instances of another modeling language's concepts. On the model-level, such links appear as references from elements in the referring model to elements in the referenced model. While this approach is versatile, e.g., it can be retrofitted into the referring language without impacting the referred language, it requires the user to comprehend models in different languages, scatters information across different models, does not enable directed alteration of elements in referred models, and may result in invalid models when referred elements are changed independently of the referring model.

A more sophisticated approach to reference-based language composition is the conversion of technology-specific language metamodels into more abstract representations. This enables the specification of correspondence relationships between concepts from heterogeneous modeling languages [48]. These correspondence relationships may anticipate conversion rules between modeling concepts, thereby making links between model elements actionable, e.g., to base the validity of a correspondence relationship on the concrete peculiarity of a referred model element. However, the same drawbacks as for inter-model references apply.

Modeling language variability [41] constitutes an approach to language composition when considering the base modeling language as one language and the delta sets of language concepts derived from activated base language features as their own languages. While modeling language variability can anticipate all possible language compositions and provide exhaustive tooling from the

[3]We do not distinguish between type and instance level in this example for simplicity reasons.

beginning, composition is constrained to the supported features of the base language.

A base language may also explicitly delegate the realization of certain modeling concepts to independently developed languages that provide concrete support for such concepts [35, 55]. This approach requires a priori reasoning about compositionality by base language designers as well as meta-languages or meta-operators that systematize the delegation of modeling concept realization. Furthermore, it may be necessary to add additional glue code for concept realization in order to align the semantics of concept-realizing languages with that of the corresponding base languages.

3.2 Language Workbenches with Composition Support

Melange [21] is a language workbench that supports model-first composition on the level of metamodel concepts including their operational semantics. However, composition of language artifacts besides metamodel implementations is out of Melange's scope. Similarly, MetaEdit+ [83] allows language integration via references between metamodel concepts. MPS [85] inherently exhibits support for model-first composition because language definition is also model-first. Concepts from the abstract syntax of a composed language can thus embed, extend, or adapt concepts from the abstract syntaxes of other languages, including related tooling like code generators.

Xtext [8] is a language workbench with grammar-first composition support, i.e., its meta language for grammar specification and subsequent metamodel derivation integrates keywords for language composition. Specifically, Xtext supports composition by importing the rules of an independent language grammar into a composed grammar and the derivation of new languages by leveraging the rules of a base language as an initial, yet extensible, rule set. Similarly to Xtext, Neverlang [16] is a grammar-first language workbench, which provides a more fine-grained, but rather complex, support for language composition based on language modules, roles, and role slicing for language feature specification.

3.3 Composed Modeling Language Families

Modeling language composition fosters the systematic creation of *modeling language families*, which constitute sets of two or more integrated languages, to enable the application of MDE to coherent parts of a problem domain.

Inter-model references (Sect. 3.1) represent a flexible means of generic language creation because they can be used to non-intrusively relate modeling languages, thus giving rise to language families by enriching or constraining modeling syntaxes and semantics. For example, reference-based composition allows for (i) constraining modeling syntaxes or model element peculiarities by linking metamodel concepts with invariants expressed in another language [72, 82]; or (ii) making relationships between diverse parts of a software architecture explicit, thereby fostering architecture comprehension and reasoning [34, 74].

When being based on a more abstract representation acting as an intermediate language to bridge between heterogeneous language concepts, inter-model references also foster independent evolution of composed languages and extensibility of language families [79]. Similarly, they facilitate the creation of modeling languages whose concepts are tailored to domain expert concerns but map to other languages' concepts of a different domain, e.g., to generate executable code [76].

Modeling language families derived from variability-based language composition (Sect. 3.1) often consist of *sibling languages*, or sub-families of such siblings, that are immediate descendants of the base language. They can be automatically derived by modelers and afterwards applied to related, yet slightly different problem sets, in the target domain [54, 57, 90].

4 Case Studies from Complex, Real-World, Software-Intensive Systems

For a better understanding of the possible uses of the different language composition techniques, we describe two specific case studies. Their size and complexity show, why different composition techniques are needed in practice.

4.1 IoT Systems

The Internet of Things connects objects with each other and with the Internet. Applications of the IoT include both industrial and consumer sectors and range from connected vehicles (and fleet tracking), to Asset Tracking, to Smart Homes. To do so, these objects are equipped with sensors and actuators. As inherently distributed applications, the development of IoT systems requires different skills than the development of classic software systems such as smartphone apps [81]. One way to manage the heterogeneity and complexity of IoT solutions is to use model-driven techniques [28, 62, 67], as they raise the level of abstraction.

MontiThings [53] is a language family for model-driven development of IoT applications. MontiThings covers the design, deployment [50], and analysis [51, 52] of the applications generated from the models. MontiThings aims to simplify the complicated development of IoT applications and abstract from the heterogeneity of IoT devices. To separate concerns and not mix, *e.g.,* technical details with high-level business logic, MontiThings consists of a family of multiple languages. The core of MontiThings is a component-and-connector (CnC) architecture description language that is used to describe the business logic of the applications. From the models of this language, MontiThings generates the C++ code for distributed applications and the scripts to package

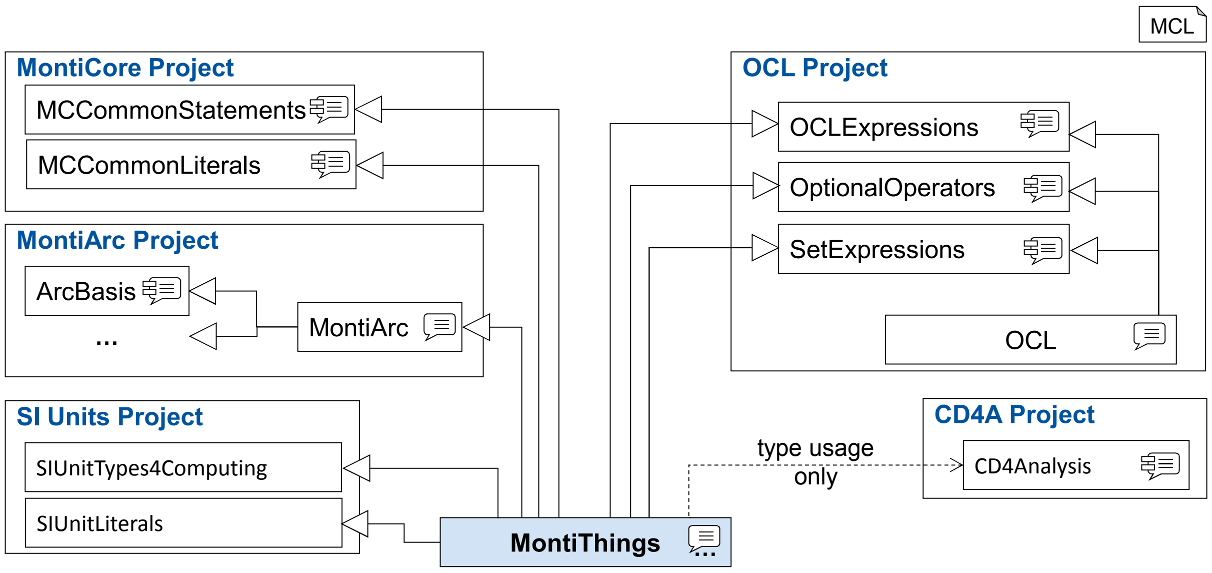


Figure 6. An overview of languages integrated by the MontiThings language (adapted from [49]).

them in the containers. Shared infrastructure such as interfaces to message brokers or the serialization of messages is provided by MontiThings. If the generated code does not meet the user's requirements, it is possible to supplement or overwrite the code with handwritten code using MontiCore's TOP mechanism. In addition to this main language, other languages exist, *e.g.,* for configuring models for specific hardware/software platforms or for defining tests.

The MontiThings language is composed of 46 grammars of the MontiCore project reusing 4371 lines of grammar [49]. Particularly noteworthy is the fact that the combined languages are not just smaller subgrammars, but modeling languages in their own right. Figure 6 shows a reduced overview of MontiThings' grammar reuse. In doing so, MontiThings uses several types of language composition: Language extension (and, thus, language inheritance), language embedding, and language aggregation (*cf.* [42]). Most importantly, MontiThings extends MontiArc [43]. MontiArc is a CnC architecture description language for simulating distributed systems. Consequently, MontiArc provides large parts of the abstract and concrete syntax of MontiThings' CnC language. However, MontiThings' generator is very different from MontiArc's generator, because MontiThings is focused on generating applications intended to be executed on real IoT devices while MontiArc is focused on simulations. In addition to the inherited language elements, MontiThings extends MontiArc with numerous elements, *e.g.,* for error handling [53] to deal with often unreliable sensor input.

MontiThings' type system is mainly based on MontiCore's Java-like type system. For primitive types, MontiThings reuses MontiCore's primitive types through language embedding. Furthermore, MontiThings uses MontiCore's SI unit language via language embedding. This enables modelers to use SI units like primitive types, *e.g.,* define a variable of type `km/h` or `°C`. If two convertible but different units are calculated together (*e.g.,* `km/h` and `m/s`), MontiThings automatically ensures that the units are converted into the same unit. More complex data structures can be defined in class diagrams. The class diagrams are specified in their own files. MontiThings can import the symbols defined in the class diagrams via language aggregation. Using aggregation, loose coupling and separation of concerns can be achieved, resulting in isolated yet synchronized views between the architectural models and their referenced types. Thus, the CD4A language could simply be replaced by another data type language as long as it conforms to MontiCore's type system. For example, if a class diagram defines a type `Photo`, variables, ports, and other elements in MontiThings models can use the type `Photo` if the artifact imports the corresponding class diagram. Besides the type system for variables, MontiThings also reuses MontiArc's type system for specifying component types. Most parts of the type check could be reused 1:1. Only in cases where the combination of languages results in new cases that the type checks of the individual languages cannot handle or languages deviate from MontiCore's defaults, individual manual adjustments have to be made.

The behavior of MontiThings components can be defined using four techniques:

1. Subcomponents
2. A Java-like language
3. Statecharts
4. Handwritten code (C++ or Python)

The ability to instantiate and connect subcomponents is a capability that the MontiThings language inherits from MontiArc. The Java-like behavior language is included in MontiThings through language embedding and is based on MontiCore's MCCommonStatements. Statecharts are another way of defining behavior. Similar to the MCCommonStatements, they are included in MontiThings via language embedding. The embedded languages are augmented with other MontiCore languages such as the OCL for writing boolean expressions. In this regard, MontiCore's common foundation of types and symbols reduces the effort of integrating languages. By providing a common denominator for common symbol types (functions, variables, etc.), languages can be reused largely unchanged, reducing the need to write adapters. Integrating OCL expressions only required us to include the statements once for the whole language, while the concepts applied to multiple locations within the language (*e.g.,* both Pre- and Postconditions, as well as within the statement language). An alternative way of defining behavior is through handwritten code. As always in MontiCore, generated code can be overridden using MontiCore's TOP mechanism. Besides overriding generated classes using C++ code via the TOP mechanism, MontiThings also has the ability to integrate Python code for behavior. In this case, MontiThings serializes data sent via ports using Google Protobuf[4] and exchanges this information with a generated Python wrapper that forwards the data to the handwritten code.

Besides its main language, MontiThings also includes a separate language for configuring components for specific targets. This language acts as a tagging language (*cf.* [40]), adding extra properties to components and ports. For example, the configuration language can be used to define that the code generator should treat a component as a single deployment unit that includes all its subcomponents instead of creating its subcomponents as independent services. This can be used, *e.g.,* to reduce communication overhead if a component is expected to be deployed on the same device. Furthermore, different variants of a component for different target platforms can be defined. For example, if the generated code is expected to be deployed on an Arduino it might require other handwritten code for accessing sensors than code intended to be deployed on a Raspberry Pi.

Moreover, the MontiThings project landscape includes a testing language inherited from MontiCore's sequence diagrams. The testing language uses MontiCore's resolving delegate mechanism to refer to symbols from the MontiThings models under test. It enables users to define white box test cases using sequence diagrams that model the interaction between a component's subcomponents. It is of course also possible to only define in- and outputs of the test case to define a black box test. A code generator independent of MontiThings' main code generator uses the sequence diagrams for C++ code transformations written against the GoogleTest framework.

The configuration and testing languages inherit from MontiThings and MontiArc, respectively. While language aggregation over symbols would have led to a better separation of languages, and external, exchangeable views, inheritance avoided the development effort of importing the symbol table. The disadvantages of this approach are the bad reusability (because of high coupling) and the long compile time of the tagging languages.

[4]Protobuf Documentation: https://protobuf.dev/, Last accessed: 11.04.2023

4.2 Assistive Systems

Assistive systems play an important role in ensuring safety and supporting individuals in a variety of settings, including work [61, 77, 91], driving [87], and daily life activities [5, 60, 64]. To be able to provide human behavior support, an assistive system needs context information [63] as well as behavior data, e.g., via activity recognition systems [58], both previously stored and real-time monitored [46]. After analyzing and reasoning about this information [3, 56], support information is provided either in a situation a person needs it or when she asks for it.

We have investigated which modeling languages are needed to apply a model-driven approach for the engineering of assistive systems and which languages are needed to use models at runtime [7]. We have used the assistive system language family to develop assistive systems to support processes in a smart kitchen as well as processes for manual assembly in production.

For the model-driven approach, we use the MontiGem [2, 38] generator framework. MontiGem was developed to support the model-based engineering of web-based information systems. It uses models in the CD4A language as input to define domain information in the data structures, models in the GUI language [37] to define user interfaces and OCL to define constraints for user input. Out of these languages, it generates the backend, frontend, and database of a web application, in this case, the core of an assistive system. Additionally, we have added hand-written components to handle relevant information during runtime, e.g., to transform data into runtime models, to reason about information, or to create support information. The support information for end users includes full sentences (in the first version in German) as well as additional pictures and acoustic information for each defined task.

To use models at runtime of the assistive system, we have defined a language family for model-based assistive systems (see Figure 7). This includes a `ContextLanguage` to define concrete objects to be used in supported processes and a `TaskLanguage` to describe the processes in a textual way.

As the set-up of assistive systems for a concrete location and tasks is time-consuming, we have developed the `ContextLanguage`. It allows us to define what tools and

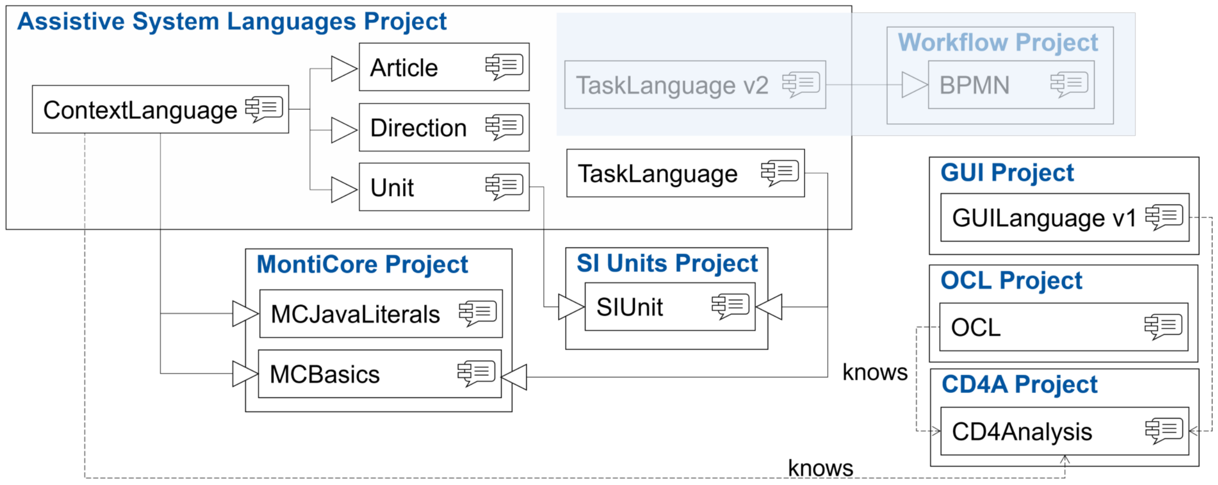
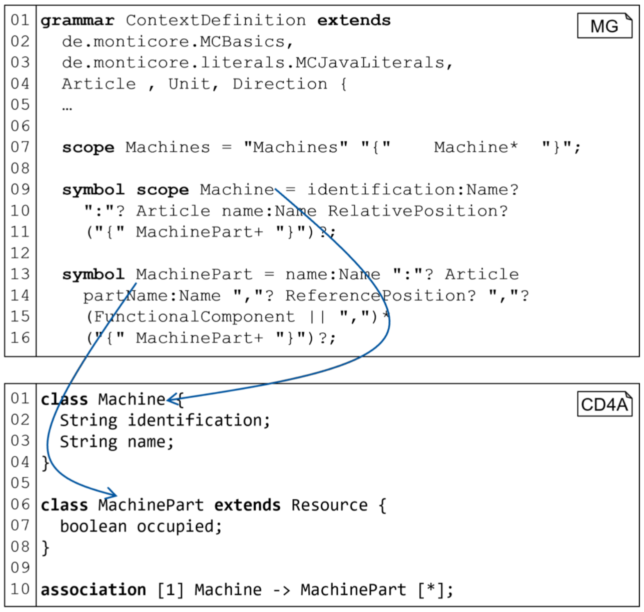


Figure 7. Overview of the Language Family for Assistive Systems

objects to use in supported tasks and where they can be found. More specifically, it allows defining objects in four object groups: Machines, Storages, Utensils, and Items. These groups describe the spatio-temporal and environmental context of a task. As we aim to use this language during runtime, these concepts also have to appear in the CD4A models as part of the concepts describing the domain. Thus, the use of the Context Language during the runtime of the assistive system relies on the existence of certain classes in the CD4A model. Figure 8 shows this relationship with an example. The concepts Machine (ContextLanguage Grammar l.9) and MachinePart (l.13) are defined in the CD4A model. As the CD4A model is used as input for generating the assistive system, we can use ContextLanguage models during runtime and store their information via data access objects in the backend into the database.

The `ContextLanguage` reuses the Java-like comments and names (`MCBasics`) and Java-like number specifications (`MCJavaLiterals`) provided by MontiCore through language embedding. This not only allows the reuse of terms like `2.2d` but also the reuse of MontiCore's type system, *e.g.*, to check whether the range in a `StepWiseComponent` is type safe (see Figure 9).

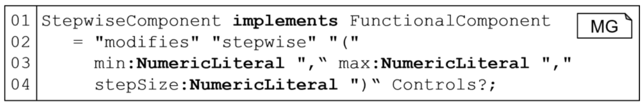
Further reusable parts relevant to the `ContextLanguage` were moved to new language components: The first version of the `Article` language component included a set of German definite articles. The `Direction` language component includes a set of phrases used to define directions relative to a certain place, e.g., left, in front of, or in the middle. To help write a more intuitive model, our languages use numbers in combination with units such as `3 l` for three liters, *e.g.*, to appropriately indicate quantities in a cooking recipe. The MontiCore `SIUnit` language fits most of our needs for the most common SI and SI-derived units [68]. Additionally, the `ContextLanguage`



```
01 grammar ContextDefinition extends
02   de.monticore.MCBasics,
03   de.monticore.literals.MCJavaLiterals,
04   Article , Unit, Direction {
05   …
06
07   scope Machines = "Machines" "{"    Machine*   "}";
08
09   symbol scope Machine = identification:Name?
10     ":"? Article name:Name RelativePosition?
11     ("{" MachinePart+ "}")?;
12
13   symbol MachinePart = name:Name ":"? Article
14     partName:Name ","? ReferencePosition? ","?
15     (FunctionalComponent || ",")*
16     ("{" MachinePart+ "}")?;
```

```
01 class Machine {
02   String identification;
03   String name;
04 }
05
06 class MachinePart extends Resource {
07   boolean occupied;
08 }
09
10 association [1] Machine -> MachinePart [*];
```

Figure 8. Relationship `ContextLanguage` grammar concepts and CD4A model concepts



```
StepwiseComponent implements FunctionalComponent
   = "modifies" "stepwise" "("
     min:NumericLiteral "," max:NumericLiteral ","
     stepSize:NumericLiteral ")" Controls?;
```

Figure 9. Extent of the `ContextLanguage` grammar embedding a MontiCore component grammar

needed to define its own set of `Units`, *e.g.*, `°Celsius` instead of `°C` or `EL` (common German word describing a spoon full of an ingredient). Here, the `Unit` component language inherits from MontiCore's `SIUnit` language adding more informal units and removing others, *e.g.,* m^3. The ContextLanguage embeds these three language components. The concepts in these language components are needed later on to assemble correct German sentences based on model information.

In the generated assistive system, we want to inform the user by providing crucial information needed for the next task. In MontiGem, the web interface is modeled using the GUILanguage v1 which we will use to model user interfaces. Models of this language refer to the data types defined in the corresponding domain model (CD4Analysis) to specify data access in the running system. OCL expressions constrain the domain model. The MontiGem framework is then responsible for generating infrastructure for data transfer and validators for each constraint. In our assistive system, we use one generic GUILanguage v1 model which presents all necessary information to the user for one step of behavior. This model is accompanied by others for domain-specific presentation tasks, e.g., providing overviews, or allowing to change settings.

The TaskLanguage can be used to model human behavior tasks in sequential order. It is especially designed to be as user-friendly as possible with not only a task order but also a natural description of how to perform those. The core elements of such a sequential order are Find tasks to instantiate a findable item, Placing, Filling and Setting tasks to alter items, as well as Waiting and Moving tasks to give an order directly to the user. Here, we again embed MCBasics to use common names and comments, MCJavaLiterals for Java-like number usage. The SIUnit language is again used on multiple occasions, for which we restrict the usage in one place to only allow time units like s or h.

As, *e.g.,* recipes or manuals, describe tasks in sequential order, we have assumed that it is sufficient if the TaskLanguage supports behavior sequences. But more complex processes might require a lot of waiting where other tasks can be done in parallel or might require a (valid) reordering based on personal preferences. Modeling languages like UML activity diagrams or the BPMN standard [69] also allow specifying the parallel ordering of tasks/activities. Since the BPMN also specifically targets human interaction, we will inherit from the textual version of the BPMN standard [25] in the next version of the language, TaskLanguage v2. This allows us to specify more real-world suitable task ordering. For this, we will only allow user tasks (removing, *e.g.,* sending or service tasks) and extend them by the task types identified in the TaskLanguage. We will also need to additionally embed MCJavaLiterals and SIUnit to fit our user-friendly notation. With these steps, we achieve a new language suitable for our needs. Models of the TaskLanguage v2 could then stand for themselves or can be used to generate multiple valid TaskLanguage models. In the generated assistive system, we could reuse a standardized workflow engine to manage our task definitions. To ensure backward compatibility, modeling in TaskLanguage (v1) could then be a starting point for a transformation in a sequential TaskLanguage v2 model.

5 Discussion

For our investigation, we have considered the notion of *viewpoint* in its broadest sense, i.e., as a conceptual means for the model-based description and reasoning about different concerns pertaining to a software system. Our consideration of the notion is thus consistent with other publications in the MDE area [6, 17, 33, 39, 65, 75], and specifically with those at the intersection of MDE and software architecture [22, 23, 31, 47, 71, 73]. Recently, Multi-Paradigm Modeling (MPM), which has its roots in simulation [84], has been discovered to greatly benefit the MDE-based development of cyber-physical systems as it enables to model and subsequently process heterogeneous parts of the system with the most appropriate MDE formalisms and workflows [4]. Hence, we perceive multi-viewpoint modeling and MPM to constitute two sides of the same coin, both aiming to tackle complex system design, development, and operation by the integration of heterogeneous modeling languages. These languages' application eventually results in models that can be analyzed and processed leveraging well-understood MDE techniques such as quality analysis, model transformation, code generation, and simulation [18].

Constructing sophisticated modeling languages and language families in heterogeneous domain use cases delivered a comprehensive set of observations, which composition technique is applicable in particular scenarios. Overall, we summarize our experiences from the two presented case studies into seven potential language engineering scenarios, depicted in Table 1. Please note that the table only reflects the conceptual composition technique and not its implementation. That is, while in MontiCore, extension and inheritance are methodically different executions of the same mechanism, or embedding always incorporates inheritance as well, they are distinguished concerning their assessment.

Adapting a single, already existing language to a use case (S1) requires only inheriting from that language. Applying modifications to the original can either be achieved by conservative extension or via inheritance. The latter also allows for overriding or restricting productions, which, in some cases, might be necessary. However, this leads to the original models not being valid in the context of the modified language anymore. If original models must be retained (S2), only conservative extension is applicable. Overall, these composition techniques are the easiest to apply since constructs are directly adopted from an existing language. In the case of extension, engineers must further methodically care to keep all modifications genuinely conservative. To facilitate this,

Table 1. Suitability assessment of the investigated language composition techniques concerning different modeling scenarios (● = suitable, ⊙ = partially suitable, ○ = not suitable)

Scenario / Use Case	**Inheritance**	**Extension**	**Embedding**	**Aggregation**
(S1) Modifying a language, tailoring it to a specific use case	●	⊙	○	○
(S2) Extending a language to a use case while maintaining the integrity of the original models	⊙	●	○	○
(S3) Combining multiple language components into a modeling language	○	○	●	○
(S4) Combining modeling languages into a language family	○	○	●	●
(S5) Constructing huge languages with different constituents	○	○	●	●
(S6) Constructing a language or language family with heterogeneous parts for interdisciplinary use	○	○	⊙	●
(S7) Modularization of model artifacts	○	○	○	●

MontiCore provides a warning when this is not the case. As these scenarios only relate to a single DSL, embedding or aggregation are unsuitable.

For scenarios that involve employing multiple languages or language components, single inheritance (and extension, respectively) is not applicable. Evolving DSLs and tailoring them towards more sophisticated applications usually implies including various modeling techniques. Language embedding combines the constituents of multiple languages into a single one, connecting the different constructs. This is especially effective when the integrated DSLs share common interfaces, enabling the automatic embedding with nearly no glue code necessary. Thus, although requiring intricate knowledge of the involved components, embedding can still be facile when prepared well. For integrating (potentially incomplete) language components (S3), e.g., MontiThings comprising various literals, statements, and expressions, language embedding is the only applicable technique. As aggregation establishes a loose coupling only, this technique does not complete the components into a fully functional DSL. On the other hand, when integrating already functional languages into a family (S4), both embedding and aggregation might be applicable. The choice mainly depends on the respective modeling goal. An integrated view can be supportive in scenarios where the same domain experts create all aspects of models (e.g., in the context language of assistive systems). On the other hand, splitting dependent constructs of a large language (S5) into separate artifacts (which is automatically achieved by language aggregation) supports the organization and structuring of larger modeling projects.

While opting for language aggregation over embedding can positively impact structuring, this effect becomes even more apparent for interdisciplinary modeling teams working together on a product (S6). Here, the composed yet separated artifacts represent different domain-specific views of the system under development. This way, domain experts can contribute without getting distracted by the information of other modeling views. This observation results from both case studies as they employ class diagrams as separated artifacts for delivering type information. Finally, language aggregation for modularizing modeling artifacts (S7) is, even while not always necessary, a technique that language engineers should consider to foster a suitable modeling project structure and avoid model cluttering.

While all composition techniques are essential, the general impression is that the more sophisticated a language becomes, the more likely it is to apply a more elaborate approach. Therefore, minor DSL modifications usually employ inheritance or extension and keep the scope within a single domain or use case. Furthermore, reusing multiple concepts requires embedding constituents of different languages appropriately. Reasons such as structuring logical units of big models into artifacts or engineering whole language families incorporating multiple viewpoints of heterogeneous domains, both following the notion of separation of concerns, require language aggregation.

With this in mind, language aggregation is also the only composition technique natively supporting multi-viewpoint modeling. One of the main challenges in multi-viewpoint modeling is maintaining consistency between the individual views [11]. Aggregation automatically fulfills this requirement as models are organized in different artifacts. Thus, each artifact represents a separate, integrated view of the overall system automatically synchronized with other domain models' elements.

As usual for experience reports, our observations are subject to threats of validity, especially concerning generalizability. We have conducted our case studies in the technological space of MontiCore and are therefore tied to its capabilities and restrictions. However, we intentionally have chosen this ecosystem as it is specifically tailored for language composition, and the proposed composition techniques are state-of-the-art. Additionally, the presented approaches can conceptually, at least partially, be found in other frameworks as well, such as MPS or Xtext. This mitigates the threat to generalizability.

6 Conclusion

In this paper, we studied the composition of heterogeneous modeling languages which were originally developed independently but yet address different concerns in the same domain. While our investigation shows that the composition of such languages is both sensible and feasible, we also found that their composition requires different techniques whose application depends on a language's use case in the envisioned composition.

In total, we considered four composition techniques, namely inheritance, extension, embedding, and aggregation, and employed them to derive integrated, non-trivial language families for two distinct case studies concerning the engineering of cyber-physical systems for IoT and assistive systems. As a result, these language families are not only practically applicable for the integrated modeling of different viewpoints on systems of the mentioned kinds but also enabled us to assess the suitability of the aforementioned composition techniques. In this context, a major finding is that embedding and aggregation are indispensable for the composition of modeling language families, and even complement each other in a natural fashion. Finally, language aggregation automatically supports establishing different viewpoints on the model level for interdisciplinary modeling tasks, making it a considerable technique for realizing multi-viewpoint modeling scenarios.

Further evolvement of the language families, e.g., to include DSLs for describing requirements or goals, and the development of language families for other domains will provide additional examples to evaluate the composition techniques.

Acknowledgements

Funded by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2023 Internet of Production - 390621612. Website: https://www.iop.rwth-aachen.de.

References

[1] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. 2017. Modeling Robotics Software Architectures with Modular Model Transformations. *Journal of Software Engineering for Robotics* 8, 1 (2017).

[2] Kai Adam, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. 2020. Enterprise Information Systems in Academia and Practice: Lessons learned from a MBSE Project. In *40 Years EMISA: Digital Ecosystems of the Future (EMISA'19) (LNI, Vol. P-304)*. GI, 59–66.

[3] Fadi Al Machot, Heinrich C. Mayr, and Judith Michael. 2014. Behavior Modeling and Reasoning for Ambient Support: HCM-L Modeler. In *Int. Conf. on Industrial, Engineering & Other Applications of Applied Intelligent Systems (IEA-AIE 2014) (LNAI)*. https://doi.org/10.1007/978-3-319-07467-2_41

[4] Moussa Amrani, Dominique Blouin, Robert Heinrich, Arend Rensink, Hans Vangheluwe, and Andreas Wortmann. 2021. Multi-paradigm modelling for cyber-physical systems: a descriptive framework. *Software and Systems Modeling* 20, 3 (2021), 611–639. https://doi.org/10.1007/s10270-021-00876-z

[5] Prashanti Angara, Miguel Jiménez, Kirti Agarwal, Harshit Jain, Roshni Jain, Ulrike Stege, Sudhakar Ganti, Hausi A. Müller, and Joanna W. Ng. 2017. Foodie Fooderson a Conversational Agent for the Smart Kitchen. In *27th Annual Int. Conf. on Computer Science and Software Engineering (CASCON '17)*. IBM, 247–253.

[6] Adil Anwar, Sophie Ebersold, Bernard Coulette, Mahmoud Nassar, Abdelaziz Kriouile, et al. 2010. A Rule-Driven Approach for composing Viewpoint-oriented Models. *Journal of Object Technology* 9, 2 (2010). https://doi.org/doi:10.5381/jot.2010.9.2.a1

[7] Nelly Bencomo, Sebastian Götz, and Hui Song. 2019. Models@run.time: a guided tour of the state of the art and research challenges. *Software and Systems Modeling* 18, 5 (2019), 3049–3082. https://doi.org/10.1007/s10270-018-00712-x

[8] Lorenzo Bettini. 2016. *Implementing Domain-Specific Languages with Xtext and Xtend* (second ed.). Packt Publishing.

[9] Mathias Blumreiter, Joel Greenyer, Francisco Javier Chiyah Garcia, Verena Klös, Maike Schwammberger, Christoph Sommer, Andreas Vogelsang, and Andreas Wortmann. 2021. Towards Self-Explainable Cyber-Physical Systems. In *22nd Int. Conf. on Model Driven Engineering Languages and Systems (MODELS '19)*. IEEE Press, 543–548. https://doi.org/10.1109/MODELS-C.2019.00084

[10] Philipp Brauner, Manuela Dalibor, Matthias Jarke, Ike Kunze, István Koren, Gerhard Lakemeyer, Martin Liebenberg, Judith Michael, Jan Pennekamp, Christoph Quix, Bernhard Rumpe, Wil van der Aalst, Klaus Wehrle, Andreas Wortmann, and Martina Ziefle. 2022. A Computer Science Perspective on Digital Transformation in Production. *Journal ACM Transactions on Internet of Things* 3 (2022). https://doi.org/10.1145/3502265

[11] Hugo Bruneliere, Erik Burger, Jordi Cabot, and Manuel Wimmer. 2019. A feature-based survey of model view approaches. *Software & Systems Modeling* 18 (2019), 1931–1952. https://doi.org/10.1007/s10270-017-0622-9

[12] Arvid Butting, Robert Eikermann, Katrin Hölldobler, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. 2020. A Library of Literals, Expressions, Types, and Statements for Compositional Language Design. *Journal of Object Technology* 19, 3 (2020), 3:1–16. https://doi.org/10.5381/jot.2020.19.3.a4

[13] Arvid Butting, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. 2021. Compositional Modelling Languages with Analytics and Construction Infrastructures Based on Object-Oriented Techniques - The MontiCore Approach. In *Composing Model-Based Analysis Tools*. Springer, 217–234. https://doi.org/10.1007/978-3-030-81915-6_10

[14] Arvid Butting, Judith Michael, and Bernhard Rumpe. 2022. Language Composition via Kind-Typed Symbol Tables. *Journal of Object Technology* 21 (October 2022), 4:1–13. https://doi.org/10.5381/jot.2022.21.4.a5

[15] Giuseppina Lucia Casalaro, Giulio Cattivera, Federico Ciccozzi, Ivano Malavolta, Andreas Wortmann, and Patrizio Pelliccione. 2022. Model-driven engineering for mobile robotic systems: a systematic mapping study. *Software and Systems Modeling* 21, 1 (2022), 19–49. https://doi.org/10.1007/s10270-021-00908-8

[16] Walter Cazzola and Edoardo Vacchi. 2013. Neverlang 2 – Componentised Language Development for the JVM. In *Software Composition*. Springer, 17–32. https://doi.org/10.1007/978-3-642-39614-4_2

[17] Federico Ciccozzi and Romina Spalazzese. 2017. MDE4IoT: Supporting the Internet of Things with Model-Driven Engineering. In *Intelligent Distributed Computing X*. Springer, 67–76.

[18] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, James Steel, and Didier Vojtisek. 2016. *Engineering Modeling Languages*. Chapman & Hall.

[19] Manuela Dalibor, Nico Jansen, Bernhard Rumpe, Louis Wachtmeister, and Andreas Wortmann. 2019. Model-Driven Systems Engineering for Virtual Product Design. In *Proc. of MODELS 2019. WS MPM4CPS*. IEEE, 430–435. https://doi.org/10.1109/MODELS-C.2019.00069

[20] Istvan David, Kousar Aslam, Sogol Faridmoayer, Ivano Malavolta, Eugene Syriani, and Patricia Lago. 2021. Collaborative Model-Driven Software Engineering: A Systematic Update. In *ACM/IEEE 24th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS)*.

[21] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. 2015. Melange: A Meta-Language for Modular and Reusable Development of DSLs. In *Int. Conf. on Software Language Engineering (SLE 2015)*. ACM, 25–36. https://doi.org/10.1145/2814251.2814252

[22] Elif Demirli and Bedir Tekinerdogan. 2011. Software Language Engineering of Architectural Viewpoints. In *Software Architecture*. Springer. https://doi.org/10.1007/978-3-642-23798-0_36

[23] Davide Di Ruscio, Ivano Malavolta, Henry Muccini, Patrizio Pelliccione, and Alfonso Pierantonio. 2010. Developing next Generation ADLs through MDE Techniques. In *32nd Int. Conf. on Software Engineering (ICSE ’10)*. ACM, 85–94. https://doi.org/10.1145/1806799.1806816

[24] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Evgeny Kusmenko, Matthias Markthaler, Philipp Orth, Karin Samira Salman, Johannes Richenhagen, Bernhard Rumpe, Christoph Schulze, Michael Wenckstern, and Andreas Wortmann. 2019. SMArDT modeling for automotive software testing. *Journal on Software: Practice and Experience* 49, 2 (February 2019), 301–328.

[25] Imke Drave, Judith Michael, Erik Müller, Bernhard Rumpe, and Simon Varga. 2022. Model-Driven Engineering of Process-Aware Information Systems. *Springer Nature Computer Science Journal* 3 (2022). https://doi.org/10.1007/s42979-022-01334-3

[26] Florian Drux, Nico Jansen, and Bernhard Rumpe. 2022. A Catalog of Design Patterns for Compositional Language Engineering. *Journal of Object Technology* 21, 4 (October 2022), 4:1–13.

[27] Robert Eikermann, Markus Look, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. 2017. Architecting Cloud Services for the Digital me in a Privacy-Aware Environment. In *Software Architecture for Big Data and the Cloud*. Elsevier Science & Technology, Chapter 12.

[28] Johan Eker, Jorn W Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Sonia Sachs, Yuhong Xiong, and Stephen Neuendorffer. 2003. Taming heterogeneity - the Ptolemy approach. *Proc. of the IEEE* 91, 1 (2003), 127–144. https://doi.org/10.1109/JPROC.2002.805829

[29] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, et al. 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures* 44 (2015), 24–47. https://doi.org/10.1016/j.cl.2015.08.007 SI on the 6th and 7th Int. Conf. on Software Language Engineering (SLE 2013, SLE 2014).

[30] Moritz Eysholdt and Heiko Behrens. 2010. Xtext: Implement Your Language Faster than the Quick and Dirty Way. In *Int. Conf. Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA ’10)*. ACM. https://doi.org/10.1145/1869542.1869625

[31] J.-M. Favre. 2004. CaCOphoNy: metamodel-driven software architecture reconstruction. In *11th Working Conf. on Reverse Engineering*.

[32] Kevin Feichtinger, Kristof Meixner, Felix Rinker, István Koren, Holger Eichelberger, Tonja Heinemann, Jörg Holtmann, Marco Konersmann, Judith Michael, Eva-Maria Neumann, Jérôme Pfeiffer, Rick Rabiser, Matthias Riebisch, and Klaus Schmid. 2022. Industry Voices on Software Engineering Challenges in Cyber-Physical Production Systems Engineering. In *27th Int. Conf. on Emerging Techn. and Factory Automation (ETFA)*. IEEE. https://doi.org/10.1109/ETFA52439.2022.9921568

[33] Robert France and Bernhard Rumpe. 2007. Model-driven Development of Complex Software: A Research Roadmap. *Future of Software Engineering (FOSE ’07)* (May 2007), 37–54.

[34] Ulrich Frank. 2002. Multi-perspective enterprise modeling (MEMO) conceptual framework and modeling languages. In *Proc. of the 35th Annual Hawaii Int. Conf. on System Sciences*. 1258–1267.

[35] Damian Frölich and L. Thomas van Binsbergen. 2022. ICola: A Compositional Meta-Language with Support for Incremental Language Development. In *15th Int. Conf. on Software Language Engineering (SLE 2022)*. ACM, 202–215. https://doi.org/10.1145/3567512.3567529

[36] Shan Fur, Malte Heithoff, Judith Michael, Lukas Netz, Jérôme Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. 2023. *Sustainable Digital Twin Engineering for the Internet of Production*. Springer Nature Singapore, 101–121. https://doi.org/10.1007/978-981-99-0252-1_4

[37] Arkadii Gerasimov, Judith Michael, Lukas Netz, and Bernhard Rumpe. 2021. Agile Generator-Based GUI Modeling for Information Systems. In *Modelling to Program (M2P)*. Springer, 113–126.

[38] Arkadii Gerasimov, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. 2020. Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise Information Systems. In *25th Americas Conf. on Information Systems (AMCIS) (AISeL)*. AIS.

[39] Javier González-Huerta, Emilio Insfran, and Silvia Abrahão. 2012. A Multimodel for Integrating Quality Assessment in Model-Driven Engineering. In *8th Int. Conf. on the Quality of Information and Communications Technology*. 251–254. https://doi.org/10.1109/QUATIC.2012.14

[40] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. 2015. Engineering Tagging Languages for DSLs. In *Conf. on Model Driven Engineering Languages and Systems (MODELS’15)*. ACM/IEEE.

[41] Hans Grönniger and Bernhard Rumpe. 2011. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems (LNCS 6662)*. Springer, 17–32.

[42] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. 2015. Composition of Heterogeneous Modeling Languages. In *Model-Driven Engineering and Software Development*, Vol. 580. Springer. https://doi.org/10.1007/978-3-319-27869-8_3

[43] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. 2012. *MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems*. Technical Report AIB-2012-03. RWTH Aachen University.

[44] David Harel and Bernhard Rumpe. 2004. Meaningful Modeling: What’s the Semantics of "Semantics"? *IEEE Computer Journal* 37, 10 (2004).

[45] Katrin Hölldobler, Oliver Kautz, and Bernhard Rumpe. 2021. *MontiCore Language Workbench and Library Handbook: Edition 2021*. Shaker Verlag. https://doi.org/10.2370/9783844080100

[46] Katrin Hölldobler, Judith Michael, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. 2019. Innovations in Model-based Software and Systems Engineering. *Journal of Object Technology* 18, 1 (2019).

[47] Eric Jouenne and Véronique Normand. 2005. Tailoring IEEE 1471 for MDE support. In *UML Modeling Languages and Applications: Satellite Activities*. Springer, 163–174.

[48] Gerti Kappel, Elisabeth Kapsammer, Horst Kargl, Gerhard Kramler, Thomas Reiter, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. 2006. Lifting Metamodels to Ontologies: A Step to the Semantic Integration of Modeling Languages. In *Model Driven Engineering Languages and Systems*. Springer, 528–542.

[49] Jörg Christian Kirchhof, Anno Kleiss, Judith Michael, Bernhard Rumpe, and Andreas Wortmann. 2022. Efficiently Engineering IoT Architecture Languages - An Experience Report (Poster). In *STAF 2022 WS Proc.: 2nd Int. WS on MDE for Smart IoT Systems (MeSS’22)*, Vol. 3250. CEUR-WS.

[50] Jörg Christian Kirchhof, Anno Kleiss, Bernhard Rumpe, David Schmalzing, Philipp Schneider, and Andreas Wortmann. 2022. Model-driven Self-adaptive Deployment of Internet of Things Applications with Automated Modification Proposals. *Transactions on Internet of Things* 3 (2022), 1–30. https://doi.org/10.1145/3549553

[51] Jörg Chrisitian Kirchhof, Lukas Malcher, Judith Michael, Bernhard Rumpe, and Andreas Wortmann. 2022. Web-Based Tracing for Model-Driven Applications. In *48th Euromicro Conf. on Software Engineering and Advanced Applications (SEAA)*. IEEE, 374–381.

[52] Jörg Christian Kirchhof, Lukas Malcher, and Bernhard Rumpe. 2021. Understanding and Improving Model-Driven IoT Systems through Accompanying Digital Twins. In *Proc. of the 20th Int. Conf. on Generative Programming (GPCE 21)*. ACM, 197–209.

[53] Jörg Christian Kirchhof, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. 2022. MontiThings: Model-driven Development and Deployment of Reliable IoT Applications. *Journal of Systems and Software (JSS)* 183 (2022). https://doi.org/10.1016/j.jss.2021.111087

[54] Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. 2014. A Metamodel Family for Role-Based Modeling and Programming Languages. In *Software Language Engineering*. Springer.

[55] Manuel Leduc, Thomas Degueule, and Benoit Combemale. 2018. Modular Language Composition for the Masses. In *Proc. of the 11th Int. Conf. on Software Language Engineering (SLE 2018)*. ACM, 47–59.

[56] Po-Sheng Li, Alan Liu, and Pei-Chuan Zhou. 2014. Context reasoning for smart homes using case-based reasoning. In *18th Int. Symp. on Consumer Electronics (ISCE'14)*. IEEE.

[57] Jörg Liebig, Rolf Daniel, and Sven Apel. 2013. Feature-Oriented Language Families: A Case Study. In *7th Int. WS on Variability Modelling of Software-Intensive Systems (VaMoS '13)*. ACM.

[58] Fadi Al Machot, Heinrich C. Mayr, and Suneth Ranasinghe. 2016. A windowing approach for activity recognition in sensor data streams. In *8th Int. Conf. on Ubiquitous and Future Networks, (ICUFN 2016)*. IEEE.

[59] Atif Mashkoor, Alexander Egyed, Robert Wille, and Sebastian Stock. 2022. Model-driven engineering of safety and security software systems: A systematic mapping study and future research directions. *Journal of Software: Evolution and Process* (2022), e2457. https://doi.org/10.1002/smr.2457

[60] Apostolos Meliones and Stavros Maidonis. 2020. DALÍ: A Digital Assistant for the Elderly and Visually Impaired Using Alexa Speech Interaction and TV Display. In *13th ACM Int. Conf. on PErvasive Technologies Related to Assistive Env. (PETRA)*. ACM, Article 37, 9 pages.

[61] Judith Michael. 2022. A Vision Towards Generated Assistive Systems for Supporting Human Interactions in Production. In *Modellierung 2022 Satellite Events*. GI, 150–153. https://doi.org/10.18420/modellierung2022ws-019

[62] Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. 2019. Towards Privacy-Preserving IoT Systems Using Model Driven Engineering. In *Proc. of MODELS 2019. WS MDE4IoT*. CEUR WS, 595–614.

[63] Judith Michael and Claudia Steinberger. 2017. Context Modeling for Active Assistance. In *ER Forum 2017 and ER 2017 Demo Track co-located with 36th Int. Conf. on Conceptual Modelling (ER 2017)*.

[64] Judith Michael, Claudia Steinberger, Vladimir A. Shekhovtsov, Fadi Al Machot, Suneth Ranasinghe, and Gert Morak. 2018. The HBMS Story - Past and Future of an Active Assistance Approach. *Enterprise Modelling and Information Systems Architectures - Int. Journal of Conceptual Modeling* 13 (2018), 345–370. https://doi.org/10.18417/emisa.si.hcm.26

[65] Parastoo Mohagheghi, Wasif Gilani, Alin Stefanescu, Miguel A. Fernandez, Bjørn Nordmoen, and Mathias Fritzsche. 2013. Where does model-driven engineering help? Experiences from three industrial cases. *Software & Systems Modeling* 12, 3 (2013), 619–639. https://doi.org/10.1007/s10270-011-0219-7

[66] Mustafa Abshir Mohamed, Moharram Challenger, and Geylani Kardas. 2020. Applications of model-driven engineering in cyber-physical systems: A systematic mapping study. *Journal of Computer Languages* 59 (2020), 100972. https://doi.org/10.1016/j.cola.2020.100972

[67] Brice Morin, Nicolas Harrand, and Franck Fleurey. 2017. Model-Based Software Engineering to Tame the IoT Jungle. *IEEE Software* 34, 1 (January 2017), 30–36. https://doi.org/10.1109/MS.2017.11

[68] David B Newell, Eite Tiesinga, et al. 2019. The international system of units (SI). *NIST Special Publication* 330 (2019), 1–138.

[69] OMG. 2013. *Business Process Model and Notation (BPMN), Version 2.0.2.* Technical Report. Object Management Group.

[70] Václav Pech. 2021. *JetBrains MPS: Why Modern Language Workbenches Matter.* Springer, 1–22. https://doi.org/10.1007/978-3-030-73758-0_1

[71] Carlos Peña and Jorge Villalobos. 2010. An MDE Approach to Design Enterprise Architecture Viewpoints. In *12th Conf. on Commerce and Enterprise Computing*. IEEE. https://doi.org/10.1109/CEC.2010.25

[72] Florian Rademacher, Sabine Sachweh, and Albert Zündorf. 2018. Towards a UML Profile for Domain-Driven Design of Microservice Architectures. In *Software Engineering and Formal Methods*. Springer.

[73] Florian Rademacher, Jonas Sorgalla, Sabine Sachweh, and Albert Zündorf. 2019. Viewpoint-Specific Model-Driven Microservice Development with Interlinked Modeling Languages. In *IEEE Int. Conf. on Service-Oriented System Engineering (SOSE)*. 57–5709. https://doi.org/10.1109/SOSE.2019.00018

[74] Florian Rademacher, Jonas Sorgalla, Philip Wizenty, Sabine Sachweh, and Albert Zündorf. 2020. Graphical and Textual Model-Driven Microservice Development. In *Microservices: Science and Engineering*. Springer, 147–179. https://doi.org/10.1007/978-3-030-31646-4_7

[75] Alberto Rodrigues da Silva. 2015. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures* 43 (2015), 139–155.

[76] Davide Di Ruscio, Ivano Malavolta, and Patrizio Pelliccione. 2014. A Family of Domain-Specific Languages for Specifying Civilian Missions of Multi-Robot Systems. In *1st Int. WS on Model-Driven Robot Software Engineering (MORSE) Co-Located with STAF 2014*. CEUR-WS, 22–31.

[77] Stefan Rüther, Thomas Hermann, Maik Mracek, Stefan Kopp, and Jochen Steil. 2013. An Assistance System for Guiding Workers in Central Sterilization Supply Departments. In *6th Int. Conf. on PErvasive Technologies Related to Assistive Env. (PETRA '13)*. ACM.

[78] Johannes Sametinger. 1997. *Software engineering with reusable components.* Springer Science & Business Media.

[79] Jesús Sánchez Cuadrado. 2012. Towards a Family of Model Transformation Languages. In *Theory and Practice of Model Transformations*. Springer, 176–191. https://doi.org/10.1007/978-3-642-30476-7_12

[80] Thomas A. Standish. 1984. An Essay on Software Reuse. *IEEE Transactions on Software Engineering* SE-10, 5 (1984), 494–497.

[81] Antero Taivalsaari and Tommi Mikkonen. 2017. A Roadmap to the Programmable World: Software Challenges in the IoT Era. *IEEE Software* 34, 1 (Jan 2017), 72–80. https://doi.org/10.1109/MS.2017.26

[82] Chouki Tibermacine, Régis Fleurquin, and Salah Sadou. 2010. A family of languages for architecture constraint specification. *Journal of Systems and Software* 83, 5 (2010), 815–831.

[83] Juha-Pekka Tolvanen and Steven Kelly. 2009. MetaEdit+: Defining and Using Integrated Domain-Specific Modeling Languages. In *Proc. of the 24th Conf. Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, 819–820.

[84] Hans L Vangheluwe and G Vansteenkiste. 1996. A multi-paradigm modeling and simulation methodology. In *Simulation in Industry*.

[85] Markus Voelter. 2013. *Language and IDE Modularization and Composition with MPS.* Springer, 383–430. https://doi.org/10.1007/978-3-642-35992-7_11

[86] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, Simon Helsen, and Krzysztof Czarnecki. 2013. *Model-Driven Software Development: Technology, Engineering, Management.* Wiley.

[87] Marco Wagner, Dieter Zöbel, and Ansgar Meroth. 2013. Model-driven development of SOA-based driver assistance systems. *ACM SIGBED Review* 10, 1 (2013), 37–42. https://doi.org/10.1145/2492385.2492392

[88] Niklaus Wirth. 1996. Extended Backus-Naur Form (EBNF). *ISO/IEC* 14977, 2996 (1996), 2–21.

[89] Enes Yigitbas, Ivan Jovanovikj, Kai Biermeier, Stefan Sauer, and Gregor Engels. 2020. Integrated model-driven development of self-adaptive user interfaces. *Software and Systems Modeling* 19, 5 (2020), 1057–1081.

[90] Steffen Zschaler, Pablo Sánchez, João Santos, Mauricio Alférez, Awais Rashid, Lidia Fuentes, Ana Moreira, João Araújo, and Uirá Kulesza. 2010. VML* – A Family of Languages for Variability Management in Software Product Lines. In *Software Language Engineering*. Springer.

[91] Doruk Şahinel, Cem Akpolat, O. Can Görür, Fikret Sivrikaya, and Sahin Albayrak. 2021. Human modeling and interaction in cyber-physical systems: A reference framework. *Journal of Manufacturing Systems* 59 (2021), 367–385. https://doi.org/10.1016/j.jmsy.2021.03.002

A Low-Code Platform for Systematic Component-Oriented Language Composition

Jérôme Pfeiffer*
jerome.pfeiffer@isw.uni-stuttgart.de
University of Stuttgart
Germany

Andreas Wortmann
andreas.wortmann@isw.uni-stuttgart.de
University of Stuttgart
Germany

Abstract

Low-code platforms have gained popularity for accelerating complex software engineering tasks through visual interfaces and pre-built components. Software language engineering, specifically language composition, is such a complex task requiring expertise in composition mechanisms and language workbenches including multi-dimensional language constituents (syntax and semantics). This paper presents an extensible low-code platform with a graphical web-based interface for language composition. It enables composition by using language components, facilitating systematic composition within language families promoting reuse and streamlining the management, composition, and derivation of domain-specific languages.

CCS Concepts: • **Software and its engineering** → **Reusability**; **Software notations and tools**.

Keywords: Software language engineering, Language composition, Language components, Low-code

ACM Reference Format:
Jérôme Pfeiffer and Andreas Wortmann. 2023. A Low-Code Platform for Systematic Component-Oriented Language Composition. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3623476.3623516

1 Introduction and Motivation

In recent years, the demand for efficient software development processes has led to the emergence and widespread adoption of low-code development platforms [1]. These platforms provide visual interfaces and pre-built components that accelerate the development of complex applications [10]. Among the intricate aspects of software development, software language engineering and composition play a vital role in achieving effective and customizable solutions, e.g., in the domain of digital twins [4]. Software language engineering involves designing and implementing domain-specific languages, while language composition combines languages. Understanding composition mechanisms and language workbenches is essential [7]. Although web-based language workbenches exist [11], limited reuse for multi-dimensional languages remains a challenge. To address this, we introduce a low-code platform based on a method for black-box language composition using language components that encompass syntax and semantics [3]. We refer to this method for systematic component-oriented language reuse as SCOLaR in the following. The platform allows language engineers to derive language components from existing projects and systematically compose them within a language family. By selecting and combining language features within the platform's language family, language engineers can create tailored languages that cater to specific application domains. Through this research, we aim to contribute to the advancement of low-code development methodologies and empower language engineers with a powerful tool to create and reuse languages in a more intuitive and efficient manner.

*The authors were supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) [grant number 441207927].

2 Systematic Component-Oriented Language Reuse

The low-code platform is grounded in the concepts of SCOLaR that uses language components encompassing the constituents of language definitions in the language workbench MontiCore [6]. Language components can be reused by their interface in language families.

2.1 Language Components

Language components [3] provide the three essential language definition constituents: (1) syntax, (2) well-formedness rules, and (3) code generators, realizing the semantic mapping between the problem and the solution domain, that are exposed by extensions in their interface. SCOLaR differentiates between required and provided extensions. Provided

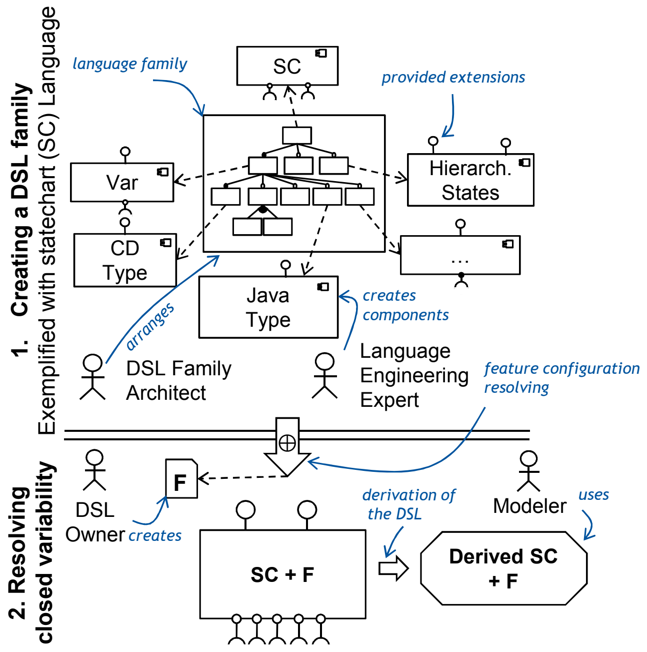


Figure 1. The systematic language composition process exemplified with a language family for statecharts.

extensions offer DSL functionality to be reused by other components. Required extensions make missing functionality of a DSL component explicit and can be either optional or mandatory. Provided and required extensions can reference productions of the grammar or a generator for a specific grammar production. Well-formedness rules are contained in sets that can act as both provided and required extensions at the same time.

2.2 Systematic Composition with Language Families

Language family architects arrange language components into a feature model representing a family of DSLs (cf. Figure 1). In this feature model, each feature either is related to a language component or is an abstract feature [9] for logical grouping. Through this relation, the language family architect decides how the components will be composed when their related features are selected. Once the language family architect completes the language family, DSL owners, who are experts of the application domains derive a suitable DSL for their application domain, by selecting appropriate features from the family in a feature configuration. The composition of two DSL components is the directed application of bindings between these components. Currently, SCOLaR supports the composition operators embedding and aggregation [8]. For the composition the provided extensions of one component are bound to required extensions of another component. The composition includes two main activities: (1) Composing the components' interfaces; and (2) Composition of the comprised language definition constituents (grammars, well-formedness rules, code generators).

3 The Low-Code Platform for Language Composition

The SCOLaR low-code platform is designed around the SCOLaR process (cf. Figure 1) and provides a graphical web-based environment to support it. This section outlines the essential requirements that the platform must meet and then delves into the workflow of the SCOLaR low-code platform.

3.1 Requirements

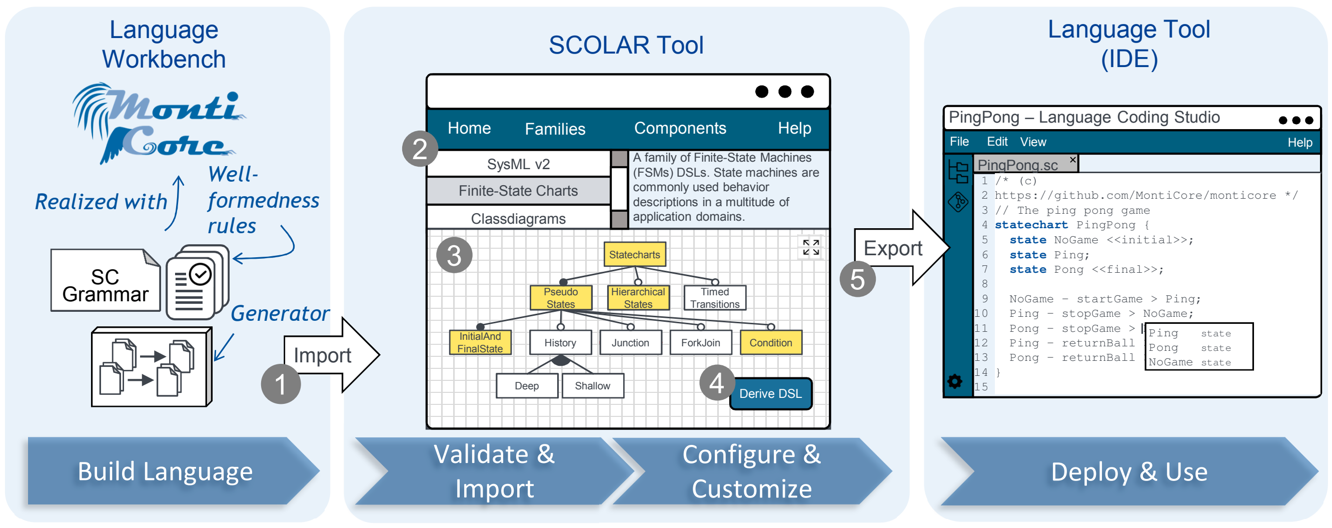
To transform the SCOLaR method from a conceptual method into a user-friendly low-code platform accessible to language architects, we have created an enhanced version of the SCOLaR process, illustrated in Figure 2. The numbers in Figure 2 refer to the requirements we derived for the low-code platform:

Req 1: Language engineers can continue developing languages in their language workbench together with the associated technology-specific artifacts. Hence, existing language projects can be imported and a language component representation is derived automatically.
Req 2: Imported language projects and created language families should be persisted, e.g., in a database.
Req 3: The low-code platform enables DSL Family Architects to create language families.
Req 4: DSL owners can configure existing language families and derive new languages by composition.
Req 5: The composed languages should be exportable for deployment.

3.2 Workflow of the Low-Code Platform

The workflow of the SCOLaR low-code platform can be divided into three steps (cf. Figure 2). Firstly, languages are constructed within a language workbench and subsequently imported into the platform. However, the only language workbench that is supported to this date is MontiCore [6]. Once imported, these languages can be reused within a language family. The configuration of the language family leads to the creation of a new language component, which includes a composed language project that can be downloaded.

3.2.1 Automatic Derivation of DSL Components. Creating a language component model is tedious and error-prone. However, in pursuit of our low-code platform's objective to empower users to compose languages with minimal manual coding, we have developed a derivation function that automatically generates language components from existing MontiCore language projects. After the automatic derivation of a language component, the low-code platform enables the customization of this component to restrict the provided extensions for generator and grammar productions. After the



A family of Finite-State Machines (FSMs) DSLs. State machines are commonly used behavior descriptions in a multitude of application domains.

```
/* (c)
https://github.com/MontiCore/monticore */
// The ping pong game
statechart PingPong {
  state NoGame <<initial>>;
  state Ping;
  state Pong <<final>>;

  NoGame - startGame > Ping;
  Ping - stopGame > NoGame;
  Pong - stopGame > |
  Ping - returnBall
  Pong - returnBall
}

```

```
Ping    state
Pong    state
NoGame  state
```

Figure 2. The systematic language composition process including the development of a language in a language workbench, the import into the SCOLaR tool, the selection of language features, and the derivation of the configured language variant.

import and customization step, the components are ready to be reused in language families. The following describes how the automatic derivation of different language component constituents is realized.

Grammar. The grammar reference is derived directly from the grammar name and its package specified in the project's grammar directory. Within this grammar, we identify that each production rule with a right-hand side serves as a provided grammar extension. Each production rule with the keyword `interface`, indicating that implementation is open for extension, becomes a required extension in the language component.

Well-formedness Rule Sets. The well-formedness rule sets are linked to particular grammar production rules. MontiCore provides infrastructure for validating model well-formedness, including checkers that allow developers to register specific well-formedness rules. In the language component, a set is defined for each checker, which precisely matches the rules registered within the corresponding checker.

Generator. In order to derive generator extensions, it is expected that generators are constructed in accordance with the concept described in [2]. This entails that generators should have explicit product and producer interfaces.

3.2.2 Creating a Language Family.

The low-code platform offers a dedicated language family workbench for composing language components. Within this workbench, users have the option to start with a blank canvas or modify an existing language family. Using a side menu, the user can add or remove features of the language family. Each feature is characterized by a name and a type, i.e., abstract or normal feature. Abstract features are used for grouping. Normal features make references to language components available in the platform's language component library. Connections between features are established and defined with types such as or, xor, and, optional, or mandatory. These connections establish bindings between the extensions of the referenced language components, linking child features to parent features. The resulting composition tree can be saved to the language family library and utilized for deriving composed languages through feature configuration in subsequent steps.

3.2.3 Configuring and Exporting Composed Languages.

To derive new languages, the initial step involves selecting a language family from the language family library. Each language family within the library comes with a comprehensive description and is visually represented as a tree structure. The user can interact with this tree by clicking on specific language features to select them. Once the desired language family configuration is established, the user can click on the "Derive DSL" button. This action triggers a validation process where the language family configuration is checked against the constraints of the feature tree. If the configuration is deemed valid, the SCOLaR framework in the backend proceeds to compose the referenced language components. Once the composition process is completed, the user is notified through a pop-up message and provided with the option to download the composed language project source files. To utilize the language, the project can be built using Maven and subsequently used with the tooling provided by the MontiCore language workbench.

4 Software Architecture

The SCOLaR low-code platform is realized as a classic three-tier architecture consisting of a persistency layer, a backend and a frontend. The software architecture is depicted in Figure 3. Our low-code platform is designed with deployability in mind, and every component of the software architecture is packaged as a Docker image. This Dockerization enables seamless and scalable deployment of our platform, allowing for efficient utilization of resources and easy management of the platform's infrastructure.

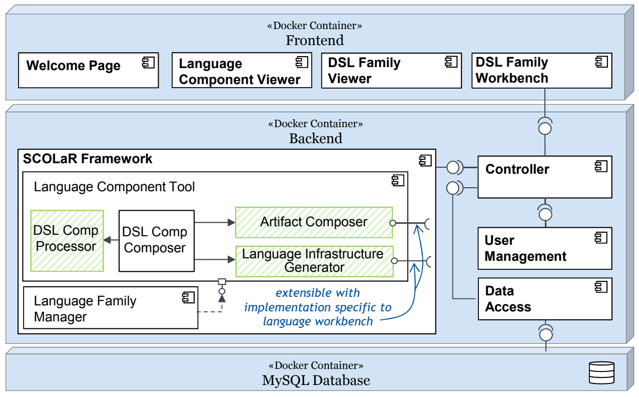


Figure 3. The 3-tier architecture of the low-code platform.

4.1 Frontend

The frontend of the SCOLaR low-code platform is developed using Vue.js and provides different components for user interaction. The `Welcome Page` enables to log into the platform and choose from the `Language Component Viewer`, the `Language Family Viewer`, and the `Language Family Workbench`, afterward. Utilizing the `Language Family Workbench` users can assemble new language families or reuse and extend existing ones following the process described in Section 3.2. With the `Language Component Viewer` language projects can be managed and imported into the platform. The `Language Family Viewer` shows available language families for configuration.

4.2 Backend

The backend of the platform is built on Java Spring Boot. The backend comprises a `Controller` for the overall workflow control of the platform, and to provide the REST API for interaction with the frontend. Additionally, the backend includes a `User Management`. Today, the SCOLaR platform supports the roles 1) engineer, that has rights to create, remove and modify language families and components, 2) and user, that can only view components, and families and configure and derive language products by selecting features of a language family already available in the language family library. For persisting imported language projects together with their associated language components, as well as created language families, the `Data Access` component persists them into a MySQL database. In addition, the existing tooling of SCOLaR is reused in the backend, to perform the processing of language component models, language families, language family configurations, and the composition of the language components and their comprised artifacts. Furthermore, there exists a language infrastructure generator that generates the composed language project and exports it as a zip file.

4.3 Database

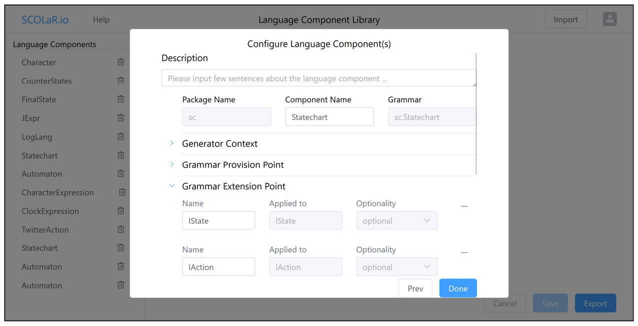
The persistency layer of the platform is implemented using MySQL. The database of the platform is used to persist all artifacts related to the SCOLaR process (cf. Figure 1), i.e., language components, their related language projects, the language family model, and users, together with roles and the associated permissions. This enables the platform to provide a library of language components and families for reuse that were imported and assembled before.

4.4 Extensibility

Since SCOLaR is subject to ongoing research, the software architecture should be refined accordingly. We see the following concepts being subject to changes that have to be taken into account in all three layers of our architecture. 1. The extension of constituents of language components and their composition according to bindings. When extending the constituents of language components, in the backend, the `DSL Comp Processor` (cf. Figure 3) has to be extended. In the frontend, the `Language Component Viewer` has to be updated. Furthermore, in the database the schema for language components has to be adapted to the changes to the language component constituents and to represent the project structures of new technological spaces. 2. When introducing new composition operators between language components, in the backend, the `Artifact Composer` for the specific artifacts of technological spaces and the `Language Infrastructure Generator` has to be implemented. Furthermore, the `Language Family Manager` has to be extended with the bindings specific to this new language composition operator. In the frontend, these changes have to be adopted by the `DSL Family Viewer` and `DSL Family Workbench`. Finally, the database schema for language families has to be updated. For all of these changes, the software architectures provides dedicated extension points or interfaces that can be extended and implemented, respectively.

5 Demonstration

This section performs a walk through the presented platform by an example of a language family for statecharts. First, the language component library together with the import mechanism is shown. Afterwards, the language family workbench where imported languages can be reused as components and arranged in a language family is presented. And at last, the language family language family for statecharts is configured to derive a specific language variant.



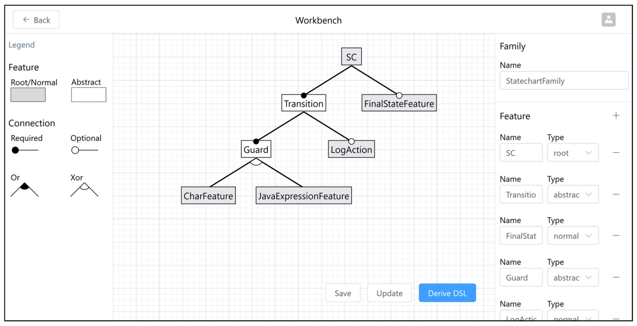
Description: Please input few sentences about the language component ...

Package Name: sc
Component Name: Statechart
Grammar: sc.Statechart

Name: IState
Applied to: IState
Optionality: optional

Name: IAction
Applied to: IAction
Optionality: optional

Figure 4. Importing languages into the SCOLaR platform and configuring the derived language component.



Name: StatechartFamily

Name	Type
SC	root
Transitio	abstrac
FinalStat	normal
Guard	abstrac
LogActio	normal

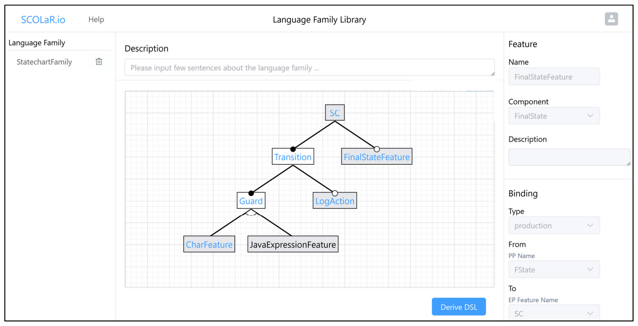
Figure 5. The workbench for creating language families.

5.1 Importing a Language Project

The first step towards reuse of existing language projects in SCOLaR is the import as language components. As mentioned in Section 3 our platform provides an automatic language component derivation mechanism. Importing language projects into the platform is possible via the language component library. By clicking the button *import*, a dialog for importing language projects opens. The selected project is uploaded to the platform, persisted, and a fitting language component is derived automatically. This language component can then be customized according to its provided and required extensions (cf. Figure 4). After the configuration, the component is persisted and available in the language component library, and the language family workbench for reuse.

5.2 Creating a Language Family

Figure 5 shows the language family workbench in the SCOLaR platform exemplified with a language family for statecharts. In the workbench, features can be removed and added, and features can be associated with language components from the library. In the view, abstract features are filled white and features backed with language components are filled grey. Editing the family is possible via the sidebar. The layout of the family is adjusted automatically whenever a feature is added or removed.



Description: Please input few sentences about the language family ...

Name: FinalStateFeature
Component: FinalState
Description:
Type: production
From PP Name: FState
To EP Feature Name: SC

Figure 6. The systematic language composition process exemplified with a language family for automatons. Selected features are highlighted blue.

5.3 Configure and Export

To derive language variants from language families, the language family library enables choosing from a set of existing language families built in the workbench. To configure a language family, the user can simply select a family in the library and choose the variant of his choice by clicking on the features in the feature tree (cf. Figure 6). By clicking the button *Derive DSL* the configuration is applied, the selected language features are composed, and the language variant is downloaded as a zip archive. After that, the user can build the language project and utilize the language to define models in his language variant.

6 Conclusion

The SCOLaR low-code platform provides graphical means for language engineers to 1) import existing languages, 2) automatically derive a language component interface, that 3) enable reuse along a language family, 4) create and configure language families for various language variants, 5) and manage language components and families for reuse. The platform is still in its early development and since SCOLaR framework is built using MontiCore this is the only language workbench supported currently. However, in the future, we plan to extend our platform by supporting other language workbenches, e.g., XText[1] and to add other language composition operators besides embedding and aggregation [5] and even composition between different compatible technological spaces. Early user reports indicated that extending the platform with more detailed error reporting, and language component and family versioning would be helpful. For the exported language variants, we plan to add LSP[2] generation, providing languages with common editor features, e.g., syntax highlighting, folding, auto-completion etc..

[1]https://www.eclipse.org/Xtext/
[2]https://microsoft.github.io/language-server-protocol/

References

[1] Alexander C Bock and Ulrich Frank. 2021. Low-Code Platform. *Business & Information Systems Engineering* 63 (2021), 733–740. https://doi.org/10.1007/s12599-021-00726-8

[2] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2018. Modeling Language Variability with Reusable Language Components. In *International Conference on Systems and Software Product Line (SPLC'18)*. ACM, Gothenburg, Sweden, 65–75. https://doi.org/10.1145/3233027.3233037

[3] Arvid Butting, Jerome Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. 2020. A Compositional Framework for Systematic Modeling Language Reuse. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (Virtual Event, Canada) *(MODELS '20)*. Association for Computing Machinery, New York, NY, USA, 35–46. https://doi.org/10.1145/3365438.3410934

[4] Manuela Dalibor, Malte Heithoff, Judith Michael, Lukas Netz, Jérôme Pfeiffer, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. 2022. Generating Customized Low-Code Development Platforms for Digital Twins. *Journal of Computer Languages (COLA)* 70 (June 2022), 101117. https://doi.org/10.1016/j.cola.2022.101117

[5] Sebastian Erdweg, Paolo G Giarrusso, and Tillmann Rendel. 2012. Language Composition Untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*. ACM, Tallinn, Estonia, 1–8. https://doi.org/10.1145/2427048.2427055

[6] Katrin Hölldobler, Oliver Kautz, and Bernhard Rumpe. 2021. *MontiCore Language Workbench and Library Handbook: Edition 2021*. Shaker Verlag.

[7] Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. 2018. Software Language Engineering in the Large: Towards Composing and Deriving Languages. *Computer Languages, Systems & Structures* 54 (2018), 386–405. https://doi.org/10.1016/j.cl.2018.08.002

[8] Jérôme Pfeiffer and Andreas Wortmann. 2021. Towards the Black-Box Aggregation of Language Components. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, ACM, Fukuoka, Japan, 576–585. https://doi.org/10.1109/MODELS-C53483.2021.00088

[9] Thomas Thum, Christian Kastner, Sebastian Erdweg, and Norbert Siegmund. 2011. Abstract Features in Feature Modeling. In *2011 15th International Software Product Line Conference*. IEEE, Munich, Germany, 191–200. https://doi.org/10.1109/SPLC.2011.53

[10] Massimo Tisi, Jean-Marie Mottu, Dimitrios S Kolovos, Juan De Lara, Esther M Guerra, Davide Di Ruscio, Alfonso Pierantonio, and Manuel Wimmer. 2019. Lowcomote: Training the Next Generation of Experts in Scalable Low-Code Engineering Platforms. In *STAF 2019 Co-Located Events Joint Proceedings: 1st Junior Researcher Community Event, 2nd International Workshop on Model-Driven Engineering for Design-Runtime Interaction in Complex Systems, and 1st Research Project Showcase Workshop co-located with Software Technologies: Applications and Foundations (STAF 2019)*. CEUR-WS.org, Eindhoven, Netherlands, 73–78.

[11] Jos Warmer and Anneke Kleppe. 2022. Freon: An Open Web Native Language Workbench. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering* (Auckland, New Zealand) *(SLE 2022)*. Association for Computing Machinery, New York, NY, USA, 30–35. https://doi.org/10.1145/3567512.3567515

Received 2023-07-07; accepted 2023-09-01

A Tool for the Definition and Deployment of Platform-Independent Bots on Open Source Projects

Adem Ait
IN3 – UOC
Barcelona, Spain
aait_mimoune@uoc.edu

Javier Luis Cánovas Izquierdo
IN3 – UOC
Barcelona, Spain
jcanovasi@uoc.edu

Jordi Cabot
Luxembourg Institute of Science and Technology
University of Luxembourg
Esch-sur-Alzette, Luxembourg
jordi.cabot@list.lu

Abstract

The development of Open Source Software (OSS) projects is a collaborative process that heavily relies on active contributions by passionate developers. Creating, retaining and nurturing an active community of developers is a challenging task; and finding the appropriate expertise to drive the development process is not always easy. To alleviate this situation, many OSS projects try to use bots to automate some development tasks, thus helping community developers to cope with the daily workload of their projects. However, the techniques and support for developing bots is specific to the code hosting platform where the project is being developed (e.g., GitHub or GitLab). Furthermore, there is no support for orchestrating bots deployed in different platforms nor for building bots that go beyond pure development activities. In this paper, we propose a tool to define and deploy bots for OSS projects, which besides automation tasks they offer a more social facet, improving community interactions. The tool includes a Domain-Specific Language (DSL) which allows defining bots that can be deployed on top of several platforms and that can be triggered by different events (e.g., creation of a new issue or a pull request). We describe the design and the implementation of the tool, and illustrate its use with examples.

CCS Concepts: • **Software and its engineering** → **Development frameworks and environments**; **Designing software**; **Open source model**.

Keywords: Open Source, Bot, Domain-Specific Language

ACM Reference Format:
Adem Ait, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2023. A Tool for the Definition and Deployment of Platform-Independent Bots on Open Source Projects. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3623476.3623524

1 Introduction

Open Source Software (OSS) projects are generally developed on social code-hosting platforms, such as GitHub or GitLab, which provide a set of tools to support the creation of software in a collaborative way. These platforms are built on top of Git and rely on the so-called pull-based development model [10], introduced by GitHub, where developers can create a copy (i.e., fork) of any project's repository and submit a pull request to the original repository to propose changes. Moreover, they provide tools to enable and foster the collaboration, such as issue trackers and forums; as well as social features such as stars, followers, and notifications.

Indeed, the development of OSS projects heavily relies on active contribution by passionate developers [18]. However, retaining and creating an active community of developers is a challenging task [24]. To address this problem, OSS projects attempt to delegate part of the work to automation tools and bots to support the development process. Nevertheless, this has several drawbacks. To begin with, it involves manual coding and expertise on the different platform APIs. Bots are therefore platform-specific making it also very time-consuming to create any type of bot that needs to interact with projects deployed on several platforms (e.g., case of mirror repositories of projects). Furthermore, the bots are usually designed to fulfill automation tasks related to the code development, even though the community behind a project is more than just the code contributors [12] and support for automatic community management would also be important to optimize project collaboration.

In this sense, this paper proposes a tool to define and deploy bots for OSS projects. The tool includes a Domain-Specific Language (DSL) to define bots that can be deployed on different platforms and that can be triggered by different events (e.g., creation of a new issue or a pull request). The set of events covered by the language is a superset of all events available on popular code-hosting platforms and their APIs, thus enabling users to define generic bots. These events also cover community events to facilitate the creation of more social bots. Beyond the tool, we also provide a DSL to define the bots, and the runtime to execute the modeled bots, translating automatically the bot behavior to calls to the underlying APIs, depending on the target platform.

The rest of the paper is structured as follows. Section 2 introduces the background and related work. Section 3 presents our proposal. Section 4 details the tool infrastructure, the language domain and syntax, and illustrates its use with an example. Section 5 describes the runtime design. Section 6 concludes the paper and presents future work.

2 Background and Related Work

This section covers the role of bots in OSS project development, the benefits of DSLs and the related work trying to use DSLs for bot definitions.

2.1 Bots in OSS Project Development

The development and success of OSS relies on the coordination and contributions by the community, usually named social coding [4]. Some studies address specific tasks in OSS project development, such as recommending developers to open tasks [23], detecting unmaintained projects [3], or predicting whether newcomers may become long-term contributors [1].

In the last years, this collaborative behavior has leveraged the use of bots to help and automatize some development tasks [11], thus reducing the workload of contributors [21] (or covering the lack of them). The idea of bots helping in software development has been explored in several works (e.g., [7, 8, 19]), which recognize their key role in addressing specific development tasks, but none of them propose solutions to create bots in a holistic and scalable way. Furthermore, some studies contemplate some drawbacks or effects of bots being a part of the project's community, such as the impact of adopting bots in pull requests code revisions [20], the problems of human-bot interactions in pull requests [22] or the interaction between software developers and a bot that recommends pull request reviewers [15].

However, these works propose concrete bots as solutions instead of mechanisms to build the bots themselves.

2.2 DSLs for the Definition of Bots for OSS Projects

Domain-Specific Languages (DSLs) are languages specially designed to help to solve a problem in a particular domain. A DSL is composed of three main elements [14]: (1) abstract syntax, which defines the concepts and relationships of the domain where the language is applied; (2) concrete syntax, which defines the notation of the language (e.g., textual, diagram-based, etc.); and (3) semantics, which defines the meaning of the language constructs. Furthermore, DSLs can be classified into external DSLs, which are generally defined by a grammar; and internal DSLs, which are embedded in a general-purpose programming language (known as host language). By using a DSL, the developer can use domain-specific constructs and therefore address the problem more efficiently [9]. We believe a DSL targeting the domain of bots for OSS would solve some of the issues commented in the introduction. So far, such DSL does not yet exist.

Some platforms offer mechanisms to define automation tasks relying on configuration languages, such as GitLab CI or, more recently released, GitHub Actions (GHA). Some works have analyzed the usage and impact of GHA [2, 6, 13] exposing its spread on this platform. These alternatives are completely platform-dependent and focus on core development tasks.

There are a couple of approaches proposing DSLs for bots, mainly focused on chatbots. Pérez-Soler et al. [16] propose a DSL, Conga, which leverages on modeling techniques to design chatbots according to a platform-independent metamodel. Xatkit [5] is a flexible multi-platform chatbot development framework, which comprises three DSLs allowing the definition of different components of a chatbot, namely: Intent DSL, Execution DSL and Platform DSL. Nevertheless, these approaches do not have primitives covering the OSS development domain, making it difficult to write bots able to manage OSS concepts, or running them.

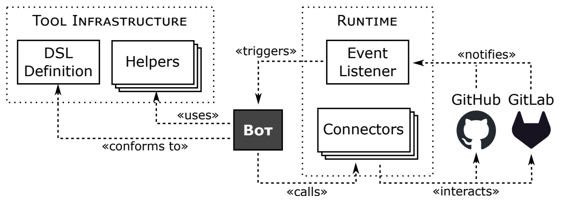


Figure 1. Architecture of our proposal.

3 Our Proposal

To the best of our knowledge there is no DSL to build bots in OSS in a way that is agnostic to the code-hosting platform as the one we propose here.

With our approach, bots are defined independently with our platform-independent tool and then can be configured to interact with any specific code-hosting platform. Figure 1 shows the architecture of our proposal. As can be seen, the *Tool Infrastructure* includes the DSL definition and helper libraries to facilitate the definition of bots in an agnostic way (i.e., helpers provide constructs to efficiently and transparently access the code-hosting platform generically). The *Runtime* is responsible for listening and tracking the events in the code-hosting platforms, triggering the bots linked to those events and executing their behavior which in turn will call the connectors to interact with the corresponding code-hosting platform. Next we describe each component.

4 Tool Infrastructure

To build the tool infrastructure, we first define the abstract syntax of the DSL, and then discuss its concrete syntax and implementation. We finalize the section with some examples of the language usage.

4.1 Abstract Syntax

The abstract syntax of our language can be clearly organized in two main sublanguages, namely: OSS domain and bot. The former covers those concepts of the domain (OSS community development in our case) required to define bots in an agnostic way, that is, independently of the code-hosting platform where the bots will be deployed; while the latter defines the core language constructs to define the bots themselves.

4.1.1 OSS Domain Sublanguage. To build the language domain, we explored existing code-hosting platforms and selected the following: GitHub and GitLab. We chose these platforms due to their activity, the number of projects they host, the ability to perform a detailed analysis of their features and their popularity. In Appendix A we provide a list of the platforms identified and eventually discarded.

Figure 2 defines the metamodel inferred from the analysis of the features and concepts of the selected platforms. Our bots will need to be able to read and get triggered by changes on those elements and update them when needed. Note that to avoid crossing lines, we sometimes express associations between classes as an attribute with the corresponding type.

The main element of the diagram is the `Repository` class. This element represents the project's repository, the central element of any code-hosting platform. The class includes the main properties available in code-hosting platforms (e.g., name, topics, stargazers, etc.). The remaining classes describe the other elements playing a role during the development process, besides the `User` hierarchy, which identifies platform users and the authors of commits, and the `Group` class which represent the ownership and contributors of the repository.

For instance, the `Contribution` class comprises issues and pull requests. Key characteristics of all contributions are its title and body, which describe its creation reason, and its state, whether it is resolved or pending for resolution, for example. Contributions can be assigned to a milestone (see `Milestone` class). The `Contribution` hierarchy includes the `Issue` and `PullRequest` classes, which represent the two types of contributions available in code-hosting platforms. While issues are designed for open discussions or feature requests in the project, pull requests are the mechanisms to accept new code changes from a branch or a fork into the repository, a process known as pull-based development [10]. The latter is composed by a set of reviews that validate or reject the proposed changes. These reviews are often supervised by the owner or an internal contributor of the project. As pull requests can be created from discussions in issues, there may be links between them.

The communication in issues and pull requests is based on comments, which are represented by the `Comment` class and hierarchy. Comments at the contribution level are represented as `ContributionComment` class, and they are used to discuss both pull requests and issues. Additionally, for pull requests, reviews may include one or more comments (see `PRReviewComment` class). Furthermore, we identified the comments of a commit as a part of the comment hierarchy, as it has common information with the other type of comments. However, these comments are usually stored directly as part of the version control system (VCS) information and visible in the commit tracking history.

Another important feature is the management of the project documentation hosted in the repository[1]. The documentation, called wiki, is composed by pages, and changes are tracked for each page.

The domain also includes the `User` hierarchy, which represents the users of the platform. We distinguish platform users and VCS users (see `PlatformUser` and `GitUser`). The former are the accounts registered in the code-hosting platform, while the latter are users only detected in the VCS tool, in our case Git (i.e., commit users). Platform users can be organized in groups, which are represented by the `Group` class, and they own a set of repositories. A key aspect of groups is the chance of assigning certain roles to the users, determining which actions can perform (see `Member` class). This facilitates the project management for corporations, organizations and other possible groups of developers.

4.1.2 Bot Sublanguage. The part of the abstract syntax devoted to define the core bot aspects of our DSL is presented in Figure 3. A `BotDefinition` represents a bot, which listens to a set of events of code-hosting platforms, defined by the `Event` class, and performs a set of actions, defined by the `Behavior` class. An `Event` has a condition (see `Condition`) which may query elements in the domain (see `from` association). Due to space limitations, we only show a subset of events (see `Event` hierarchy). The `Behavior` hierarchy includes `Executes` and `Creates`, which may also query domain elements. We describe these elements below. Finally, the `DomainElement` concept is the superclass of all elements in the metamodel of Figure 2.

4.2 Concrete Syntax & Implementation

We designed our DSL as a textual language, and implemented it as an internal DSL in Java to leverage on the Java ecosystem and its existing libraries. As a textual language, the language definition is driven by a set of statements, which are identified by keywords. Being an internal DSL, we relied on fluent interfaces using the method chaining pattern [9] to enable the language statements. The language currently

[1] GitLab creates a separate Git repository.

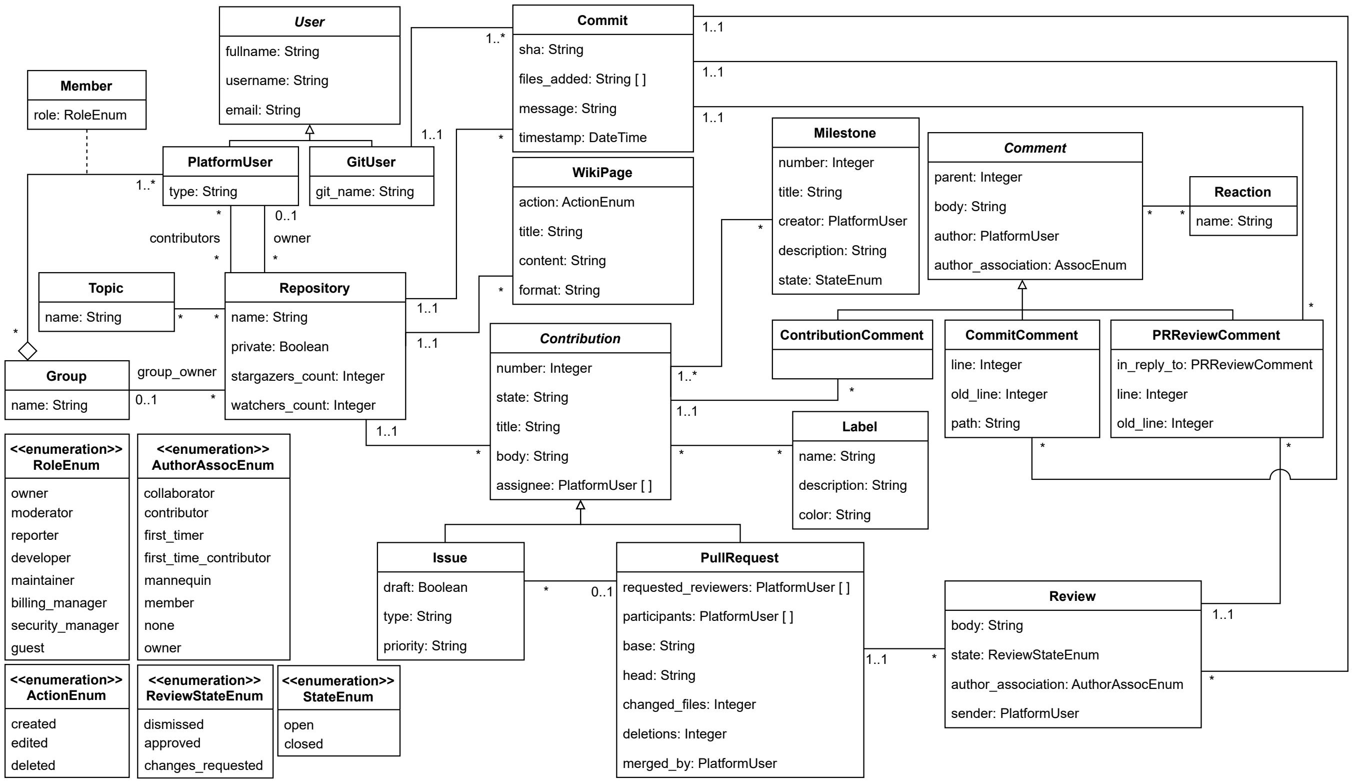


Figure 2. OSS domain metamodel.

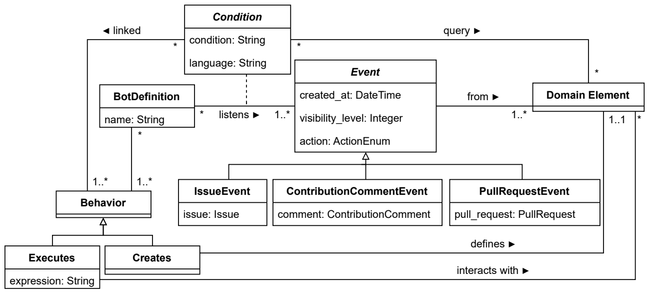


Figure 3. Bot metamodel.

includes five statements using the corresponding keywords, namely, `createBot`, `on`, `creates`, `executes`, and `validate`. In the following, we describe each statement following a function-based format (i.e., keyword name and parameters).

`createBot(name)` This statement sets the bot name.

`on(event, condition)` This statement defines the event and the optional conditional statement triggering the bot. Several on statements may be used if a bot is triggered by several events.

`creates(element)` This statement specifies the domain element that must be created once an event triggers the bot. One or more domain elements can be created. Constructors to build each domain element are provided.

`executes(code)` This statement defines the bot behavior to be executed when it is triggered by an event. Unlike the previous statement, the `executes` statement accepts a lambda expression that must be executed as part of the bot behavior.

`validate()` This statement ends the definition of the bot and validates the bot definition.

A bot definition must use the keywords in a specific order. The first statement must be `createBot` and one or more `on` statements afterward. Each set of `on` statements must be followed by either a `creates` or an `executes` keyword. The `creates` or `executes` keyword can be followed by either a `validate` keyword, which finalizes the bot definition; or another set of `on` statements, thus defining a new set of triggering conditions. Note that any bot definition must always finalize with a `validate` keyword.

As a Java internal language, we leverage on the host language to be able to ignore most newlines, thus improving the readability and making debugging easier. To enforce the order of the keywords, we use progressive interfaces, that is, the usage of using multiple interfaces to drive and enforce a fixed sequence of method-chaining calls. However, one of the disadvantages of using method chaining and progressive interfaces is the finishing problem, summarized to the lack of a clear end-point to a method chain. To mitigate this problem, we added the `validate` statement, which closes

Listing 1. Example using `creates` statement.

```
1  exampleBot = Bot.createBot("Commenter")
2    .on(Event.CONTRIBUTION_COMMENT, (payload) -> {
3      Issue issue = DomainHelper.digestPayload(payload,
           domainClass:Issue.class);
4      return issue.getNum_comments() == 1;
5    })
6    .creates(CreateHelper.createComment(body:"Thanks␣for␣your␣
         contribution!"));
7    .validate();
```

the bot definition but introduces syntactic noise. To alleviate the situation, the `validate` statement also validates that the bot definition and state is correct.

4.3 Example

To illustrate the use of our language, we show two examples of simple bots, which (1) thanks the author of the first comment in an issue and (2) notify project contributors when a pull request is created without requested reviewers. Listings 1 and 2 show these examples, respectively. Listing 1 illustrates the use of the `creates` statement, while Listing 2 uses the `executes` statement.

In both examples, the name is a unique Java string. Note that the event is defined among a set of predefined events, and it is declared using Java enumerations. Along with the declared event, the conditional statement is represented as a Java lambda expression, accessing to the platform entities via the `DomainHelper`. The `DomainHelper` facilitates the extraction of any domain element from the payload of the webhook notification, thus liberating developers from building and navigating the domain elements. For instance, line 3 in Listings 1 extracts the issue related to the event, while line 3 in Listing 2 does so for the pull request. In both examples, conditional triggers are defined with Java comparison operators with the attributes of the retrieved element. At last, each bot includes the definition of the behavior of the bot, via a `creates` and `executes` statements, respectively.

In Listing 1, note that the `creates` statement allows the developer to define the bot behavior effectively. To this aim, our approach provides the so-called `CreateHelper`, which implements typical behavior when creating elements, in the example, the creation of a comment in the issue linked to the event. Note that more complex behavior should be defined by using the `executes` statement. On the other hand, the `executes` statement showed in Listing 2, allows the user to be more precise, thus enabling the definition of more complex actions. In this case, we rely on the `Member` domain element to recover the set of project maintainers to be notified.

5 Runtime

The execution of bots is governed by the Runtime (cf. Figure 1). The Runtime includes an *Event Listener* to track the events from code-hosting platforms, and to trigger the execution of the corresponding bot, and a set of *Connectors* to interact with the code-hosting platforms APIs.

Listing 2. Example using `executes` statement.

```
1  exampleBot = Bot.createBot("MailNotifier")
2    .on(Event.PULLREQUEST, (payload) -> {
3      PullRequest pr = DomainHelper.digestPayload(payload,
           domainClass:PullRequest.class);
4      return pr.getRequested_reviewers().isEmpty();
5    })
6    .executes((payload) -> {
7      Repository repo = DomainHelper.digestPayload(payload,
           Repository.class);
8      ArrayList<Member> members = repo.getGroup().getMembers().
           stream().filter((m) -> m.getRole() == RoleEnum.
           MAINTAINER);
9      for (Member m : members) {
10       message.addRecipient(m.getEmail());
11     }
12     String body = "Hi␣developer,␣there␣is␣a␣new␣pull␣request␣
           with␣no␣requested␣reviewers␣in␣your␣repo:" + repo.
           getName();
13     message.SetContent(body);
14     Transport.send(message);
15   })
16   .validate();
```

We have implemented the Runtime as a web application, able to track and listen events from code-hosting platforms via webhooks. The Event Listener only triggers the bot if the event conditions defined in the *on* statement are fulfilled. Being an internal DSL, the event listener delegates the execution flow to the *creates* or *executes* statement of the bot. The execution of these statements calls the corresponding connector, which is in charge of mapping the bot actions to the code-hosting platform API calls.

6 Conclusion

In this paper, we have presented a tool for defining and deploying bots independently of the code-hosting platform. For this, we have defined a language as an internal DSL in Java. Bots defined with our tool can be easily deployed in potentially any code-hosting platform via the Runtime, which currently supports GitHub and GitLab. We have illustrated the use of our approach with several examples.

This is the first step of a more ambitious vision towards providing every OSS project with a swarm of bots able to collaborate among them and with the community members to ensure the project's long-term sustainability. Along this line, future work includes extending our bots with NLP capabilities and LLM connectors for more advanced interactions, the ability to model bots' orchestrations and their collaboration, and coordination towards a common goal, e.g., involving ecosystems of projects deployed over multiple repositories and platforms. Works on the swarm robotics domain (e.g., [17]), can be useful to adapt swarm algorithms and communication methods into our domain.

Acknowledgements

This work is part of the project TED2021-130331B-I00 funded by MCIN/AEI/10.13039/501100011033 and European Union NextGenerationEU/PRTR; and BESSER, funded by the Luxembourg National Research Fund (FNR) PEARL program, grant agreement 16544475.

A. Platform selection

Table 1. Discarded platforms considered for building the language domain.

Platform	URL
Gitea	https://gitea.io/en-us/
Codeberg	https://codeberg.org/
BitBucket	https://bitbucket.org/
SourceForge	https://sourceforge.net/
HuggingFaceHub	https://huggingface.co/
ProjectLocker	https://www.projectlocker.com/
Launchpad	https://launchpad.net/
Assembla	https://get.assembla.com/
Beanstalk	https://beanstalkapp.com/
Savannah	https://savannah.gnu.org/
RepositoryHosting.com	https://repositoryhosting.com/
Codebase	https://www.codebasehq.com/
SourceRepo	http://sourcerepo.com/
Gerrit	https://www.gerritcodereview.com/
Backlog	https://nulab.com/backlog/
Codegiant	https://codegiant.io/home
Kallithea	https://kallithea-scm.org/
RhodeCode	https://code.rhodecode.com/

References

[1] Lingfeng Bao, Xin Xia, David Lo, and Gail C. Murphy. 2021. A Large Scale Study of Long-Time Contributor Prediction for GitHub Projects. *IEEE Trans. Softw. Eng.* 47, 6 (2021), 1277–1298.

[2] Tingting Chen, Yang Zhang, Shu Chen, Tao Wang, and Yiwen Wu. 2021. Let's Supercharge the Workflows: An Empirical Study of GitHub Actions. In *Int. Conf. on Quality Software*. 1–10.

[3] Jailton Coelho, Marco Túlio Valente, Luciano Milen, and Luciana Lourdes Silva. 2020. Is this GitHub project maintained? Measuring the level of maintenance activity of open-source projects. *Inf. Softw. Technol.* 122 (2020), 106274.

[4] Laura A. Dabbish, H. Colleen Stuart, Jason Tsay, and James D. Herbsleb. 2012. Social coding in GitHub: transparency and collaboration in an open software repository. In *ACM Conf. on Computer Supported Cooperative Work*. 1277–1286.

[5] Gwendal Daniel, Jordi Cabot, Laurent Deruelle, and Mustapha Derras. 2020. Xatkit: A Multimodal Low-Code Chatbot Development Framework. *IEEE Access* 8 (2020), 15332–15346.

[6] Alexandre Decan, Tom Mens, Pooya Rostami Mazrae, and Mehdi Golzadeh. 2022. On the Use of GitHub Actions in Software Development Repositories. In *IEEE Int. Conf. on Software Maintenance*. IEEE, 235–245.

[7] Linda Erlenhov, Francisco Gomes de Oliveira Neto, and Philipp Leitner. 2020. An empirical study of bots in software development: characteristics and challenges from a practitioner's perspective. In *Int. Conf. on the Foundations of Software Engineering*. 445–455.

[8] Linda Erlenhov, Francisco Gomes de Oliveira Neto, Riccardo Scandariato, and Philipp Leitner. 2019. Current and future bots in software development. In *Int. Workshop on Bots in Software Engineering @ ICSE*. 7–11.

[9] Martin Fowler. 2011. *Domain-Specific Languages*. Addison-Wesley.

[10] Georgios Gousios, Martin Pinzger, and Arie van Deursen. 2014. An exploratory study of the pull-based software development model. In *Int. Conf. on Software Engineering*. 345–355.

[11] Philipp Hukal, Nicholas Berente, Matt Germonprez, and Aaron Schecter. 2019. Bots Coordinating Work in Open Source Software Projects. *Computer* 52, 9 (2019), 52–60.

[12] Javier Luis Cánovas Izquierdo and Jordi Cabot. 2022. On the analysis of non-coding roles in open source development. *Empir. Softw. Eng.* 27, 1 (2022), 18.

[13] Timothy Kinsman, Mairieli Santos Wessel, Marco Aurélio Gerosa, and Christoph Treude. 2021. How Do Software Developers Use GitHub Actions to Automate Their Workflows?. In *IEEE Int. Working Conf. on Mining Software Repositories*. 420–431.

[14] Anneke Kleppe. 2008. *Software Language Engineering*. Addison-Wesley.

[15] Zhenhui Peng, Jeehoon Yoo, Meng Xia, Sunghun Kim, and Xiaojuan Ma. 2018. Exploring how software developers work with mention bot in GitHub. In *Int. Symposium of Chinese CHI*. 152–155.

[16] Sara Pérez-Soler, Esther Guerra, and Juan de Lara. 2020. Model-Driven Chatbot Development. In *Int. Conf. on Conceptual Modeling*, Vol. 12400. 207–222.

[17] Melanie Schranz, Martina Umlauft, Micha Sende, and Wilfried Elmenreich. 2020. Swarm Robotic Behaviors and Current Applications. *Frontiers Robotics AI* 7 (2020), 36.

[18] Ravi Sen, Siddhartha S. Singh, and Sharad Borle. 2012. Open Source Software Success: Measures and Analysis. *Decis. Support Syst.* 52, 2 (2012), 364–372.

[19] Margaret-Anne D. Storey and Alexey Zagalsky. 2016. Disrupting developer productivity one bot at a time. In *ACM SIGSOFT*. 928–931.

[20] Mairieli Wessel, Alexander Serebrenik, Igor Wiese, Igor Steinmacher, and Marco A Gerosa. 2020. Effects of Adopting Code Review Bots on Pull Requests to OSS Projects. In *IEEE Int. Conf. on Software Maintenance*. 1–11.

[21] Mairieli Santos Wessel, Bruno Mendes de Souza, Igor Steinmacher, Igor Scaliante Wiese, Ivanilton Polato, Ana Paula Chaves, and Marco Aurélio Gerosa. 2018. The Power of Bots: Characterizing and Understanding Bots in OSS Projects. *Proc. ACM Hum. Comput. Interact.* 2, CSCW (2018), 182:1–182:19.

[22] Mairieli Santos Wessel and Igor Steinmacher. 2020. The Inconvenient Side of Software Bots on Pull Requests. In *Int. Conf. on Software Engineering*. 51–55.

[23] Joicymara Xavier, Autran Macedo, and Marcelo de Almeida Maia. 2014. Understanding the popularity of reporters and assignees in the Github. In *Int. Conf. on Software Engineering and Knowledge Engineering*. 484–489.

[24] Minghui Zhou and Audris Mockus. 2015. Who Will Stay in the FLOSS Community? Modeling Participant's Initial Behavior. *IEEE Trans. Softw.* 41, 1 (2015), 82–99.

Received 2023-07-07; accepted 2023-09-01

Online Name-Based Navigation for Software Meta-languages

Peter D. Mosses
P.D.Mosses@tudelft.nl
TU Delft
Delft, Netherlands
Swansea University
Swansea, UK

Abstract

Software language design and implementation often involve specifications written in various esoteric meta-languages. Language workbenches generally include support for precise name-based navigation when browsing language specifications *locally*, but such support is lacking when browsing the same specifications *online* in code repositories.

This paper presents a technique to support precise name-based navigation of language specifications in online repositories using ordinary web browsers. The idea is to generate *hyperlinked twins*: websites where *verbatim copies* of specification text are enhanced with hyperlinks between name references and declarations. By generating hyperlinks directly from the name binding analysis used internally in a language workbench, online navigation in hyperlinked twins is automatically consistent with local navigation.

The presented technique has been implemented for the Spoofax language workbench, and used to generate hyperlinked twin websites from various language specifications in Spoofax meta-languages. However, the applicability of the technique is not limited to Spoofax, and developers of other language workbenches could presumably implement similar tooling, to make their language specifications more accessible to those who do not have the workbench installed.

CCS Concepts: • **Software and its engineering** → **Integrated and visual development environments**; **Software libraries and repositories**; • **Information systems** → **Browsers**.

Keywords: code navigation, hyperlinked twins, language specifications, meta-languages, language workbenches

SLE '23, October 23–24, 2023, Cascais, Portugal
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0396-6/23/10.
https://doi.org/10.1145/3623476.3623528

ACM Reference Format:
Peter D. Mosses. 2023. Online Name-Based Navigation for Software Meta-languages. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3623476.3623528

1 Introduction

Name-based navigation is a significant aspect of software language engineering. IDEs generally include support for precise name-based navigation when browsing code *locally*, but such support is lacking *online* when using ordinary web-browsers on code repositories.

Here, we suggest to generate *hyperlinked twin websites* from code repositories. The code on the website should look the same as it does in an IDE, and the hyperlinks should support the same name-based navigation as the IDE.

Software *meta-languages* are a particularly important special case of software languages, and language workbenches implement name-based navigation for the meta-languages that they use. Moreover, a language workbench is likely to provide an API to access ASTs and name binding analyses, facilitating generation of hyperlinked twin websites.

To illustrate the suggested technique, the Spoofax language workbench [4] has been used to generate hyperlinked twins from various language specifications in Spoofax meta-languages.[1] This involved writing only a small amount of code in the Spoofax meta-language Stratego. The code uses generic AST traversals to generate HTML from parsed and analysed specifications, and a simple API for accessing name binding information. The code is available on GitHub.[2]

The rest of this section expands on the above points. Section 2 then explains the main steps of the generation process, which may be of interest to developers of other language workbenches. Section 3 briefly mentions some details specific to the use of Spoofax. Section 4 concludes, and discusses future work. Appendix A shows how a fragment of a language specification looks in Spoofax, in a GitHub repository, and in the hyperlinked twin generated from that repository.

[1] https://pdmosses.github.io/hyperlinked-twins/
[2] https://github.com/pdmosses/sdf/tree/sle23/org.metaborg.meta.lang.template/trans/generation/docs/

1.1 Name-Based Code Navigation

Software languages generally include *declarations* that bind names to entities, and *references* to those entities using the declared names. Name-based navigation between declarations and references is essential for browsing and exploring code in software languages.

Manual name-based navigation can be tedious and error-prone: it may require scrolling, or entering text in search boxes. It becomes significantly more difficult when declarations can be in different files from references to them – particularly when code is divided into hundreds of files, perhaps with a complicated import relationship.

Integrated software development environments (IDEs) support name-based navigation when locally browsing or editing code. When a reference to a name is selected, the IDE allows navigation directly to the relevant declaration(s). When a declaration is selected, the IDE may also support navigation directly to some or all the references to it.

Often, a name can be used in more than one declaration in the same project – either in different namespaces (e.g., types and constructors) or in different parts of the project. Support for name-based navigation using simple textual search may then be significantly inferior to precise navigation using name binding analysis, due to false positives in search results.

Support for name-based navigation is often weak in online code repositories when using ordinary web browsers. GitHub repositories currently support search-based code navigation in about a dozen mainstream programming languages [3], but precise name-based navigation in only one language [1]: Python. GitHub's implementation of precise online name-based navigation requires specifying the name binding analysis of the language in terms of stack graphs [2]. Apart from the significant amount of expertise and effort required for that, a potential drawback of GitHub's approach may be the difficulty of validating that the navigation in the repository accurately reflects the name-binding analysis implemented in compilers. In any case, precise navigation on GitHub seems likely to be limited to a few major programming languages, despite the possibility for language developers to contribute support for further languages [6].

1.2 Software Meta-languages

A *meta-language* is a language for specifying languages (primarily their syntax and semantics). A *software meta-language* is simply a meta-language for specifying software languages. Specifications of major software languages can be large, and difficult to navigate. Moreover, unfamiliarity with a particular software meta-language can hinder manual name-based navigation in language specifications – especially when name binding in the meta-language differs significantly from that in conventional programming languages.

Development and validation of software language specifications is supported by software language workbenches, which generally implement precise name-based navigation. However, that navigation is not generally available for such language specifications when browsing them in online repositories using ordinary web browsers. To browse a language specification with precise name-based navigation, users then need to install a workbench locally and download a copy of the repository.

1.3 Prior Examples of Hyperlinked Twins

The reference manuals of most current programming languages are available online in HTML or PDF, and can be browsed using ordinary browsers. There, hyperlinks already support name-based navigation in grammars that specify language syntax. When the hyperlinks are generated from repositories containing the plain text of the grammars, the reference manuals may then be regarded as hyperlinked twins.

The author has previously developed support for precise name-based navigation of language specifications online: the CBS-beta website,[3] which was generated from CBS specifications whose syntax and name binding were specified in Spoofax meta-languages. In [5] he speculated that the approach used to generate the CBS-beta website might be applicable to other software meta-languages; the present paper confirms that, but it turned out not to be possible to reuse the implementation of the generation process directly: the code involved case analysis on the constructs of CBS, and would need to be almost completely reimplemented for each meta-language.

Various other specification frameworks provide tool support for generating hyperlinked websites from specifications. For example, the web version of an online book [7] includes hyperlinked pages generated from (literate) Agda source code. If web versions of source code in other specification languages can be generated using the same tool support, it would be interesting to compare the generation process with that outlined here.

2 Generating Hyperlinked Twin Websites

The aim is tool support for online name-based navigation of language specifications in ordinary web browsers. The main idea is to generate web pages where verbatim copies of the specifications are enhanced with hyperlinks between name references and declarations. By generating the hyperlinks directly from analyses used internally in language workbenches, online navigation in language specifications is automatically consistent with local navigation.

The proposed technique has been implemented in the Spoofax language workbench, with only modest effort, as outlined in Section 3; it might be possible to implement it in other language workbenches in much the same way.

[3]https://plancomps.github.io/CBS-beta/

Suppose that some language workbench is to generate a hyperlinked website from the plain code of a language specification found online in some GitHub repository. The suggested technique is to proceed as follows.

Requirements. The language workbench needs to parse and analyse the plain language specification. Unless the workbench can directly access the repository online, a local clone is required; and to add the source files for the generated website to the repository using pull-requests, the clone will need to be published as a fork of the repository.

If the language specification is in meta-languages supported by the workbench, it can already parse and analyse them. However, the results also need to be accessible for transformation to HTML. (That should always be possible when the implementation of the meta-languages in the workbench is bootstrapped.) If the specification uses external meta-languages, those languages need to be loaded into the workbench before proceeding.

The following steps are to be applied to a complete language specification project.

Creating ASTs. To support generation steps that involve tree traversal, the first step is to parse the language specification files and create corresponding abstract syntax trees (ASTs). The generation process is to be completely independent of the detailed structure of the ASTs (and hence of the meta-language used for specification). The ASTs might correspond closely to parse trees, or they could be 'de-sugared' to remove semantically-irrelevant structure such as white space, line breaks, and literal terminal symbols (depending on the language).

However, the ASTs must support the addition of name binding information to nodes that correspond to declarations and references. Such nodes also need to reveal the start and end positions of their source text.

The language workbench may automatically parse files and generate their ASTs, otherwise this step needs to be explicitly executed.

Adding name binding analysis. Based on the relevant name binding analysis for the meta-language, this step should ensure that all declarations and references can be detected when traversing the ASTs. Each declaration node needs to provide the source text of the declared name; each reference node needs to provide not only the name, but also the declaration(s) to which the reference has been resolved.

In general, a reference may resolve to a declaration in a different file; and a declaration of a single name may be spread across multiple files.

As with generating ASTs, a language workbench may automatically analyse files and add the resulting information to their ASTs, otherwise this step needs to be explicitly executed. The remaining steps are specific to the generation of hyperlinked websites, but could also be made automatic.

Generating plain HTML.. The obvious way to generate HTML that renders exactly as some plain source text is to enclose the text in `<pre><code>...</code></pre>` tags. In general, this preserves the white space (i.e., indentation and line breaks) of the source text – assuming that the rendering uses a fixed-width font.

The source text might also contain the characters '<', '>', and '&', which are all treated specially in HTML. These need to be replaced by the corresponding HTML entities '`&lt;`', '`&gt;`', and '`&amp;`', respectively.

Subsequent steps are to enclose parts of the source text in tags for hyperlinks and highlighting. To avoid the need for obtaining the source text of all nodes in an analysed AST, plain HTML can be generated gradually, by copying characters from the source file to the generated file while traversing the AST (top down, left to right).

Generating hyperlinks. To generate hyperlinks between declarations and references, the relevant tags can be inserted whenever the traversal reaches the corresponding node.

When the node is a declaration of name N at position P, the element `<span id="`*N_P*`">`*N*`</span>` provides a unique target for references that resolve to this declaration of N. The inclusion of the position P ensures that the ID of the tag is unique in the generated file

Similarly, when a reference to name N resolves to a single declaration of N at position P in file F, the anchor element `<a href="`*F*`#`*N_P*`">`*N*`</a>` renders as the desired hyperlink to the declaration.

In general, a reference to a single name may resolve (unambiguously) to multiple declarations, possibly located in multiple files. Similarly, multiple references may resolve to the same declaration(s). Such information can be added to HTML elements as a `title` attribute, which is usually displayed by HTML browsers as a tooltip while hovering over the element. (Pop-ups or modals could support links to multiple targets, but might be too distracting due to the high density of names in language specifications.)

Generating highlighting. Independently of name-based navigation, language workbenches use syntax highlighting to enhance code readability. To make code rendered on the generated website look the same as in a workbench, the website needs to replicate the colours and fonts that it uses.

Websites often highlight code in many software languages automatically. For example, GitHub highlights code in its repositories for hundreds of languages, using Tree-sitter[4] parsing and context-aware token scanning to recognise different kinds of language construct – also coping gracefully with incomplete or syntactically ill-formed code.

When a code editor of a language workbench supports the same automatic highlighting framework as a website, it might seem attractive to exploit it, and avoid the need for

[4] https://tree-sitter.github.io/tree-sitter/

adding highlighting markup when generating web pages. However, this seems incompatible with the simple approach adopted here for generating hyperlinks in HTML. In any case, websites seldom support automatic highlighting for software *meta*-languages.

So here, highlighting is added to generated HTML using tags of the form `<span class="C">...</span>`, where *C* indicates the (syntactic or lexical) sort of the enclosed text. The rendering of the text – font colour, style, and weight – can then be specified in CSS (generated from data in the language workbench).

Generating a website. When generating a website from code in a repository, it is natural to generate a separate web page for each code file, and copy the directory structure. The website navigation panel can then display the directory structure as a tree, with links to the individual pages as leaves. The detailed rendering of the navigation panel on the website is not so important, because name-based navigation reduces (or even eliminates) the need for drilling down through the directory structure of a code project when browsing or exploring code online.

Static site generators (SSGs) such as MkDocs[5] and Jekyll[6] can generate websites automatically from HTML files. Metadata can be prefixed to the HTML content as so-called front matter, e.g., specified in YAML. HTML can also be embedded directly in Markdown, which facilitates the inclusion of headings and links in the generated source files for the website. An important advantage of relying on an SSG to generate web pages from Markdown is that the resulting HTML can be expected to render properly in any (modern) web browser, on mobile devices as well as desktop and laptop computers.

Figure 1 illustrates the form of the generated HTML. It is a single line from a source file for a hyperlinked twin website (here wrapped to fit the page width).

3 Using Spoofax

The Spoofax Language Workbench[7] currently uses three main meta-languages: SDF3 for syntax, Statix for name binding, and Stratego for transformation. The meta-languages are themselves specified using Spoofax meta-languages (including the now-deprecated SDF2, NaBL, and NaBL2). A further meta-language is ESV, for specifying editor services, including syntax highlighting details. The specifications of all the meta-languages are available as Spoofax language projects on GitHub in repositories of the MetaBorg organisation.[8]

[5]https://www.mkdocs.org
[6]https://jekyllrb.com
[7]https://spoofax.dev
[8]https://spoofax.dev/references/

The Spoofax language workbench is implemented as an Eclipse plugin. To implement generation of hyperlinked websites for an external language specified using Spoofax meta-languages, it is possible to add the required code to the language specification using the plugin. (That is how the CBS-beta website was generated, based on the specifications of the CBS meta-language in SDF3 and NaBL2.)

To add the required code to a Spoofax meta-language such as SDF3, however, it is necessary to build the complete baseline version for bootstrapping Spoofax-2, following the steps explained in the documentation on Spoofax Development.[9] By adjusting the version number in the dependency specification of the relevant meta-language, Spoofax can be used to parse, analyse, and transform its own specifications.

Spoofax provides a Stratego API for reading text from a file, and for parsing it to produce an AST. The parser is generated automatically from the SDF3 specification of the language when the language project is built. The API also supports analysing the name binding of all the files in an Eclipse project, and adding the analysis as annotations on the AST nodes, which can also be accessed using Stratego. And it supports accessing the source text of nodes in the AST, which is based on origin-tracking. The same API includes strategies for obtaining the character positions of name declarations and references.

The generation of a web page with hyperlinks from each source file in a project is specified as a generic traversal in Stratego, independently of the syntax of the language.

For example, Figure 2 shows the Stratego code for generating HTML from references.

Currently, there is no Stratego API for accessing the kinds of individual lexical tokens determined by parsing. As a workaround, highlighting markup is added using pattern matches on the source text (expressed by Stratego strategy combinators) and rendered using CSS generated from an ESV specification. The result corresponds closely to the highlighting in Spoofax.

The documentation site theme used for the main Spoofax documentation website (Material for MkDocs[10]) automatically generates a navigation panel with the same structure as the source project, with language-independent configuration. However, the underlying MkDocs SSG transforms directory names; a plugin[11] is required to ensure that the rendered links in the navigation panel show the untransformed names.

It is straightforward to deploy the generated web pages to GitHub Pages using Actions. Versioned web pages could also be deployed for different releases or branches.[12]

[9]https://spoofax.dev/howtos/development/
[10]https://squidfunk.github.io/mkdocs-material/
[11]https://github.com/lukasgeiter/mkdocs-awesome-pages-plugin
[12]https://squidfunk.github.io/mkdocs-material/setup/setting-up-versioning/

```
  <a href="../AssignmentOperators.sdf3#FieldAccess_938_949" id="FieldAccess_331_342"
title="Referenced at ../AssignmentOperators.sdf3 line 30; ../Disambiguation.sdf3 line 57;
line 16">FieldAccess</a>.<span class="cons_Constructor"><span id="QSuperField_343_354"
title="Not referenced locally, nor via imports">QSuperField</span></span> = &lt;&lt;<a
href="../../names/Names.sdf3#TypeName_145_153" id="TypeName_359_367" title="Defined
at ../../names/Names.sdf3 line 11, 21, 22">TypeName</a>&gt;<span class="cons_String">
.super.</span>&lt;<a href="../../lexical/Identifiers.sdf3#Id_141_143" id="Id_376_378"
title="Defined at ../../lexical/Identifiers.sdf3 line 15, 23">Id</a>&gt;&gt;
```

Figure 1. A fragment of a generated source file for a hyperlinked twin.

```
// gen-node-markup embeds a reference node in an <a> element with a link to the corresponding definition
// - assumes that the name of a reference is a string
// - links to the first definition of the reference

gen-node-markup(|ins, outs, repo-name, cons, ast-list):
  (p, node) -> r
  where
    defs@[def1|_] := <is-string; use-to-defs; sort-by-origin> node
  ; (q, r) := <origin-offset> node
  ; id-attr := $[[node]_[q]_[r]]
  ; gen-tokens-markup(| ins, outs, <subti> (q, p))
  ; direct-path := <direct-path> (<origin-file> node, <origin-file> def1)
  ; (s, t) := <origin-offset> def1
  ; href := $[[direct-path]#[node]_[s]_[t]]
  ; origin-info := <gen-origin-info> (<origin-file> node, defs)
  ; title := <string-replace(|"  ", " ")> $[Defined at [origin-info]]
  ; <fputs> ($[<a href="[href]" id="[id-attr]" title="[title]">], outs)
  ; gen-copy(| ins, outs, <subti> (r, q))
  ; <fputs> ($[</a>], outs)
```

Figure 2. Stratego code for generating HTML from references.

4 Conclusion and Future Work

Using the technique presented in this paper, hyperlinked twin websites have been successfully generated from the syntax of several Spoofax meta-languages (SDF3, NABL, NaBL2, Statix) and from the name binding specification of NaBL.[13] A future release of Spoofax should support generation of hyperlinked twins from code in all the Spoofax meta-languages, so hyperlinked twins can be published for all repositories that use Spoofax language specifications. It may also be possible to support meta-languages used in other frameworks.

Acknowledgments

Gabriël Konat, Daniël Pelsmaeker, and Jeff Smits provided crucial assistance for developing generation of hyperlinked twins using Spoofax. The comments and suggestions of the anonymous reviewers helped improve the paper.

A Appendix

The screenshot in Figure 3 shows how a file from an SDF3 specification of Java looks when editing it in Spoofax. Figure 4 shows how the same file looks when browsing it on GitHub, and Figure 5 shows browsing it on the generated hyperlinked twin. Both Spoofax and the hyperlinked twin support name based navigation in SDF3, in contrast to GitHub.

[13] https://pdmosses.github.io/hyperlinked-twins/

References

[1] Douglas Creager. 2021. Precise Code Navigation for Python, and Code Navigation in Pull Requests. Blog page, https://github.blog/2021-12-09-precise-code-navigation-python-code-navigation-pull-requests.

[2] Douglas A. Creager and Hendrik van Antwerpen. 2023. Stack Graphs: Name Resolution at Scale. In *Eelco Visser Commemorative Symposium (EVCS 2023) (Open Access Series in Informatics (OASIcs), Vol. 109)*, Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 8:1–8:12. https://doi.org/10.4230/OASIcs.EVCS.2023.8

[3] GitHub 2023. About Navigating Code on GitHub. Docs page, https://docs.github.com/en/repositories/working-with-files/using-files/navigating-code-on-github, accessed 2023-09-10.

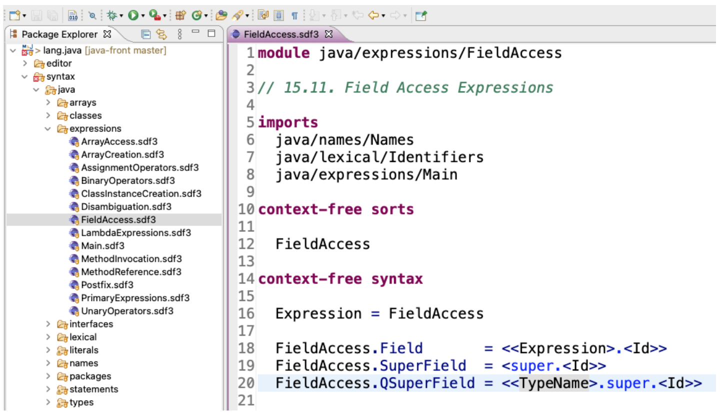
[4] Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax Language Workbench. In *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, 237–238. https://doi.org/10.1145/1869542.1869592

[5] Peter D. Mosses. 2023. Using Spoofax to Support Online Code Navigation. In *Eelco Visser Commemorative Symposium (EVCS 2023) (Open Access Series in Informatics (OASIcs), Vol. 109)*, Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 21:1–21:12. https://doi.org/10.4230/OASIcs.EVCS.2023.21

[6] Patrick Thomson. 2022. Bringing Code Navigation to Communities. Blog page, https://github.blog/2022-04-29-bringing-code-navigation-to-communities.

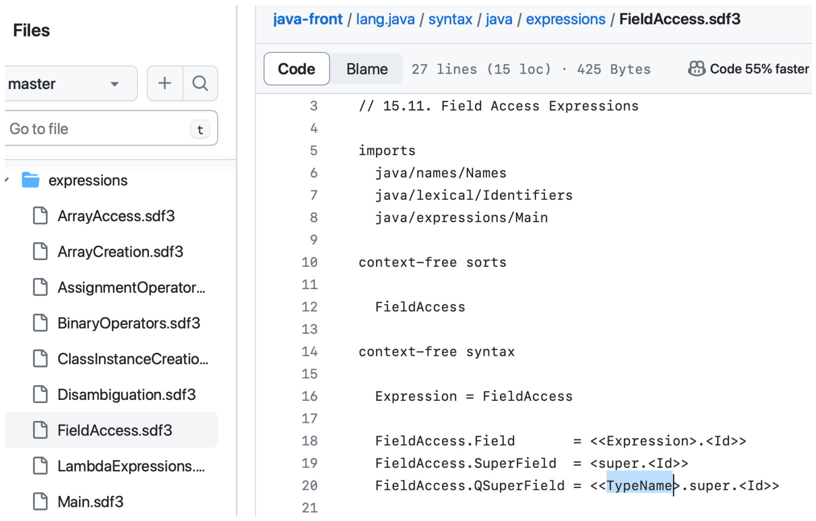
[7] Philip Wadler, Wen Kokke, and Jeremy G. Siek. 2022. *Programming Language Foundations in Agda.* Online book. https://plfa.inf.ed.ac.uk/22.08/

Received 2023-07-07; accepted 2023-09-01



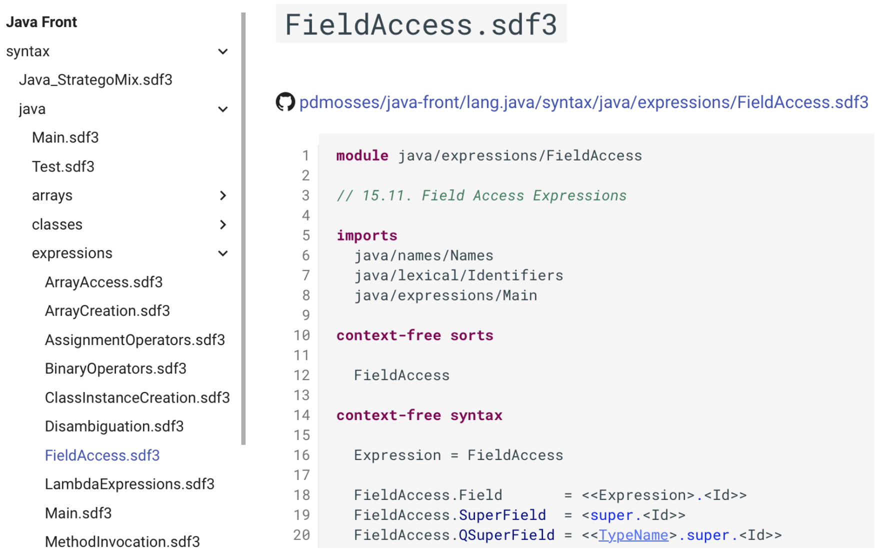
```
module java/expressions/FieldAccess

// 15.11. Field Access Expressions

imports
  java/names/Names
  java/lexical/Identifiers
  java/expressions/Main

context-free sorts

  FieldAccess

context-free syntax

  Expression = FieldAccess

  FieldAccess.Field       = <<Expression>.<Id>>
  FieldAccess.SuperField  = <super.<Id>>
  FieldAccess.QSuperField = <<TypeName>.super.<Id>>

```

Figure 3. Editing a file in the Spoofax language workbench.



```
// 15.11. Field Access Expressions

imports
  java/names/Names
  java/lexical/Identifiers
  java/expressions/Main

context-free sorts

  FieldAccess

context-free syntax

  Expression = FieldAccess

  FieldAccess.Field       = <<Expression>.<Id>>
  FieldAccess.SuperField  = <super.<Id>>
  FieldAccess.QSuperField = <<TypeName>.super.<Id>>

```

Figure 4. Browsing the same file in a GitHub repository.



```
module java/expressions/FieldAccess

// 15.11. Field Access Expressions

imports
  java/names/Names
  java/lexical/Identifiers
  java/expressions/Main

context-free sorts

  FieldAccess

context-free syntax

  Expression = FieldAccess

  FieldAccess.Field       = <<Expression>.<Id>>
  FieldAccess.SuperField  = <super.<Id>>
  FieldAccess.QSuperField = <<TypeName>.super.<Id>>
```

Figure 5. Browsing the same file in the hyperlinked twin.

Practical Runtime Instrumentation of Software Languages: The Case of SciHook

Dorian Leroy
CEA, DAM, DIF
F-91297, Arpajon, France
Université Paris-Saclay, CEA DAM DIF, LIHPC
91297, Arpajon, France
dorian.leroy@cea.fr

Benoit Combemale
University of Rennes
Rennes, France
benoit.combemale@irisa.fr

Benoît Lelandais
CEA, DAM, DIF
F-91297, Arpajon, France
Université Paris-Saclay, CEA DAM DIF, LIHPC
91297, Arpajon, France
benoit.lelandais@cea.fr

Marie-Pierre Oudot
CEA, DAM, DIF
F-91297, Arpajon, France
Université Paris-Saclay, CEA DAM DIF, LIHPC
91297, Arpajon, France
marie-pierre.oudot@cea.fr

Abstract

Software languages have pros and cons, and are usually chosen accordingly. In this context, it is common to involve different languages in the development of complex systems, each one specifically tailored for a given concern. However, these languages create *de facto* silos, and offer little support for interoperability with other languages, be it statically or at runtime. In this paper, we report on our experiment on extracting a relevant behavioral interface from an existing language, and using it to enable interoperability at runtime. In particular, we present a systematic approach to define the behavioral interface and we discuss the expertise required to define it. We illustrate our work on the case study of SciHook, a C++ library enabling the runtime instrumentation of scientific software in Python. We present how the proposed approach, combined with SciHook, enables interoperability between Python and a domain-specific language dedicated to numerical analysis, namely NabLab, and discuss overhead at runtime.

CCS Concepts: • **Software and its engineering** → **Source code generation**; **General programming languages**; **Domain specific languages**; • **Applied computing**;

Keywords: language interoperability, instrumentation, C++, Python, domain-specific languages, scientific computing

ACM Reference Format:
Dorian Leroy, Benoit Combemale, Benoît Lelandais, and Marie-Pierre Oudot. 2023. Practical Runtime Instrumentation of Software Languages: The Case of SciHook. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3623476.3623531

1 Introduction

Software languages are tools providing specific abstractions to support developers in describing efficient solutions (i.e., software systems) to their problems. According to the abstractions provided, a given software language is more or less relevant for a specific concern. For instance, in the field of scientific computing, it is common to use C or C++ for implementing efficient simulation models, complemented by Python to help in data processing or debug instrumentation. This leads to polyglot development of software systems [1].

In this polyglot development context, existing approaches are either focusing on the use of specific libraries unifying on a single language runtime (*e.g.,* Truffle/GraalVM [3, 10], LLVM [5], WebAssembly [4]), or with ad-hoc bindings between different language runtimes defined at the program level (*e.g.,* CORBA [8], CCA [1], CoLoRS [13]). While the former limits to the use of a specific execution platform, the latter requires to define bindings at the program level. This either prevents the use of specific language runtimes (*e.g.,* specific C++ compilers such as GCC, or mainstream Python interpreters like CPython or Pypy), or imposes the overhead of defining *ad-hoc* bindings for each new program.

In this paper, we introduce an approach to support interoperability between different language runtimes (interpreters and compilers) through specific interfaces defined at the language level. We present our approach to define such language interfaces for NabLab[1], a Domain-Specific Language (DSL) for scientific computing, and report on our experience using it with SciHook[2], our C++ library to enable Python-based runtime instrumentation for scientific software.

Using SciHook, computational scientists can write analyses as Python scripts that will run during the simulation, also called in-situ analyses [12], with the full power of Python's libraries for scientific computing [9]. Such analyses can access the data they require in-memory, which allows to turn off expensive input/ouput operations (I/Os) and speeds up the simulation time significantly [2]. Furthermore, with write access to the execution context of the simulation, instruments can implement complex behaviors varying on a case-by-case basis such as the physical behavior of the environment of a simulation, without needing to recompile the simulator.

We experimented the use of SciHook with simulation models implemented in NabLab [6], a compiled DSL for numerical analysis. Based on these experiments, we discuss the required effort for the definition of the behavioral interfaces, the resulting performance at runtime, and the suitability of runtime instrumentation coupled with software language interoperability.

We demonstrate the practicality of enabling interoperability between different languages through well-defined and purposefully designed language behavioral interfaces, with a limited overhead. We also show how the combination of interoperability and runtime instrumentation capability opens up new usage scenarios for the instrumented programs.

The remainder of this paper is as follows. Section 2 presents the motivation behind this work. Section 3 details the proposed approach. Section 4 discusses how we applied the approach to NabLab, using SciHook to enable the instrumentation of NabLab programs in Python. Section 5 presents our evaluation of the overhead induced by the approach over a selection of use cases. Section 6 discusses related works, and Section 7 provides concluding remarks.

2 Motivation

In this paper, we use the field of scientific computing and the software languages used in that field as our illustrative example. The two main use cases for language interoperability in scientific computing are (*i*) C++ and Fortran, to call legacy Fortran code from C++ code [1], and (*ii*) Python and C/C++, to pilot efficient C/C++ libraries from Python-based GUIs or with libraries such as SciPy [1, 11]. In both use cases, interoperability is mainly used to provide program-level bindings for black box software components, and does not allow to interact with the internal state of such components.

Yet, specific operations might be more easily or robustly implemented in a different software language, or require scriptability to access the execution state of the component at runtime. For example, in the context of complex simulation codes, this allows to submit scripts to process data *in-situ* [2], waiving the need to persist complete data sets to disk. Another use case is to expose the execution state at a mathematical level of abstraction, in a language well-suited for mathematical operations (*e.g.,* Python with NumPy support). This in turn enables debugging and domain-specific property monitoring at an adequate level of abstraction for numerical analysts, while keeping the computation-intensive parts of the simulator efficient.

Unfortunately, support for such a gray-box usage of software language interoperability is tedious and error-prone to implement, and does not contribute directly to the business logic of the application. This can be dissuasive for practitioners of scientific computing, who are not software engineers and/or might not have the manpower to spare on these concerns. To remedy this, we present a systematic approach to define and realize a *behavioral interface* dedicated to instrumentation for existing software languages. We then illustrate the approach through a case study using SciHook, a C++ library to enable Python-based runtime instrumentation, and provide performance measurements for a range of relevant use cases for scientific computing.

3 Systematic Runtime Instrumentation of Software Languages

We define runtime instrumentation as the dynamic introduction of code at specific points in the execution of a program, and with controlled access to a subset of the execution state of the program. We detail below our proposed approach for enabling this at the language level, which relies on the systematic definition of *behavioral interfaces* for software languages, that are then realized through an *instrumentation runtime* supporting software language interoperability.

3.1 Defining the Behavioral Interface

In this paper, we adopt a definition for language behavioral interfaces similar to the definition given in [7], albeit more restricted. Indeed the behavioral interfaces defined with our approach consist of the set of language-level events that are exposed by any program written with that language, and that provide their execution context as a parameter. The process for defining such behavioral interfaces is as follows.

Identify execution events. The first step consists in identifying the set of abstract syntax tree (AST) nodes whose execution will result in the emission of an execution event, and crafting a static analysis to extract this set of AST nodes from the AST of a program. At runtime, the set of exposed execution events can then be queried by instruments, allowing them to register to and unregister from these events, and thus be triggered by their emission.

Determine execution contexts. The second step consists in providing the means for instruments to operate on the

[1]https://github.com/cea-hpc/NabLab
[2]https://github.com/cea-hpc/scihook

current execution state when they are triggered by the emission of an execution event. To this end, instruments must be provided an *execution context* when triggered.

However, depending on the purpose for which the behavioral interface is defined, it might not be desirable to expose the complete internal state of a program to its instruments. For example, in the context of execution events emitted on a method call, a behavioral interface designed for regular users might only expose the public fields and methods of the containing object, whereas one designed for developers or expert users might also expose private fields and methods.

Thus, extracting the proper execution context of each execution event of the behavioral interface requires a static analysis tailored to the purpose of the interface.

3.2 Specification of the Instrumentation Runtime

In the remainder of this section, we differentiate the *host language* from the *instrumentation language*. In the context of a given program, the host language is the language used to write the program being instrumented, which we refer to as the *host program*. Instrumentation languages are the languages used to write the instrumentation code (*i.e.*, the instruments) for the host program. We describe below the API that the instrumentation runtime must provide.

Event declaration. The runtime must provide a way for the host program to declare events to which instruments can subscribe. This allows to specify which parts of the application can be instrumented. Note that events can be declared in different granularities, to open more or less parts of the application to instrumentation, similarly to logging levels.

Event subscription. Conversely, the instrumentation runtime must provide a way for the instruments to query, subscribe to, and unsubscribe from execution events. This allows to write instruments that are able to dynamically activate and deactivate themselves, and to identify specific subsets of the exposed execution events to which register.

Event emission. Finally, the runtime must offer a way for the host program to emit execution events, thereby executing the instruments registered to these events, passing along the corresponding execution context. That way, the instrumentation runtime acts as a bridge between host language runtime and instrumentation language runtime.

3.3 Realizing the Interface

Host languages must then provide facilities as part of their infrastructure to derive the instrumentation interface of any program, and realize it through the API of the instrumentation runtime, exposing (*i*) the different execution events to which instruments can register, and (*ii*) a wrapper exposing the associated execution contexts to instruments.

Depending on the host language and on the software language used to implement the instrumentation runtime, using the API of the instrumentation runtime might require the use of foreign function interface or similar technologies.

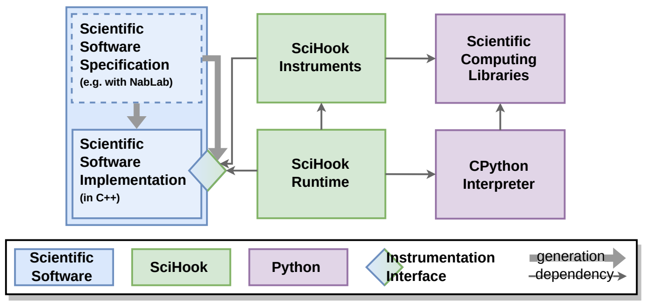

Scientific Software Specification (e.g. with NabLab)
Scientific Software Implementation (in C++)
SciHook Instruments
SciHook Runtime
Scientific Computing Libraries
CPython Interpreter
Scientific Software
SciHook
Python
Instrumentation Interface
generation
dependency

Figure 1. Overview of SciHook.

Finally, interoperability bindings must be defined over the exposed execution contexts so they can be accessed from the desired instrumentation languages. These can be defined systematically for each execution context once bindings for the basic types manipulated by the host language are defined.

4 The Case of SciHook

In this section, we first provide an overview of SciHook, our C++/Python instrumentation runtime. We then discuss the work required to apply the proposed approach to NabLab, a DSL with compiler back-ends targeting C++, using SciHook as the instrumentation runtime.

4.1 SciHook Overview

Figure 1 provides an overview of SciHook. On the left is a piece of scientific software, which can be implemented directly in C++, or generated from its specification when written in a language transpiling to C++ (such as NabLab).

To leverage SciHook, this piece of scientific software provides an instrumentation interface, as defined in the previous Section. This instrumentation interface can be generated directly for a given program (in C++), or it can be generated from the software specification (using NabLab).

At the center of Figure 1 are the SciHook runtime and its registered instruments. Using the SciHook API, applications define their execution events as specified in their instrumentation interface, and trigger those events during the execution. Conversely, SciHook instruments use the SciHook API to register and unregister to the runtime, listing their triggering events from the instrumentation interface.

The SciHook runtime stores the registered events and instruments, and triggers instruments upon the emission of events to which they are registered. Triggered SciHook instruments are provided with the execution context of the application, on which they can perform read and possibly write operations, depending on how the context was exposed, as well as call functions exposed as part of the context.

To achieve this, the SciHook runtime depends on the CPython interpreter, as shown on the right of Figure 1, to which it delegates the execution of SciHook instruments. This means that, when writing instruments, SciHook users have access to the vast ecosystem of libraries for scientific computing such as NumPy, Matplotlib, Numba, and so on [9].

With access to scientific computing libraries and to the execution state of the application, users can craft analyses that are easily plugged into the application and that can be turned on or off at runtime. In addition, the separate "instrumentation state" (*i.e.,* the heap of the CPython interpreter) enables unanticipated, execution-wide analyses (*e.g.,* monitoring temporal properties), without relying on I/Os.

Beyond analysis and debugging, write access to the execution context also allows to configure simulation workflows with Python scripts, from input and output data processing, to simulation initialization, to system behavior specification.

4.2 Experimentation with NabLab

NabLab is a DSL for scientific computing allowing numerical analysts to define their numerical schemes at a level of abstraction close to discrete mathematics, and then generate the corresponding C++ simulator. The C++ code generation infrastructure handles system-level concerns such as memory and programming paradigm (GPU, CPU, MPI, etc.). In this experiment, it is extended to derive the instrumentation interface from the NabLab program, and realize it.

Identifying Execution Events. The execution events exposing the instrumentation points of a NabLab program include calls to the jobs defined in a NabLab program (*i.e.,* its callable entities), as well as all variables writes. Job call events are emitted before and after the triggering calls, and write events are emitted before and after the triggering writes.

In the case of variable writes, we define two kinds of events: global writes and local writes. Global write events are emitted when writing to any global variables. Local write events are emitted when writing to a variable (global or local) in the context of a specific Job. Thus, when a global variable is written to, two write events are emitted before and after the write: a global one and a local one.

As a design decision we made when applying the approach to NabLab, when a variable is written to inside a loop but declared outside of that loop, the corresponding write events are only emitted before/after the entire execution of the loop. This allows to only be notified before and after the complete update of arrays or accumulator variables.

Exposing the Execution Context. With execution events identified, the corresponding execution contexts must be computed to be exposed as part of the instrumentation interface. In the case of job call events, only the global variables and parameters of the call are exposed in the execution context. In the case of global write events, only global variables are considered in the execution context, whereas in the case of local write events, local variables are included as well, as are the parameters provided to the encompassing job.

Realizing the Interface. We realize the instrumentation interface during the C++ code generation.

First, we insert calls to the SciHook runtime to declare the execution events exposed by the interface. Next, we generate each distinct execution context as a C++ struct holding

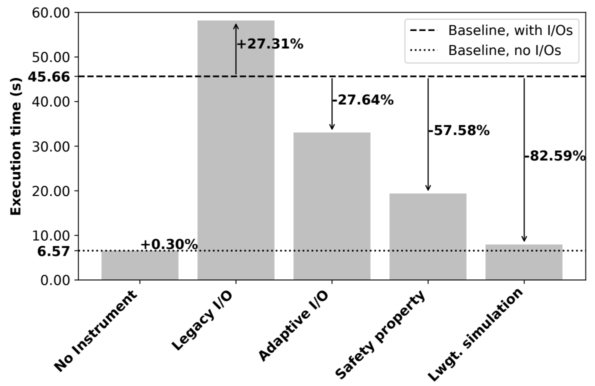


Figure 2. Performance measurements of NabLab model instrumented with SciHook in various use cases.

references to the variables accessible therefrom. We then generate code instantiating these structs at each new execution context, and calls to the SciHook runtime triggering execution events with their execution context.

Finally, we generate Python bindings for these structs, exposing the variables they encapsulate to registered instruments. We also generate a Python-facing interface to expose these execution events to Python-based instruments.

Library for Code Generation. To apply the approach, we developed an Xtend library of around 900 lines of code for analyzing NabLab programs and generating C++ instrumentation code, which we added to the code generation infrastructure of NabLab, also written in Xtend. This library provides facilities for computing the set of execution events of a NabLab program, the set of corresponding execution contexts, and the set of concrete types that must be exposed to Python. To mesh well with scientific computing Python libraries, we exposed the array types of NabLab as NumPy arrays, thereby avoiding expensive copy operations.

The code generation library also provides facilities to generate the Python bindings for the execution contexts, and the CMake build files integrating SciHook into the application. This code generation library is disabled by default and, when enabled, places all instrumentation code between `#ifdef`/`#ifndef` directives. As a result, C++ code is generated without instrumentation by default, and when generated, the instrumentation must be turned on at compile-time, allowing the instrumented code to be used in production.

5 Performance Evaluation

In this Section we evaluate and discuss the overhead induced by SciHook-based instrumentation of NabLab in various use cases. For each use case, we measured the execution time of 30 runs of the same simulation, and provide the average execution time in Figure 2, as well as its relative overhead with regard to the baseline execution time. We performed the measurements on a 11th Gen Intel® Core™ i5-1145G7 @ 2.60GHz × 8, on Ubuntu 20.04.4.

Baseline. For this evaluation, we consider two baselines (dashed lines in Figure 2): the average execution time of the non-instrumented code with, and without I/Os. As we use SciHook instruments instead of post-processing to address the use cases, we disable the built-in I/Os. However, we still compare execution times against the "with I/Os" baseline, as they are mandatory for addressing the use cases with non-instrumented code, through post-processing. Note that we do not consider the overhead of this post-processing in our comparison.

No instrument. This use case shows the overhead induced by the instrumentation alone, without any instrument registered. For the evaluated model, the overhead with regard to the "no I/O" baseline is minimal as it stands at 0.30%.

Legacy I/O. This use case reproduces the original I/Os by calling the original C++ code responsible for I/Os, but through Python bindings, via a SciHook instrument. We estimate that the 27.31% induced overhead is due to the back-and-forth between the Python interpreter and the simulator, and to the absence of link-time optimization. This shows that SciHook is not best used to naively reproduce core functionalities of a C++ simulator such as I/Os. However, the dynamic nature of SciHook allows developers to write adaptive instruments, as discussed next.

Adaptive I/O. This use case leverages the separate Python interpreter state to adapt the frequency of I/Os at runtime, dividing their frequency by 10 once the maximum temperature over the simulation domain goes below 75% percent of its starting value. In our case, this happens after 733 iterations out of 1628, with 895 iterations remaining, yielding 27.64% shorter execution times. However, this requires to be able to determine the "points of interest" of a simulation.

Safety property. In this use case, we monitor a safety property ensuring that the difference between a computed quantity of interest (temperature in this case) and its reference value never exceeds a given threshold across the simulated domain. The 57.58% shorter execution time corresponds to runs where the property is never violated, and is thus monitored during the entire execution.

Lightweight simulation. This use case exemplifies the use of scientific computing for exploratory purposes, where simulations are run in a fast and iterative process. All input data are provided through SciHook instruments, and a plot of the final state of the simulation is the only produced output, meaning no time is spent on I/Os. The 82.59% shorter execution time is an important speed-up compared to non-instrumented code, which allows to quickly obtain insights on a simulated physics problem.

These results demonstrate the practicality of enabling runtime instrumentation at the language level, and instrumenting scientific software to perform analyses during the execution, reducing the need for highly sequential workflows relying on writing data to disk and reading it back in another tool. In particular, the use case of lightweight simulation greatly benefits from this, as the approach allows to prototype simulations quickly. In the case of simulations where the output is kept as a reference and analyzed multiple times by a variety of tools, the I/O-intensive approach remains best, as long as the computing infrastructure is able to handle the amount of data produced by the simulation.

6 Related Work

We identified two categories of related works in the context of enabling software language interoperability.

The first category regroups approaches providing interoperability inside a single language runtime, through a unified intermediate representation used by all supported languages. This is the case of Truffle/GraalVM [14], LLVM [5], or WebAssembly [4]. This means that interoperability with specific language runtimes such as Pypy, CPython, GCC, or the Intel compiler must be implemented at the program level. In comparison, our proposed approach aims to support interoperability between language runtimes.

The second category regroups approaches like CORBA [8] and CCA [1], which relies on interfaces defined for each component to allow them to communicate, whether they are defined in the same language or not. However, as interfaces are defined at the component level, each new component necessitates the definition of its interface, and its implementation by the component. In our proposed approach, the interface is instead defined at the language level, and realized at compile-time, and can thus be reused for each new component defined with that language.

7 Concluding Remarks and Perspectives

In this paper, we presented an approach to support interoperability between different software languages, relying on the definition of language behavioral interfaces, and on the use of an instrumentation runtime to realize those interfaces. We demonstrated the approach on the NabLab DSL, using SciHook as our instrumentation runtime to realize the interface and provide interoperability between C++ (the target language of NabLab), and Python. The ability to instrument scientific software in Python allowed for greater agility when prototyping and debugging, illustrating the benefits of opening language-induced silos to other languages.

From here, we envision several threads of future work. A first perspective is to explore the interplay between the instrumentation interface of a language and testing frameworks, with the goal to provide testing support out-of-the-box to languages exposing an instrumentation interface. Another perspective is to explore solutions based on just-in-time compilation to reduce the time spent in the Python interpreter, to circumvent the pitfalls of Python such as its global interpreter lock, minimizing the crossing of language boundaries, and optimizing Python code implementing complex behaviors and/or acting as glue between native libraries.

References

[1] David E Bernholdt, Benjamin A Allan, Robert Armstrong, Felipe Bertrand, Kenneth Chiu, Tamara L Dahlgren, Kostadin Damevski, Wael R Elwasif, Thomas GW Epperly, Madhusudhan Govindaraju, et al. 2006. A component architecture for high-performance scientific computing. *The International Journal of High Performance Computing Applications* 20, 2 (2006), 163–202.

[2] Hank Childs, Janine Bennett, Christoph Garth, and Bernd Hentschel. 2019. In Situ Visualization for Computational Science. *IEEE Computer Graphics and Applications* 39, 6 (2019), 76–85.

[3] Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, Mikel Luján, and Hanspeter Mössenböck. 2018. Cross-Language Interoperability in a Multi-Language Runtime. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 40, 2 (2018), 1–43.

[4] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 185–200.

[5] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

[6] Benoit Lelandais, Marie-Pierre Oudot, and Benoit Combemale. 2018. Fostering Metamodels and Grammars within a Dedicated Environment for HPC: the NabLab Environment (Tool Demo). In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*. 200–204.

[7] Dorian Leroy, Erwan Bousse, Manuel Wimmer, Tanja Mayerhofer, Benoit Combemale, and Wieland Schwinger. 2020. Behavioral Interfaces for Executable DSLs. *Software and Systems Modeling* 19 (2020), 1015–1043.

[8] Object Management Group. [n. d.]. Common Object Request Broker Architecture. https://www.omg.org/spec/CORBA.

[9] Travis E Oliphant. 2007. Python for Scientific Computing. *Computing in science & engineering* 9, 3 (2007), 10–20.

[10] Michael Van De Vanter, Chris Seaton, Michael Haupt, Christian Humer, and Thomas Würthinger. 2018. Fast, Flexible, Polyglot Instrumentation Support for Debuggers and Other Tools. *The Art, Science, and Engineering of Programming* 2, 3 (2018), 14–1.

[11] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature methods* 17, 3 (2020), 261–272.

[12] Venkatram Vishwanath, Mark Hereld, and Michael E Papka. 2011. Toward Simulation-Time Data Analysis and I/O Acceleration on Leadership-Class Systems. In *2011 IEEE Symposium on Large Data Analysis and Visualization*. IEEE, 9–14.

[13] Michal Wegiel and Chandra Krintz. 2010. Cross-Language, Type-Safe, and Transparent Object Sharing for Co-Located Managed Runtimes. *ACM Sigplan Notices* 45, 10 (2010), 223–240.

[14] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. 187–204.

Received 2023-07-07; accepted 2023-09-01

Author Index