Well-Formed and Scalable Invasive Software Composition

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Dipl.-Inf. Sven Karol
geboren am 2. November 1981 in Berlin

Gutachter:
Prof. Dr. rer. nat. habil. Uwe Aßmann
(Technische Universität Dresden)

Prof. Dr. Welf Löwe
(Linnæus University, Växjö, Schweden)

Fachreferent:
Prof. Dr.-Ing. habil. Hans-Ulrich Karl
(Technische Universität Dresden)

Tag der Einreichung: 17. November 2014
Tag der Verteidigung: 18. Mai 2015

Confirmation

I confirm that I independently prepared this thesis with the title *Well-Formed and Scalable Invasive Software Composition* and that I used only the references and auxiliary means indicated in the thesis.

Dresden, November 17, 2014

Dipl.-Inf. Sven Karol

Abstract

Software components provide essential means to structure and organize software effectively. However, frequently, required component abstractions are not available in a programming language or system, or are not adequately combinable with each other. *Invasive software composition (ISC)* is a general approach to software composition that unifies component-like abstractions such as templates, aspects and macros. ISC is based on *fragment composition*, and composes programs and other software artifacts at the level of syntax trees. Therefore, a unifying *fragment component model* is related to the context-free grammar of a language to identify extension and variation points in syntax trees as well as valid component types. By doing so, *fragment components* can be composed by transformations at respective extension and variation points so that always valid composition results regarding the underlying context-free grammar are yielded.

However, given a language's context-free grammar, the composition result may still be incorrect. Context-sensitive constraints such as type constraints may be violated so that the program cannot be compiled and/or interpreted correctly. While a compiler can detect such errors after composition, it is difficult to relate them back to the original transformation step in the composition system, especially in the case of complex compositions with several hundreds of such steps. To tackle this problem, this thesis proposes *well-formed ISC*—an extension to ISC that uses *reference attribute grammars (RAGs)* to specify fragment component models and *fragment contracts* to guard compositions with context-sensitive constraints. Additionally, well-formed ISC provides *composition strategies* as a means to configure composition algorithms and handle interferences between composition steps.

Developing ISC systems for complex languages such as programming languages is a complex undertaking. Composition-system developers need to supply or develop adequate language and parser specifications that can be processed by an ISC composition engine. Moreover, the specifications may need to be extended with rules for the intended composition abstractions. Current approaches to ISC require complete grammars to be able to compose fragments in the respective languages. Hence, the specifications need to be developed exhaustively before any component model can be supplied. To tackle this problem, this thesis introduces *scalable ISC*—a variant of ISC that uses *island component models* as a means to define component models for partially specified languages while still the whole language is supported. Additionally, a scalable workflow for *agile composition-system development* is proposed which supports a development of ISC systems in small increments using modular extensions.

All theoretical concepts introduced in this thesis are implemented in the *Skeletons and Application Templates* framework SkAT. It supports "classic", well-formed and scalable ISC by leveraging RAGs as its main specification and implementation language. Moreover, several composition systems based on SkAT are discussed, e.g., a well-formed composition system for Java and a C preprocessor-like macro language. In turn, those composition systems are used as composers in several example applications such as a library of parallel algorithmic skeletons.

Acknowledgments

First of all, I would like to thank my supervisor Uwe Aßmann who gave me the opportunity to work as a research assistant and later also as a PhD student in the Software Technology Group of TU Dresden. During all these years, he enabled me to get an insight to plenty of different research areas in software engineering and to discover the non-obvious relations between them. For this, I am deeply grateful. I also would like to thank my best friend and colleague Christoff Bürger. I remember an uncountable number of challenging and fruitful discussions that helped me to develop a better understanding of what computer science is all about. I believe without his friendship and all these discussions, I never would have taken the long road of a research career.

During my six years in the Software Technology Group, I had the pleasure of working with many great personalities. Until he left the group in 2008, Steffen Zschaler guided me in paper writing and research best practices. In 2008 and 2009, I mainly worked in the EMFText team together with Jendrik Johannes, Mirko Seifert, Christian Wende and Florian Heidenreich. During these two years, we had many controversial discussions on the "quo vadis" of EMFText, which I believe made the tool a very successful one in the end. In retrospect, I learned a lot about team-building mechanisms as well as on the organization of software projects. At the end of 2009, Christoff Bürger and I started our research on the application of attribute grammars in model-driven software engineering, which gave rise to the JastEMF tool. I am still very proud of the outcome of this collaboration and hope that this research may be continued in the future. In 2010 and 2011, I had a severe personal crisis which many people helped me to get through. Uwe and Christoff lent me their ears and especially Hannes Strass helped me a lot in the beginning. I also would like to thank may family and my girlfriend Maria Ristau for their support and patience during that very difficult time. In 2011, I finally decided to stay in the group and started working on the topic of this thesis. Michael Thiele joined me as a student to write his Master's thesis on model weaving with static attribute grammars. I remember an uncountable number of half-day workshops where we discussed the dangerous shallows of attribute grammars and composition. At the end of 2011, Gerhard Goos—one of the veterans of computer science in Germany and Uwe's former PhD supervisor—visited our group for a workshop. I had the chance to give him a presentation on my topic. The discussion and his comments aftwards strengthened my belief that it is a worthwhile one and encouraged me to continue my work. In 2012, I had the pleasure of working with Claas Wilke and Julia Schroeter on a shared topic. Both already were in the middle of their PhDs and we frequently discussed about our progress and time constraints. With Julia, I also worked in the "Informatik Förderverein" FFFI where she took a lot of work and responsibilities, which often helped me to get the head free in this respect. In 2013, I was able to

finish the first usable versions of the SkAT tool and it turned out that my ideas really worked out. Yes! Moreover, Christoff and I had two times the chance to visit Görel Hedin's group in Lund. We had plenty of discussions on efficient attribute evaluation and the JastAdd system, which is used in the SkAT implementation. In the beginning of 2014, I left the Software Technology Group to completely focus on my thesis writing. Johannes Mey took over my responsibilities in the orchestration path of the CfAED project and does a great job since then.

Of course, many other people in the group and the departement crossed my way. I would like to thank Sandro Schmidt, Ronny Fritzsche and Matthias Niederhausen of the Multimedia Technology Group for organizing all the Pizza rounds and keeping an eye on Bruno the bear. I also thank Jan Polowinski, Andreas Bartho and Sebastian Götz for many interesting conversations, their support and motivation from time to time. And I would not like to forget our secretary Katrin Heber for writing all these coffee-payment reminders.

A very special thanks goes to Christoff Bürger, Johannes Mey, Christoph Seidl, Sandro Schmidt and Christa Brueckner who supported me by providing reviews and discovering mistakes.

And last but not least I would like to thank my parents Michael and Elke for all their long-term support and all their parental care during the last three decades. All my love goes to Maria and our newborn sunshine Emelie. Maria greatly backed me in all these difficult times with her optimism and always encouraged me to keep on writing.

Sven Karol,
Dresden, November 2014.

Publications

This thesis is partially based on the following peer-reviewed publications:

- Sven Karol, Christoff Bürger, and Uwe Aßmann [2012]. "Towards Well-Formed Fragment Composition with Reference Attribute Grammars." In: *Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE '12)*. New York, NY, USA: ACM, pp. 109–114. ISBN: 978-1-4503-1345-2. DOI: `10.1145/2304736.2304755`

- Sven Karol, Matthias Niederhausen, Daniel Kadner, Uwe Aßmann, and Klaus Meißner [2011]. "Detecting and Resolving Conflicts Between Adaptation Aspects in Multi-staged XML Transformations." In: *Proceedings of the 11th ACM Symposium on Document Engineering (DocEng '11)*. New York, NY, USA: ACM, pp. 229–238. ISBN: 978-1-4503-0863-2. DOI: `10.1145/2034691.2034738`

The following peer-reviewed publications cover work that is closely related to the content of the thesis, but not contained herein:

- Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende [2013]. "Model-Based Language Engineering with EMFText." In: *Generative and Transformational Techniques in Software Engineering IV*. vol. 7680. Lecture Notes in Computer Science. Berlin / Heidelberg, Germany: Springer, pp. 322–345. ISBN: 978-3-642-35991-0. DOI: `10.1007/978-3-642-35992-7_9`

- Uwe Aßmann, Andreas Bartho, Christoff Bürger, Sebastian Cech, Birgit Demuth, Florian Heidenreich, Jendrik Johannes, Sven Karol, Jan Polowinski, Jan Reimann, Julia Schroeter, Mirko Seifert, Michael Thiele, Christian Wende, and Claas Wilke [2012]. "DropsBox: the Dresden Open Software Toolbox." In: *Software & Systems Modeling* 13.1, pp. 133–169. ISSN: 1619-1366. DOI: `10.1007/s10270-012-0284-6`

- Christoff Bürger, Sven Karol, Christian Wende, and Uwe Aßmann [2011]. "Reference Attribute Grammars for Metamodel Semantics." In: *Proceedings of the Third International Conference on Software Language Engineering (SLE 2010)*. Vol. 6563. Lecture Notes in Computer Science. Berlin / Heidelberg, Germany: Springer, pp. 22–41. ISBN: 978-3-642-19439-9. DOI: `10.1007/978-3-642-19440-5_3`

- Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, Michael Thiele, Christian Wende, and Claas Wilke [2011]. “Integrating OCL and Textual Modelling Languages.” In: *Workshops and Symposia at MODELS 2010, Reports and Revised Selected Papers*. Vol. 6627. Lecture Notes in Computer Science. Berlin / Heidelberg, Germany: Springer, pp. 349–363. ISBN: 978-3-642-21209-3. DOI: `10.1007/978-3-642-21210-9_34`

- Christoff Bürger, Sven Karol, and Christian Wende [2010]. “Applying Attribute Grammars for Metamodel Semantics.” In: *Proceedings of the International Workshop on Formalization of Modeling Languages (FML ’10)*. New York, NY, USA: ACM, 1:1–1:5. ISBN: 978-1-4503-0532-7. DOI: `10.1145/1943397.1943398`

- Sven Karol, Martin Heinzerling, Florian Heidenreich, and Uwe Aßmann [2010]. “Using Feature Models for Creating Families of Documents.” In: *Proceedings of the 10th ACM Symposium on Document Engineering (DocEng ’10)*. New York, NY, USA: ACM, pp. 259–262. ISBN: 978-1-4503-0231-9. DOI: `10.1145/1860559.1860618`

- Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende [2009a]. “Derivation and Refinement of Textual Syntax for Models.” In: *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2009)*. Vol. 5562. Lecture Notes in Computer Science. Berlin / Heidelberg, Germany: Springer, pp. 114–129. ISBN: 978-3-642-02673-7. DOI: `10.1007/978-3-642-02674-4_9`

- Matthias Niederhausen, Sven Karol, Uwe Aßmann, and Klaus Meißner [2009]. “Hyper-Adapt: Enabling Aspects for XML.” in: *Proceedings of the 9th International Conference on Web Engineering (ICWE ’09)*. Vol. 5648. Lecture Notes in Computer Science. Berlin / Heidelberg: Springer, pp. 461–464. ISBN: 978-3-642-02817-5. DOI: `10.1007/978-3-642-02818-2_38`

Contents

1 Introduction

Divide and conquer is one of the most essential principles in the engineering of complex software systems. It directly correlates with the *separation of concerns (SoC)* as the most basic strategy for problem solving in computer science, which Dijkstra felicitously characterized as “yet the only available technique for effective ordering of one’s thoughts” [Dijkstra 1982, p. 61]. In software engineering, *components* are essential to structure systems, hide implementation detail, to “conquer” complex problems and separate concerns. Components occur on all levels of a software system. They can be binary, distributed and commercial [Szyperski 2002], they can be reusable libraries of standard algorithms for different platforms [Mcilroy 1969] or even any piece of program code or model [Nierstrasz and Tsichritzis 1995; Aßmann 2003]. Besides structuring, software components are a key enabling technology for variability and reuse in software systems. Therefore, software components provide *interfaces* making their signatures and data types, interaction protocols and *contracts* [Meyer 1992], as well as variation and extension points explicit. Depending on the component kind and composition technology, interfaces may be realized differently. For example, in commercial software, interfaces are often used to keep implementation detail as trade secrets and to decouple software systems. However, in the engineering process of software it is necessary to (de-)compose software at the level of source code so that a plain *black-box* view on components often is not adequate as it does not help to organize program sources efficiently. Consequently, other methodologies to software composition such as *invasive software composition (ISC)* [Aßmann 2003] allow for composition at the level of program elements. ISC transforms structural representations of programs by applying *composition operators* at extension and variation points according to a *gray-box component model* which supports interfaces but does not hide the implementation. This thesis proposes two extensions to ISC. *Well-formed ISC* primarily addresses the problem that programs may not be well-formed after composition with the ISC method (i.e., may not be executable). Complementarily, *scalable ISC* is concerned with reducing the development efforts that are required to implement composition systems based on ISC.

Composition in software engineering. In software engineering, two incarnations of *divide and conquer* are typically used to arrange software components: *hierarchical decomposition (HiDec)* and the *separation of cross-cutting concerns (SoCC).* HiDec is about structuring components using relations that describe hierarchies such as *part of.* The reason for the application of HiDec is simple: in the physical world, people usually expect one object that occurs as a part of another object to be no part of a third unrelated object at the same time—a keyboard is always part of only one piano. In the virtual world of software, this is also expected as software reflects the real world. Thus, programs are typically organized in similar ways. For example, artifacts in object-oriented languages such as Java and C# are organized along hierarchical namespaces that must even be reflected in the folder structure in some cases (e.g., in Java). Moreover, classes are hierarchically organized type declarations that contain methods, fields and other classes. To foster reuse in hierarchically structured programs, composition mechanisms such as class-based inheritance, delegation between objects as well as file inclusion and namespace imports are common approaches.

Since every part of a program requires an address to be locatable, architecting a software system without HiDec seems difficult. However, with HiDec only one decomposition is possible at a time, which is typically far from being an optimal solution for all *concerns* of a software. Besides the functional *core*, there are also concerns that are *cross-cutting* in nature with respect to *HiDec* decomposition of a program [Tarr et al. 1999] that has been chosen by a programmer. During the last two decades, novel approaches for separating concerns orthogonal to the technical decomposition have been researched. The most prominent approach that supports the *SoCC* is *aspect-oriented programming (AOP)* [Kiczales et al. 1997]. A system's property is an aspect "if it can not be cleanly encapsulated in a generalized procedure" [Kiczales et al. 1997, p. 226], where "generalized procedure" denotes a class, method or similar kind of abstraction. A very typical example of a cross-cutting concern is debugging via printing log statements to console or a file. For fine-grained logging, programmers bloat their code with log statements that are, on the one hand, *tangled* with functional code which makes it worse to be read and understood. On the other hand, the very similar pieces of code are *scattered* over the program and must be maintained with the evolving program. Typical AOP realizations [Kiczales et al. 2001; Bergmans and Aksit 2001; Mezini and Ostermann 2003] allow developers to encapsulate cross-cutting concerns in separate components without the need to change the code of the core program. Moreover, the *multi-dimensional separation of concerns (MDSoC)* approach [Tarr et al. 1999; Ossher and Tarr 2001] unifies HiDec and SoCC by providing a methodology for separating all concerns of a program (i.e., functional and cross-cutting ones) along equally ranked dimensions in a unified way so that the concepts of the programming language are not dominating the physical structure of a program's components.

Component technology by metaprogramming. The research documented by this thesis deals with a special kind of component technology based on *metaprogramming*. Metaprograms are programs that *modify*, *generate* or *transform* other programs at compile time (i.e., *static* metaprogramming) or runtime (i.e., *dynamic* metaprogramming). Component abstractions for HiDec or SoCC can be realized as metaprograms (cf. [Aßmann 2003]). Besides that, it is a key enabling technology for some accompanying software-composition approaches. *Template metaprogramming (TMP)* is a way to express variability and to increase reuse of software components [Czarnecki and Eisenecker 2000]. Templates allow developers to realize *blueprint components*. Blueprint components are generic pieces of program code that have to be parametrized with usage-specific bindings and thus can be used in various different contexts. For example, the standard library of C++ provides several predefined class and function templates that can be parametrized with specific types (e.g., lists, matrices) or adjusted to a specific platform. Macros are a special class of static metaprograms that are integrated transparently into programs of a host language [Weise and Crew 1993; Brabrand and Schwartzbach 2002] while very different possibilities to design and implement macro languages exist. *Syntactic* macros as in Scheme [Kelsey et al. 1998] or LISP [Steele Jr. 1990] are extensions of the corresponding programming language. Thus, syntactic macro languages work on the syntax-tree representation of a program so that macros are not allowed to perform transformations that violate the language grammar and always yield syntactically correct programs. Other macro languages such as the C preprocessor (CPP) only work on the lexical level, i.e., on the token-stream representation of a program and do *not* depend on the actual language grammar. Hence, lexical macro languages are out-of-the-box portable to other languages without being explicitly designed for them or even without being designed for any specific language.

Frameworks for component technology. Using standard metaprogramming for developing component abstractions is an elaborate task as it does not provide a systematic framework for component-based systems in software engineering. Therefore, general approaches for modeling and implementing software-composition abstractions emerged (e.g., [Batory et al. 2004; Erwig and Walkingshaw 2011]). This thesis is centered around ISC [Aßmann 2003] as a general metaprogramming-based approach to software composition regarding it as both—implementation technique and conceptual framework. ISC is suitable for the implementation of arbitrary HiDec, SoCC and TMP systems and guarantees composition correctness at the level of context-free syntax. ISC has its focus on the practical engineering of software-composition systems. The first ISC system was COMPOST—a Java framework [Aßmann 2003; Heuzeroth et al. 2006]. It has an extensible component model, and compositions are described as Java programs. COMPOST provides a default implementation of a composition system for Java which supports templates and AOP. The work of [Henriksson 2009] implemented ISC as a generative approach based on context-free grammars called *universal invasive software composition (U-ISC)*, which has been used to realize several composition systems for *domain-specific languages (DSLs)*. Afterwards, the work of [Johannes 2011] introduced the approach of *universal invasive software composition* *for typed graphs (U-ISC/Graph)*, which is suitable for realizing composition systems in the model-driven engineering (MDE) [Kent 2002] domain. U-ISC/Graph is based on the observation that graphical models are usually graph-structured and metamodels provide means to specify such languages.[1]

Similar to most of the other composition approaches above—the standard approaches to template metaprogramming and syntax macros—ISC works at the syntactic level. Consequently, invasive composition always yields syntactically correct results. However, a consideration of syntax alone is often not sufficient. For example, C++ templates are well-known for causing cryptic error messages in the C++ compiler that are hard to debug. This is caused by the separation of the template expansion from the actual compilation, which makes them unaware of most parts of the static language semantics encoded in the C++ compiler. In contrast, the AOP implementation AspectJ [Kiczales et al. 2001] and the corresponding editors of the AspectJ Development Tools (AJDT) [Eclipse Foundation 2013a] show how an integration with large parts of Java's static semantics can be employed beneficially to improve on error reporting and usability.

1.1. Thesis Topic

Today's frameworks for metaprogramming-based composition systems lack systematic approaches to tie language semantics with composition abstractions. Thus, the main objective of this thesis is to develop an approach that enables a systematic integration of composition abstractions with static language semantics. The proposed solution—*well-formed ISC*—uses reference attribute grammars (RAGs) [Hedin 2000] as a basic model for specifying and implementing ISC concepts. RAGs are a special kind of attribute grammar (AG) [Knuth 1968; Knuth 1971], and a well-know specification formalism and implementation tool for the semantics of formal languages. Using RAGs, the thesis directly improves on the ISC approaches of [Aßmann 2003] and [Henriksson 2009].

Another equally important problem when developing metaprogramming-based composition systems are *interferences* [Kniesel and Bardey 2006] between single steps of a composition. ISC frameworks should provide means to make composition transparent to their users in such a way that they can understand how a composition result is achieved and how it can be reproduced deterministically. Therefore, this thesis extends ISC with *strategies* (cf. *strategic program transformations*, [Visser 2004]) to parametrize compositions and to better handle interference related problems.

While the unifying model of ISC and well-formed ISC is an advanced general approach to fragment composition, it also requires advanced knowledge of developers to apply the technology to develop new composition abstractions for their fragment languages. One potential way to reduce complexity is to provide DSLs to specify those abstractions, as shown by the works

[1]Of course, with the U-ISC and U-ISC/Graph approaches, the term *metaprogramming* has to be put into a broader context, since a DSL or a model-based language is not necessarily a program, but still something that has an execution semantics or translational semantics.

of [Henriksson 2009] and [Johannes 2011]. However, DSLs also imply a trade-off between simplicity and expressibility. For example, the approach of [Henriksson 2009] provides a set of simple DSLs to specify composition systems but it is also restrictive, as it will be discussed in a later chapter. This thesis aims at providing a more flexible way of composition-system specification and will use a set of *reusable RAG patterns* to specify composition systems that can be realized by common RAG tools.

Another source of complexity depends on the language a composition system should be developed for. Syntax and semantics of programming languages often tend to be complex and ambiguous so that it becomes difficult (or even impossible) to use a complex approach like ISC or well-formed ISC. For example, it is well-known that Java's syntax and semantics is simple enough to be covered by generic (generative) approaches such as AGs and parser generators. Contrastingly, C++ is more complex because of its enormous number of language features and potentially ambiguous concepts. Hence, to also be able to support languages like C++ with invasive composition systems, this thesis proposes *minimal ISC*, which basically works on the lexical level and can handle incomplete language specifications. The approach is also suitable for an agile style of composition system development as it can be extended with constructs up to full ISC support. The combination of minimal, classic and well-formed ISC with an agile style of composition-system development is called *scalable ISC* in this thesis.

The results of this thesis are implemented as a proof of validity as part of the *Skeletons and Application Templates (SkAT)* framework on the basis of the JastAdd RAG tool [Hedin 2011]. SkAT for Java (SkAT4J) is a well-formed composition system for Java which imitates the original look and feel of the COMPOST ISC implementation, but adds semantics awareness based on RAGs and composition strategies. SkAT4J is used to implement a code-generation use case for a domain-specific code-generation framework. Furthermore, it is employed to realize a library of parallel patterns (cf. *algorithmic skeletons* [Cole 1989]) as compositional abstractions for writing parallel applications in Java. The scalable ISC approach is supported by three smaller SkAT-based systems. Most notably, the *Universal Extensible Preprocessor (UPP)* is a CPP-like language-independent macro processor, which can be extended with additional composition abstractions and partial target-language support.

1.2. Thesis Outline

Including this introduction, the thesis is divided into nine chapters, which are summarized below.

Chapter 2 introduces an example of an ISC-based code generator for a domain-specific language. The purpose of this example is three-fold. First, readers become more familiar with the very basic terminology of ISC. Second, based on the scenario, a detailed problem analysis is conducted, discussing benefits and drawbacks of classic ISC systems w.r.t. soundness and implementation effort. From the problem analysis, the thesis objectives are derived. Third, in the course of this thesis the example is used as a composition scenario for the developed SkAT framework as well as in a comparative study of existing ISC systems.

Chapter 3 on background and terminology prepares the ground for understanding the approaches discussed and developed in this thesis. It introduces formal languages, grammars, trees and graphs. Moreover, tree-based language-processing schemes are discussed as well as the kinds of trees used in the core of fragment components. Most importantly, the chapter introduces *Simple Attribute Grammar Specification Language (SimpAG)*—the RAG specification language later used in the chapters on well-formed and scalable ISC.

Chapter 4 discusses the state of the art in ISC and introduces a formal model of fragment component models and the basic ISC composition operators. Existing implementations of ISC are discussed and compared. Moreover, they are evaluated w.r.t. the formal model by discussing their specification languages and composition realizations.

Chapter 5 introduces well-formed ISC. Using SimpAG, a RAG-based specification approach for fragment component models is developed by providing specific attribution patterns and generic equations performing typical component-model tasks. Moreover, composition operators are modeled as an extension to the fragment component model and three configurable composition algorithms are developed and provided as composition strategies. Additionally, the chapter introduces *fragment contracts* as means to specify pre- and postconditions for single composition steps and invariants for verification. Fragment contracts are integrated with the RAG-based component model and incorporate context-sensitive constraints of the component language.

Chapter 6 presents the SkAT framework. SkAT transfers the approach of well-formed ISC into a JastAdd-based framework consisting of multiple reusable RAG modules. In comparison to the classic ISC systems, SkAT adds fragment contracts and composition strategies while providing a conventional Java-based composition API. In the Chapter's second part, SkAT4J is discussed as a well-formed fragment composition system for Java (version 1.5). The system is then used to implement the domain-specific code generator of Chapter 2, which is also compared to the respective implementations based on the existing ISC systems discussed in Chapter 4. Moreover, SkAT4J is used to implement a small fragment library of parallel algorithmic skeletons.

Chapter 7 discusses minimal and scalable ISC. Again, SimpAG is used as a specification framework for the developed approaches. To support component languages partially, *island fragment component models* are introduced and used in three example composition systems. Moreover, a workflow for scalable composition system development is proposed. The approach is based on *island grammars* [Moonen 2001].

Chapter 8 discusses and analyzes related work that has not been investigated in one of the other chapters.

Chapter 9 is draws conclusions w.r.t. the achieved research results, and discusses open issues and potential future work.

2 A Motivating Example

ISC is a fragment-composition approach based on context-free grammars (CFGs). The notion of *fragment* stems from the BETA programming language, where fragments are syntax-based modules for program modularization [Madsen et al. 1993]. A fragment module[1] is “associated with a name and a syntactic category . . . [and] a string of terminal and nonterminal symbols” [Madsen et al. 1993, p. 259] derived from the respective syntactic category, i.e., a correctly typed code snippet with placeholders. Nonterminal symbols in a fragment are called *slots* and define a range where other fragments may be added to the program [Madsen et al. 1993, p. 258]. In ISC, the notions of *declared hooks* and slots are used interchangeably [Aßmann 2003]. Additionally, ISC provides the concept of *implicit hooks*, which denotes parts of a fragment that are not explicitly declared for variation, but are implicitly derived from the fragment structure.

In the following, a short example on an invasive composition system is discussed. Afterwards, immanent problems of the ISC approach, and especially its incarnations in the COMPOST systems and the universal approaches by [Henriksson 2009] and [Johannes 2011], are discussed.

2.1. A Code Generation Use Case

To emphasize how the ISC approach basically works, an academic example that illustrates the implementation of a generative *business application framework (BAF)* is developed subsequently. The example is implemented in Java and provides a textual DSL to ease the specification of business domain models and to generate Java code.

[1]Fragment modules are also called *fragment forms* in BETA.

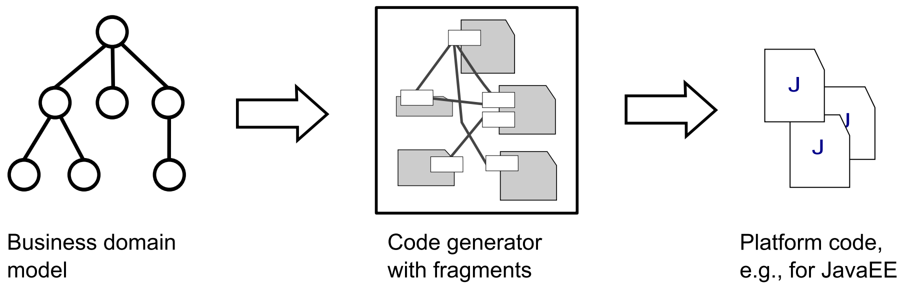

J
J
J
Business domain model
Code generator with fragments
Platform code, e.g., for JavaEE

Figure 2.1.: An ISC-based code generator for domain objects of a business application.

Figure 2.1 shows the basic code generation process in the BAF. It starts from some model of the business domain provided by the application developer. While it does not matter how the model was created (e.g., using some graphical or textual modeling editor), the model representation needs to be compatible with the generator, which must be able to read the input. The task of the code generator is to transform the input model into source code of an executable program that runs within the business application framework. In the case under consideration, the emitted code is Java, but could also be any other programming language. Since the generator is based on ISC, it has several Java fragments with slots and hooks as basic building blocks. The fragments are parametrized and extended by inspecting the input business model and executing a composition (meta)program. Languages used to implement fragments (Java in the BAF case) are called *fragment languages* or *component languages*, respectively.

The code emitted by the BAF generator reuses some basic classes of the framework's application programming interface (API). Listing 2.1 shows an excerpt of this API which provides an abstract Java class `BusinessObject` as a super class of all business objects that should be plugged into the framework. It has a unique `id` for storing the object in a database and an abstract method `asString` for generating a String representation of a business object, e.g., for logging purposes. The second class `Person` provides an API for roles (e.g., customer) that may occur in business applications with typical properties such as `firstName`, `lastName` and `age`.

To generate a business application, the framework has to be instantiated by specifying a business model. Since a full-fledged business model would go beyond the scope of a simple example application, only the role-model part is considered. Listing 2.2 contains a textual role-model specification in this DSL. It simply defines three basic stakeholders (cf. [Czarnecki 1999]). An `Employee` has a date that specifies when he was employed by the company and has a certain weekly workload. A `Customer` has a certain default discount on the regular price while a `Shareholder` of the company has a certain percentage of ownership. Finally, a combined role `EmployeeCustomer` is defined that represents an employee who also acts as a customer with a default discount of 20 percent.

The textual DSL specification is parsed by a parser and transformed into an object graph as shown in Figure 2.2. Typically, such textual DSLs are implemented using textual modeling

```
public abstract class BusinessObject {
    private String id;

    public String getID(){return id;}
    public void setID(String id){this.setID(id);}

    public abstract String asString();
}

public abstract class Person extends BusinessObject {
    private String firstName;
    private String lastName;
    private int age;

    public String getFirstName() {return firstName;}
    public void setFirstName(String firstName)
            {this.firstName = firstName;}

    public String getLastName() {return lastName;}
    public void setLastName(String lastName)
            {this.lastName = lastName;}

    public String getName() {return lastName + ", " + firstName;}

    public int getAge() {return age;}
    public void setAge(int age) {this.age = age;}
}
```

Listing 2.1: Basic business objects in the business application framework.

```
roles{
    object Employee:
         employed : Date
         workload : Hours
    object Customer:
         discount : Percentage
    object Shareholder:
         shares : Percentage
    object EmployeeCustomer is_a Employee,Customer:
         discount : Percentage[20]
}
```

Listing 2.2: Excerpt of a business domain model specified using a textual DSL.

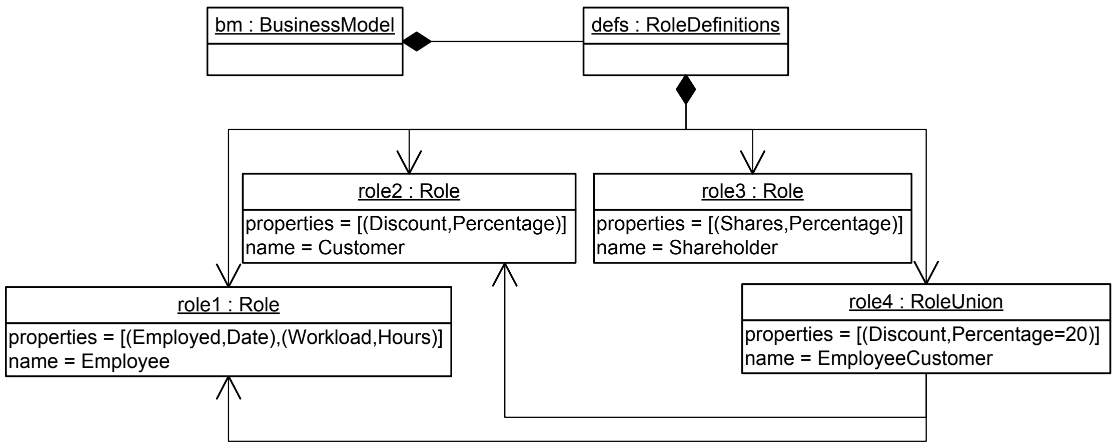


Figure 2.2.: Object-graph representation of the textual specification in Listing 2.2.

frameworks like EMFText [Heidenreich et al. 2009a], which is used in the BAF. The object graph is the input of the code generator. Since the generator relies on ISC, it maintains a set of Java fragments as basic building blocks for the framework-specific code that shall be emitted. Figure 2.3 gives an overview of the composition steps and fragments involved. Initially, the object graph is evaluated by the generator and, for each role definition in the specification, a set of template parameters is derived. The parameter values are bound to slots in the fragments.

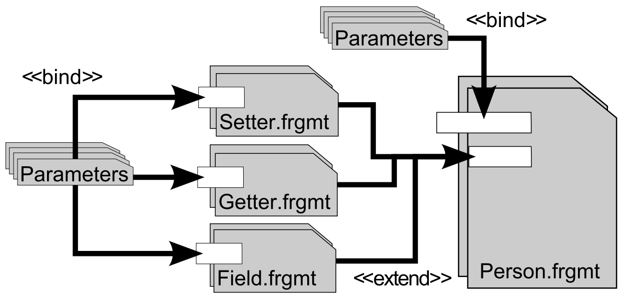


Figure 2.3.: The basic fragment-composition process as a data-flow graph.

In the following, a detailed example is discussed, which shows how and with what values the `Person` fragment is parametrized to generate a basic class body for `Employee` objects. Afterwards, the instantiation of the `Setter`, `Getter` and `Field` fragments and how they extend the initial class body is explained. Listing 2.3 shows the template fragment which is used to generate Java implementations from the role definitions in the business-model specification. It eventually inherits from `Person` declared in Listing 2.1 and has a basic implementation for `asString`. For parametrization, the template provides three slots marked up using `[[...]]`

```
1 //fragment person.frgmt
2 public class [[Type]] extends Person {
3     public String asString(){
4         String v = [[TypeName]];
5         v+= "[[Pfx]] id:" + getID();
6         v+= "[[Pfx]] name:" + getName();
7         return v;
8     }
9 }
```

Listing 2.3: The basic template for person business objects.

```
//fragment person_employee.frgmt
public class Employee extends Person {
    public String asString(){
        String v = "Employee";
        v+= "\n id:" + getID();
        v+= "\n name:" + getName();
        return v;
    }
}
```

Listing 2.4: Instance of person.frgmt for the employee role definition.

parentheses. The parenthesis separate slots from the rest of the language and are also called *syntactic hedges* in literature [Vinju 2005; Arnoldus 2011]. The slots occurring in Listing 2.3 are explained below.

- `Type` is the placeholder of a type signature (i.e., the role name).
- `TypeName` is a placeholder of a type name as a String literal or expression.
- `Pfx` is a placeholder of a default prefix in each line which is produced in the `asString` method.

In a first intermediate step, the code generator uses the template in Listing 2.3 to internally create an initial class body for each role in the business model. For this, the template is parsed and represented as a syntax tree. Consequently, it has to be *syntactically well-formed*, which is a clear advantage over standard string-based template engines (e.g., StringTemplate [Parr 2006]). For example, if a defect like missing double quotes in Line 6 is assumed (e.g., `v+= [[Pfx]] id:" + getID();`), the ISC-based template engine would recognize this as defect code and issue a message to the user, whereas a string-based engine would continue processing, finally emitting defect code.

Listing 2.4 shows the initialized class body generated for the `Employee` role. `Type` has been replaced by a qualified identifier using the invasive *bind* composition operator. Observe that in order to replace `Type` correctly, *bind* expects a qualified identifier subtree. For the `TypeName` slot, an expression node is expected which is bound to a string expression node (`"Employee"`) by the code generator. Finally, occurrences of `Pfx` were replaced by the newline character so that `asString`'s output is nicely formatted.

After the initial class bodies were created, they have to be *extended* with fields and access methods that represent the properties of the modeled business objects in appropriate ways. Listing 2.5 provided the code of the most basic member templates. As the template before, it provides some slots for parametrization:

- `Type` is again used for type signatures,
- `Field` marks placeholders for attribute identifiers,

```
//fragment getter.frgmt
public [[Type]] [[GetSfx]]() {
    return [[Field]];
}

//fragment setter.frgmt
public void [[SetSfx]]([[Type]] [[Field]]){
    this.[[Field]] = [[Field]];
}

//fragment field.frgmt
private [[Type]] [[Field]];
```

Listing 2.5: Basic class member templates for role fields.

```
//fragment getter_workload.frgmt
public int getWorkload() {
    return workload;
}

//fragment setter_workload.frgmt
public void setWorkload(int workload){
    this.workload = workload;
}

//fragment field_workload.frgmt
private int workload;
```

Listing 2.6: Instances of class member templates.

- `GetSfx` and `SetSfx` are slots for different accessor methods (i.e., get and set methods) of field declarations.

Listing 2.6 shows an instantiation of the member templates for the `workload` property of `Employee`. All occurrences of `Type` have been bound to the primitive Java type `int`, which is used by the BAF to represent properties declared as `Hours` in the model. This can be done in similar ways for other kinds of properties. Furthermore, `Field` slots are simply bound to the property name in all places and the names for get and set methods are generated accordingly.

Since the instantiated member fragments are still standalone, they have to be added to the according class fragment. In the above mentioned case, this would be the just generated `Employee` class. To achieve this, the extension mechanism of ISC can be used, which allows users to safely *extend* fragments at certain *hooks*. Hooks are derived from the structure of a fragment and need not to be declared explicitly using hedge symbols. The class-`members` hook is a typical hook of Java compilation units (cf. [Aßmann 2003, p. 127]). In this example, the code generator uses the *extend* composition operator to add the fragments of Listing 2.6 to the basic `Employee` class.[2] Note that this way of adding code fragments is conceptually very similar to inter-type declarations in AOP. For interested readers, the `Employee` class emitted by the code generator can be inspected in Listing A.1 of Appendix A.1.

While the code for the `Customer` and `Shareholder` roles is generated in the same way as for `Employee`, several options for composing the `EmployeeCustomer` class become emergent since `EmployeeCustomer` requires some form of multiple inheritance which is not supported by Java directly. A solution for this is the application of *mixin composition* [Bracha and Cook 1990], since mixins can be realized using ISC. [Bracha and Cook 1990] consider mixins as abstract subclasses and use mixin composition as a mechanism to model arbitrary kinds of inheritance. Today, the term mixin is generally used to describe class-merge operations, which do not rely on the standard inheritance mechanisms of the host language. For

[2]If a COMPOST like approach to ISC was assumed, the composition operator could be invoked by a method call, e.g., $extend(person\_employee, field\_workload)$.

instance, *inter-type declarations* used in AspectJ [Kiczales et al. 2001], *traits* like in the Scala programming language [Odersky et al. 2008] and mixin templates such as in C++ are *mixin-like* composition abstractions.

As a first step, a basic class body for the `EmployeeCustomer` role(s) is instantiated from the general class template. The following convention can be applied: instead of inheriting from `Person` it directly inherits from `Employee` as this is the first concept in the role composition declaration of `EmployeeCustomer`'s definition (cf. Listing 2.2, Line 9). To support this, the generator can use a variant of the template in Listing 2.3 that provides an implicit or declared super-class slot (e.g., `[[Super]]` instead of `Person` in Line 2). Afterwards, the generator reuses the member fragments for the `Customer discount` property which were originally instantiated from the member templates in Listing 2.5, and mixes them into the `EmployeeCustomer` class body via the *extend* composition operator. To achieve the default `discount` of 20 percent (cf. Listing 2.2, Line 10), a new constructor that sets the values accordingly can simply be added to the generated class. Alternatively, a variant of `field.frgmt` could be initialized to the default value by adding a slot to the fragment. Finally, to make the type hierarchy more consistent, the classes generated for each role definition could be divided into a Java interface class and an implementation class. Since Java supports multiple inheritance of interfaces, `EmployeeCustomer` would have an interface inheriting from both—the `Employee` and the `Customer` interfaces. Interested readers may look up potentially emitted code for `EmployeeCustomer` in Listing A.2 of Appendix A.1.

Several limitations of the ISC approach that have not been tackled appropriately yet will be discussed and analyzed in the next section.

2.2. Problem Analysis and Thesis Objectives

By definition, the compositional approach of ISC strictly adheres to the CFG of the component language and, thus, ensures syntactic correctness of the output. Further static checks such as name resolution, type checks, control-flow analysis etc. are outsourced to the compilation (interpretation) phase by the component-language compiler (interpreter) [Aßmann 2003, p. 152]. However, one can easily imagine multiple scenarios in which such a delayed check is not a sufficient or adequate solution for checking fragment compatibility. Considering use cases with hundreds or more fragments that are composed via a complex composition program, even a single error—e.g., caused by a contradiction with the type system—makes it difficult to find the problem's origin. Furthermore, as integrated development environments (IDEs) such as the Java Development Tools (JDT) of Eclipse [Eclipse Foundation 2013c] are getting better and better in providing comfort for users and developers by improving live feedback and error reporting facilities, post-compilation approaches seem to be behind the times (of course, usage of IDEs also depends on personal preferences). Standard compilers are not designed to support the implementation of such responsive and interactive IDEs. Instead, a complex IDE normally has its own compiler frontend (i.e., mostly the parsing and static analysis part of a compiler), with more adequate incremental analysis algorithms and an API that makes this information available to the actual editor and other clients. Thanks to the API provided by the JDT, it was possible for the developers of the AspectJ AOP tool [Kiczales et al. 2001] to implement the AJDT [Eclipse Foundation 2013a] as an extension of the JDT and reuse its syntactic and semantic analysis algorithms. The development of tools such as the AspectJ compiler and especially the AJDT is a complex undertaking and requires lots of software engineering and compiler-construction knowledge. Unfortunately, to the best of this author's knowledge, no generic approach on how to integrate static analysis algorithms with fragment composition systems has been developed yet. Consequently, one of the central research questions of this thesis is:

How to integrate static analysis algorithms with fragment composition systems to validate compositions in advance? [Q1]

As an example on what can be checked with such a semantics-aware composition system consider again the examples in Section 2.1, especially the `TypeName` slot in Listing 2.3. The bind composition operator was used to replace the slot with a simple string providing the type name of the composed class. Although this replacement looks straight-forward at a first glance, this composition step is only valid, because it fulfills two necessary conditions. First, the bound fragment has the right "syntactic type" w.r.t. the language grammar, namely `Expression`. If the fragment type would be different, e.g., `Statement`, the invasive composition system would refuse this composition step. Second, the bound expression also has the right type w.r.t. the type system of Java, since it always yields values of type `String`, which is the expected type of the assignment. If the bound expression would yield a different type like `int` (e.g., $3 + 4$), the composition system would execute this composition step, but the type checker in the compiler would report a static typing error. In contrast, a type-system-aware composition system can perform such checks during composition and provide cause-related error messages.

Another general problem in fragment composition systems is the interactions and interference between single composition steps. In AOP, incarnations of this problem are called *aspect interactions* [Douence et al. 2002; Durr et al. 2007] in general or *weaving interactions* [Kniesel and Bardey 2006]. In ISC, similar interactions between composition steps occur. For example, consider a logging statement that should be inserted via the extend composition operator in all methods with write access in the `Employee` class developed in Section 2.1. Depending on the order of execution and the point of time the logging extension is applied, some slots or hooks of the accessor methods inserted later may be missed accidentally by the composition, which is called an interference. This is due to the fact that compositional points change their *contexts* when the fragment is composed with another fragment. Since composition interactions occur in almost every composition scenario, simple solutions such as automatically avoiding or even forbidding them seems not reasonable. Instead, users should be supported with possibilities to influence the composition process so that only user-intended interactions occur. Hence, a second important research question of this thesis is as follows:

How to make fragment composition transparent in such a way that composition interactions can be discovered, understood and managed? [Q2]

A third important problem of ISC lies in the effort of implementing language-specific composition systems. While string-based template approaches are applicable to generate code for arbitrary fragment languages, ISC-based composition systems have to be implemented for each of these languages. Hence, given a number of composition languages x (e.g., slots, macros or aspects) and a number of object languages y, $x * y$ composition systems have to be developed to support all x composition languages and all y object languages in comparison to x string-based composition systems. Moreover, during the past decade it turned out that also the effort of realizing practical composition systems is generally high. Implementing such a system involves knowledge from language engineering and language composition because the component language is enriched with compositional constructs. For many real-world programming languages this results in conflicts with their specification which must be treated by the composition-system developer. However, having multiple languages supported is essential for any application that intends to use ISC to generate code for more than one target platform or architecture. For instance, the BAF code generator may have alternative fragment components to generate code for C# and C++. Hence, it would require two additional composition systems to support these languages in a code-generation backend whereas it is not sure if it is even possible to realize an ISC system that fully supports the additional languages (e.g., if the language specification is not covered by a corresponding CFG). Hence, the third important research question of this thesis is as follows:

How to reduce the effort of implementing heterogeneous fragment composition systems and how to make the approach more scalable with respect to complex languages? [Q3]

Three of the thesis' objectives immediately derive from the three research questions above by reformulating them:

[O1] Integrate static semantics analysis algorithms with fragment composition systems for validation (*[Q1]*).

[O2] Develop an approach that makes fragment composition more transparent and helps users to better understand and manage their compositions (*[Q2]*).

[O3] Lower the effort of implementing fragment composition systems for multiple component languages or very complex languages (*[Q3]*).

The three objectives above are considered as the thesis' main objectives. Besides that, it has two secondary objectives that support and validate the research on the main objectives.

[O4] Provide implementations and applications demonstrating that the approaches suggested by this work are practically feasible.

[O5] Evaluate existing approaches to fragment composition and compare them with the approaches of this thesis to demarcate the progress achieved by this work from the previous state of the art.

Although called secondary, the two additional objectives are not less important. Since this is a thesis with a focus on software engineering, an implementation proves that the concept is valid and can directly be used by people different from the author. Moreover, an in-depth analysis of previous work validates previous research results by confirming or refuting them and helps to understand differences and improvements.

Addressing the objectives above, this thesis makes several major research contributions. The next section discusses those contributions and explains how they relate to the objectives.

2.3. Contributions in Detail

This thesis has four main contribution blocks (or "work packages"), each diverging into multiple compartments. While some of them have already been mentioned in Chapter 1, the enumeration below provides a completed overview and explains how they relate to the research questions imposed above.

[C1] **Well-formed invasive software composition.** This contribution includes several extensions to the classic model of ISC and has three major compartments.

a) *Fragment component models based on reference attribute grammars* are the main entrance point for well-formed ISC and a "door opener" to support context-sensitive properties (*[O1]*) and extensible component models (*[O3]*).

b) *Advanced composition technique based on strategies and rewrites.* The thesis suggests several strategies of composition-program interpretation. These strategies use the component model's information on composition operators, compositional points and attribute dependencies to perform the actual composition. This makes the composition process more transparent and configurable (*[O2]*).

c) *Fragment Contracts* are a conceptual framework that integrates context-sensitive constraints with the composition and the fragment language. It gives access to the component language's type system and other semantic properties, and checks if specific compositions obey these rules (*[O1]*).

[C2] **Scalable invasive software composition.** This contribution breaks the strict separation between classic ISC, well-formed ISC and string-based composition approaches. Scalable ISC has the following compartments:

a) *Minimal invasive software composition* lifts string-based composition to ISC and introduces the notion of a *minimal fragment component model*, which is applicable to compose fragments of any language at the downside of losing any syntactic or semantic validation (*[O3]*).

b) *Island fragment component models* allow to specify component models with *partial* syntactic validation of component and composition-language constructs (*[O3]*). In contrast to a sublanguage-based component model, this approach can process the *whole* language.

c) *Agile composition-system development* leverages the extensible specification approach of *[C1]* combining minimal, well-formed and classic ISC. It allows developers to adjust language awareness as needed and to develop ISC systems flexibly in small iterations (*[O3]*).

[C3] The Skeletons and Application Templates (SkAT) framework. This is the reference implementation of the developed ISC approaches. Based on the JastAdd RAG system [Ekman and Hedin 2007a], SkAT has the following compartments and byproducts (all *[O4]*):

a) *A generic implementation of Contributions [C1a]–[C1c]* that can be instantiated for arbitrary fragment languages.

b) *A well-formed composition system supporting Java* as a real-world component language. It provides an API for writing composition programs and fragment contracts to perform semantic checks.

c) *A library of parallel algorithmic skeletons* [Cole 1989] realized as Java fragments based on the Java composition system. Currently, three skeletons are supported.

d) *A generic implementation of Contributions [C2a] and [C2b]* that supports island fragment component models and scalable ISC.

e) *Examples of three extensible island composition systems*, including a minimal slot language, a language for variant markup in arbitrary component languages and an extensible macro language mimicking the well-known preprocessor of the C programming language.

[C4] A consolidating review of the state of the art in invasive software composition. The review has the following compartments:

a) *A formalized tree-based model of classic ISC* is given which precisely defines the ingredients of ISC-based fragment composition systems. The concepts of this model are used in the comparison of the respective systems (*[O5]*).

b) *The BAF example as a cross-cutting case study* is implemented in four different ISC frameworks (including SkAT) to explain and compare their specification languages w.r.t. expressiveness and support of the defined ISC features (*[O4,O5]*).

c) *A detailed analysis of existing ISC frameworks* and a comparison with the SkAT tool, which has been developed in context of this thesis (*[O5]*).

Having described the thesis' main contributions, the next chapter discusses background and terminology.

3
Background and Notation

In this chapter, the conceptual framework and terminology of this thesis are defined. Section 3.1 gives an overview of the very basics of formal languages like grammars and trees. The purpose of the section is to provide an entry point to the very broad field of formal language theory and familiarize with the basic notions and notations used in the thesis. Section 3.2 contains an abstract discussion on different parsing techniques used in practice. A basic understanding on how practical parsers and parser generators work is essential to successfully design and implement fragment composition systems for textual languages. Moreover, the section discusses how fragment artifacts like programs, documents and models are typically represented as data structures in programs and relates this to the theoretical concepts introduced in Section 3.1. Finally, in Section 3.3, a definition of attribute grammars (AGs) is given and reference attribute grammars (RAGs) are discussed, as well as the state of the art of RAG systems. Moreover, a notational framework for RAGs is introduced which is used in later chapters to describe RAG-based ISC.

3.1. Basic Definitions

This section introduces concepts that are frequently used in this thesis and that are well-known in literature on language theory and compiler construction [Leeuwen 1994; Berstel and Boasson 1990; Schöning 2001; Aho et al. 1986; Goos 1997]. Readers who are familiar with context-free grammars, syntax trees and parsing may skip this section and continue with Section 3.2 or Section 3.3, using this section for looking up specific notations.

3.1.1. Formal Languages and Context-Free Grammars

In computer science, formal languages and grammars are used as a common concept for specifying the *syntax* of programming languages or DSLs. A formal language $\mathcal{L}$ is a set of words over a finite set of symbols—the alphabet Σ. Symbols are atomic units, e.g., letters, numbers, objects or even events. Grammars are rule bases for such languages. A formal grammar consists of terminals (i.e., elements in the alphabet), nonterminals and productions (the grammar rules). In general, a production specifies a replacement step in a derivation chain and consists of an arbitrary sequence (length > 1) of terminals and nonterminals on the left-hand side, and an arbitrary sequence of terminals and nonterminals on the right-hand side. A special subclass are CFGs with a single nonterminal on the left-hand side of a production. Formally, they are defined as follows:

Definition 3.1 (context-free grammar):
A context-free grammar (CFG) is a 4-tupel (N,Σ,P,S) with

- N a finite set of nonterminals,
- Σ a finite set of terminals and $\Sigma \cap N = \emptyset$,
- P a finite set of productions and $P \subseteq N \times (\Sigma \cup N)^*$ and $\forall n \in N \, \exists \gamma \in (\Sigma \cup N)^* :$ $n \to \gamma \in P$,
- S a start symbol $S \in N$.

The derivation relation $\Rightarrow_G \subseteq (\Sigma \cup N)^* \times (\Sigma \cup N)^*$ of G is $\Rightarrow_G = \{(\alpha n \beta, \alpha \gamma \beta) \mid \alpha, \beta \in (\Sigma \cup N)^* \wedge n \to \gamma \in P\}$. The reflexive transitive closure of $\Rightarrow_G$ is denoted by $\Rightarrow_G^*$. If $S \Rightarrow_G^* \alpha$, then α is called a *sentential form* of G. A derivation step $\alpha n \beta \Rightarrow_G \alpha \gamma \beta$ is called left-most (right-most) derivation step if $\alpha \in \Sigma^*$ ($\beta \in \Sigma^*$). A derivation $d : \alpha \Rightarrow_G \ldots \Rightarrow_G \gamma$ is called left-most (right-most) derivation if all derivation steps in d are left-most (right-most) derivation steps. G is called unambiguous, if for any two left-most (or right-most) derivations $d_1 : S \Rightarrow_G \ldots \Rightarrow_G \alpha$ and $d_2 : S \Rightarrow_G \ldots \Rightarrow_G \alpha$ it holds that $d_1 = d_2$. Otherwise, G is called ambiguous. ◇

A CFG specifies a context-free language that is generated by $\Rightarrow_G$:

Definition 3.2 (context-free language):
Let $G = (N, \Sigma, P, S)$ be a context-free grammar and $n \in N$. Then $L(G) = \{\lambda \in \Sigma^* \mid S \Rightarrow_G^* \lambda\}$ is the context-free language defined by G. Further, $L_{|n}(G) = \{\lambda \in \Sigma^* \mid n \Rightarrow_G^* \lambda\}$ is called the n-language projection of $L(G)$ and $G_{|n} = (N_{|n} \subseteq N, \Sigma_{|n} \subseteq \Sigma, P_{|n} \subseteq P, n)$ with $L(G_{|n}) = L_{|n}(G)$ is the n-projection of G. Note that $L_{|S}(G) = L(G_{|S}) = L(G)$. ◇

Sometimes, it is useful to require some kind of normal form for a given CFG. *Reduced CFGs* have a separated start symbol, each nonterminal is reachable from the start symbol and the language projection of each nonterminal is nonempty:

Definition 3.3 (reduced context-free grammar (cf. [Berstel and Boasson 1990])):
A CFG $G = (N, \Sigma, P, S)$ is called reduced if

- $\forall \alpha, \beta \in (\Sigma \cup N)^* \, : \, n \to \alpha S \beta \notin P$,
- $\forall n \in N \, \exists \alpha, \beta \in (\Sigma \cup N)^* \, : \, S \Rightarrow_G^* \alpha n \beta$,
- $\forall n \in N \, : \, L_{|n}(G) \neq \emptyset$. $\diamond$

The following example contains a simple (reduced) CFG that specifies a simple Boolean expression language:

Example 3.1 (context-free grammar/language).
Let $G = (\{E, O, A, T\}, \{\&, |, f, t\}, P, E)$ be a CFG, where

$$\begin{aligned} P = \{ & E \to O, && (p_0)\\ & O \to O \,|\, O, && (p_1)\\ & O \to A, && (p_2)\\ & A \to A \,\&\, A, && (p_3)\\ & A \to T, && (p_4)\\ & T \to t, && (p_5)\\ & T \to f\}. && (p_6) \end{aligned}$$

The word $t \,|\, f \,\&\, t$ can be generated by $\Rightarrow_G$ as follows: $E \overset{p_0}{\Rightarrow}_G O \overset{p_1}{\Rightarrow}_G O \,|\, O \overset{p_2}{\Rightarrow}_G A \,|\, O \overset{p_4}{\Rightarrow}_G T \,|\, O \overset{p_5}{\Rightarrow}_G t \,|\, O \overset{p_3}{\Rightarrow}_G t \,|\, A \,\&\, A \overset{p_4}{\Rightarrow}_G t \,|\, T \,\&\, A \overset{p_6}{\Rightarrow}_G t \,|\, f \,\&\, A \overset{p_4}{\Rightarrow}_G t \,|\, f \,\&\, T \overset{p_5}{\Rightarrow}_G t \,|\, f \,\&\, t$. Hence, $E \Rightarrow_G^* t \,|\, f \,\&\, t$. The language generated by G is $L(G) = \{t,\, f,\, t \,|\, f,\, f \,|\, t,\, t \,|\, t,\, f \,|\, f,\, t \,\&\, f,\, f \,\&\, t,\, t \,\&\, t,\, f \,\&\, f,\, t \,|\, f \,\&\, t,\, \ldots\}$. The A-language projection is $L_{|A}(G) = \{t,\, f,\, t \,\&\, f,\, f \,\&\, t,\, t \,\&\, t,\, f \,\&\, f,\, t \,\&\, f \,\&\, f,\, \ldots\}$. $\diamond$

3.1.2. Extended Backus Naur Form

The standard notation for CFGs turned out as impractical for the specification of huge grammars such as for programming languages. Hence, in many cases it is practical to use comprehensible notations like the Extended Backus Naur Form (EBNF). In this thesis, the following EBNF notation is used:

Definition 3.4 (EBNF grammar notation):
Let $G = (N, T, P, S)$ be an EBNF grammar with N, T, P the finite sets of nonterminals, terminals, productions and S the start symbol. Further, let $\mathcal{L}_{ident} = \{\mathrm{a} \ldots \mathrm{z}, \mathrm{A} \ldots \mathrm{Z}, \_, 0 \ldots 9\}^+$ the language of acceptable EBNF identifiers and $\mathcal{L}_{text} = \Sigma^+_{\text{ASCII}}$ the language of arbitrary strings over the ASCII alphabet. Then

- $N \subset \mathcal{L}_{ident}$,
- $T \subset \mathcal{L}_{keywords} \cup \mathcal{L}_{tokens}$ with

  - $\mathcal{L}_{keywords} = \{\texttt{"}\lambda\texttt{"} \mid \lambda \in \mathcal{L}_{text}\}$ the set of keywords (name and value of a keyword are always λ),
  - $\mathcal{L}_{tokens} = \{\texttt{<}\lambda\texttt{>} \mid \lambda \in \mathcal{L}_{ident}\}$ the set of token symbols (the name of a token is λ, its value is usually different from λ),
- $P \subset N \times \mathcal{L}_{exp}$,
- $\mathcal{L}_{exp}$ is a *regular-expression* language over $T \cup N$. $\mathcal{L}_{exp}$ is recursively defined as follows:
  - $\epsilon \in \mathcal{L}_{exp}$ (empty sequence),
  - if $\alpha \in T \cup N$, then also $\alpha \in \mathcal{L}_{exp}$ (atom),
  - if $\alpha \in \mathcal{L}_{exp}$, then also $\texttt{(}\alpha\texttt{)} \in \mathcal{L}_{exp}$ (grouping),
  - if $\alpha \in T \cup N$ or $\alpha = \texttt{(}\beta\texttt{)}$ where $\beta \in \mathcal{L}_{exp}$, then also $\alpha\texttt{?} \in \mathcal{L}_{exp}$ (optional),
  - if $\alpha \in T \cup N$ or $\alpha = \texttt{(}\beta\texttt{)}$ where $\beta \in \mathcal{L}_{exp}$, then also $\alpha\texttt{*} \in \mathcal{L}_{exp}$ (repeat ≥ 0 times),
  - if $\alpha \in T \cup N$ or $\alpha = \texttt{(}\beta\texttt{)}$ where $\beta \in \mathcal{L}_{exp}$, then also $\alpha\texttt{+} \in \mathcal{L}_{exp}$ (repeat ≥ 1 times),
  - if $\alpha, \beta \in \mathcal{L}_{exp}$, then also $\alpha\textvisiblespace\beta \in \mathcal{L}_{exp}$ (sequence),
  - if $\alpha, \beta \in \mathcal{L}_{exp}$, then also $\alpha \mid \beta \in \mathcal{L}_{exp}$ (alternative).

Observe that `::=` is used instead of $\rightarrow$ in productions. An EBNF grammar is called *flat* if no grouping expressions occur in any production (i.e., $\mathcal{L}_{exp}$ would be defined without the grouping recursion in this case).

Tokens are special types of terminal symbols. While keywords symbolize themselves, tokens are symbolic abstractions for the most basic signatures of a language like variable names and type signatures. As such, they are typically specified themselves as languages over a some alphabet using regular expressions.

EBNF grammars are typically specified as lists of productions. By default, the nonterminal on the left-hand side of the first production is considered as the start symbol. The sets of terminals and nonterminals can be derived from the symbols used in the productions at hand. $\diamond$

The example below contains an EBNF grammar for simple Boolean expressions which generates the same language as the plain CFG in Example 3.1:

Example 3.2 (EBNF grammar).
Let G_E be the EBNF grammar with the following productions:

```
Expression ::= Or
Or ::= Or "|" Or | And
And ::= And "&" And | Term
Term ::= "t" | "f"
```

Since the alternative operator | is a shorthand for multiple productions with the same nonterminal on the left-hand side, a mapping of G_E to G in Example 3.1 is straight-forward and can be achieved by eliminating $|$, which results in G'_E, and by mapping terminals and nonterminals in G'_E to their correspondences in G. ◇

The right-hand sides of EBNF productions are essentially *regular expressions* [Schöning 2001] over terminal and nonterminal symbols that specify a *regular language*. Since a regular expression can be mapped to an equivalent CFG, any EBNF grammar can be transformed into an equivalent CFG if each right-hand side nonterminal has a corresponding EBNF production. Since infinitely many mappings are possible, the subsequently described informal transformation rules are applied "implicitly" when definitions or descriptions that may hold for both—plain CFGs and EBNF grammars—are discussed in this thesis.

Let $G_E = (N_E, T_E, P_E, S_E)$ be an EBNF grammar and $G = (N, \Sigma, P, S)$ the CFG derived from G_E. If P_E only contains productions $\texttt{n ::=}\,\gamma$ where γ is ϵ or is a sequence over $T_E \cup N_E$, G immediately derives with $N = N_E$, $\Sigma = \{\lambda \,|\, \texttt{"}\lambda\texttt{"} \in T \vee \texttt{<}\lambda\texttt{>} \in T\}$, $P = \{n \rightarrow \delta \,|\, \texttt{n ::=}\,\gamma \in P_E$ and δ is isomorphic to γ by the mappings of N to N_E and T to $\Sigma\}$. Otherwise, if G_E contains alternatives, groupings, optionals or repetitions, these have to be desugared:

- If a grouping $grp = \texttt{(}\alpha\texttt{)}$ occurs on a right-hand side of a production, a fresh nonterminal n_{fresh} and a production $n_{fresh}\ \texttt{::=}\,\alpha$ are added to G_E while grp is replaced with n_{fresh}.

Example 3.3 (desugaring groupings).
```
A ::= ("b"|<c>)* "a"?   => A ::= BOrC* "a"?
                           BOrC ::= "b"|<c>
```
◇

- If no groupings are contained in productions of G_E, repetitions can be eliminated by transforming them into directly recursive nonterminals. If $\alpha\texttt{*}$ or $\alpha\texttt{+}$ occur on a right-hand side of any production, a fresh nonterminal $\alpha\texttt{List}$ is added to N_E and the productions $\alpha\texttt{List ::=}\,\alpha\,\alpha\texttt{List}$ (recursive rule) and $\alpha\texttt{List ::=}\,\alpha$ (terminating rule) are added to P_E. All occurrences of $\alpha\texttt{+}$ ($\alpha\texttt{*}$) are replaced by $\alpha\texttt{List}$ ($\alpha\texttt{List?}$).

Example 3.4 (desugaring repetitions).
```
A ::= BOrC* "a"?       => A ::= BOrCList "a"?
BOrC ::= "b"|<c>          BOrC ::= "b"|<c>
                          BOrCList ::= BOrC
                          BOrCList ::= BOrC BOrCList
```
◇

- If no groupings and repetitions are contained in productions of G_E, optionals can be eliminated by adding a fresh nonterminal and production. If $\alpha\texttt{?}$ occurs on a right-hand side of any production, a fresh nonterminal $\alpha\texttt{Opt}$ is added to N_E and two productions

α`Opt ::=`ϵ and α`Opt ::=`α are appended to P_E. All occurrences of α`?` in all productions p are replaced with α`Opt`.

Example 3.5 (desugaring optionals).
```
A ::= BOrCList "a"?              A ::= BOrCList aOpt
BOrC ::= "b"|<c>                 BOrC ::= "b"|<c>
BOrCList ::= BOrC           =>   BOrCList ::= BOrC
BOrCList ::= BOrC BOrCList       BOrCList ::= BOrC BOrCList
                                 aOpt ::= "a"
                                 aOpt ::= ϵ
```
◇

- If no groupings, repetitions and optionals are contained in the productions of G_E, alternatives can be eliminated by removing each production containing $k > 1$ alternatives n `::=` γ_1 `|` ... `|` γ_k and adding a fresh production n `::=` γ_i for each alternative.

Example 3.6 (desugaring alternatives).
```
A ::= BOrCList aOpt              A ::= BOrCList aOpt
BOrC ::= "b"|<c>                 BOrCList ::= BOrC
BOrCList ::= BOrC           =>   BOrCList ::= BOrC BOrCList
BOrCList ::= BOrC BOrCList       aOpt ::= "a"
aOpt ::= "a"                     aOpt ::= ϵ
aOpt ::= ϵ                       BOrC ::= "b"
                                 BOrC ::= <c>
```
◇

As a convention, it is further assumed that the transformation described above retains the order of the productions so that they can be addressed uniquely using subscripts, e.g., $\texttt{BOrCList}_1$, $\texttt{BOrCList}_2$, $\texttt{aOpt}_1$, $\texttt{aOpt}_2$, $\texttt{BOrC}_1$ and $\texttt{BOrC}_2$ in Example 3.6.

3.1.3. Graphs

Graphs are one of the most important data structures in computer science which occur in mostly every program. Data structures linked with pointers, software models, object nets, database schemes and others are essentially graphs. Basically, a graph is a collection of objects (called *nodes* or *vertices*) which are connected via linking objects (called *edges*). Typically, nodes and edges are associated with names (called *labels*). The following definition gives a general notion of labeled graphs and directed labeled graphs:

Definition 3.5 ((directed) labeled graph):
A labeled graph H is a 4-tuple $(V, E, \mathrm{Lab}, \mathcal{L}_\Sigma)$ with

- V a finite set of vertices (nodes) with $|V| \geq 0$,

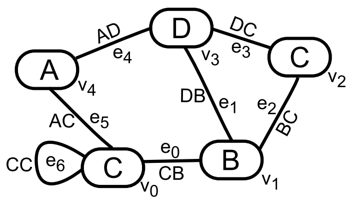


Figure 3.1.: A labeled graph.

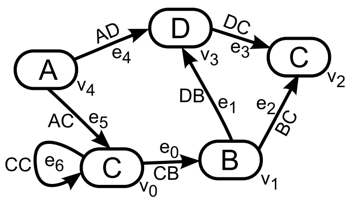


Figure 3.2.: A directed labeled graph.

- E the set of edges, where
  - E may be any set of 2-elementary sets on $V : E \subseteq \{\{u, v\} \mid u, v \in V\}$ (then H is called an *undirected* labeled graph or just "graph"),
  - E may be a relation $E \subseteq V \times V$ (then H is called a *directed* labeled graph),
- $\mathcal{L}_\Sigma$ the set of labels (or label alphabet),
- $\mathrm{Lab} : V \cup E \to \mathcal{L}_\Sigma$ the labeling function, which assigns a label to each node in V and edge in E.

If $v \in V$ can be reached from $u \in V$ by traversing at least one edge of H, then the corresponding sequence of 2-tuples $seq = (v_1, e_1), \ldots, (v_i, e_i), \ldots, (v_n, e_n)$ with $v_1 = u$ and $e_n = \{v_n, v\}$ (or (v_n, v) for directed graphs) is called a *path* between u and v, where $1 \leq i \leq n$, $n \in \mathbb{N}$ is the number of path segments, $v_i \in V$, $e_i \in E$, and for any two $(v_k, e_k), (v_l, e_l)$ in *seq*: $e_k \neq e_l$ (i.e., an edge is only traversed once). For directed graphs, a path between u and v exists if (u, v) is in the transitive closure E^+ of E. H is called *connected*, if there exists a path between all two different nodes $u, v \in V$. H is called *cyclic*, if for any node $v \in V$ there exists a path from v to itself. Otherwise, H is called *acyclic*. If H is a directed graph, then H is cyclic if $(v, v) \in E^+$. If H is a directed graph, then $\mathrm{Out}_H : V \to 2^E$ and $\mathrm{In}_H : V \to 2^E$ collect a node's outgoing and incoming edges: $\forall u, v \in V : (u, v) \in \mathrm{Out}_H(u) \wedge (u, v) \in \mathrm{In}_H(v) \iff (u, v) \in E$. Moreover, for any two nodes $u, v \in V$ with $(u, v) \in \mathrm{Out}_H(u)$, v is called a *successor* of u while u is a called *predecessor* of v. $\diamond$

Graphs are typically represented by graphical figures using lines, arrows and ovals. Example 3.7 discusses two exemplary directed and undirected graphs as well as their corresponding representations as figures.

Example 3.7 ((directed) labeled graph).
Let $H = (V, E, \mathrm{Lab}, \mathcal{L}_\Sigma)$ be the graph with the following components:

- $V = \{v_0, \ldots, v_4\}$,
- $E = \{e_0 = \{v_0, v_1\}, e_1 = \{v_1, v_3\}, e_2 = \{v_1, v_2\}, e_3 = \{v_3, v_2\}, e_4 = \{v_4, v_3\},$

$e_5 = \{v_4, v_0\}, e_6 = \{v_0, v_0\}\}$,

- $\mathcal{L}_\Sigma = \{A, B, C, D, AA, AB, AC, AD, BA, BB, BC, BD, CA, CB, CC, CD, DA, DB, DC, DD\}$,
- $\text{Lab} = \{(v_0, C), (v_1, B), (v_2, C), (v_3, D), (v_4, A), (e_0, CB), (e_1, DB), (e_2, BC), (e_3, DC), (e_4, AD), (e_5, AC), (e_6, CC)\}$.

Figure 3.1 contains a visual representation of H. Nodes are denoted as oval shapes and edges are represented by lines linked the corresponding edges. Figure 3.2 represents H as a directed graph H', where $E = \{e_0 = (v_0, v_1), e_1 = (v_1, v_3), e_2 = (v_1, v_2), e_3 = (v_3, v_2), e_4 = (v_4, v_3), e_5 = (v_4, v_0), e_6 = (v_0, v_0)\}$. The orientation of directed edges is represented by arrows instead of lines. H is a connected and cyclic graph. In contrast, H' is not connected (e.g., there is no path to v_4 from any other node in H') and, if e_6 was removed, H' would also be acyclic. $\diamond$

The graphs considered in this thesis are typically directed graphs, i.e., the set of edges E of a graph H is a relation. Moreover, Definition 3.5 allows exactly one edge from a node u to another node v of H so that the maximum number of edges is $|V|^2$ for undirected graphs and $2 * |V|^2 - |V|$ for directed graphs. Realistic graph-based data structures in programs usually do not obey such restrictions, since it is typical to have more than one edge (in one direction) between two nodes. To support practical use cases, the set of edges E in Definition 3.5 has to be defined as a *multiset*, which makes the defined graph a *multigraph*. A multiset M contains 2-tuples as entries with one element the unique value val and a natural number as a counter denoting the number of occurrences of val in M. For instance, if there were two directed edges between v_0 and v_1 in H' of Example 3.7, this would be denoted as $((v_0, v_1), 2)$. The labeling function Lab of a multigraph also has a slightly different signature considering the occurrence of an edge: $\text{Lab} : V \cup E \times \mathbb{N} \rightarrow \mathcal{L}_\Sigma$.

The following definition specifies the subgraph relation between two graphs:

Definition 3.6 (subgraph):
Let $H = (V, E, \text{Lab}, \mathcal{L}_\Sigma)$ and $H' = (V', E', \text{Lab}', \mathcal{L}'_\Sigma)$ be graphs. Then H' is called a subgraph of H ($H' \subseteq H$) and H a host graph of H', if $V' \subseteq V$, $E' \subseteq E$, $\text{Lab}' \subseteq \text{Lab} : V' \cup E' \rightarrow \mathcal{L}'_\Sigma$ and $\mathcal{L}'_\Sigma \subseteq \mathcal{L}_\Sigma$. $\diamond$

Hence, a subgraph contains some nodes and edges of its host graph, where the edges may only link nodes of the subgraph. Moreover, the labeling function and the label alphabet must be defined accordingly in such a way that labels are assigned to all nodes and edges of the subgraph.

3.1.4. Trees and Syntax Trees

Syntax trees are essential data structures for processing and visualizing information captured in context-free languages. Basically, a directed tree is a directed graph consisting of *nodes* that are connected via an injective *parent-child* relation such that each node has a certain number of *child* nodes. Conversely, each node—except the root node—has a unique *parent*. Nodes without children are called *leaf* nodes. Nodes and edges of a tree can be assigned with *labels*—strings over some label alphabet. In this work, the following definition for labeled trees is used:

Definition 3.7 (directed labeled tree):
Let $T = (V, E, \mathrm{Lab}, \mathcal{L}_\Sigma)$ be a directed, labeled graph. T is a directed labeled tree, if

- T is acyclic,
- E is left-unique (i.e., $\forall v \in V : |\operatorname{In}_T(v)| \leq 1$),
- all nodes must be reachable from a distinct root: $\exists r \in V\, \forall v \in V \setminus \{r\} : |\operatorname{In}_T(r)| = 0 \wedge (r, v) \in E^+$.

Let $(u, v) \in E$, then u is called the *parent* of v and v is called a *child of* u. The function $\mathrm{Chil}_T : V \rightarrow 2^V$ collects a node's children: $\forall u, v \in V : v \in \mathrm{Chil}_T(u) \iff (u, v) \in \mathrm{Out}_T(u)$. The function $\mathrm{Par}_T : V \rightsquigarrow V$ provides a node's parent and $\perp$ for the root of T.

If E is a partially ordered set w.r.t. $<_E = \bigcup\limits_{v \in V} <_v$, where $<_v$ is a total order on the outgoing edges $\mathrm{Out}_T(v)$ of v, then T is an *ordered* tree. A total order $<_V$ on the nodes of T then derives from $<_E$ with $<_V = E^+ \cup \{(u, v) \,|\, e_1 : (p, u), e_2 : (p, v) \in E \wedge e_1 <_E e_2 \wedge u \neq v\} \cup \{(u, v) \,|\, (p_u, u), (p_v, v) \in E^+ \wedge e_1 : (p, p_u), e_2 : (p, p_v) \in E \wedge e_1 <_E e_2 \wedge p_u \neq p_v\}$. ◇

Trees in this thesis are typically assumed to be ordered. If not, it will be stated explicitly.

Labels of edges do not have to be unique by Definition 3.5. However, considering the typical dot notation (`[node].[label]`) for child access via edge labels, these must be unique w.r.t. the context of a node and its children. Let $T = (V, E, \mathrm{Lab}, \mathcal{L}_\Sigma)$ be a labeled tree, and $\mathrm{Lab}_v \subseteq \mathrm{Out}_T(v) \times \mathcal{L}_\Sigma$ a subrelation of Lab which assigns labels to outgoing edges of v. If Lab_v is injective, an edge $e \in \mathrm{Out}_T(v)$ can be denoted correctly via its label yielding a partial function $\mathrm{dot} : V \times \mathcal{L}_\Sigma \rightsquigarrow V$ with $\mathrm{dot}(v, \mathit{lab}) = x \iff (v, x) \in \mathrm{Out}_T \wedge \mathrm{Lab}((v, x)) = \mathit{lab}$ and $\perp$ otherwise. In infix notation $v.\mathit{lab} = x$. If Lab_v is not injective, the notation cannot be used for path navigation in general. However, in the case of ordered trees, indexes can be used to distinguish multiple occurrences of a label in context of a node. Hence, a modified $\mathrm{dott} : V \times \mathcal{L}_\Sigma \times \mathbb{N} \rightsquigarrow V$ can be derived: $\mathrm{dott}(v, \mathit{lab}, n) = x \iff (v, x) \in \mathrm{Out}_T \wedge \mathrm{Lab}((v, x)) = \mathit{lab} \wedge \mathrm{idx}(v, x, \mathit{lab}) = n$, which is $v.\mathit{lab}_n = x$ in infix notation. Consider $\mathrm{Out}_T(v, \mathit{lab}) \subseteq \mathrm{Out}_T(v) = \{(v, y) \,|\, (v, y) \in \mathrm{Out}_T(v) \wedge \mathrm{Lab}_v((v, y)) = \mathit{lab}\}$ and $<'_v$ the total order w.r.t. $\mathrm{Out}_T(v, \mathit{lab})$ derived from $<_V$. Then $\mathrm{idx} : V \times V \times \mathcal{L}_\Sigma \rightarrow \mathbb{N}$ can be derived as follows: $\mathrm{idx}(v, y, \mathit{lab}) = |\{e \,|\, e \in \mathrm{Out}_T(v, \mathit{lab}) \wedge e <'_v (v, y)\}| + 1$ (i.e., the index is the number of "smaller" edges + 1).

Like graphs, it is easy to visualize trees as figures. In typical graphical notations, nodes and edges are represented as graphical objects like ellipses, lines or arrows between them. Labels can be represented by text which is rendered close to the objects. The following example shows a labeled tree as defined above. It also gives a graphical representation of the tree based on the visualization metaphors used in this thesis.

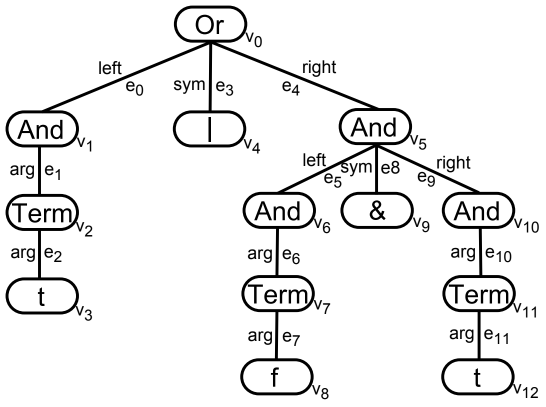


Figure 3.3.: A labeled tree.

Example 3.8 (labeled tree).
Let $T = (V, E, \mathrm{Lab}, \mathcal{L}_\Sigma)$ be a tree with the following components:

- $V = \{v_0, \ldots, v_{12}\}$,
- $E = \{e_0 = (v_0, v_1), e_1 = (v_1, v_2), e_2 = (v_2, v_3), e_3 = (v_0, v_4), e_4 = (v_0, v_5), e_5 = (v_5, v_6), e_6 = (v_6, v_7), e_7 = (v_7, v_8), e_8 = (v_5, v_9), e_9 = (v_5, v_{10}), e_{10} = (v_{10}, v_{11}), e_{11} = (v_{11}, v_{12})\}$,
- $\mathcal{L}_\Sigma = \{\mathit{Or}, \mathit{And}, \mathit{Term}, |, \&, t, f, \mathit{left}, \mathit{right}, \mathit{arg}, \mathit{sym}\}$,
- $\mathit{Lab} = \{(v_0, \mathit{Or}), (v_1, \mathit{And}), (v_2, \mathit{Term}), (v_3, t), (v_4, |), (v_5, \mathit{And}), (v_6, \mathit{And}), (v_7, \mathit{Term}), (v_8, f), (v_9, \&), (v_{10}, \mathit{And}), (v_{11}, \mathit{Term}), (v_{12}, t), (e_0, \mathit{left}), (e_1, \mathit{arg}), (e_2, \mathit{arg}), (e_3, \mathit{sym}), (e_4, \mathit{right}), (e_5, \mathit{left}), (e_6, \mathit{arg}), (e_7, \mathit{arg}), (e_8, \mathit{sym}), (e_9, \mathit{right}), (e_{10}, \mathit{arg}), (e_{11}, \mathit{arg})\}$,
- v_0 is the root of T.

T has five leaf nodes: $v_3, v_4, v_8, v_9, v_{12}$. The visual representation of T is shown in Figure 3.3. Although T is a directed graph, plain lines instead of arrows (cf. Example 3.7) were used to represent edges as their direction can be derived from the vertical and horizontal arrangement of the nodes in Figure 3.3. ◇

Observe that Figure 3.3 contains annotations for both—labels and object identities (i.e., v_i, e_i). If not stated otherwise, in the remainder of this work the object identities (i.e. v_i, e_j) can be omitted from graphical tree representations in favor of a better readability.

Based on the $\subseteq$ relation on sets and the subgraph relation, the subtree relation on trees emerges as follows:

Definition 3.8 (subtree):
Let $T = (V, E, \text{Lab}, \mathcal{L}_\Sigma)$ and $T' = (V', E', \text{Lab}', \mathcal{L}'_\Sigma)$ be trees. Then T' is a subtree of T, if, and only if, T' is a subgraph of T. T' is called complete, if all leaf nodes of T' are also leaf nodes of T. Otherwise, T' is called an incomplete subtree. $\diamond$

If graph-based data structures are processed in a program, developers often take the advantage of a *spanning tree* which links all of a graph's nodes and makes them accessible from a distinct root node as an entry point for algorithms that need to traverse the data structure in a certain order. Formally, a spanning tree is defined as follows:

Definition 3.9 (spanning tree):
Let $H = (V, E, \text{Lab}, \mathcal{L}_\Sigma)$ and $H' = (V', E', \text{Lab}', \mathcal{L}'_\Sigma)$ be directed graphs. H' is called a spanning tree of H if, and only if, it is a subgraph of H and a directed tree with $V' = V$. $\diamond$

In practice, a graph with a spanning tree frequently also is a multigraph with an explicit distinction between a specific predetermined spanning tree and the remaining compartments of the graph. These parts also form a subgraph which is called an *overlay graph* of the spanning tree:

Definition 3.10 (overlay graph):
Let $H = (V, E, \text{Lab}, \mathcal{L}_\Sigma)$ be a directed graph and $T = (V, E', \text{Lab}', \mathcal{L}'_\Sigma)$ a spanning tree of H. Further, let $O = (V, E'', \text{Lab}'', \mathcal{L}_\Sigma)$ be a subgraph of H. If $E'' \cap E' = \emptyset$ and $E = E'' \cup E'$, then O is called the overlay graph of T in H. $\diamond$

Besides the graphical representation as shown in Example 3.8, trees can be represented using a more compact term notation:

Definition 3.11 (term representation of a labeled tree):
Let $\mathcal{L}_\Sigma$ be some finite label alphabet. Then

- for all $l \in \mathcal{L}_\Sigma$: $l[\,]$ is a *tree term*,
- for any $k+1$ labels $l, l_1, \ldots, l_k \in \mathcal{L}_\Sigma$ and $k \in \mathbb{N}$ tree terms $t_1, \ldots, t_k$, $t = l[l_1 : t_1, \ldots, l_k : t_k]$ is a tree term and $t_1, \ldots, t_k$ are the *subterms* of t.

$\mathcal{T}(\mathcal{L}_\Sigma)$ is called the set of all tree terms over $\mathcal{L}_\Sigma$.

Given a term $t \in \mathcal{T}(\mathcal{L}_\Sigma)$, an equivalent graph representation $T = (V, E, \text{Lab}, \mathcal{L}_\Sigma)$ for this term can be derived by the following construction (the compartments of T are assumed initially empty and $c := t$):

(1) let $c = l[l_1 : t_1, \ldots, l_k : t_k]$ be the "current" tree term with $k \in \mathbb{N}_0$ subterms,

(2) if $k > 0$, then for each subterm t_i of c, where $i \in \mathbb{N}$ and $1 \leq i \leq k$, construct a graph-based representation $T_i = (V_i, E_i, \text{Lab}_i, \mathcal{L}_\Sigma)$ via (1) with $c := t_i$, afterwards, continue with (3),

(3) create a fresh node v_0 and compose the compartments of the T_i obtained from (2):

- $V = \{v_0\} \cup \bigcup\limits_{i=1}^{k} V_i$,
- $E = \bigcup\limits_{i=1}^{k} E_i \cup \{e_i\}$ and $<_E = \bigcup\limits_{i=1}^{k} <_{E_i} \cup \bigcup\limits_{i=2}^{k} \{(e_{i-1}, e_i)\}$, where $e_i = (v_0, r_i)$ and r_i the root of T_i,
- $\mathrm{Lab} = \{(v_0, l)\} \cup \bigcup\limits_{i=1}^{k} \mathrm{Lab}_i \cup \{(e_i, l_i)\}$, where $e_i = (v_0, r_i)$ and r_i the root of T_i. $\diamond$

Since terms and graphs are equivalent representations of trees, they will be used interchangeably depending on what is more appropriate in a certain situation.

Each sentence or sentential form σ generated by a CFG can be represented by a tree which is called a syntax tree (or derivation tree) with respect to σ. Syntax trees are constructed by

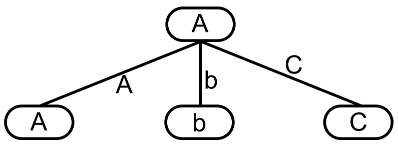


Figure 3.4.: Graphical tree representation of $A \to A\,b\,C$.

successively composing them from the local trees induced by the productions of the corresponding CFG. For example, a production $p_0 = A \to A\,b\,C$ yields a tree $t_0 = A[A : A[\,], b : b[\,], C : C[\,]]$. As a short-hand notation, empty brackets may be left and, if edge and node labels are the same, the edge label may be left and assumed implicitly, so that $A[A : A[\,], b : b[\,], C : C[\,]] \equiv A[A[\,], b[\,], C[\,]] \equiv A[A, b, C]$. Figure 3.4 shows the graphical representation of t_0 according to Definition 3.11.

The following definition introduces *syntax trees*.

Definition 3.12 (syntax tree):
Let $G = (N, \Sigma, P, S)$ be a reduced CFG. A tree-derivation relation $\Rrightarrow_G \subseteq \mathcal{T}(\mathcal{L}_\Sigma) \times \mathcal{T}(\mathcal{L}_\Sigma)$ with $\mathcal{L}_\Sigma = \Sigma \cup N$ can be derived from $\Rightarrow_G$ to generate tree terms:

$$\begin{aligned}\Rrightarrow_G =& \{(n[\,],\, n[\alpha_1[\,], \ldots, \alpha_{np}[\,]]) \mid n \to \alpha_1 \ldots \alpha_{np} \in P\} \cup \\ & \{(l[\ldots n[\,] \ldots],\, l[\ldots n[\alpha_1[\,], \ldots, \alpha_{np}[\,]] \ldots]) \mid n \to \alpha_1 \ldots \alpha_{np} \in P\}\end{aligned}$$

where $l, \alpha_i \in \mathcal{L}_\Sigma$, $1 \le i \le np$ and $i, np \in \mathbb{N}$ and $n \in N$. $\Rrightarrow_G^*$ denotes the reflexive transitive closure of $\Rrightarrow_G$. Let $\sigma = \sigma_1 \ldots \sigma_m$ be a sentential form with $m \in \mathbb{N}$ ordered symbols and $t \in \mathcal{T}(\mathcal{L}_\Sigma)$ where $T = (V, E, \mathrm{Lab}, \mathcal{L}_\Sigma)$ is the graph-based representation of t according to Definition 3.11 with $\mathcal{L}_\Sigma = \Sigma \cup N$ and $\mathrm{Lab}(r) = S$ where $r \in V$ is the root of T. Consider a concrete leftmost derivation $S \overset{p1}{\Rightarrow}_G \gamma_1 \overset{p2}{\Rightarrow}_G \ldots \overset{pk\text{-}1}{\Rightarrow}_G \gamma_{k-1} \overset{pk}{\Rightarrow}_G \sigma$ and a leftmost tree derivation $S \overset{p1}{\Rrightarrow}_G t_1 \overset{p2}{\Rrightarrow}_G \ldots \overset{pk\text{-}1}{\Rrightarrow}_G t_{k-1} \overset{pk}{\Rrightarrow}_G t$ where $j, k \in \mathbb{N}$, $1 \le j \le k$ and $pj \in P$ are the productions applied in each derivation step. t is called a *syntax tree* (*derivation tree*) of σ where $\sigma_1 \ldots \sigma_m$ are labels of the m empty subterms $\sigma_1[\,] \ldots \sigma_m[\,]$ in t and the corresponding leaf nodes of T. $\diamond$

The following example demonstrates how a syntax tree can be generated by a CFG.

Example 3.9 (syntax tree).
Let $G = (\{E, O, A, T\}, \{\&, |, f, t\}, P, E)$ be the CFG used in Example 3.1 and $E \Rightarrow_G^* t \,|\, f \,\&\, t$.
The corresponding syntax tree for $t \,|\, f \,\&\, t$ can be generated by $\Rrightarrow_G$ as follows:

$$\begin{array}{rlll}
E[\,] & \Rrightarrow_G & E[O] & (\text{by } p_0) \\
 & \Rrightarrow_G & E[O[O, |, O]] & (\text{by } p_1) \\
 & \Rrightarrow_G & E[O[O[A], |, O]] & (\text{by } p_2) \\
 & \Rrightarrow_G & E[O[O[A[T]], |, O]] & (\text{by } p_4) \\
 & \Rrightarrow_G & E[O[O[A[T[t]]], |, O]] & (\text{by } p_5) \\
 & \Rrightarrow_G & E[O[O[A[T[t]]], |, O[A, \&, A]]] & (\text{by } p_3) \\
 & \Rrightarrow_G & E[O[O[A[T[t]]], |, O[A[T], \&, A]]] & (\text{by } p_4) \\
 & \Rrightarrow_G & E[O[O[A[T[t]]], |, O[A[T[f]], \&, A]]] & (\text{by } p_6) \\
 & \Rrightarrow_G & E[O[O[A[T[t]]], |, O[A[T[f]], \&, A[T]]]] & (\text{by } p_4) \\
 & \Rrightarrow_G & E[O[O[A[T[t]]], |, O[A[T[f]], \&, A[T[t]]]]] & (\text{by } p_5)
\end{array}$$

◇

Definition 3.13 introduces syntax graphs which decorate syntax trees with additional edges:

Definition 3.13 (syntax graph):
Let $H = (V, E, \mathrm{Lab}, \mathcal{L}_\Sigma)$ be a directed graph and $T = (V', E', \mathrm{Lab}', \mathcal{L}'_\Sigma)$ be a syntax tree with respect to a CFG $G = (N, \Sigma, P, S)$ and sentential form σ with $S \Rrightarrow_G^* \sigma$. H is called a syntax graph of σ if T is a spanning tree of H. ◇

The tree derivation relation of desugared EBNF is equivalent to $\Rrightarrow_G$ of Definition 3.12 so that $\Rrightarrow_G$ is also used for EBNF derivations. As an example, reconsider the EBNF grammar G_E of Example 3.2. Except edge labels, the labeled syntax tree T in Figure 3.3 can be derived from Or, i.e., $Or \Rrightarrow_{\mathrm{G_E}}^* T$.

For navigation purposes, it is common to support edge-label declarations in EBNF grammars. An extension of the EBNF and the tree-derivation relation is sketched below:

Definition 3.14 (EBNF with labels):
A label-aware variant of EBNF can be obtained from Definition 3.4 by adopting $\mathcal{L}_{exp}$ in Definition 3.4 in such a way that a label declaration is allowed in front of any atom, grouping optional or repetition, i.e.,

- if $\alpha \in N \cup T$, $\alpha = \texttt{(}\beta\texttt{)}$, $\alpha = \texttt{(}\beta\texttt{)?}$, $\alpha = \texttt{(}\beta\texttt{)*}$ or $\alpha = \texttt{(}\beta\texttt{)+}$, and $\beta \in \mathcal{L}_{exp}$ and $l \in \mathcal{L}_{ident}$, then also $l : \alpha \in \mathcal{L}_{exp}$.

Given a desugared EBNF grammar G_E with labels and G the corresponding CFG, a modified derivation relation $\Rrightarrow_{GE}$ constructs T similar to $\Rrightarrow_G$, but uses the label names associated with terminals and nonterminals in productions of G_E. ◇

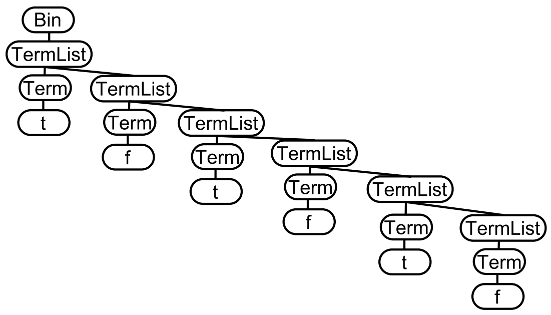


Figure 3.5.: A binary list.

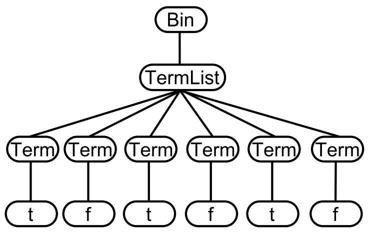


Figure 3.6.: A compact list.

3.1.5. Lists

A list is a primitive tree which contains a collection of similar nodes in a sequence. For ISC, lists are important to support fragment extensions. For example, consider the member declarations of a class declaration. To add new members to the class by composition, the composition system must be aware of the structure of the tree and were to insert a new node. In CFGs, lists like the members list are usually modeled using directly recursive nonterminals. For example, $AList \rightarrow AList\, b\, C$ is a direct *left* recursion, $AList \rightarrow C\; AList\; B$ is a direct *central* recursion and $AList \rightarrow C\, b\; AList$ is a direct *right* recursion over $AList$. The last recursion step is given by a nonrecursive production such as $AList \rightarrow A$ where A is the terminating entry of the list.

Definition 3.15 (list):
Let $G = (N, \Sigma, P, S)$ be a reduced CFG and $T = (V, E, \mathrm{Lab}, \mathcal{L}_\Sigma)$ with root r a syntax tree w.r.t. G. T is a list, if $h = \mathrm{Lab}(r)$ is a directly recursive nonterminal with at least one nonrecursive production $h \rightarrow \gamma$ where $\gamma \in (\Sigma \cup N - \{h\})^*$ and all recursive h-productions have exactly one recursive occurrence of h, i.e., $\forall (h \rightarrow \alpha\, h\, \beta) \in P : \alpha, \beta \in (\Sigma \cup N - \{h\})^*$. More specifically, T is also called a h-list. A node $v \in V$ is called a *list node*, if the complete subtree $T' = (V', E', \mathrm{Lab}', \mathcal{L}'_\Sigma)$ of T with root v is a list. $\diamond$

As a simplifying convention in this thesis and without loss of generality, it is assumed that lists induced by plain CFGs are generated by directly right recursive nonterminals with one or more recursive productions where each production represents an “entry type”, and one or more terminating productions. This convention is also suitable for repetitions in EBNF grammars and the desugarings presented in Section 3.1.2 which also yield right-recursive productions: an EBNF repetition yields a list and list nodes in a syntax tree. Moreover, if the two right-recursive productions of a CFG are of the form $n \rightarrow \alpha\, n$ and $n \rightarrow \alpha$, where n is the recursive nonterminal and α the “entry” terminal or nonterminal, then trees induced by n may be written in a compact notation. Hence, any $n - list\; n[\alpha[\ldots], n[\alpha[\ldots], n[\ldots]]]$ translates into a flattened form $n[\alpha[\ldots], \alpha[\ldots], \alpha[\ldots], \ldots]$.

Example 3.10 (list).
Let G be the EBNF grammar with the following productions:

```
Bin ::= Term+
Term ::= "t" | "f"
```

`Bin` $\Rightarrow_G^* T$ generates the syntax tree T w.r.t. `Bin` $\Rightarrow_G^* tftftf$ in Figure 3.5 where `TermList` is the recursive nonterminal. The corresponding term s is

$$\begin{aligned} s =& \mathit{Bin}[\mathit{TermList}[\mathit{Term}[t], \mathit{TermList}[\mathit{Term}[f], \mathit{TermList}[\mathit{Term}[t], \\ & \mathit{TermList}[\mathit{Term}[f], \mathit{TermList}[\mathit{Term}[t], \mathit{TermList}[\mathit{Term}[f]]]]]]]]. \end{aligned}$$

Alternatively, in compact notation this is equivalent to

$$s = \mathit{Bin}[\mathit{TermList}[\mathit{Term}[t], \mathit{Term}[f], \mathit{Term}[t], \mathit{Term}[f], \mathit{Term}[t], \mathit{Term}[f]]].$$

A complementary graphical representation of s is shown in Figure 3.6. ◇

To distinguish between the visible representation of an artifact and the abstract structure of the corresponding data, the notions of *concrete syntax* and *abstract syntax* as well as *concrete syntax tree (CST)* and *abstract syntax tree (AST)* are common. In comparison to CSTs, ASTs typically have a different grammar describing the data structure rather than textual language. Moreover, ASTs are typically transformed and enriched in further language-processing steps yielding graph-structured representations called *abstract syntax graphs (ASGs)*.

The following section describes the relation of concrete and abstract syntax in more detail. It gives a short overview on common parsing technologies. Furthermore, the notions of abstract syntax and *metamodels* are discussed to provide a basic understanding on the technical basis of the ISC approach developed in this thesis and in which domains it may be applied since ISC is not limited to textual languages only.

3.2. On Parsing, Unparsing and Abstract Syntax

Up to this point, it was discussed how a CFG *generates* a language and syntax trees. An equally relevant problem is the *word problem*, i.e., the decision on if a given sequence of terminal symbols is in the language. For any context-free language $\mathcal{L}$, a push-down automaton (PDA) A can be specified in such a way that A *accepts* all words in $\mathcal{L}$ and declines all others, i.e., $L(A) = \mathcal{L}$. Informally, a PDA is a state machine that reads symbols in Σ from an input stream, and pushes and pops symbols (which can, but do not have to be in Σ) onto a stack. State transitions depend on the uppermost stack symbol and the next symbol to be read from an input stream (i.e., the word). Given a CFG G with $L(G) = \mathcal{L}$, a corresponding PDA A can be generated automatically, so that $L(G) = L(A) = \mathcal{L}$ (cf. [Schöning 2001]). A A is called *deterministic* if for each state of A there is at most one possible successor state for any given stack entry and input symbol.

Otherwise, A is called *nondeterministic*. A context-free language $\mathcal{L}$ is called deterministic if there exists a deterministic PDA that accepts $\mathcal{L}$ (cf. [Schöning 2001]).

A parser is a program (or function) that uses or simulates a PDA derived from a given CFG to (1) automatically solve the word problem for a given input string and (2) to construct a corresponding syntax tree to enable further processing of the given input. Therefore, the parser consumes the symbols on the input stream in a strict order (usually from left to right) and constructs a left-most or right-most derivation, if possible. The most common approaches for implementing or generating parsers from a given CFG are $LL(k)$ and $LR(k)$ where the first L stands for the left-to-right input stream consumption while the second L (R) stands for left-most (right-most) derivation and k is the number of symbols the parser may look-ahead in the input stream to make a decision. k is usually larger than 0 and, depending on the concrete implementation approach, it is fixed or variable. LL parsers are frequently constructed as top-down algorithms using the *recursive descent* approach which simulates the PDA stack via recursive method calls. In contrast, LR parsers are constructed as *bottom-up* parsers using a so-called item automaton—a PDA with a special construction. The class of languages that can be accepted by $LR(k)$ parsers corresponds to the class of deterministic context-free languages (cf. [Schöning 2001]). For further details on how parsers can be generated from CFGs it is kindly referred to the common literature on compiler construction (e.g., [Aho et al. 1986; Wilhelm and Maurer 1997]).

3.2.1. Parsing Schemes

From a theoretical point of view, $LR(k)$ parsers are superior over $LL(k)$ as the class of languages that can be described with grammars an $LR(k)$ parser can be generated from is larger than the class of $LL(k)$ languages. However, in practice no clear preference of one or the other is visible. Prominent representatives from the class of $LR(k)$ generators are YACC (Yet Another Compiler Compiler) [Johnson 1975] and its various reincarnations which are based on the $LALR(1)$ optimization of $LR(1)$. A well-known example of a recursive descent LL parser generator is ANTLR (ANother Tool for Language Recognition) [Parr and Quong 1995].

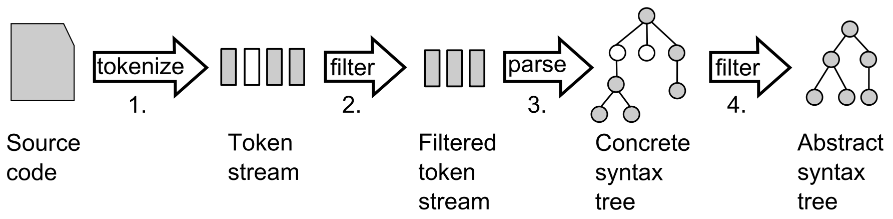


Figure 3.7.: The typical parsing scheme in four phases.

Figure 3.7 shows a logical view on the parsing process which is typically employed in parser software. Phase (1) is called *tokenization* (or also *lexical analysis* or *scanning*). The input stream of bytes is read and converted into a stream of tokens such as keywords or identifiers. Phase (2) filters certain kinds of tokens from the token stream, usually this is white space consiting of blanks and line breaks. Phase (3) is the typical parsing phase (or also *syntactic analysis*): the parser reads the filtered stream of tokens and derives the corresponding concrete syntax tree (CST). Finally, during phase (4), a more compact equivalent abstract syntax tree (AST) representation is derived from the CST, e.g., by removing redundant information from the tree and converting it into an appropriate data structure for further processing.

Frequently, the mentioned phases are not cleanly separated. In general, the CST is not constructed explicitly, instead an AST is constructed directly via *semantic actions* that are annotated to each production. Also, white-space tokens are discarded immediately during lexical analysis instead of performing an extra filter step. In the classical parser generator approaches as discussed above, tokenization is based on regular expressions and *deterministic finite automatons (DFAs)*—a subset of PDAs that do not use a stack. DFA-based tokenization makes parsing more efficient by decreasing PDA sizes and provides preparsing disambiguation techniques such as the principle of the *longest match* (cf. [Wilhelm and Maurer 1997]). However, a strict separation of tokenization and parsing often turns out to be problematic when languages are to be composed or if disambiguation depends on parser state [Karol and Zschaler 2010]. To enable more sophisticated

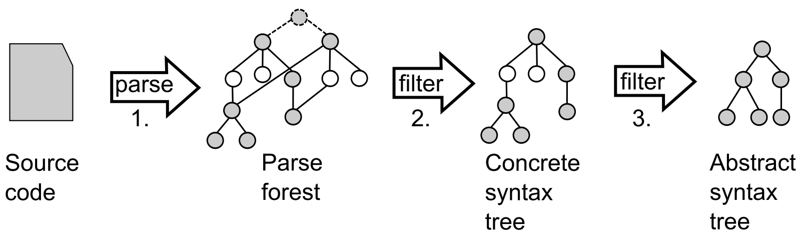


Figure 3.8.: A typical SGLR parsing scheme.

means for disambiguation, several more recent parsing algorithms have emerged. Generalized LR parsing (GLR) [Tomita 1991] maintains a set of ambiguous LR states during parsing and, thus, may emit multiple CSTs as a parse forest for the same input. The scannerless generalized LR parsing (SGLR) [Brand et al. 2002] approach uses the capabilities of GLR to also handle lexical disambiguation depending on a parser's state and often is considered as the most powerful parsing technology in the literature when it comes to language modularization and expressiveness. However, SGLR not always is a perfect solution for ambiguous languages. This becomes obvious if the SGLR parsing scheme in Figure 3.8 is investigated. The problems are twofold: SGLR parsers always build up an internal CST data structure to represent the parse forest. From the forest, one valid representation has to be selected and it still needs to be determined which one is the "right" one—sometimes a non-trivial problem. Finally, after selecting the concrete CST, it has to be converted into an adequate AST.

On the other end of the scale, parsing expression grammars (PEGs) and *packrat parsing* [Ford 2004] have proven to be an expressive approach for scannerless parsing and disambiguation. Packrat parsing is a typical recursive descent algorithm—similar to LL, but with sophisticated backtracking and caching mechanisms. A prackrat parser usually has no preceding lexical

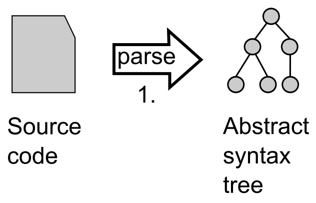


Figure 3.9.: A typical PEG/packrat parsing scheme.

analysis phase and directly produces an AST (cf. Figure 3.9). Although related, PEGs are more expressive than normal EBNF grammars or CFGs in general. If a textual language is specified as a PEG, exactly one derivation of a string is selected during parse, yielding a unique parse tree. Like deterministic parsers, parsing is still achieved in linear time by using memoization [Ford 2004]—a caching mechanism which recalls any successfully parsed substring in relation with its original position in the input string. Although PEGs and EBNF look similar, the former is more a "parser programming language" while EBNF is declarative and lacks any notion of order or precedence in derivation steps. Hence, the derivation relation of PEGs is different from the derivation relation of CFGs.

Syntactically, the main differences between EBNF and PEGs are as follows (see [Ford 2004] for a complete specification):

- Choices in PEGs are sequentially ordered, which is reflected by using the choice operator "`/`" instead of "`|`".
- Left and right side of a production are separated with "`<-`" instead of "`::=`".
- Syntactic predicates for matching the next characters in the input without actually consuming the input. PEGs support "`&`" (*and*) as well as "`!`" (*not*) as predicate operators.
- PEG-based parsing of the repetition operators (`*` and `+`) and the optional operator (`?`) is greedy, i.e., consumption is preferred over reduction.

In this thesis, PEGs are used as a basic technology for realizing the Minimal-ISC approach presented in Chapter 7. In other implementation parts of the thesis with focus on textual languages, classical LL and LR approaches are used for parsing.

3.2.2. Representations of Abstract Syntax

Abstract syntax is the program-internal representation of an artifact for further processing. As already anticipated in the previous sections, abstract syntax can be represented as ASTs but also as a general, graph-based data structure such as an ASG. There is no definitive rule on the abstraction level of abstract syntax. Thus, it is possible that the CST of a program is equivalent to its AST or they could also be completely different. Although the above terminology originally stems from the communities of compiler construction and software language engineering, it has much broader implications. Grammars—as a specification vehicle with various incarnations—and the above-mentioned data structures are in broad use in software applications. Grammar-related software is also frequently called *grammarware* in literature [Klint et al. 2005b].

ASTs and ASGs are *typed* according to an abstract-syntax specification formalism. In the following, the eXtensible Markup Language (XML) [World Wide Web Consortium (W3C) 1998–2009] and the Essential Meta Object Facility (EMOF) [Object Management Group (OMG) 2011c] are discussed as commonly and practically used approaches to specify abstract syntax. Afterwards, it is introduced how abstract syntax is specified in this thesis and how it relates to the XML and EMOF standards.

XML Trees as Abstract Syntax Representation

In the sense of [Klint et al. 2005b], XML-based applications as they are widely-used in industry can be considered grammarware applications. Boiled down to grammarware terminology, XML is a standardized format for the efficient serialization, de-serialization and processing of ASTs. While XML represents arbitrary labeled trees, schema languages such as the XML Schema Definition Language (XSD) [World Wide Web Consortium (W3C) 2012] define namespaces by restricting the set of possible labels in an XML file to a certain alphabet and by defining how nodes may be nested (i.e., the edge relation). XML schemata are strongly related to *regular tree grammars* [Murata et al. 2005]. Regular tree grammars can be considered as a special kind of CFGs, related to EBNF. Productions in regular tree grammars are restricted to the form $n \to t\,r$ where $n \in N$, $t \in T$ the label of the local root and r is a regular expression over N. In contrast to CFGs, regular tree grammars are typically used to *validate* ASTs that already have been instantiated from some source or program. For more detailed discussions on the relation of XML, XSD and CFGs, see [Berstel and Boasson 2002] and [Brüggemann-Klein and Wood 2004]. Figure 3.10 roughly shows a general processing scheme of applications that use XML

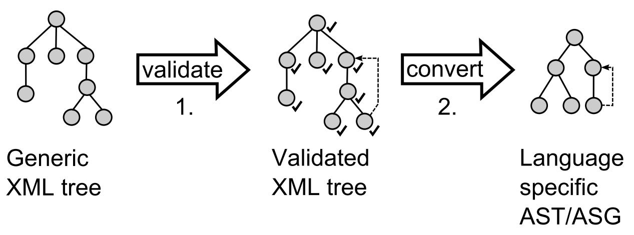


Figure 3.10.: A typical processing scheme of an XML-based application that uses an XSD–programming language binding.

trees and XML schemata to load data from their repositories. The generic XML tree on the left is an AST which represents a tree that adheres to the Domain Object Model (DOM) [World Wide Web Consortium (W3C) 1998–2012]—a general, standardized model to represent tree structures in memory at application runtime. The generic XML tree is typically generated by some standard XML parser which itself obeys to the general parsing scheme in Figure 3.7. Note that the separation of phases in XML parsers is not necessarily as strict as presented since Figure 3.7 contains a conceptual view on parse processes. In a first process step, the XML schema comes into play. Typically, an XSD validation engine traverses the DOM tree and compares it to the grammar specified in the XSD. XSD has features that go beyond the expressiveness of regular tree grammars. Key-value references are a context-sensitive feature to specify distinct reference edges to connect distant nodes in the DOM tree. Furthermore, the validated DOM tree contains typed attribute values (i.e., token values in terms of CFGs). In the second process step of Figure 3.10, the DOM tree is converted into a specific AST data structure supported by the target environment. In statically typed object-oriented languages like Java, such data structures are represented as object trees that are typed by a set of classes and a convenient programming interface. XML bindings that convert DOM trees into specific ASTs can—of course—be implemented by deriving classes from the XML schema by hand, or easily be generated from an XSD specification by a XML library providing a binding generator, e.g., the Java Architecture for XML Binding (JAXB) [Sun Microsystems, Inc. 2009]. The binding generator uses the regular tree grammar in the schema and the type information to derive an adequate static class hierarchy that can be serialized and deserialized automatically.

Models as Abstract Syntax

While XML is a platform-independent format for tree-shaped documents and a vehicle for data exchange, it does not define a general metadata API required for the interoperability of tools. The need for a standardized metadata API was addressed by the Object Management Group (OMG) in the Model-Driven Architecture (MDA) proposal [Object Management Group (OMG) 2003] and especially the Meta Object Facility (MOF) [Object Management Group (OMG) 2011c]. Relying on the MOF, the OMG defined a bunch of other standards and generic languages for transformation, querying and modeling (cf. [Object Management Group (OMG) 2003–2013]), implemented by various tool vendors on different platforms. Most notably, the MOF is used as a definition language for the Unified Modeling Language (UML) [Object Management Group (OMG) 2011d].

In the MOF standard, two compatibility layers are defined—the EMOF and the Complete Meta Object Facility (CMOF). Both are constrained subsets of the UML, while EMOF is considered a subset of CMOF. The distinction between EMOF and MOF was made for the following reasons [Object Management Group (OMG) 2011c, p. 31]:

- EMOF provides features that “closely correspond[s] to the facilities found in OOPLs [(object oriented programming languages)] and XML”.

- EMOF allows users to define “simple metamodels [...] using simple concepts while supporting extensions [...] for more sophisticated metamodeling using CMOF”.

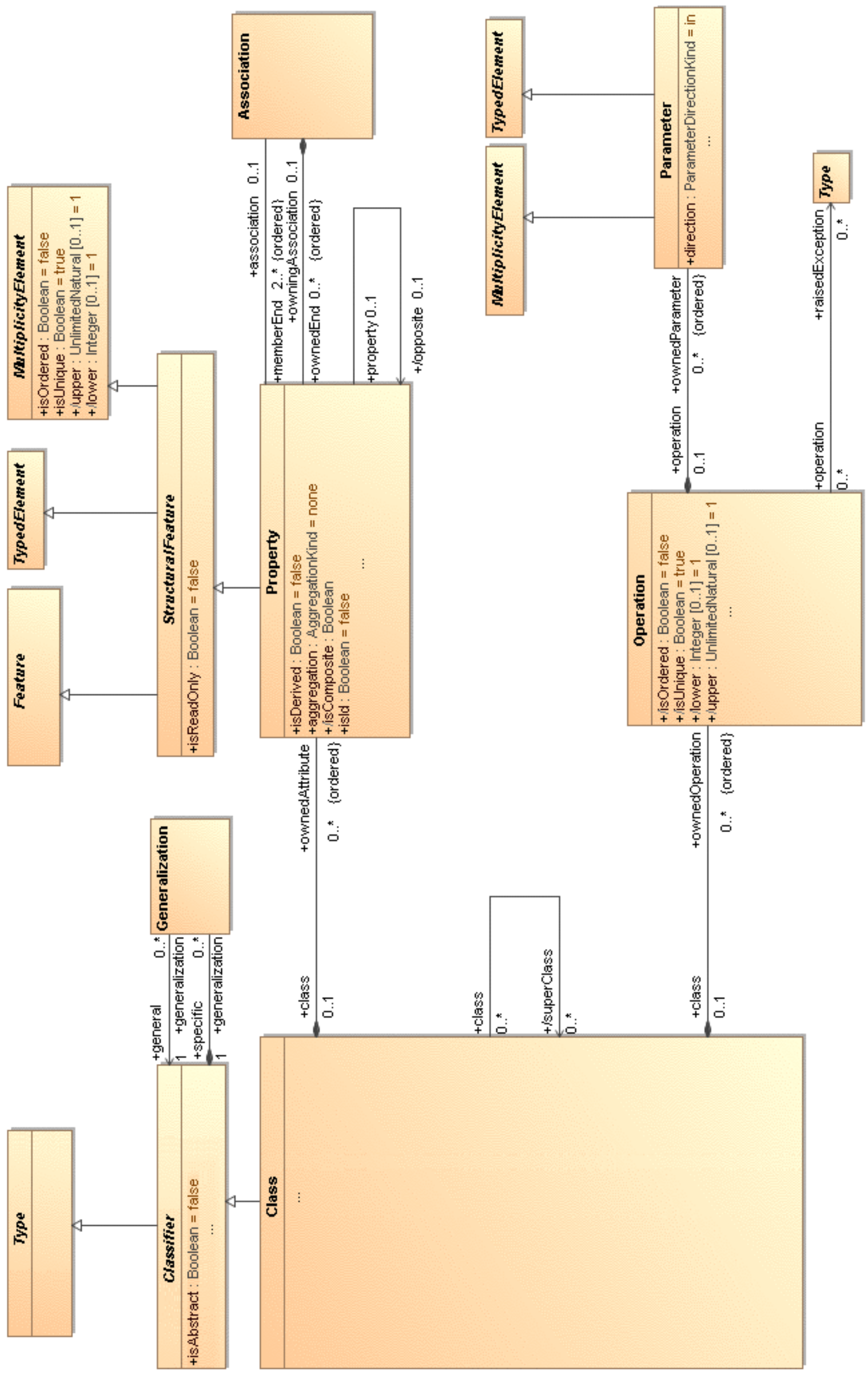
- EMOF is specified in the standard as a complete merged model. This way, the EMOF metamodel “can be used to bootstrap metamodel tools rooted in EMOF without requiring an implementation of CMOF and package merge semantics.”

Hence, EMOF enables tool interoperability by providing a compact metamodeling standard which is easy to implement and self-contained since no other concepts from the MOF or UML standards are required. Figure 3.11 shows the EMOF metamodel as an UML class diagram. The basic ingredients one also finds in object oriented programming languages. EMOF has `Types`, `Classes` and `Generalization` for specifying static type hierarchies. Similar to object-oriented languages, `Classes` can be abstract or concrete and `Generalization` corresponds to multiple inheritance modulo dynamic dispatch. Furthermore, EMOF supports `PrimitiveType` definitions, `Enumerations` and `Packages` which are not included in Figure 3.11 but are also in the standard. `Classes` may have `Properties` as `StructuralFeatures` and `Operations` as `BehavioralFeatures`. `Properties` may be related with an `Association` which represents a binary relation. Observe that the number objects a `Property` can be associated with is determined by providing it a `lowerBound` and an `upperBound`. The kind of a `Property` in EMOF can only be `composite` or `none`. A `composite Property` makes the objects owned by a corresponding instance dependent of the existence of the *container* (i.e., the owning object). Observe that even if a unique object may have an arbitrary number of composite `Properties`, each object can have at most one container. Consequently, if the container is deleted from the model, all transitively owned objects are also deleted. In contrast, such restrictions do not apply for *non-containment* properties (i.e., `none` kind `Properties`). `Properties` that are not part of `Association` ends are simple values typed by primitive types. Primitive types are not classes in the metamodel, but typically basic types such as strings and Integers. In EMOF, the specification of those is lent from the XSD definition [Object Management Group (OMG) 2011c] so that EMOF values are compatible with XML terminals.

Although the standard specification of EMOF does not strictly define how to structure metamodels, containment associations are typically used to model tree structures since the relation between container and contained object obviously resembles the parent–child relation of labeled trees as in Definition 3.7. Hence, containments can be used to construct tree-based models if the following requirements on EMOF metamodels and models are imposed:

- Each `Class` in the metamodel must participate as contained type in at least one `containment Association`, except `Classes` that are supposed to be root node types.
- Each object (i.e., instance of a `Class` in the metamodel)—except the root of the tree-based model—must be contained in a container such that each object is reachable from the root object.

Note that the first requirement is not strict: for metamodels not fulfilling it, a *tree-completion* operator could be used which simply adds a `Class` R to the metamodel and by adding an *unbounded* `containment Association` between R and each of the not contained non-root



MultiplicityElement
+isOrdered : Boolean = false
+isUnique : Boolean = true
+upper : UnlimitedNatural [0..1] = 1
+lower : Integer [0..1] = 1
TypedElement
Feature
StructuralFeature
+isReadOnly : Boolean = false
Association
+association 0..1
+memberEnd 2..* {ordered}
+owningAssociation 0..1
+ownedEnd 0..* {ordered}
+property 0..1
+/opposite 0..1
Property
+isDerived : Boolean = false
+aggregation : AggregationKind = none
+isComposite : Boolean
+isId : Boolean = false
...
+ownedAttribute 0..* {ordered}
TypedElement
MultiplicityElement
Parameter
+direction : ParameterDirectionKind = in
...
Type
+raisedException 0..*
+operation 0..1
+ownedParameter 0..* {ordered}
+operation 0..*
Operation
+isOrdered : Boolean = false
+isUnique : Boolean = true
+lower : Integer [0..1] = 1
+upper : UnlimitedNatural [0..1] = 1
...
+ownedOperation 0..* {ordered}
Generalization
+general 1
+generalization 0..*
+specific 1
+generalization 0..*
Type
Classifier
+isAbstract : Boolean = false
...
Class
...
+class 0..1
+class 0..*
+/superClass 0..*
+class 0..1

Figure 3.11.: Concepts and relations in EMOF [Object Management Group (OMG) 2011c].

`Classes`. Based on `containment Associations` and the above requirements, it can be concluded that tree-based models are ASTs if no non-`containment Associations` are specified in the corresponding metamodel and no non-`containment` links are in the model. If this restriction is left out, such links may occur in the tree-based model yielding an ASG structure with a *unique* spanning tree (cf. Definition 3.9).

Models in MDE are often rendered and edited via a graphical user interface, e.g., the class diagram in Figure 3.11 is a graphical representation of the underlying class model. For serialization and model exchange purposes, tools commonly store models as XML files. Therefore, OMG provided the XML Metadata Interchange (XMI) standard to provide a generic XML-based format for meta descriptions and the exchange of models [Object Management Group (OMG) 2011b]. While the meta part of XMI is well-defined by the corresponding XSD schema, it does

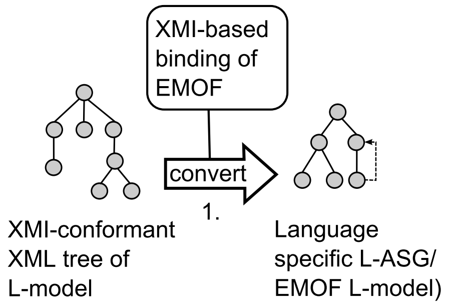

XMI-based binding of EMOF
convert
1.
XMI-conformant XML tree of L-model
Language specific L-ASG/ EMOF L-model)

Figure 3.12.: The typical processing scheme of a model-based application that uses the EMOF XMI schema binding.

not cover a generic schema for the model data. Consequently, a mapping between MOF and XMI is provided in the mapping standard (cf. [Object Management Group (OMG) 2011b]) which can be used to derive XML schemata from models and allows tool vendors to realize language bindings like with JAXB. Figure 3.12 emphasizes the processes step that converts a generic XML DOM tree of an instance of a metamodel L into an object tree of L in a bound language using the mapping specified in the XMI standard.

Similar to XML, tree-based metamodels are related to EBNF and therefore CFGs. In [Alanen and Porres 2003], an automatic mapping form EBNF to MOF and its inverse is discussed. Naturally, while all EBNF grammars have at least one representation as a tree-based metamodel, general metamodels cannot be represented by a CFG. If tree-shaped metamodels without inheritance are considered, an EBNF grammar can be derived directly (cf. [Alanen and Porres 2003]). Furthermore, the author of [Kunert 2008] discusses a semi-automatic mapping from EBNF to MOF which allows users to tailor the metamodel which is derived from a grammar (e.g., by introducing inheritance or extra classes). The dominant tree structure in EMOF models enables the usage of models as a representation of abstract syntax for textual languages with the benefits of tool integration with model-based software, e.g., UML modeling platforms and the reuse of existing model-based tools. EMFText [Heidenreich et al. 2009a] is a tool which uses the

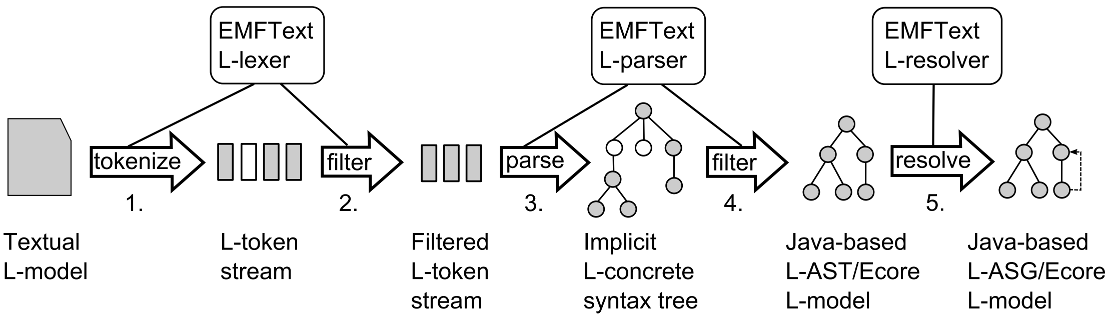


Figure 3.13.: Processing scheme of EMFText.

relation between EMOF and grammarware to generate advanced model-based textual editors for the Eclipse Modeling Framework (EMF) [Eclipse Foundation 2013b; Steinberg et al. 2009]. The parsing process of a hypothetical language L using EMFText and EMF is shown in Figure 3.13 on the basis of the general parsing scheme in Figure 3.7. Syntax specifications in EMFText are based on regular expressions (for tokens) and EBNF (for the context-free parts of the language) and are tightly coupled with the metamodel. From the syntax specification and the according metamodel, EMFText generates a lexer and a parser component. The lexer component covers the phases (1) and (2) of the process. The parser directly generates the AST from the token stream input (phases (3) and (4)). Similar to the XML case, during phase (5) references are resolved, which has to be implemented by hand in most cases.

3.2.3. Abstract Syntax in this Thesis

In this thesis, EBNF is used as a specification formalism of abstract and concrete syntax. This has the advantage of having a grammar language for ISC fragments which is directly suitable for using it with RAGs and still is semantically close to the domains of web languages (XML) and modeling languages (EMOF). However, to be more close to an object-oriented notion of abstract syntax and to support extensible AST grammars, it is beneficial to include the notation of *abstract nonterminals* and *nonterminal inheritance* in the AST specification formalism:

Definition 3.16 (EBNF with nonterminal inheritance):
A flat EBNF grammar with nonterminal inheritance is a 5-tuple $G = (N, T, P, S, A)$ where N and T are equivalent to Definition 3.4, $A \subset N$ is a set of abstract nonterminals, $S \in N \setminus A$ and P is $P \subset \mathcal{L}_{nt} \times \mathcal{L}_{exp}$, where $\mathcal{L}_{exp}$ is given by Definition 3.4 and Definition 3.14 (with labels), and $\mathcal{L}_{nt}$ is defined as follows:

- if $n \in N \setminus A$, then $n \in \mathcal{L}_{nt}$,
- if $n \in A$, then $@n \in \mathcal{L}_{nt}$ (abstract declaration),
- if $n \in N$ and $m \in A$ then $n \triangleright m \in \mathcal{L}_{nt}$ (inheritance declaration).

If not stated otherwise, the following constraints are defined with respect to G and $n, m, q \in N$:

- for each $n \in A$ there is one abstract declaration $@n ::= \ldots \in P$ or $@n \triangleright m ::= \ldots \in P$,
- for any two productions $n \triangleright m ::= \ldots \in P$ and $n \triangleright q ::= \ldots \in P$ it holds that $q = m$,
- for any two productions $n \triangleright m ::= \ldots \in P$ and $q ::= \ldots \in P$ it holds that $q \neq n$,
- the inheritance relation $\triangleright$ is acyclic, i.e., a nonterminal may not inherit from itself. ◇

The semantics of EBNF grammars with nonterminal inheritance is as before given by desugaring transformations translating the inheritance relations into normal EBNF according to Definition 3.4 which itself is defined by a mapping to plain CFGs. If no abstract nonterminals are in the grammar, the EBNF-to-CFG transformations described in Section 3.1.2 can be applied immediately. Otherwise, let $G = (N, T, P, S, A)$ be an EBNF grammar with abstract nonterminals.

- Let $n \in A$ be an abstract nonterminal and $k \in \mathbb{N}_0$ nonterminals m with at least one inheritance declaration $m \triangleright n ::= \ldots \in P$. If $k > 0$ and n occurs on a right-hand side of a production $p = q ::= \alpha_1 | \ldots | \alpha_i | \ldots | \alpha_l \in P$ with $l \in \mathbb{N}$ alternatives and $\alpha_i = \delta\, n\, \gamma$, then a fresh production $p' = q ::= \alpha_1 | \ldots | \alpha_{i-1} | \beta_1 | \ldots | \beta_k | \alpha_{i+1} | \ldots | \alpha_l$ replaces p in P where $\beta_j = \delta\, m\, \gamma$ are k new alternatives for each m that inherits from n.

Example 3.11 (desugaring of abstract productions).
```
S ::= (A)* A       S ::= (B)* B | (B)* C | (C)* B | (C)* C
@A ::= "a"    =>   @A ::= "a"
B▷A ::= <b>        B▷A ::= <b>
C▷A ::= <c>        C▷A ::= <c>
```
◇

- If no abstract nonterminals are contained on the right-hand sides of productions in P, for each inheriting production $p = \alpha \triangleright n ::= \delta \in P$ and $@n ::= \gamma \in P$ where $n \in A$, $\alpha = @m$ (if $m \in A$) or $\alpha = m$ (if $m \in N \setminus A$), a fresh production p' replaces p in P. Assume that $\delta = \delta_1 | \ldots | \delta_k$ has $k \geq 1$ top-level alternatives. If γ has one alternative, then $p' = \alpha ::= \gamma\, \delta_1 | \ldots | \gamma\, \delta_k$. If γ has more than one alternative, then $p' = \alpha ::= (\gamma)\, \delta_1 | \ldots | (\gamma)\, \delta_k$. Finally, all abstract productions in P and the corresponding nonterminals in A are eliminated from the grammar.

Example 3.12 (desugaring of inheriting productions).
```
S ::= (B)* B | ...      S ::= (B)* B | ...
@A ::= "a"          =>  B ::= "a" <b>
B▷A ::= <b>             C ::= "a" <c>
C▷A ::= <c>
```
◇

Nonterminal inheritance brings EBNF close to the technical realization of abstract syntax in object-oriented languages. Abstract nonterminals can directly be realized by mapping normal

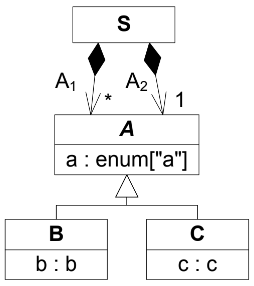


Figure 3.14.: Representation of the initial EBNF grammar of Example 3.11 in a UML class-diagram-like notation.

nonterminals to classes using abstract classes for binding abstract nonterminals and class-based inheritance for mapping nonterminal inheritance. This way, members (induced by the node labels) of the abstract nonterminal can be accessed from inheriting nonterminals via the built-in polymorphic dispatch of object-oriented languages. Other grammar-based abstract-syntax languages such as XML and EMOF are bound to object-oriented implementation languages in similar ways. For example, abstract `EClass`es in the EMF realization of EMOF (cf. Section 3.2.2) are technically backed by abstract Java classes, interfaces and Java inheritance. Language bindings of XSD specifications for XML such as JAXB (cf. Section 3.2.2) typically map abstract complex XML types and inheritance between complex types to classes and inheritance concepts in object-oriented languages. Consequently, a UML class-diagram-like notation can be used to visualize EBNF grammars with nonterminal inheritance and labels. This is exemplified in Figure 3.14. Nonterminals are represented as classes, abstract nonterminals are abstract classes. Containment references symbolize nonterminals on the right-hand side while class properties represent keywords and tokens. However, the ordering of nonterminals is not reflected in the diagram, but can still be obtained from the original grammar.

Furthermore, nonterminal inheritance is also supported by state-of-the-art RAG implementations like JastAdd which is used in Chapters 6 and 7 of this thesis. Hence, the theoretical results of this thesis can be transferred easily to an implementation framework. Finally, nonterminal inheritance is actually a component model for EBNF which allows for modular grammars where abstract nonterminals play the roles of slots where nonterminals of other modules can be bound and integrated into the language via inheriting productions. This is essential for building extensible fragment component models based on RAGs.

3.2.4. Unparsing Schemes

Unparsing (frequently also called *pretty printing*) is an issue for any metaprogramming approach that transforms textual languages using their ASTs or models: when a transformation (e.g., fragment composition) is completed, the structure needs to be printed back to a file from as a stream of tokens. Figure 3.15 sketches the logical structure of the unparsing process in more detail.

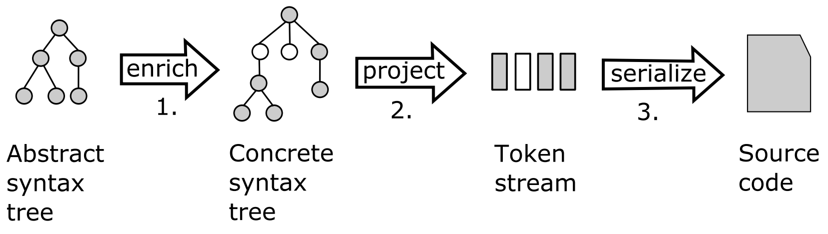


Figure 3.15.: A typical unparsing scheme in three phases.

First, the AST (or model) needs to be converted into a CST in such a way that the abstracted information is "concretized". For example, this may include information on the formatting of expressions in case brackets of nested expressions were removed before, converting terminal-values typed by the type system of the AST's host language, or putting nodes in an appropriate order if the AST is not ordered. In a second step, an output token stream must be projected from the CST. This typically involves adding layout information such as white-space characters formatting the output with blanks, line-break characters and indentation. Finally, the token stream is printed sequentially as characters to a file (or another serialization target).

In practice, the three stages above are often virtually reduced into a single stage. During this stage, the AST is traversed once printing characters directly to a corresponding output stream of a file handle. Figure 3.16 sketches this compact scheme. Commonly, the traversal algorithm

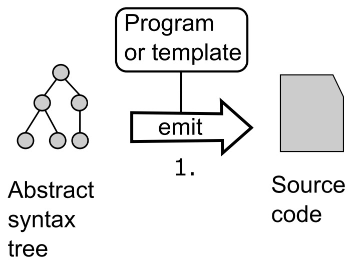


Figure 3.16.: Pretty printing with programs and/or templates.

and the character-emitting code is implemented in the AST's host programming language using print statements and the language's operations on strings (e.g., concatenation). However, it is also common to use a string-emitting template engine (i.e., TMP abstractions) with the advantage of having the object code readable in a parametrizable template. Other approaches provide semi-formal languages for text-layout computation [Jonge 2000], e.g., box layout which is similar to layout descriptions of user interfaces or websites. The previously discussed EMFText tool also has a simple layout language built into its syntax specification language.

Often, it is an advantage to preserve the layout information from the input and reuse it for pretty-printing, e.g., for source-code refactoring tools [Jonge and Visser 2012]. But also fragment composition systems such as those discussed in this thesis profit from layout preservation since users normally expect that indentation and white spaces of fragment components are replicated in composed fragments.

As a final remark, please observe that pretty printing is only a minor topic in this thesis and will not receive a detailed attention in the respective chapters.

Having discussed all required concepts related to context-free languages, the following section discusses RAGs and introduces the corresponding notation used in this thesis.

3.3. Reference Attribute Grammars (RAGs)

Attribute grammars (AGs) are a well-known formalism that can be used construction to specify parts and algorithms of compiler frontends. As introduced by Knuth [Knuth 1968; Knuth 1971], AGs are an extension of CFGs enabling the specification of computations and the distribution of data between distant nodes in syntax trees of the underlying context-free grammar. Therefore, each node of a syntax tree carries a set of *attributes*. These attributes can be either *inherited* (passing information from top to bottom) or *synthesized* (passing information from bottom to top). To compute the actual attribute values, an evaluator traverses the attributed syntax tree and evaluates attributes according to a semantic rule specified in the AG which may only directly depend on attributes in the local context of the node the attribute is associated with.

3.3.1. Basic Attribute Grammars

Following [Kühnemann and Vogler 1997] and [Wilhelm and Maurer 1997], Definition 3.17 introduces AGs as a formal concept.

Definition 3.17 (attribute grammar):
An attribute grammar (AG) is an 8-tuple G=(G_0,Syn,Inh,Syn_x,Inh_x,K,Ω,Φ) with the following constituents:

- $G_0 = (N, \Sigma, P, S)$, a reduced CFG,
- Syn and Inh, the finite, disjoint sets of synthesized and inherited attributes,
- $\mathrm{Syn}_x : N \to 2^{Syn}$, a function that assigns a set of synthesized attributes to each nonterminal in G_0,
- $\mathrm{Inh}_x : N \to 2^{Inh}$, a function that assigns a set of inherited attributes to each nonterminal in G_0,
- K, a set of attribute types,
- $\Omega : Inh \cup Syn \to K$, a function assigning each attribute a a $\kappa \in K$,
- Φ, a set of semantic functions $\varphi_{(p,i,a)}$ related to *attribute occurrences* or *local attributes* (p, i, a), where $p = p_0 \to p_1 \dots p_{np} \in P$, $i \in \{0, \dots, np\}$ and $a \in \mathrm{Syn}_x(p_i) \cup \mathrm{Inh}_x(p_i)$.

Let $p = p_0 \to p_1 \dots p_{np} \in P$ be a production of G and $i \in \{0..np\}$. The following sets and functions are provided:

- $Att = Syn \cup Inh$, the set of all attributes in G,
- $\mathrm{Att}_x(y) = \mathrm{Syn}_x(y) \cup \mathrm{Inh}_x(y)$ if $y \in N$ and $\mathrm{Att}_x(y) = \emptyset$ if $y \in \Sigma$, a function that provides the set of all attributes associated with y,
- $\mathrm{Att}(p) = \{(p, i, a) \mid a \in \mathrm{Syn}_x(p_i) \cup \mathrm{Inh}_x(p_i)\}$, a function that provides the set of attribute occurrences of p,
- $\mathrm{Att}_{\text{in}}(p) = \{(p, 0, a) \mid a \in \mathrm{Inh}_x(p_0)\} \cup \{(p, i, a) \mid a \in \mathrm{Syn}_x(p_i) \wedge i \geq 1\}$, a function that provides the set of incoming attribute occurrences of p,
- $\mathrm{Att}_{\text{out}}(p) = \mathrm{Att}(p) \setminus \mathrm{Att}_{\text{in}}(p)$, a function that provides the set of outgoing attribute occurrences of p.

For each occurrence $(p, i, a) \in \mathrm{Att}_{\text{out}}(p)$ of an attribute a, a semantic function $\varphi_{(p,i,a)} \in \Phi$ must be defined in such a way that $\varphi_{(p,i,a)}$ only depends on the local attributes $\mathrm{Att}(p)$ of p. Let $D(\varphi_{(p,i,a)}) = \{(q_1, i_1, a_1), \ldots, (q_m, i_m, a_m)\} \subseteq \mathrm{Att}(p)$ with $m \in \mathbb{N}$ be the set of local attributes that $\varphi_{(p,i,a)}$ depends on. The signature of $\varphi_{(p,i,a)}$ is $\Omega(a_1) \times \ldots \times \Omega(a_m) \rightarrow \Omega(a)$. $\diamond$

Figure 3.17 shows a tree representation of a production $p = p_0 \rightarrow p_1 \ldots p_{np}$ with the respective sets of attribute occurrences. The tree's root is labeled with the left-hand side nonterminal p_0 of p while its leaf nodes correspond to the right-hand side of p. If not stated otherwise, in Figure 3.17 and in the remainder of this work, the following convention is used: inherited attributes are drawn on the left of a node while synthesized attributes can be found on the right of it. Hence, $\mathrm{Att}_{\text{in}}(p) = \{(p, 0, a_0^0), (p, 1, a_1^2), \ldots, (p, i, a_i^0), \ldots, (p, np, a_{np}^1), (p, np, a_{np}^2)\}$ and $\mathrm{Att}_{\text{out}}(p) = \{(p, 0, a_0^1), (p, 1, a_1^0), (p, 1, a_1^1), \ldots, (p, np, a_{np}^0)\}$.

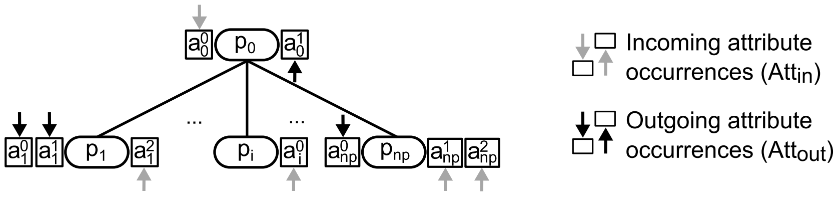


Figure 3.17.: Attributed tree representation of some production p.

Production trees T_p of a production p as shown in Figure 3.17 can be considered as basic building blocks of *attributed syntax trees*. Copies thereof may occur multiple times in different *contexts*. For example, reconsider the syntax tree in Figure 3.3 which corresponds to a sentence generated by the EBNF grammar in Example 3.2. The tree of the second alternative of the production `Or ::= ...| And` occurs in three different contexts in the syntax tree (nodes v_1, v_6, v_{10}). Let $T = (V, E, \mathrm{Lab}, \mathcal{L}_\Sigma)$ be a syntax tree generated by $\Rrightarrow_G$ from an attributed CFG G. Each occurrence of $T_p = p_0[p_1, \ldots, p_{np}]$ in T is called an instance of p in T. Each instance is associated with the set $\mathrm{Att}(p)$ and the corresponding semantic functions. Moreover, each node $v \in V$ of T holds a set of *attribute instances* $\mathrm{AttI}_x(v) = \{(v, a) \mid a \in \mathrm{Att}_x(\mathrm{Lab}(v))\}$. An AG defines the values of these instances.

The semantic functions $\varphi_{(p,i,a)} \in \Phi$ induce attribute dependencies. For each production p, a corresponding directed local dependency graph D_p can be derived whose nodes are the attribute occurrences in $\mathrm{Att}(p)$ and whose edges represent the attribute dependencies induced by the semantic functions associated with the outgoing occurrences $\mathrm{Att}_{\mathrm{out}}(p)$ of p. A full example of an AG will be discussed in Section 3.3.5.

Based on an AG's local dependencies, a circularity test for attributes that transitively depend on themselves can be conducted by composing the local dependency graphs of the underlying CFG with each other (cf. [Knuth 1968; Knuth 1971]). In classic AG systems, cyclic dependencies are typically considered harmful since circular attributes do not have a general semantics. However, modern AG systems usually support circularity to gain expressiveness (cf. Section 3.3.3).

3.3.2. Evaluation of Attributed Syntax Trees

Attribute instances are evaluated by *attribute evaluators*—tools that can be generated from or interpret a given AG specification. Traversing a syntax tree, attribute evaluators compute the values of attributes using the semantic functions $\varphi_{(p,i,a)} \in \Phi$ and the attribute dependencies induced by these functions. Out of two different perspectives on these dependencies, two classes of attribute grammar systems have emerged. *Static evaluators* use precomputed evaluation orders to provide space and time-efficient computation boundaries. Their implementation is usually based on generic AST walkers that can be parametrized with an evaluation order. Like the circularity test, the order-computation algorithms use the local dependency graphs D_p of productions p. Absolutely non-circular attribute grammars (ANCAGs) [Kennedy and Warren 1976] rely on the computation of partial orders over attribute occurrences for a subclass of acyclic AGs. At evaluation time, the partial order is mapped to attribute instances and composed with additional dependency information which is only available from contexts in a syntax tree. Using the precomputed information, the number of visits of nodes and redundant reevaluation of attribute instances is reduced drastically in comparison with an uninformed evaluation. Ordered attribute grammars (OAGs) [Kastens 1980] rely on the computation of total orders over attribute occurrences, which results in a total order of attribute-instance visits at evaluation time and an optimal evaluation of attributes. Although this goes on expressiveness (OAGs are a subclass of ANCAGs), OAGs have been considered a sufficient class to implement semantic analysis of most common programming languages for a long time.

In contrast, *demand-driven* (or *dynamic*) attribute-evaluation strategies have been considered to be too inefficient at the same time as they are inherently nondeterministic w.r.t. evaluation orders which remains implicitly encoded in the semantic functions. However, they still have some advantages. While standard static evaluators always evaluate complete trees, demand-driven evaluation is invoked when an attribute value is actually requested somewhere in the tree and only computes the values of the attribute instances which the requested attribute instance depends on. As it does not depend on a statically precomputed evaluation order, dynamic evaluation is a suitable strategy for evaluating arbitrary acyclic AGs. Furthermore, performance optimizations are possible using caching mechanisms (also called *lazy evaluation*). With caching, a value is only computed once and afterwards stored at the attribute instance, and returned each time the attribute is requested again. Of course, caching increases the memory footprint of the AG. However, on a modern machine this is typically acceptable.

3.3.3. Attribute Grammar Extensions

AGs have been applied to create implementations of static semantics for various kinds of languages [Paakki 1995]. This includes standard programming languages such as Java [Ekman and Hedin 2007b] as well as modeling languages [Heidenreich et al. 2013]. Because of this broad range of use cases during the last decades, several extensions to the basic AG formalism by Knuth have emerged—most of them with practical intentions.

Higher-order attribute grammars (HOAGs) [Vogt et al. 1989] permit attributes that have syntax trees as values. Hence, instead of computing primitive values according to the host of the AG system only—as originally suggested by [Knuth 1968]—special attributes that are marked as *nonterminal attributes* can compute new trees according to the underlying CFG. This is useful to make implicit information in the syntax tree (e.g., computed by attributes or in terminals) explicit as a subtree to ease further computations. Object-oriented AG systems allow for more flexible and intuitive AST declarations [Hedin 1989; Grosch 1990; Sloane et al. 2010] by using typical features of object-oriented programming languages. Although object orientation does not influence the basic formalism, object-oriented AGs have a huge impact on the implementation and attribute specification style. Using classes for typing ASTs gives a better integration with the host language of the AG system and proved more flexible than plain AG systems. For instance, the system can use abstract classes and object-oriented dispatching for attribute look-up. Circular AGs [Farrow 1986] enable the evaluation of cyclic attribute grammars based on *fixpoint* computation [Tarski 1955]. Circularity allows using AG tools for also implementing recursive static analysis algorithms such as *control-flow analysis* [Nilsson-Nyman et al. 2009].

Two other important extensions are concerned with attribute access at distant nodes. Remote AGs [Boyland 2005] enable access to attribute instances at distant nodes from a local context. RAGs [Hedin 2000] allow to pass references to AST nodes through attribute instances and to access attributes of that node in different contexts. Using reference attributes and object-oriented ASTs, several analysis algorithms (e.g., *name analysis*) can be specified more elegantly in comparison to classic AGs. Collection attributes [Magnusson 2007] are a declarative notation for specifying attributes that aggregate values on a syntax tree. For example, they can be used for collecting a set of nodes with certain properties.

However, the extensions above are not covered by the original AG algorithms and thus cannot (or only with strong restrictions) be implemented using static dependency graphs and precomputed attribute evaluation orders of the static algorithms. Consequently, many of the current AG systems that support features like reference and collection attributes rely on *demand-driven* strategies which is currently the most established implementation strategy for RAGs. Prominent examples are Silver [Wyk et al. 2008], JastAdd [Ekman and Hedin 2007a] and Kiama [Sloane 2011]. One of the most important properties of these systems is their support for modular specifications, which enables language developers to separate concerns like name or type analysis and to create extensible language processors [Dueck and Cormack 1990; Ekman 2006; Wyk et al. 2007].

Moreover, the RAG concepts can be mapped to standardized metamodeling languages such as EMOF [Bürger and Karol 2010; Bürger et al. 2011] to compute references and to perform static-semantics computations on model instances. This is possible because EMOF models typically provide a unique spanning tree, which is specified by EMOF's containment relation, which is similar to flat EBNF grammars for abstract syntax (cf. Section 3.2.2). The JastEMF tool [Software Technology Group 2013] implements JastAdd's RAGs for the EMF and Java-based EMOF metamodels. Hence, technically, the RAG-based approach to ISC developed in this thesis can also be applied to modeling languages.

3.3.4. Attribute Grammars and Tree Manipulations

Besides the classic field of application of AGs—compiler construction—there are also works applying them in more interactive scenarios like editor implementation for IDEs. In an IDE where a file is edited interactively, syntactic and semantic analysis algorithms running in the background need to be highly responsive. Hence, parsers used in such editors are usually *incremental* and thus avoid a complete parse and AST reconstruction. Since edit operations are mostly local changes in a file, changes to the underlying AST are often minimal. Consequently, not all attribute values of the AST have to be recomputed, but only those in newly added parts and those which depended on values that have been deleted or moved.

Incremental AG evaluators [Reps et al. 1983; Maddox III 1997] support such scenarios by employing dependency graphs to reduce the recomputation effort. [Söderberg 2012] uses dynamic dependency tracking to approximate the set of attribute caches of an RAG that have to be flushed when the AST is edited.

Besides editing, ASTs can also be manipulated operationally using *rewrites*. *Rewritable reference attribute grammars (ReRAGs)* [Ekman and Hedin 2004]—as supported by JastAdd—rewrite the AST interlaced with attribute evaluation. Like edits, rewrites can interfere with cached attribute values and may cause inconsistent states (i.e., attributions) of the AST. A dependency graph available at runtime helps to discover such inconsistencies. The *reference attribute grammar controlled rewriting (RACR)* approach [Bürger 2012] provides sophisticated dependency-graph management with a nearly optimal approximation of which cached attribute values have to be flushed and recomputed.

This thesis combines RAGs with *ISC-based composition operators*. Composition operators are considered as special kinds of AST rewrites which are restricted according to a given *component model* (cf. Section 4.1.2).

3.3.5. SimpAG: A Simple RAG Language

In Chapters 5 and 7, a simple attribute grammar specification language will be used to specify RAGs-based component models. The language is called *Simple Attribute Grammar Specification*

Language (SimpAG) and is introduced subsequently. A SimpAG specification always consists of a flat EBNF grammar without grouping constructs, a list of attribute declarations and equations with semantic functions computing the attribute values.

EBNF Interpretation and Restrictions

The grammar language of SimpAG corresponds to the EBNF dialect given by Definitions 3.4 and 3.14. Additionally, nonterminal inheritance and abstract nonterminals are supported in correspondence with Definition 3.16 of Section 3.2.3. To exemplify the usage of SimpAG, the *Logic "Programming" Language (LogProg)* is developed as an example in this section. Its EBNF grammar is shown in Example 3.13 below, comparing its compact and desugared forms.

Example 3.13 (LogProg EBNF grammar and its desugared version).
Consider the simple LogProg program in Listing 3.1 which allows its users to write lists of propositional logic formulas.

```
main {
    a = t;
    b = a | f;
    result = b;
 }
```

Listing 3.1: A simple LogProg program.

The syntax of LogProg is defined by the following EBNF grammar G_{Log}:

```
Program ::= "main" "{" Stmt+ "}"
Stmt ::= Name "=" Expr ";"?
Expr ::= Or
Or ::= left:Or "|" right:And | And
And ::= left:And "&" right:Term | Term
Term ::= "t" | "f" | Name | "!" Name
Name ::= id:<ident>
```

Observe that `ident` tokens are strings of $\{a \ldots z, A \ldots Z, \_, ., 0 \ldots 9\}^+$. By applying the EBNF desugaring transformations described in Section 3.1.2, the following grammar G'_{Log} is obtained:

```
Program ::= "main" "{" StmtList "}"
StmtList₁ ::= Stmt
StmtList₂ ::= Stmt StmtList
Stmt ::= Name "=" Expr SemOpt
SemOpt₁ ::= ϵ
SemOpt₂ ::= ";"
```

```
Expr ::= Or
Or₁ ::= left:Or "|" right:And
Or₂ ::= And
And₁ ::= left:And "&" right:Term
And₂ ::= Term
Term₁ ::= "t"
Term₂ ::= "f"
Term₃ ::= Name
Term₄ ::= "!" Name
Name ::= ident:<ident>
```

The subscript indexes are used in SimpAG to address the alternatives as different contexts that can be implied by a nonterminal. Since no special EBNF constructs remain in G'_{Log} (except keywords and labels), an attribution for LogProg can be specified. ◇

Declarations and Equations

Inspired by JastAdd, SimpAG provides a simple syntax for attribute declarations while pseudo code mixed with mathematical expressions is used in the equations. Hence, SimpAG is independent of a specific implementation language so that SimpAG RAGs are specified as a blueprint to be ported to arbitrary real-world AG tools that support SimpAG's features.

For basic attribute declarations, SimpAG has the following syntax:

$$\mathbf{syn}\ \texttt{[type] [nt].[attribute]} \tag{3.1}$$
$$\mathbf{inh}\ \texttt{[type] [nt].[attribute]} \tag{3.2}$$

Attribute declarations either start with the keyword **syn** for synthesized attributes (cf. Declaration 3.1) or with **inh** for inherited attributes (cf. Declaration 3.2). `attribute` stands for the declared attribute's name as an arbitrary identifier string. `type` represents the type of the value computed by the attribute. The types supported by SimpAG are discussed later in this section. Finally, `nt` denotes a nonterminal of the AG's underlying context-free grammar.

As a short-hand notation to declare the same attribute w.r.t. multiple nonterminals at once, set notation can be used as shown in Declarations 3.3 and 3.4 below:

$$\mathbf{syn}\ \texttt{[type]}\ \{\texttt{[nt]},\ldots,\texttt{[nt]}\}.\texttt{[attribute]} \tag{3.3}$$
$$\mathbf{inh}\ \texttt{[type]}\ \{\texttt{[nt]},\ldots,\texttt{[nt]}\}.\texttt{[attribute]} \tag{3.4}$$

Semantic functions are generally specified using the keyword **fun**, a left-hand side to associate the equation with an attribute occurrence and context, and a right-hand side where pseudo code provides an "implementation". Equation 3.5 below shows the syntax of synthesized equations in SimpAG:

$$\mathbf{fun}\ \texttt{[nt]}_{\texttt{[idx]}}.\texttt{[attribute]} = \texttt{[}\textit{pseudo code}\texttt{]} \tag{3.5}$$

Synthesized equations are associated with a nonterminal `nt` and an `attribute` the equation is provided for. Optionally, the nonterminal can be associated with an index `idx` if a particular production/alternative of `nt` is addressed. The right-hand side is separated by an equation symbol and is typically provided by using common mathematical expressions mixed with *`pseudo code`*. It must be "compatible" with the attribute's type given by its declaration. For example, if the type in the declaration is `X`, the implementation has to generate a value of type `X`. In case `nt` is defined by only one alternative (i.e., a single production in the desugared grammar), attribute declaration and equation may converge to a single declaration with a right-hand side equation.

Equations of inherited attributes have a similar but more complex syntax. Since inherited attributes must be defined w.r.t. to an AST node's parental context, SimpAG supports two equation variants of specifying them as shown in Equation 3.6 and Equation 3.7 below:

$$\textbf{fun}\ \texttt{[nt]}_{\texttt{[idx]}}\texttt{.child}_{\texttt{[idx]}}\texttt{.[attribute]} = \texttt{[}\textit{pseudo code}\texttt{]} \tag{3.6}$$

$$\textbf{fun}\ \texttt{[nt]}_{\texttt{[idx]}}\texttt{.[label]}_{\texttt{[idx]}}\texttt{.[attribute]} = \texttt{[}\textit{pseudo code}\texttt{]} \tag{3.7}$$

Using $\texttt{child}_{\texttt{[idx]}}$, an equation for `attribute` in context of the nonterminal at index `idx` of `nt`'s right-hand side nonterminals is defined. If an attribute is declared at *all* of `nt`'s children, $\texttt{child}_{\texttt{all}}$ defines the equation for all of these contexts. Alternatively, the child nonterminal can be described via its `label` and, if the label is used more than once, with an index `idx` in such a way that these uses can be distinguished in order of their occurrence in the respective `nt` production.

Standard inherited equations hold for a single parent-child context. However, inherited attributes are often just passed down the tree to a specific node without transforming the value in between. Hence, inherited equations would need to be chained, which bloats the AG specification with no-op equations. To avoid this, SimpAG supports broadcasting equations as shown in Equations 3.8 and 3.9 below:

$$\textbf{fun}\downarrow \texttt{[nt]}_{\texttt{[idx]}}\texttt{.child}_{\texttt{[idx]}}\texttt{.[attribute]} = \texttt{[}\textit{pseudo code}\texttt{]} \tag{3.8}$$

$$\textbf{fun}\downarrow \texttt{[nt]}_{\texttt{[idx]}}\texttt{.[label].[attribute]} = \texttt{[}\textit{pseudo code}\texttt{]} \tag{3.9}$$

The semantics of broadcasting is as follows: if a broadcasting equation is given for a specific context c_1, its value is passed down to any instance of the same attribute reachable in the context's subtree which is *not* provided with its own context-dependent equation. If an instance of the same attribute in c_1's subtree has its own inherited equation, this value is used for the local context c_2 while the value of c_1 is sill broadcasted down in c_2's subtree. In contrast, if c_2 also is a broadcasting equation then c_2's value is broadcasted down its subtree.

Types and Values

SimpAG supports a set of types which can be further distinguished in three subsets. *Value types* are primitive types like Integer, Boolean and String. *Structure types* are lists and maps. *Complex types* are nodes typed by the nonterminals of the AST grammar. The following enumeration explains SimpAG's types in more detail:

- SimpAG supports a set of primitive types:
  - `bool`$_\perp$ is a 3-valued type representing logical values. *true* represents the Boolean value “true”, *false* represents the Boolean value “false” and $\perp$ is the undefined value,
  - `int`$_\perp$ denotes integers $\{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$ and $\perp$,
  - `string`$_\perp$ is the set of strings Σ^* over the set of SimpAG characters (or alphabet) Σ. As a simplification, set of characters supported by the Unicode Standard [The Unicode Consortium 2014] is assumed. $\perp$ is again the undefined value.

  For primitive types, value semantics is assumed by default.
- Moreover, the following structures are supported:
  - key–value pairs in SimpAG are surrounded by angle brackets and are declared as follows: `<[key type],[value type]>`. <*key*,*value*> denotes a pair instance with a *key* and a *value*.
  - Lists are typed and marked up with “$\star$”: `[entry type]`$\star$. Within semantic functions, lists are declared using square brackets: `[`$value_1, \ldots, value_n$`]`.
  - Maps: by combing list and pair types, maps can be easily declared as lists of pairs. Hence, `<[key type],[value type]>`$\star$ declares a map type. Consequently, in semantic functions, maps are written as lists:
    `[`<key_1,$value_1$>$, \ldots,$ <key_n,$value_n$>`]`. Each key in a map is unique so that there are never two pairs with the same key. Moreover, if a new pair is added to a map and a pair with the same key is already in the map, the old pair is removed and the new pair is added to the map.
- Complex types: AST nodes are the only complex types in SimpAG, where type names and structure definition are specified by the set of nonterminals N and productions of the underlying CFG. Hence, if `X` is the type signature used in an attribute declaration and `X` is not a primitive type or structure then `X` $\in N$. Typically, a node computed by an equation is a *reference* to an already existing node in the AST. Hence, by default complex types have a *reference semantics* where the AST is the corresponding “heap”. However, in rare cases it is necessary to construct a fresh node and subtree in an equation. Then, *value semantics* is used locally in such a way that the current attribute instance becomes the “address” of the fresh node (i.e., the root of a new subtree). Observe that complex-type attributes are called *reference attributes* if they have a reference semantics and *nonterminal attributes* if they have a value semantics.

An Example Attribution of Logic Programs

Example 3.14 shows how SimpAG can be used to specify an attribute semantics evaluator to compute values of programs or, broader, ASTs. Therefore, the LogProg language developed in Example 3.13 is reused and improved.

Example 3.14 (SimpAG attribution).
In the following, a SimpAG attribute grammar AG_{Log} is developed that specifies how LogProg expressions are to be evaluated (i.e., how values of primitive logic terms in LogProg are computed). AG_{Log} declares four attributes: `val` holds the computed values of LogProg compartments like `Terms` and `Programs`, `env` helps to look up declared variables and `en` and `ident` provide the basic values at leaf nodes. As AGs inherently provide a natural way of modularizing semantic concerns [Dueck and Cormack 1990], these attributes allocate two concerns—namely *evaluation* including the `val` attribute and *name analysis* including `env`, `en`, and `ident`. In the following, AG_{Log} is specified along these two concerns.
Evaluation: Below, `val` is declared as an attribute of type $\texttt{bool}_\perp$:

$$\textbf{syn}\ \texttt{bool}_\perp\ \{\texttt{Program,StmtList,Stmt,Expr,Or,And,Term}\}\texttt{.val}$$

With an exception of `SemOpt` and `Name`, `val` is declared for all nonterminals of G_{Log} using SimpAG's set-based short-hand notation.
Next, the four `Term` contexts are specified:

$$\textbf{fun}\ \texttt{Term}_1\texttt{.val} = \mathit{true}$$
$$\textbf{fun}\ \texttt{Term}_2\texttt{.val} = \mathit{false}$$
$$\textbf{fun}\ \texttt{Term}_3\texttt{.val} = \begin{cases} \mathit{envVal} & \textbf{if}\ \texttt{<child}_1\texttt{.id,}\mathit{envVal}\texttt{>} \in \texttt{env}, \\ \perp & \textbf{else}. \end{cases}$$
$$\textbf{fun}\ \texttt{Term}_4\texttt{.val} = \begin{cases} \neg\mathit{envVal} & \textbf{if}\ \texttt{<child}_1\texttt{.id,}\mathit{envVal}\texttt{>} \in \texttt{env}, \\ \perp & \textbf{else}. \end{cases}$$

While $\texttt{Term}_1$ and $\texttt{Term}_2$ just represent the atomic Boolean values, the `val` specification of the $\texttt{Term}_3$ alternative retrieves the value from the first child $\texttt{child}_1\texttt{.ident}$ and performs a look-up by using the `env` attribute provided by the *name analysis* parts of G_{Log}. In the case the look-up fails, the attribute evaluates to $\perp$ (i.e., invalid). $\texttt{Term}_4$ is similar to the evaluation of $\texttt{Term}_3$, but negates the value obtained from the environment.
The following equations define the evaluation rules of LogProg expressions:

$$\begin{aligned}
\textbf{fun}\ \texttt{And}_1\texttt{.val} &= \begin{cases} \perp & \textbf{if}\ \texttt{left.val} = \perp, \\ \perp & \textbf{if}\ \texttt{right.val} = \perp, \\ \texttt{child}_1\texttt{.val} \wedge \texttt{child}_3\texttt{.val} & \textbf{else}. \end{cases} \\
\textbf{fun}\ \texttt{And}_2\texttt{.val} &= \texttt{child}_1\texttt{.val} \\
\textbf{fun}\ \texttt{Or}_1\texttt{.val} &= \begin{cases} \perp & \textbf{if}\ \texttt{left.val} = \perp, \\ \perp & \textbf{if}\ \texttt{right.val} = \perp, \\ \texttt{child}_1\texttt{.val} \vee \texttt{child}_3\texttt{.val} & \textbf{else}. \end{cases} \\
\textbf{fun}\ \texttt{Or}_2\texttt{.val} &= \texttt{child}_1\texttt{.val} \\
\textbf{fun}\ \texttt{Expression.val} &= \texttt{child}_1\texttt{.val}
\end{aligned}$$

$$
\begin{aligned}
&\textbf{fun}\ \texttt{Stmt.val} &&= \texttt{child}_3\texttt{.val}\\
&\textbf{fun}\ \texttt{StmtList}_1\texttt{.val} &&= \texttt{child}_1\texttt{.val}\\
&\textbf{fun}\ \texttt{StmtList}_2\texttt{.val} &&= \texttt{child}_2\texttt{.val}\\
&\textbf{fun}\ \texttt{Program.val} &&= \texttt{child}_3\texttt{.val}
\end{aligned}
$$

The `And` and `Or` contexts are simply realized by the corresponding logical operators with a certain distinction. Since $\texttt{Term}_3$ may evaluate $\perp$, this has to be handled by the evaluator as error cases. Therefore, $\vee$ and $\wedge$ evaluate to $\perp$ in case one of their arguments is $\perp$. The value of a `Stmt` is determined by the value of its `Expr` child. The value of the whole LogProg program is given by the value of the last `Stmt` in the program which is just passed trough the AST to the `Program` root.

Name analysis: Below, `en` and `id` are declared and specified:

$$
\begin{aligned}
&\textbf{syn}\ \texttt{<string}_\perp\texttt{,bool}_\perp\texttt{>}\ \texttt{Stmt.en} = \texttt{<child}_1\texttt{.id,child}_3\texttt{.val>}\\
&\textbf{syn}\ \texttt{string}_\perp\ \texttt{Name.id} \qquad\qquad = \texttt{ident}
\end{aligned}
$$

The `en` and `id` attributes given above initialize the atomic values at the leaf nodes in LogProg ASTs. The value of `id` in the `Name` context is derived from the actual identifier terminal `ident`. Based on that, the `en` attribute generates a pair at each `Stmt` node with the identifier string of its left-hand side as a key and evaluated expression of the right-hand side as a value.

Next, `env` is declared as an inherited attribute which is associated with all nonterminals of G_{Log} (G'_{Log}) except `Program`, `SemOpt` and `Name`:

$$\textbf{inh}\ \texttt{<string}_\perp\texttt{,bool}_\perp\texttt{>}\star\ \{\texttt{StmtList,Stmt,Expr,Or,And,Term}\}\texttt{.env}$$

As specified below, the `env` attribute distributes a map as a look-up table through LogProg trees:

$$
\begin{aligned}
&\textbf{fun}\downarrow\texttt{Program.child}_3\texttt{.env} &&= \texttt{[]}\\
&\textbf{fun}\downarrow\texttt{StmtList}_2\texttt{.child}_2\texttt{.env} &&= \texttt{[child}_1\texttt{.en]} \sqcup \texttt{env}\\
&\textbf{fun}\ \texttt{StmtList}_1\texttt{.child}_1\texttt{.env} &&= \texttt{env}\\
&\textbf{fun}\ \texttt{StmtList}_2\texttt{.child}_1\texttt{.env} &&= \texttt{env}\\
&\textbf{fun}\ \texttt{Stmt.child}_3\texttt{.env} &&= \texttt{env}\\
&\textbf{fun}\ \texttt{Expr.child}_1\texttt{.env} &&= \texttt{env}\\
&\textbf{fun}\ \texttt{Or}_1\texttt{.child}_1\texttt{.env} &&= \texttt{env}\\
&\textbf{fun}\ \texttt{Or}_1\texttt{.child}_3\texttt{.env} &&= \texttt{env}\\
&\textbf{fun}\ \texttt{Or}_2\texttt{.child}_1\texttt{.env} &&= \texttt{env}\\
&\textbf{fun}\ \texttt{And}_1\texttt{.child}_1\texttt{.env} &&= \texttt{env}\\
&\textbf{fun}\ \texttt{And}_1\texttt{.child}_3\texttt{.env} &&= \texttt{env}
\end{aligned}
$$

$$\textbf{fun}\ \texttt{And}_2\texttt{.child}_1\texttt{.env} = \texttt{env}$$

As an inherited attribute, `env` is initialized within the `Program` top level context and then passed through the list of `Stmts` provided by the chained `StmtList` list nodes. In context of `StmtList`$_2$'s second child, the value of the left `Stmt` is added to the environment using the $\sqcup$ append operator and the synthesized `en` attribute already described above. This realizes a simple *declare-before-use* semantics to variables in LogProg, i.e., a logic variable has to be declared before it is used in another expression. The remaining equations of `env` occurrences are only for passing down the environment through the AST to provide the name look-up functionality where it is needed—in `Term` contexts. These semantic functions can also be omitted from the specification since they can be derived automatically by AG systems as the first two equations are broadcasting equations.

Listing 3.1 of Example 3.13 shows a small LogProg program with three statements. A corresponding attributed syntax tree is shown in Figure 3.18, including data-flow edges. The statement list of the program contains three statements, which is reflected by three `Stmt` equivalent subtrees in Figure 3.18. Starting with an empty environment, the `Expr` statement in Line 2 corresponds to the leftmost `Stmt` subtree and evaluates $true$. A suitable entry for the environment is created at this node. Line 3 contains an `Or` expression corresponding to the central `Stmt` subtree. The AG evaluator looks up variable `a` in the environment using the `env` attribute. Since `a` evaluated to $true$ above, `b` also evaluates to the same value and its entry is added to the environment. The last statement in Line 4 assigns `result` the value of `b`, i.e., $true$. According to the AG specification, the value of `result` is propagated to the root node of the attributed AST and can be emitted to a client as the actual result of the program. $\diamond$

Observe that the above example is a “classic” AG in Knuth’ sense: it does not contain any reference or nonterminal attributes and has an obvious static-computable evaluation order since the attribute dependencies are acyclic. If reference attributes are used as they are provided by the JastAdd system which is used in the later chapters, name analysis could also be realized via passing references to `Stmt` nodes as “declaration objects” through the AST and performing a *demand-driven* evaluation of LogProg programs.

3.4. Summary and Conclusions

This chapter provided the background terminology, concepts as well as notation required to understand the approaches reevaluated and developed by this thesis. Trees and grammars are the basis of fragment composition and ISC since any fragment component essentially has a tree typed by some context-free grammar in its core. Besides parsers, many other tools use context-free grammars or related tree-definition approaches to define tree-based data structures including important industrial standards such as XML, XML schema and EMOF. Hence, besides plain text-based languages such as programming languages, XML files, models with graphical representation and other tree-based languages are potential application areas of ISC.

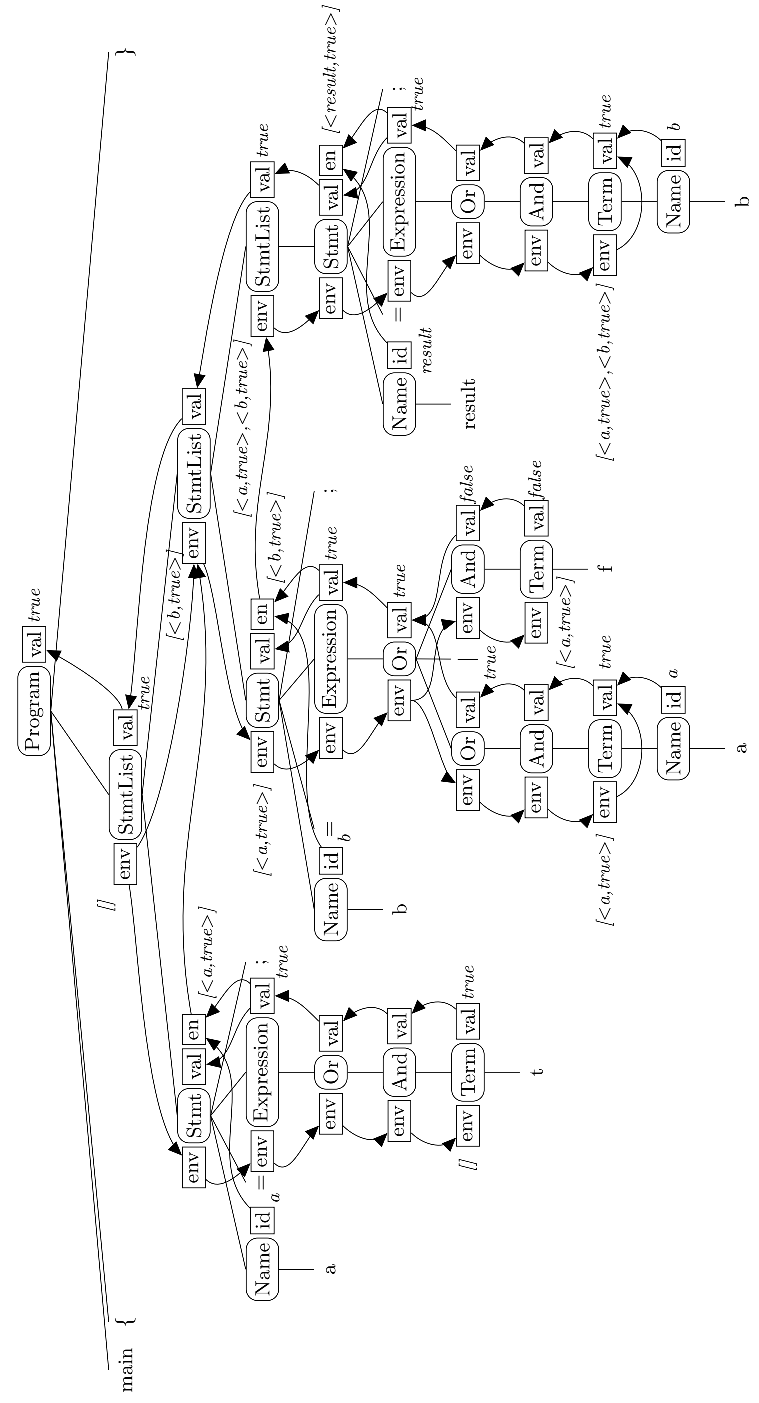


Figure 3.18.: Attributed syntax tree of the logic program in Listing 3.1.

However, as the main invention of this thesis is *well-formed ISC*, trees and CFGs are necessary but insufficient concepts w.r.t. context-sensitive computations on trees. RAGs support such computations via attributes, equations and reference attributes that induce a directed graph on trees. Consequently, RAGs have been chosen as the main specification language of well-formed ISC. Moreover, it has been discussed that RAGs are extensible and naturally modularized along semantic concerns. Thus, RAGs are also the basics of the second invention of this thesis: *scalable ISC*.

Before well-formed and scalable ISC are introduced, Chapter 4 recapitulates the classical model of ISC and discusses two implementation approaches—COMPOST and Reusew*air*—and compares them. Moreover, a modified model of ISC for graphs and its implementation—Reuseware—is discussed and evaluated.

4 Approaches to Invasive Software Composition

"We must do more than create new techniques. We must understand the old ones" [Potts 1993].

In the first part of this chapter, the basic constituents of composition systems in general and invasive composition systems in particular are discussed. While the basic terminology stems from [Aßmann 2003], a formal *fragment component model* definition is developed that serves as a comparative framework throughout the chapter. Afterwards, three approaches to ISC are presented and analyzed by applying them to implement the *business application framework (BAF)* example of Chapter 2.

4.1. Invasive Fragment Composition Systems

In this section, a generic model of ISC is developed. To classify ISC among general approaches to component-based software engineering, it at first follows a general classification scheme for composition systems. Afterwards, the concepts of ISC w.r.t. this scheme are presented. Finally, a formal definition of fragment component models based on grammars and syntax trees is given.

4.1.1. Ingredients of Composition Systems

Component abstractions for the separation of cross-cutting concerns (SoCC), hierarchical decomposition (HiDec) and template metaprogramming (TMP) have to be supported by composition

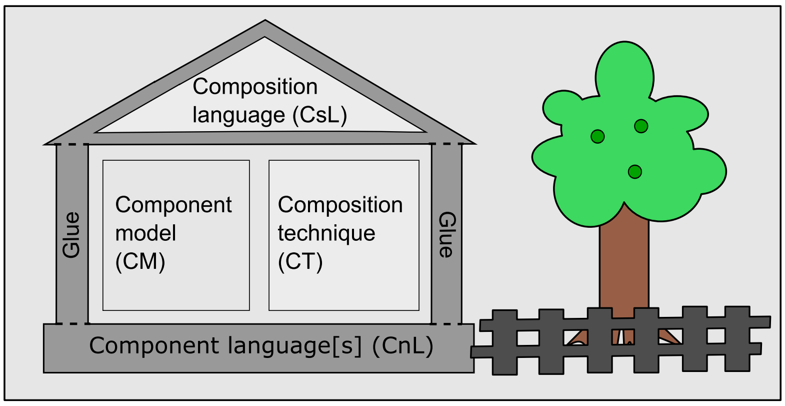

Composition language (CsL)
Glue
Component model (CM)
Composition technique (CT)
Glue
Component language[s] (CnL)

Figure 4.1.: The house of software composition and the precious garden of software.

systems. Although composition systems can be quite different in purpose, a general terminology can be used to describe their compartments: Figure 4.1 symbolizes a general architectural schema on the ingredients of composition systems—the *house of software composition*. The basic notions stem from [Nierstrasz and Meijler 1995; Aßmann 2003]. In the following, the basic ingredients—compartments of the house—are described.

Component model (CM). A CM specifies look and shape of components. It determines how components can be accessed, how they are represented (e.g., as source code or compiled, in binary form) and provides abstractions for reuse. Typically, it can be distinguished between *black-box*, *white-box* and *gray-box* CMs. The former hides a component's implementation details behind a well-defined, usually typed and well-documented interface. This is also called the principle of *information hiding* [Szyperski 2002]. White-box models are at the other end of the scale: white-box composition requires deep knowledge of a component's internals and protocols. Gray-box models share properties of both—providing a well-defined interface *and* revealing implementation details [Aßmann 2003].

Composition technique (CT). The CT determines how components are composed, e.g., by providing a composition formalism, patterns or a standardized protocol. It also determines how components communicate. The CT provides basic composition operators that can be used for composition.

Composition language (CsL). Components can be put together by writing recipes in an adequate CsL. This can be, for example, a standard programming language, a programming-language extension or a standalone CsL. Expressiveness of such languages is an important issue. It should be powerful enough to express all the composition scenarios supported by the CM. A well-designed CsL enables efficient specifications and is easy to learn.

Component language (CnL). This is the actual language used to develop component implementations. Depending on the CM, the CnL might be completely invisible to the developer who uses the composition system. For example, if it is a black-box system, the CnL might be hidden behind a language-independent interface. In contrast, *white-box* approaches require detailed knowledge about the CnL and its semantics. Hence, in the latter case, the CM is strongly coupled with the CnL while in the former case the coupling can be loose.

Glue. The *glue* integrates the constituents of composition systems. It relates CM and CT with CnL and CsL. In white-box approaches, it is common to glue concepts together by language-model integration. That is, the CM is materialized as a formal or semi-formal specification, e.g., using a grammar or metamodel, and meta-composed with the CnL and CsL. This is different in black-box systems. Since components are communicating through well-defined interfaces and protocols, the glue materializes as connectors.

Example 4.1 explains the differences between white and black-box composition. Afterwards, in the next section, the terminology given above is used to introduce the basic model of invasive software composition.

Example 4.1 (black-box vs. white-box composition systems).
The Common Object Request Broker Architecture (CORBA) is a well-known composition system with binary black-box components [Object Management Group (OMG) 2012b]. The component model of CORBA is specified semi-formally in [Object Management Group (OMG) 2012a] and has a number of implementations in common programming languages such as Java and C++. Component interfaces in CORBA have to be specified in a language-independent way using the also standardized Interface Description Language (IDL). Hence, a component in CORBA is a "specific, named collection of features that can be described by an IDL component definition or a corresponding structure in an Interface Repository" [Object Management Group (OMG) 2012a, p. 5]. The component implementation can be realized in any programming language (i.e., CnL) that provides a CORBA library implementation. The *glue* between IDL and CnL is provided by programming-language-specific component-model implementations which—amongst other responsibilities—can generate CnL specific code from IDL specifications. Since CORBA components are language- and location-independent, they are wired together transparently by *connectors*. A connector converts data types and handles component invocations using adequate lower-level standards, e.g., remote method invocation (RMI).

In contrast to CORBA, the already-mentioned aspect-oriented programming (AOP) is a white-box composition approach. AOP is *asymmetric*, that is, a set of self-defined *core* components is composed with incomplete *aspects* [Kiczales et al. 1997] augmenting the core with additional program code. This is specified by advice functions [Teitelman 1966]. In contrast to systems like CORBA, AOP implementations are typically tightly integrated with a concrete programming language (cf. [Kiczales et al. 2001; Aracic et al. 2006]). The glue between CnL and CM is realized via language extension. The process of composing aspects and core is called *weaving*.◇

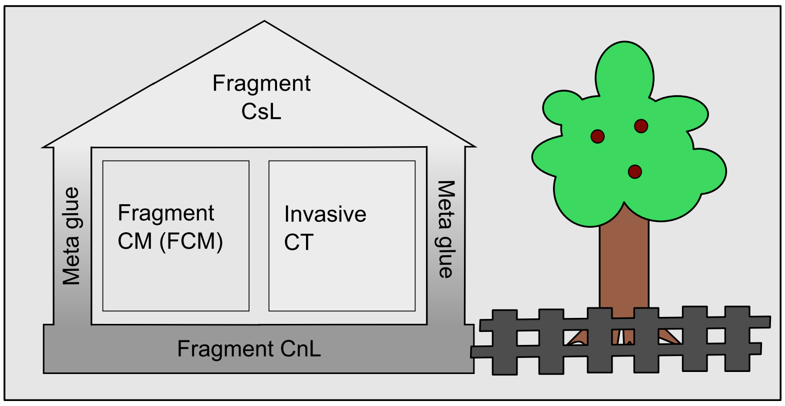


Figure 4.2.: The house of invasive software composition.

4.1.2. Ingredients of Invasive Software Composition

> *"Invasive Software Composition (ISC) composes software components by program transformation. Standard composition treats components as immutable black boxes and plugs them together as they are. Invasive software composition goes one step further and transforms components when they are embedded in a reuse context. In this way, components can be adapted more appropriately to reuse requirements. The components need not to be adapted by hand, instead invasive composition operators do all the work" [Aßmann 2003, p. 108].*

ISC is a general approach for composing *fragment components*. The quotation above and the notion of *fragment forms* in the BETA programming language [Kristensen et al. 2007] indicate that fragment components are potentially underspecified pieces of program code, derived from a nonterminal of the programming language that may contain placeholders for variation, and implicit points of extension. The term "program" not only covers full-fledged programming languages, but also DSLs and modeling languages that have a tree-based abstract syntax (cf. Section 3.2.2). Hence, fragment components can be *fragments* of some DSL "program" [Henriksson 2009], an underspecified XML document [Pop et al. 2005], or a model [Heidenreich et al. 2009d; Johannes 2011].

Figure 4.2 contains the *house of invasive software composition* whose compartments are the basic ingredients of ISC described below.

Fragment CnL. The CnL is the very fundament of any ISC system. It defines the upper bounds of what the fragment CM allows for composition. The CnL specification determines the set of potential fragment types by its nonterminals and potential positions of variation and extension points by its structure.

Fragment CM. Typically, only certain concepts of the CnL are adequate fragment component types. Therefore, the *fragment component model (FCM)* defines a subset of the CnL nonterminals as valid fragment types (cf. [Henriksson 2009]). Furthermore, not all parts of a fragment should be freely replaceable or extensible. Therefore, the FCM provides a composition interface that consists of sets of compositional points called *hooks*. [Aßmann 2003] distinguishes between *declared* and *implicit* hooks of a fragment. The essential difference between them is that the former must be declared explicitly by some placeholder while the latter are derived "implicitly" from the fragment structure. There is a difference between *list hooks* and *non-list hooks*. List hooks indicate compositional points with lists of similar objects (e.g., statements) while non-list hooks indicate compositional points with just one single object. Since hooks declared by placeholders are typically non-list hooks, they are called *slots* while list-hooks are called just hooks in this thesis (cf. [Madsen et al. 1993; Henriksson 2009]).

Invasive CT. The ISC model includes a set of basic *composition operators* which can be parametrized with a compositional point and a fragment to perform a *program transformation*. *Bind* transforms a host fragment at a given slot by replacing (i.e., *binding*) it with the copy of a given argument fragment. Hence, after composition, the slot is no longer available in the host fragment. The *extend* operator transforms a host fragment at a hook by extending it with the copy of a given argument fragment. After composition, the hook is still available in the host fragment and can be extended further. A *composer* applies one or more composition operators at one or more points of a fragment. Often, ISC systems provide a collection of *basic* or *primitive composers* that realize a single composition step, e.g., a bind composer provides a corresponding composition operation.[1] Users can instantiate the basic composers to implement *complex composers* and *composition programs*. Alternatively, composition programs are called *composition recipes*.

Fragment CsL. The fragment CsL is provided by ISC systems to support users in writing complex composers and composition programs. The fragment CsL can be any kind of language as long as it may serve the purpose of composition specification and execution. The CsL model is freely chosen by the composition-system developer who should choose composition abstractions that are adequate w.r.t. their specific domain. Composition abstractions supported by the CsL are typically realized as complex composers built into the composition system—i.e., a composition abstraction's semantics manifests as a sequence of composition operator calls at runtime. This way, the CsL can be a simple imperative language, aspect-oriented, a macro language, textual or graphical and is backed by a formal FCM. Moreover, an integration with the CnL is also possible.

[1]Observe that the terms "composition operator" and "composer" are often used interchangeably. This results from the fact that a "composition operator" is an abstract transformational concept to express compositions while a "composer" is its *reusable* implementation in a specific composition system.

Meta glue. Of course, fragment CsL and the fragment CT need to be integrated with the CnL and the FCM. This is indicated in Figure 4.2 by the two *meta glue* pillars. Depending on the actual constitution of the composition system, *composer signatures* are related with concrete fragment types in the CnL. Abstractions and concepts of the CsL have to be realized as complex composers by the CT, including mappings to the responsive parts in the CnL so that the CsL is aware of the syntactic types (nonterminals) of the CnL.

4.1.3. ISC$_{\text{core}}$: The Core of Invasive Software Composition

In the following, a formal definition for FCMs which uses context-free grammars (CFGs) and syntax trees is provided. It contributes to the thesis in multiple ways. It provides a clear distinction of all parts of a fragment CM and allows to use these clearly named parts to identify their realizations in the existing ISC approaches discussed afterwards. Even more important, the definition can be used to compare the ISC approaches developed in this thesis—well-formed ISC and minimal ISC—with the existing ones.

Definition 4.1 (fragment component model):
A fragment component model FCM is a 6-tuple (G,$\mathcal{S}$,$\mathcal{H}$,$\mathcal{F}$,$\int$,$\mathcal{L}_\int$) with

- $G = (N, \Sigma, P, S)$ a CFG (or a flat EBNF grammar) of the CnL,
- $\mathcal{S} \subseteq N$ the finite set of *slot candidates*,
- $\mathcal{H} \subseteq N$ the finite set of *hook candidates*,
- $\mathcal{F} \subseteq N$ the finite set of *fragment-component candidates*,
- $\int = \int_\mathcal{S} \cup \int_\mathcal{H} \cup \int_\mathcal{F}$ is a set of partial *point-identification* functions assigning identifiers to nodes that represent slots ($\int_\mathcal{S}$), hooks ($\int_\mathcal{H}$) and fragments ($\int_\mathcal{F}$) in syntax trees,
- $\mathcal{L}_\int$ a set of *point identifiers* (e.g., defined by some signature-language grammar).

If $n \in \mathcal{H}$, then n has to be a list nonterminal (cf. Definition 3.15 and Section 3.1.5).

Subsequently, let $T = (V, E, \text{Lab}, R, \mathcal{L}_\Sigma) \in \mathcal{T}(\mathcal{L}_\Sigma)$ be a syntax tree w.r.t. $G_{|n} = (N_{|n}, \Sigma_{|n}, P_{|n}, n)$, so that $n \Rightarrow^*_G T$, and $name \in \mathcal{L}_\int$ an identifier. The compounds of $\int$ are defined as follows:

- $\int_\mathcal{S}(T, v) : \mathcal{T}(\mathcal{L}_\Sigma) \times V \rightsquigarrow \mathcal{L}_\int$ so that
$$\int_\mathcal{S}(T, v) = \begin{cases} name & \text{if } \text{Lab}(v) \in \mathcal{S} \text{ and } v \in V \text{ is a \textit{slot} in } T \\ \bot & otherwise \end{cases},$$
- $\int_\mathcal{H}(T, v) : \mathcal{T}(\mathcal{L}_\Sigma) \times V \rightsquigarrow \mathcal{L}_\int$ so that
$$\int_\mathcal{H}(T, v) = \begin{cases} name & \text{if } \text{Lab}(v) \in \mathcal{H} \text{ and } v \in V \text{ is a \textit{hook} in } T \\ \bot & otherwise \end{cases},$$

- $\int_{\mathcal{F}}(T) : \mathcal{T}(\mathcal{L}_{\Sigma}) \rightsquigarrow \mathcal{L}_{\int}$ so that
$$\int_{\mathcal{F}}(T) = \begin{cases} name & \text{if } \mathrm{Lab}(R) \in \mathcal{F} \text{ and } T \text{ is an } \textit{FCM fragment component} \\ \bot & otherwise \end{cases} . \qquad \diamond$$

Although the above definition uses CFGs as the basic language-specification formalism, this is not meant as a strong restriction. As shown in Chapter 3.1, without loss of generality, the definition also applies for EBNF-like grammars and other CFG-based concepts for concrete and abstract syntax such as XML and EMOF. Moreover, any tree specification language that can be mapped to a CFG is covered by the definition.

The following example shows how Definition 4.1 is applied to a concrete language.

Example 4.2 (fragment component model).
Reconsider the grammar G_{Log} of the LogProg language defined in Example 3.13. In this example, an FCM for LogProg was defined. As requirements, the FCM should support the extensibility of statement lists via an according hook and it should support genericity of logic expressions. Let $FCM_{\text{Log}} = (G_{\text{Log}}, \mathcal{S}, \mathcal{H}, \mathcal{F}, \int, \mathcal{L}_{\int})$ be the LogProg fragment component model. To model the according component types in FCM_{Log}, the corresponding nonterminals have to be added to the $\mathcal{F}$ candidate set:

$$\mathcal{F} = \{Stmt, Program, Expr\}$$

To model genericity of expressions, the corresponding nonterminal has to be added to the $\mathcal{S}$ candidate set:

$$\mathcal{S} = \{Expr\}$$

To model the extensibility of LogProg programs with statements, the corresponding list nonterminal has to be added to the $\mathcal{H}$ candidate set:

$$\mathcal{H} = \{StmtList\}$$

The above sets define those nonterminals which can deliver appropriate fragment components, hooks and slots. To identify which *instances* of them are components, hooks or slots, the point identification function $\int$ needs to be defined. The language of point identifiers $\mathcal{L}_{\int}$ shall be equivalent to the codomain of LogProg's `ident` tokens:

$$\mathcal{L}_{\int} = \{\mathrm{a} \ldots \mathrm{z}, \mathrm{A} \ldots \mathrm{Z}, \_, ., 0 \ldots 9\}^{+}$$

For the naming of fragment components, $\int_{\mathcal{F}}$ shall derive the name of a component given as a syntax tree T from the name of the tree's source file. Therefore, it is assumed that LogProg trees are obtained from files in some directory and the file name can be accessed via a system function $\mathrm{fname}(T) : \mathcal{T}(\mathcal{L}_{\Sigma}) \rightarrow \mathcal{L}_{\int}$. Hence, if T is a fragment tree according to $\mathcal{F}$ and Definition 4.1

(i.e., it is an instance of $G_{\text{Log}|\mathit{Stmt}}$, $G_{\text{Log}|\mathit{Expr}}$ or G_{Log}) then:

$$\int_{\mathcal{F}}(T) = \text{fname}(T)$$

$\int_{\mathcal{S}}$ shall recognize *Expr* slots by the suffix "*Slot*" and by extracting the portion of the identifier in front of the suffix. Since *Expr* nodes do not carry naming information themselves, this information has to be extracted from the tree. Moreover, only *Expr* trees of a specific shape shall be recognized as slots, which only have one leaf node labeled with an *ident* token. According to the nonterminals of G_{Log}, those trees have the shape $\mathit{Expr}[\mathit{Or}[\mathit{And}[\mathit{Term}[\mathit{Name}[\mathit{ident}]]]]]$. If T and its node $v \in V$ fulfill the conditions given in Definition 4.1 (i.e., v is an *Expr* node) and if the subtree rooted by v has the above shape, then:

$$\int_{\mathcal{S}}(T, v) = \texttt{if}(\text{Lab}(v) = \mathit{Expr})\,\texttt{then}\,(\text{pfx}(v.\mathit{Or}.\mathit{And}.\mathit{Term}.\mathit{Name}.\mathit{ident}))\,\texttt{else}\,(\bot),$$

where pfx simply extracts the prefix value of the *ident* token if it recognizes the slot suffix, otherwise it evaluates $\bot$.

$\int_{\mathcal{H}}$ shall recognize *StmtList* hooks. Hence, if the conditions related to $\int_{\mathcal{H}}$ given by Definition 4.1 hold, then

$$\begin{aligned}\int_{\mathcal{H}}(T, v) = \texttt{if}(\neg(\text{Lab}(\text{Par}_T(v)) = \mathit{StmtList}) \wedge \text{Lab}(v) = \mathit{StmtList}) \\ \texttt{then}\,(\mathit{StmtList})\,\texttt{else}\,(\bot)\end{aligned}$$

The following LogProg program is a fragment component with respect to the above defined component model.

```
1  main {
2      a = initSlot;
3      b = c;
4      result = a & !b | !a & b;
5   }
```

Listing 4.1: `simple.fgmt`: a LogProg fragment component.

The syntax tree obtained from the fragment is identified by $\int_{\mathcal{F}}$ as a fragment component with the name `simple.fgmt`. $\int_{\mathcal{S}}$ identifies an *Expr* slot named `init` on the right side of the statement in Line 2. The list of statements ranges from Line 2 to Line 4 and is identified as a *StmtList* hook by $\int_{\mathcal{H}}$. ◇

As in the cases of CFGs and AGs, it is also convenient for FCMs to have an according specification language. Such languages may provide different degrees of freedom to specify the FCM constituents. For example, it can be imagined that a small and simple DSL for such CMs could only provide features to determine the sets of point and fragment candidates (i.e., $\mathcal{S}$, $\mathcal{H}$, $\mathcal{F}$) while keeping the corresponding parts of $\int$ inaccessible using some predefined determination pattern like "a node is always a slot (hook, fragment) if its label is contained in $\mathcal{S}$ ($\mathcal{H}$, $\mathcal{F}$)". Alternatively, either $\mathcal{S}$ or $\mathcal{H}$ could be left empty, which leads to a clear distinction between SoCC abstractions like aspects (i.e., $\mathcal{S} = \emptyset$) and TMP abstractions as in the standard library of C++ (i.e., $\mathcal{H} = \emptyset$), or fragment composition in BETA.

Given an FCM, the bind composition operator of ISC transforms fragment components at their slots by replacing them with another fragment. Thereby, the operator ensures the syntactic integrity of the resulting components according to what is specified in the FCM. Definition 4.2 gives a precise specification of bind:

Definition 4.2 (bind composition operator):
Let $FCM = (G, \mathcal{S}, \mathcal{H}, \mathcal{F}, \int, \mathcal{L}_\int)$ be a fragment component model, where $G = (N, \Sigma, P, S)$ is a CFG. The *bind* composition operator $\dot{\pi}_{FCM}(T_1, T_2, \mathrm{m}) : \mathcal{T}(\mathcal{L}_\Sigma) \times \mathcal{T}(\mathcal{L}_\Sigma) \times (\mathcal{T}(\mathcal{L}_\Sigma) \to 2^V) \rightsquigarrow \mathcal{T}(\mathcal{L}_\Sigma)$ is a partial function composing any two FCM components $T_1 = (V_1, E_1, \mathrm{Lab}_1, \mathcal{L}_\Sigma)$ and $T_2 = (V_2, E_2, \mathrm{Lab}_2, \mathcal{L}_\Sigma)$ with $T_1, T_2 \in \mathcal{T}(\mathcal{L}_\Sigma)$ at points $\mathrm{m}(T_1) \subset V_1$ by replacing the subtrees induced by nodes in $\mathrm{m}(T_1)$ with copies of T_2. The result of an application of $\dot{\pi}_{FCM}$ is a valid syntax tree w.r.t. G or undefined ($\perp$), if the transformation cannot yield a valid tree.

More formally, let $f_1 = \mathrm{Lab}_1(R_1)$, $f_2 = \mathrm{Lab}_2(R_2)$ and $\mathcal{Q} = \mathrm{m}(T_1)$ where $f_1 \Rrightarrow^*_G T_1$, $f_2 \Rrightarrow^*_G T_2$ and R_1, R_2 the root nodes of T_1 and T_2. $\dot{\pi}_{FCM}$ is given as follows:

$$\dot{\pi}_{FCM}(T_1, T_2, \mathrm{m}) = \begin{cases} T_1[\mathcal{Q}/T_2] = T_1' & \text{if } \mathcal{Q} \neq \emptyset \\ & \quad \text{and } f_1, f_2 \in \mathcal{F} \\ & \quad \text{and } \int_\mathcal{F}(T_1) \neq \perp \wedge \int_\mathcal{F}(T_2) \neq \perp \\ & \quad \text{and } \forall v \in \mathcal{Q} : \int_\mathcal{S}(T_1, v) \neq \perp \wedge v \neq R_1 \\ & \quad \text{and } f_1 \Rrightarrow^*_G T_1', \\ T_1 & \text{if } \mathcal{Q} = \emptyset, \\ \perp & otherwise. \end{cases}$$

$T_1[\mathcal{Q}/T_2] = f_1[\ldots\, t_1' \,\ldots\, t_i' \,\ldots\, t_k' \,\ldots]$ is the replacement of all $k = |\mathcal{Q}|$ subtrees (slots) $t_i = l_i[\ldots]$ in $T_1 = f_1[\ldots\, t_1 \,\ldots\, t_i \,\ldots\, t_k \,\ldots]$ by $t_i' = T_2$ where $R_i \in \mathcal{Q}$ is the root of t_i and $1 \leq i \leq k$. $\diamond$

The following example demonstrates the bind composition operator applied to LogProg fragment components.

Example 4.3 (bind composition operator).
Reconsider FCM_{Log} defined in Example 4.2 and the LogProg fragment component `simple.fgmt` in Listing 4.1. Let T_{core} be the syntax tree that corresponds to `simple.fgmt`:

$$T_{core} = Program[main, `\{\text{'}, StmtList[Stmt[Name[ident], =, \underline{Expr[Or[And[Term[ident]]]]}, ;], \ldots], `\}\text{'}]$$

As a fragment to be bound, let `ex.fgmt` be the fragment containing the LogProg expression `t` and T_{ex} with $w(T_{ex}) = t$ the corresponding syntax tree $T_{ex} = Expr[Or[And[Term[t]]]]$.

Moreover, the matching function m shall be defined as:

$$\mathrm{m}(T_{core}) = \{Program.StmtList.Stmt_1.Expr\} = \{v\}$$

Given T_{core}, T_{ex} and m, the bind operator $\dot{\pi}_{FCM_{\mathrm{Log}}}$ can be applied to compose the two trees:

$$\begin{aligned}\dot{\pi}_{FCM_{\mathrm{Log}}}(T_{core}, T_{ex}, \mathrm{m}) = T_{core}[\{v\}/T_{ex}] = Program[main, `\{', StmtList[\\ Stmt[Name[ident], =, \underline{Expr[Or[And[Term[t]]]]}, ;], \ldots], `\}']\end{aligned}$$

Finally, the composed tree needs to be serialized to its LogProg string representation which is the fragment in Listing 4.1 with `slotIdent` replaced by `t`. $\diamond$

The extend composition operator can be defined in a way similar to the bind operator. However, unlike bind, extend does not replace complete subtrees, but inserts them at a certain position. Definition 4.3 gives a precise specification of extend.

Definition 4.3 (extend composition operator):
Let $FCM = (G, \mathcal{S}, \mathcal{H}, \mathcal{F}, \int, \mathcal{L}_\int)$ be a fragment component model, where $G = (N, \Sigma, P, S)$ is a CFG. The *extend* operator $\ddot{\pi}_{FCM}(T_1, T_2, \mathrm{m}, x) : \mathcal{T}(\mathcal{L}_\Sigma) \times \mathcal{T}(\mathcal{L}_\Sigma) \times (\mathcal{T}(\mathcal{L}_\Sigma) \to 2^V) \times \mathbb{N} \rightsquigarrow \mathcal{T}(\mathcal{L}_\Sigma)$ is a partial function composing any two FCM fragment components $T_1 = (V_1, E_1, \mathrm{Lab}_1, \mathcal{L}_\Sigma)$ and $T_2 = (V_2, E_2, \mathrm{Lab}_2, \mathcal{L}_\Sigma)$ with $T_1, T_2 \in \mathcal{T}(\mathcal{L}_\Sigma)$ at points $\mathrm{m}(T_1) \subset V_1$ by inserting copies of T_2 at the position x to list nodes in $\mathrm{m}(T_1)$. The result of applications of $\ddot{\pi}_{FCM}$ is a valid syntax tree w.r.t. G or undefined if the transformation would not yield a valid syntax tree.

In the following, let $f_1 = \mathrm{Lab}_1(R_1)$, $f_2 = \mathrm{Lab}_2(R_2)$ where $f_1 \Rrightarrow^*_G T_1$, $f_2 \Rrightarrow^*_G T_2$, $\mathcal{Q} = \mathrm{m}(T_1)$, and R_1, R_2 the root nodes of T_1 and T_2. $\ddot{\pi}_{FCM}$ is defined as follows:

$$\ddot{\pi}_{FCM}(T_1, T_2, \mathrm{m}, x) = \begin{cases} T_1[\mathcal{Q}/T_2, x] = T_1' & \text{if } \mathcal{Q} \neq \emptyset \\ & \quad \text{and } f_1, f_2 \in \mathcal{F} \\ & \quad \text{and } \int_{\mathcal{F}}(T_1) \neq \bot \wedge \int_{\mathcal{F}}(T_2) \neq \bot \\ & \quad \text{and } \forall v \in \mathcal{Q} : \int_{\mathcal{H}}(T_1, v) \neq \bot \wedge v \neq R_1 \\ & \quad \text{and } f_1 \Rrightarrow^*_G T_1', \\ T_1 & \text{if } \mathcal{Q} = \emptyset, \\ \bot & otherwise. \end{cases}$$

$T_1[\mathcal{Q}/T_2, x] = f_1[\ldots\, t_1' \,\ldots\, t_i' \,\ldots\, t_k' \,\ldots]$ denotes the replacement of all $k = |\mathcal{Q}|$ sublists (hooks) $t_i = l_i[s_1, \ldots, s_j, \ldots, s_n]$ of $T_1 = f_1[\ldots\, t_1 \,\ldots\, t_i \,\ldots\, t_k \,\ldots]$ by an extended list t_i' where $R_i \in \mathcal{Q}$ is the root of t_i, $1 \leq i \leq k$, $j, n \in \mathbb{N}$, $1 \leq j \leq n$ and n is the number of list entries s_j in t_i. The t_i' are constructed as follows: if $1 \leq x \leq n$, then $j = x$ and $t_i' = l_i[s_1, \ldots, s_{j-1}, s_j', s_{j+1}', \ldots, s_{n+1}']$ where $s_j' = T_2$ and $s_{j+1}' = s_j$, …, $s_{n+1}' = s_n$. Otherwise, $t_i' = l_i[s_1, \ldots, s_j, \ldots, s_n, s_{n+1}']$ where $s_{n+1}' = T_2$. $\diamond$

The following example shows how the extend operator can be applied to a LogProg fragment component.

Example 4.4 (extend composition operator).
Given FCM_{Log} defined in Example 4.2, the LogProg fragment component `simple.fgmt` in Listing 4.1 *after* binding the slot in Example 4.3 and T_{core} the syntax tree corresponding to `simple.fgmt`. This example is focused on the *StmtList* hook, i.e:

$$T_{core} = Program[main, \text{‘\{’}, StmtList[Stmt[\ldots], Stmt[\ldots], Stmt[\ldots]], \text{‘\}’}]$$

For the extension of T_{core}, consider the fragment component `st.fgmt` with the contents `c = f;` and the corresponding syntax tree T_{st}:

$$T_{st} = Stmt[Name[ident], \text{‘=’}, Expr[Or[And[Term[Name[ident]]]]], ;]$$

A matching function m matches the *StmtList* hook:

$$\mathrm{m}(T_{core}) = \{Program.StmtList\} = \{v\}$$

Given T_{core}, T_{st} and m, the extend composition operator can be applied to extend T_{core} with T_{st}. In contrast to bind, a list position needs to be provided to determine where T_{st} is inserted into the $StmtList$. Since `st.fgmt` contains a declaration of `c`, it seems reasonable to insert it at a position preceding the second statement which uses `c` to compute `b` (cf. Listing 4.1). Therefore, the fragment is added at the first position which is also called *prepending* in ISC terminology:

$$\ddot{\pi}_{FCM_{\text{Log}}}(T_{core}, T_{st}, \mathrm{m}, 1) = T_{core}[\{v\}/T_{st}, 1] = Program[main, \text{‘\{’}, StmtList[$$
$$Stmt[Name[ident], \text{‘=’}, Expr[Or[And[Term[Name[ident]]]]], ;],$$
$$Stmt[\ldots], Stmt[\ldots], Stmt[\ldots]], \text{‘\}’}]$$

After binding `initSlot` in Example 4.3 and extending *StmtList* in this example, the composition result can be found in Listing 4.2 below.

```
main {
    c = f;
    a = t;
    b = c;
    result = a & !b | !a & b;
 }
```

Listing 4.2: The `simple.fgmt` fragment component after composition.

Since a is $true$ and b is $false$ because of c, the result of the program is $true$. ◇

In the following, Definition 4.1 is complemented with a notion for well-definedness. This avoids inconsistent models such as FCMs that allow slots but do not support corresponding fragment components.

Definition 4.4 (well-defined fragment component model):
Let $FCM = (G, \mathcal{S}, \mathcal{H}, \mathcal{F}, \int, \mathcal{L}_{\int})$ be a fragment component model. FCM is called *well-defined* under the following conditions:

- G is a reduced grammar (cf. Definition 3.3). That is, fragments cannot be empty, each fragment type must be reachable from the start symbol S in G and a fragment must not derive S in any derivation.
- For any FCM fragment component $T = (V, E, \mathrm{Lab}, \mathcal{L}_\Sigma)$ and $v \in V$, it holds that $\int_{\mathcal{S}}(T, v) \neq \bot \rightarrow \int_{\mathcal{H}}(T, v) = \bot$. That is, a node should not be a slot and a hook at the same time (*exclusiveness of point categories*).
- For each slot candidate $c \in \mathcal{S}$, there exist fragment candidates $f, g \in \mathcal{F}$ and fragment components $T_1 = f[\ldots t \ldots]$, $T_2 = g[\ldots]$, where $t = (V', E', \mathrm{Lab}', \mathcal{L}_\Sigma) = c[\ldots]$ is a subtree of T_1 with root R' which is a slot with $\int_{\mathcal{S}}(T_1, R) \neq \bot$ so that a fragment component T_1' can be obtained by binding t in T_1 with T_2, i.e., $\dot{\pi}_{FCM}(T_1, T_2, \{(T_1, R')\}) = T_1'$ (*composability of slots*).
- For each hook candidate $c \in \mathcal{H}$, there exist fragment candidates $f, g \in \mathcal{F}$ and fragment components $T_1 = f[\ldots t \ldots]$, $T_2 = g[\ldots]$ where $t = (V', E', \mathrm{Lab}', \mathcal{L}_\Sigma) = c[e_1, \ldots, e_k]$ is a list in T_1 with root R' which is recognized as a hook (i.e., $\int_{\mathcal{H}}(T_1, R') \neq \bot$) so that a fragment component T_1' can be obtained by extending the list in the subtree induced by R' in T_1 with T_2, i.e., $\ddot{\pi}_{FCM}(T_1, T_2, \{(T_1, R')\}, 1) = T_1'$ (*composability of hooks*). ◇

In the following example, the component model of Example 4.2 is identified as a well-defined component model.

Example 4.5 (well-defined fragment component model).
Reconsider FCM_{Log} of Example 4.2. FCM_{Log} is a well-defined component model because (1) G_{Log} is a reduced CFG, (2) $\mathcal{F} \cap \mathcal{H} = \emptyset$ so that for all fragment components $T = (V, E, \mathrm{Lab}, \mathcal{L}_\Sigma)$ and $v \in V$, v never is a hook *and* a slot, (3) slots are composable by Example 4.3 and (4) hooks are composable by Example 4.4. ◇

For the sake of completeness, the following two sketched definitions introduce the notions of *composition program* and *composition system*.

Definition 4.5 (composition program):
A composition program consists of a set of composer signatures which can be applied for composition, a stack of already executed compositions and a decision function which selects the next applicable operation from the set of composer signatures and the stack of already executed compositions. ◇

Definition 4.6 finally defines fragment composition systems and their compartments (cf. [Aßmann 2003, p. 119]):

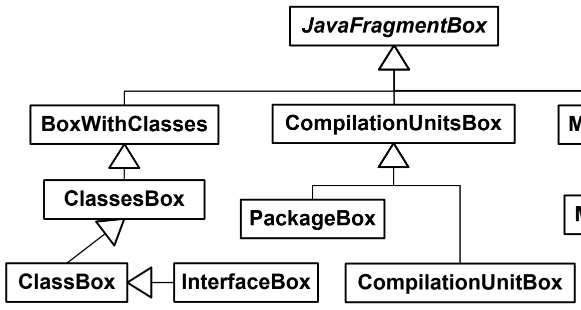


Figure 4.3.: Excerpt from the Boxology of COMPOST (cf. [Aßmann 2003, p. 296]).

Definition 4.6 (fragment composition system):
A fragment composition system (or invasive composition system) is a triple consisting of a fragment component model, a set of primitive composition operators and complex composers, and a composition language. ◇

In the remainder of this chapter, existing approaches on the implementation of invasive composition systems are evaluated and compared.

4.2. The COMPOST Approach

COMPOST has been the first demonstrator implementation of ISC [Aßmann 2003] and provides an invasive composition system for Java 1.4 implemented in Java. It is built around the RECODER tool suite [Ludwig 2002; Heuzeroth et al. 2013]—a parsing, refactoring and transformation engine for Java.

4.2.1. Component Model and Composition Process

As RECODER provides the model and the according classes of Java's abstract syntax, the COMPOST Java FCM is realized as an extra abstraction layer on top of these classes and the RECODER tooling. This layer is called a *Boxology* and includes the basic implementation of a fragment component API—the *fragment boxes*—and the basic composers, and it determines the occurrences of slots and hooks in Java programs. Furthermore, it already provides some built-in complex SoC abstractions like complex composers for mixin-based inheritance (i.e., HiDec), aspects and views (i.e., SoCC), and also generics (i.e., TMP).

Figure 4.3 gives an overview of the available standard fragment boxes for Java in COMPOST as a UML class diagram. Essentially, COMPOST supports `CompilationUnitBox`es, `Class-Box`es, `MethodBox`es and `StatemenBox`es as basic fragment components encapsulating compilation units, classes, methods and statements. Fragment boxes provide a basic API to write composition programs in Java. They provide convenience methods for finding and composing slots and hooks, which ease the implementation process for composition-system users (i.e., the developers of composition programs). The advantage of using Java as a composition language is twofold. First, the composition system inherits the expressive power of a full imperative programming language. Thus, arbitrary composition programs can be written, only restricted by the supported FCM and the composition API. Second, component language and composition language are equivalent. Hence, the system allows developers to write *staged* composition programs (i.e., composition programs that generate composition programs), which also may reuse already developed composition abstractions (e.g., complex aspect or template composers).

The composition process introduced by COMPOST is crucial to understand the way ISC-based systems generally work and which tasks have to be generally conducted by a fragment composition system. Below, the process is described in general as well as COMPOST-specifically. The process passes through the following phases (cf. [Aßmann 2003, Chapter 4]):

Initialization. The composition system has to be instantiated and initialized. In COMPOST, this has to be done in the composition program by a corresponding Java statement which instantiates a new `FragmentCompositionSystem` object as shown below:

```
FragmentCompositionSystem cSys = new FragmentCompositionSystem("...");
```

Fragment loading. The composition-system instance can be used to load fragment components from the file system as well as strings and other representations. Depending on the fragment-identification function $\int_{\mathcal{F}}$ implemented by the system, a fragment may be recognized as a component and can be loaded. In COMPOST, Java boxes can be loaded via the API provided by the `FragmentCompositionSystem`:

```
FragmentBox box = cSys.createBox("...<resourceName>...");
```

Point identification. Slots and hooks (and potentially other compositional points as shown later) of a box have to be identified using the implementation of the point-identification functions—$\int_{\mathcal{S}}$ and $\int_{\mathcal{H}}$—of the system. COMPOST uses *hook-identification tables* to look up AST nodes which have been identified as compositional points. The table can be accessed through the `FragmentBox` API:

```
Hook hook = box.findHook("...<hookName>...");
```

For hook declarations, COMPOST provides a specific naming scheme, which originates from the *Hungarian Notation* conventions by [Simonyi 1976] and is recognized by the point-identification functions. Declared hooks in COMPOST are prefixed with `generic`, followed by the actual name of the point and a suffix which denotes the fragment type of the point. The following example illustrates this scheme. It is a `ClassDeclaration` whose name can be parametrized by binding the slot `T`, which is an `Identifier`.[2]

```
public class genericTIdentifier {}
```

[2]The underline notation is adopted from the ISC book [Aßmann 2003] to improve the readability of fragment-box listings. It is not an integral part of the markup.

Transformation. In this phase, the actual composition is executed by binding and extending slots and hooks. Depending on the composition-system implementation, this may happen at a designated composition time, immediately or in a mixed composition style. Transformations in COMPOST are realized using the transactional transformation engine provided by RECODER. A composition step can be triggered via the `Hook` API:

```
hook.extend(...<fragment>...) // for list-like hooks
hook.bind(...<fragment>...) // for slot-like hooks
```

COMPOST distinguishes *declared hooks* and *implicit hooks*, but has no explicit slot concept. Typically, declared hooks correspond to the slots in Definition 4.1 (as they are node-based) while implicit hooks correspond to the hooks in the same definition (as they are list-based).

Emitting. Finally, fragments have to be emitted. Depending on the purpose of the composition system, fragments are simply printed as text to the file system (*pretty printing*). However, they can also be emitted as an AST or model, e.g., for further processing in a transformation chain. In COMPOST, fragment boxes are printed back to the file system. This can be triggered via the `CompositionSystem` API:

```
cSys.getJavaCompositionSystem().printAll();
```

Instantiations of the composition process can be simple non-recursive programs with distinct initialization step, several composition steps and a distinct serialization phase. However, composition processes can also form composite transformation chains with complex interdependencies. For example, a circular composition program may reuse composed fragments from a previous composition program and may itself produce new input fragments for those composition programs it depends on.

In the following, COMPOST is used to implement the BAF code generator as a simple composition program.

4.2.2. The Business Application Generator in COMPOST

To implement the BAF generator with COMPOST, the basic fragment components of Chapter 2 have to be slightly rewritten, since COMPOST relies Hungarian Notation to mark up slots instead of the syntactic hedges originally proposed in Chapter 2. Listing 4.3 shows a variant of the BAF template for `Person` business objects. The class name can be parametrized via the `generic`Type`Identifier` slot which has the name `Type` and can be replaced by a valid Java identifier. The super class can be parametrized by binding the `generic`Super`Superclass` slot which has the name `Super`, with a reference identifier denoting a concrete super class. The remaining slots `generic`TypeName`Constant` and `generic`Pfx`Constant` represent string constants that should contribute the class name (`TypeName`) and a line prefix (`Pfx`) to

```
1 public class genericTypeIdentifier extends genericSuperSuperclass {
2    public String asString(){
3       String v = genericTypeNameConstant;
4       v+= genericPfxConstant + " id:" + getID();
5       v+= genericPfxConstant + " name:" + getName();
6       return v;
7    }
8 }
```

Listing 4.3: `Person.jbx`—the COMPOST equivalent of the business-object base template `Person.frgmt` in Listing 2.3.

```
/* Field.jbx */
private genericTType genericFieldNameIdentifier;

/* Getter.jbx */
public  genericTType genericGetSfxIdentifier() {
    return genericFieldNameIdentifier;
}

/* Setter.jbx */
public void genericSetSfxIdentifier(genericTType genericFieldNameIdentifier){
     this.genericFieldNameIdentifier = genericFieldNameIdentifier;
}
```

Listing 4.4: `Field.jbx`, `Setter.jbx` and `Getter.jbx`—the COMPOST equivalent of the property accessory templates in Listing 2.5.

the output of the `asString()` method.[3] In contrast to the original `Person.fgmt`, the lines that produce the output for the `ID` and `name` properties–Lines 4 and 5—use additive expressions on strings as slots within constants are not considered by COMPOST.

The code of the field fragment (`Field.jbx`) and the corresponding accessor methods (`Getter.jbx` and `Setter.jbx`) is shown in Listing 4.4. It has also been rewritten according to the COMPOST naming schemes. The identifier slots are one-to-one mappings to the slots in the scenario description. `genericFieldNameIdentifier` denotes the placeholder for the actual name of the field. Moreover, `genericGetSfxIdentifier` as well as `genericSetSfxIdentifier` are the slots for the accessor-method names. The actual Java type of the field can be parametrized using the `genericTType` slot which can be replaced by an AST node representing a reference to a type declaration. The name `T` is used instead of `Type` to avoid confusion with the `Type` suffix.

The composition program using COMPOST's Java API to express the composition can be investigated in Listing 4.5. The composition system is initialized in Line 4 while the model (cf. Listing 2.2) is loaded in Line 4. After loading, for each `RoleDefinition` in the model, an adequate Java implementation is composed from the above fragments (cf. Lines 10–28). First, the

[3]Originally, the COMPOST version 0.78e, which was used for the implementation, did not support `Constant` slots. Thus, it was extended with an according hook class.

```
1  public void compositionProgram() throws IOException {
2
3      /* initialize/get the current composition system instance */
4      FragmentCompositionSystem compositionSystem = getCompositionSystem("baf/in/");
5
6      /* load the business model with EMFText */
7      BusinessModel bm = loadBusinessModel("baf/in/model.bm");
8
9      /* for each role definition generate a Java class */
10     for(RoleDefinition role: preOrder(bm.getRoleDefinitions())){
11
12         /* copy and rename Person.jbx */
13         CompilationUnitBox box = getPersonBoxForRole(role.getName());
14
15         /* find and bind declared hooks (slots) in box */
16         box.findHook("genericTypeIdentifier").bind(role.getName());
17         box.findHook("genericNameConstant").bind("\"" + role.getName() + "\"");
18         box.findHook("genericPfxConstant").bind("\"\\n\"");
19
20         /* compose the super class */
21         composeSuperClass(box,role);
22
23         /* compose fields and accessory methods for properties */
24         composeFieldsAndAccessors(box,role);
25
26         /* extend asString according to properties of super roles */
27         composeAsString(box, role);
28     }
29     /* print all fragments to the output directory */
30     compositionSystem.getCompositionSystemManager().setOutputPath("baf/out/");
31     compositionSystem.getJavaCompositionSystem().printAll();
32 }
```

Listing 4.5: The BAF code generator as COMPOST composition program.

contents of `Person.jbx` are copied, a `CompilationUnitBox` is created and its box name is set to the name of the `RoleDefinition`. Second, the `Type`, `Name` and `Pfx` slots are bound to the role name and the line-prefix string (cf. Lines 16–18). In the third step, a subcomposer `composeSuperClass` which composes a super class and weaves constructors is invoked (cf. Line 21). It will be explained in more detail in the next paragraph. During the iteration's fourth step, the subcomposer `composeFieldsAndAccessors` composing and weaving fields and accessor methods for the `RoleDefinitions` properties is invoked (cf. Line 24). It will be explained in more detail after the next paragraph. Afterwards, `composeAsString` is called to extend the `methodExit` hook of the `asString` operation with string-building code according to the `PropertyDefinitions` of the super roles (cf. Line 27). Finally, after all `RoleDefinitions` were processed by the composition program, the preferred output location is set and all fragments are printed to files in that location.

The composition program `composeSuperClass` which computes the actual super class of the generated Java class and extends it with a constructor, is shown in Listing 4.6. Basically, it makes a decision on the actual super class depending on the number of super `RoleDefi-`

```
1  private void composeSuperClass(CompilationUnitBox box, RoleDefinition role) {
2
3      /* request number of super roles declared for the current role */
4      int roleNum = role.getSuperRoles().size();
5
6      /* if no super roles defined --> bind Person to Super slot */
7      if(roleNum == 0) {
8          box.findHook("genericSuperSuperclass").bind("Person");
9      }
10     /* if at least one super role defined --> bind the first to Super slot */
11     else if(roleNum > 0) {
12         box.findHook("genericSuperSuperclass")
13             .bind(role.getSuperRoles().get(0).getName());
14
15         /* if two or more super roles defined --> mix in the others */
16         for(int i = 1; i < roleNum; i++) {
17             box.extend(getPersonBoxForRole(role.getSuperRoles().get(i).getName()));
18             removeDuplicateMethods(box);
19         }
20     }
21 }
```

Listing 4.6: Complex super-class composer of the COMPOST-based BAF generator.

`nitions` that have been declared for the given `RoleDefinition` in the current BAF model. If no super role is declared, `Person` will be bound as the super class (cf. Lines 7–9). The second case occurs if one or more super `RoleDefinitions` have been declared in the model (cf. Lines 11–20). The super class is bound to the name of the `RoleDefinition` declared first in the super role list. Accordingly, the emitted Java class inherits from the implementation class generated for the super role. Hence, no extra code has to be woven for this `RoleDefinition`. However, since Java only supports single inheritance, the approach for the remaining `RoleDefinitions` is different by using COMPOST's built-in class-based `extend` operator (cf. Line 17). The `extend` operator "mixes" the members of the argument class into the `ClassBox`(es) of the target `CompilationUnitBox`. Hence, the code which is generated for the super `RoleDefinition` is added also to the class body of the extending `RoleDefinition`. After the extend operator has been executed, duplicate methods are deleted, e.g., `asString`. This has to be done manually because COMPOST does not check context-sensitive constraints.

The contents of `composeFieldsAndAccessors` can be investigated in Listing 4.7. For each `PropertyDefinition` owned by a role, the composition program generates the corresponding output using the accessor templates of Listing 4.4 and extending the intermediate `CompilationUnitBox`. First, variables that carry the names of the property's field accessor methods and the default value are initialized (cf. Lines 8–10). In a second conditional composition step in Lines 13–26, the box is extended with the default-setting code if this is specified in the model. In this example, a complex constructor-based variant to realize this has been chosen: if a default value is present, the members list of the generated class is extended with a parameterless default constructor. The constructor itself is then extended at its `methodExit` hook with a call

```
private void composeFieldsAndAccessors(
                    CompilationUnitBox box, RoleDefinition role) throws IOException {

    /* iterate over the given role's properties */
    for(PropertyDefinition prop:role.getProperties()){

        /* prepare set and get method names, request property default value */
        String setSfx = "set" + toFirstUpper(prop.getName());
        String getSfx = "get" + toFirstUpper(prop.getName());
        String defValue = prop.getType().getDefault();

        /* if a default value is declared, add a setSfx call to the constructor */
        if(defValue != null){

            /* extend the implicit box members hook with an empty constructor */
            if(box.findHook("constructors") == null){
                box.findHook("members")
                    .extend("public " + role.getName() + "()" +
                    " { \nsuper();\nSystem.out.println(\"setting defaults ...\");}");
            }

            /* extend the implicit methodExit hook of the constructor */
            box.findHook("genericTypeIdentifier." + role.getName() + ".methodExit")
                .extend("set" + toFirstUpper(prop.getName()) +
                        "(" + prop.getType().getDefault() + ");");
        }

        /* if a property is not declared in a super role --> extend the box */
        if(!isShadowed(prop)){

            /* extend asString operation with property-specific code */
            box.findHook("genericTypeIdentifier.asString.methodExit")
              .extend("v+= \"\\n\" + \" " + prop.getName() + ":\" + "+getSfx+"();");

            /* extend box members with field and method boxes */
            box.findHook("members").extend(createFieldBox());
            box.findHook("members").extend(createSetterBox());
            box.findHook("members").extend(createGetterBox());

            /* bind the slots of the new members */
            box.findHook("genericTType").bind(prop.getType().getTargetType());
            box.findHook("genericNameIdentifier").bind(prop.getName());
            box.findHook("genericSetSfxIdentifier").bind(setSfx);
            box.findHook("genericGetSfxIdentifier").bind(getSfx);
        }

        /* commit the changes to RECODER to be safe in the next iteration*/
        box.commit();
}   }
```

Listing 4.7: Complex members composer of the COMPOST-based BAF generator.

to the set method of the corresponding field. Next, the field and accessor fragments are appended to the members list of the class (cf. Lines 29–45). A testing predicate—`isShadowed`—makes sure that if the property is already declared in a super role, it is not added to the currently processed role. The `asString` operation is extended with an expression statement appending property-specific information to the emitted object-representation string (cf. Lines 32 and 33). After extending the `members` hook of the `FragmentBox`, the fragment's slots are bound with the prepared type and identifier names. As a final step in each iteration, the changes are *committed* to the underlying RECODER engine (cf. Line 48). This operation ensures that fragments with the same hooks can be parametrized with values of the next property during the next iteration.

The resulting code emitted by the composition program is nearly equivalent to the code as it should have been expected from the description in Chapter 2. The exact code can be inspected in Appendix A.2.

4.2.3. Evaluation

In this section, it has been shown that COMPOST is suited to implement the BAF code generation backend. The emitted code is nearly equivalent to what was sketched in Chapter 2. COMPOST supports hook extension, slot binding and mixin-based composition. Compositional points can be addressed with path-like expressions and, since Java is used as a composition language, conditional composition (`if`), fragment iteration (`for`) and recursive composition programs are supported. Thus, COMPOST is powerful enough to express arbitrary compositions which are allowed w.r.t. its Boxology.

COMPOST has some downsides. Technically, it is out of maintenance and only supports the version 1.4 of Java. Hence, it would not be able to process fragment components containing foreach loops and generics introduced in Java 1.5. Moreover, the component model for Java is rather fixed and difficult to extend because it directly relies on the API and internals of RECODER. Consequently, to change COMPOST's $\int$ or the set of point identifiers $\mathcal{L}_\int$ (cf. Definition 4.1), the composition-system developer has to deal with these internals. This also comprises an understanding on how and when RECODER internally performs AST rewrites (compositions).

Besides the specific RECODER and Java binding, the Boxology is a universal API for ISC-based fragment composition and does not depend on a specific FCM. In principle, it can be used to develop adapters for arbitrary languages. Thus, creating a language-specific binding of COMPOST is creating subclasses of the Boxology classes and adapt the Boxology API to a fragment-language implementation. In [Pop et al. 2005], such a binding has been described for an XML-based DSL. The approach developed in this thesis picks up on the Boxology concept by providing reusable object-oriented RAG modules as a Boxology-like API with attributes.

Naturally, creating language-specific adapters for COMPOST is a cumbersome and implementation-saddled task. A more declarative and transparent approach could help to ease the implementation of ISC-based systems. The approaches of the next two sections are focused on providing declarative specification languages for fragment composition systems based on CFGs and metamodels.

4.3. Universal Invasive Software Composition

Universal invasive software composition (U-ISC) and its implementation Reusew*air* have been introduced by [Henriksson 2009]. U-ISC is a *grammarware* [Klint et al. 2005b] approach to ISC and eases the creation of fragment composition systems by providing declarative specification languages for component models and for syntactic language extensions with the purpose to introduce reuse-enabling concepts. Starting point for developing composition systems with U-ISC is the CFG G that specifies the abstract syntax of the fragment language. G is extended with slot and reuse nonterminals so that a reuse grammar G' is obtained. From G', the U-ISC implementation Reusew*air* for Java generates a basic infrastructure for composer implementation in plain Java. The U-ISC approach has been successfully applied to implement composition-system prototypes in several domains, e.g., semantic web or language processing [Aßmann et al. 2007]. In comparison to the COMPOST approach, U-ISC introduced the following novel concepts to ISC:

Declarative component-model specification. In contrast to the ISC approach by [Aßmann 2003], the U-ISC approach comprises the *component-model specification language (C_mSL)* to specify fragment component models. The C_mSL provides concepts to *slotify* certain nonterminals in the underlying grammar of the fragment language which is also called a *base grammar* in U-ISC. Slotification is a grammar transformation which, for a given nonterminal, introduces an alternative nonterminal and productions representing slots. Second, nonterminals of the base grammar can be declared as *fragment types*.

Visitor stubs and composition algebra. Basic composition operators such as *bind* and *extend* are realized as a *composition algebra* and implemented as a Java class in the Reusew*air* tool. This algebra is a basic API of composition operators that has to be used in complex composers for achieving complex compositional behavior. Complex composers are also implemented completely in Java. To support this, Reusew*air* generates an implementation stub, according to the well-known *visitor* design pattern (cf. [Gamma et al. 1995]). Choosing Java as a host language for composition seems to be a viable choice as it makes a full programming language available for writing composition programs.

CFG-based reuse-language extensions. As an extension to plain U-ISC component models, [Henriksson 2009] proposes *embedded* U-ISC to add composer declarations to the base language. Therefore, C_mSL supports direct manipulation of the base grammar by *construct injection* and marking nonterminals as composer signatures via *construct annotation*. Using injection and annotation, composer signatures can be embedded in a fragment language in such a way that complex composers can be “called” in the reuse language. For more detailed description of construct injection and construct annotation with C_mSL, it is kindly referred to [Henriksson 2009, p.115]. The semantics of composer declarations has to be implemented using the basic visitor stubs and the API of the composition algebra.

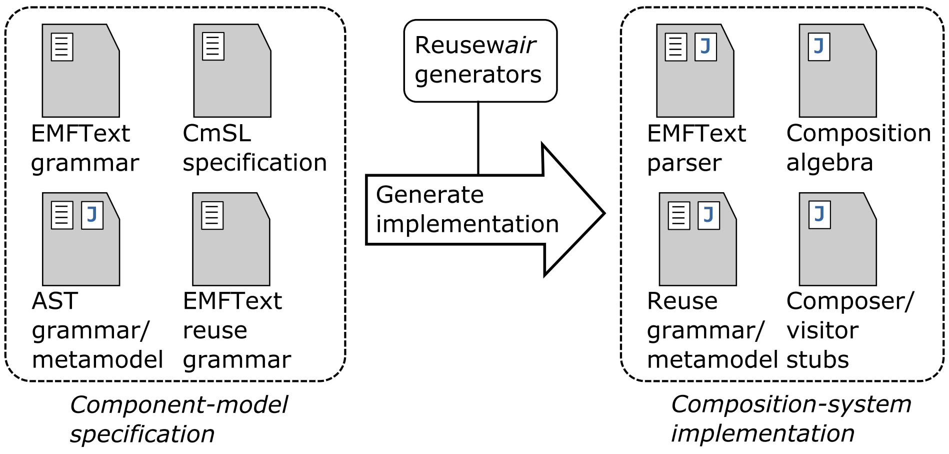


Figure 4.4.: Component-model code generation in Reuse*wair*.

Figure 4.4 summarizes Reuse*wair*'s generative approach. In a broad sense, the component-model specification consists of a $\mathrm{C_mSL}$ specification, an AST grammar of the CnL and a corresponding grammar for concrete syntax. Similar to the BAF model, the AST grammar is realized as an EMF-based metamodel while the concrete syntax is based on EMFText [Heidenreich et al. 2009a].[4] Based on the $\mathrm{C_mSL}$ specification, Reuse*wair* modifies the original AST grammar by injection and derives an extended grammar with reuse concepts. If slots or other concepts have been introduced into the extended AST grammar, a concrete-syntax grammar for these concepts has to be provided additionally. Reuse*wair*'s target language for generated artifacts is Java. For generating AST and parser implementations, it uses the EMF code generator for Java and the EMFText parser generator. From the $\mathrm{C_mSL}$, it generates a Java implementation of a basic *composition algebra* and a set of *Java fragment interfaces* as types of that algebra. Similar to fragment boxes in COMPOST, the fragment interfaces hide implementation detail, provide an API for loading and storing, as well as an API for accessing typical ISC composition operators. Finally, a rudimental visitor implementation is generated. The main purpose is to provide a simple infrastructure to traverse the ASTs of fragments and to invoke in-place compositions.

In the following subsections, parts of the BAF generator will be implemented using Reuse*wair*.[5]

4.3.1. A Component Model for Java$^-$

To realize the BAF code generator in Reuse*wair*, the *Java*$^-$ model used in [Henriksson 2009] has been extended with concepts required to realize the example (e.g., additive expressions and

[4]Historically, a rudimentary version of EMFText was an integral part of Reuse*wair* and all of its predecessors.

[5]Since Reuse*wair* has been discontinued in 2009, an intermediate version (0.6.0) from the repository [Software Technology Group and DevBoost GmbH 2013b] was used which the author of this thesis modified and improved to work with the BAF example and EMFText 1.4.1.

```
1  extends file:java.cs @ java as file:rjava.cs .
2
3  % declare which nonterminals should have slot alternatives
4  slotify java.QualifiedName.
5  slotify java.Identifier.
6  slotify java.StringValue.
7  slotify java.IntegerValue.
8
9  % declare which nonterminals are fragment types
10 fragtypes {
11         java.CompilationUnit, java.MethodDeclaration,
12         java.Statement, java.VariableDeclaration, java.Modifier,
13         java.StringValue, java.Identifier, java.QualifiedName,
14         java.ClassDeclaration, java.IntegerValue, java.AttributeDeclaration
15 }
```

Listing 4.8: C$_m$SL specification declaring a fragment component model for Java$^-$.

assignments). The improved Java$^-$ is a small subset of Java 1.4, including classes, methods, variable declarations, statements, assignments and additive expressions over strings and numeric values. Similar to all formal language implementations based on EMFText, Java$^-$ comes with parsing and printing support. Consequently, the generated implementation can load and store Java$^-$ fragments out of the box.

Listing 4.8 shows the C$_m$SL component model for the BAF example. In Line 1, the grammar of Java$^-$ is imported and the location of the target reuse grammar—`rjava`—is declared. The set of syntactic fragment-component types is declared between Lines 10 and 15. For each entry, Reusew*air* generates a fragment interface for composition. Slots are declared using the keyword `slotify`. Lines 4–7 contain the `slotify` declarations of the BAF example. For instance, `QualifiedName` and `Identifier` slots are used as placeholders for class names and variable names. *Slotification* is a transformation that takes an AST grammar G_{base} and produces a modified base grammar G'_{base} and a reuse grammar G_{reuse}. The transformation works as follows: consider a nonterminal $B \in G_{\text{base}}$, a slotification statement `slotify B` in some C$_m$SL specification, and the following two productions in G_{base}:[6]

```
B ::= $\gamma$
A ::= $\alpha$ child:B $\beta$
```

The algorithm introduces an abstract nonterminal `BAbstract` which replaces `B` in all of its contexts (i.e., right-hand sides of a production) and has `B` as an alternative. Consequently, in the modified base grammar G'_{base} the original productions are replaced by the following ones:

```
A ::= $\alpha$ child:BAbstract $\beta$
B$\triangleright$BAbstract ::= $\gamma$
@BAbstract ::= $\epsilon$
```

[6]The notation for abstract syntax used in this thesis is defined in Section 3.2.3. α, β, γ are arbitrary sentential forms.

```
1  SYNTAXDEF rjava
2  FOR <...location of abstract grammar/metamodel $G_{\text{reuse}}$...>
3
4  // declare which Java nonterminals can be parsed
5  START java.CompilationUnit, java.StringValue, java.AttributeDeclaration,
6           java.MethodDeclaration, java.QualifiedName, java.Statement
7
8  // importing the base grammars (abstract and concrete syntax)
9  IMPORTS {
10    java:<...location of abstract grammar/metamodel $G'_{\text{base}}$...>
11       WITH SYNTAX javamm <java.cs>
12 }
13
14 // declare the slot syntax with syntactic hedges
15 RULES {
16  IdentifierSLOT ::= "[[" name "]]";
17  QualifiedNameSLOT ::= "[["name "]]";
18  IntegerValueSLOT ::= "[i:" name ":i]";
19  StringValueSLOT ::= "[s:" name ":s]";
20
21  VariationPointName ::= name[IDENT];
22 }
```

Listing 4.9: Definition of an EMFText concrete syntax grammar for slots in Java^{-}.

The transformation is applied for every `slotify` statement in the C_mSL specification, and for each context at hand the concrete nonterminal is replaced with the abstract one. The reuse grammar G_{reuse} is an extension to G'_{base} and introduces a slot alternative `BSlot` which "inherits" from `BAbstract` and `Slot` from the internal Reuse*wair* component metamodel:

$$\texttt{BSlot} \triangleright \texttt{BAbstract,Slot ::=} \; \epsilon$$
$$\texttt{Slot ::= <name> <type>?}$$

Considering the BAF example, the above transformation is applied to the according nonterminals in Listing 4.8. Four abstract nonterminals are added to the abstract syntax of Java^{-} and four slot nonterminals are introduced into the reuse grammar of Java^{-}: `QualifiedNameSLOT`, `IdentifierSLOT`, `StringValueSLOT` and `IntegerValueSLOT`. Based on these types, all slots of the BAF example can be specified, except slots within strings (cf. `Pfx` in Listing 2.3) as these kinds of slots are not supported by Reuse*wair*.

Using Reuse*wair*'s slotify operator, the abstract syntax is ready to be used for composition. However, still the textual representation of slots needs to be specified as an EMFText grammar, which is shown in Listing 4.9. Lines 5–6 specify that the parser accepts the given nonterminals as starts symbols. The actual syntax of slots is specified in the `RULES` section, between Lines 15–22.

Mainly, the same hedge symbols (`"[[...]]"`) are used to mark up slots in Java^{-} programs. However, to distinguish `IntegerValueSlots` and `StringValueSlots` from the others deterministically, `"[i:...:i]"` and `"[s:...:s]"` are used as alternative markup. This is required by the underlying parsing approach of EMFText which does not support resolving ambiguities in its specification language.

Based on the specifications presented so far, an implementation stub of a composition system can be generated.

4.3.2. The Business Application Generator in Reusew*air*

From the component model and the extended grammars, Reusew*air* generates the basic stubs and API in Java. Subsequently, these are used to realize the BAF code generator.

Listing 4.10 contains the basic composition program. It loads the business model from the specification file and then subsequently processes `RoleDefinitions` in the model (cf. Lines 7–40). During each iteration, a new parametrizable copy of the basic `person.rjava` is loaded and instantiated as an `ICompilationUnit` fragment. The basic version of this fragment is shown in Listing 2.3. It is slightly modified to be used with Reusew*air* and Java^{-}. Like in the COMPOST case, the `Pfx` slot is moved to its own subexpression (`[s:Pfx:s]`). A parametrization of basic slots occurs between Lines 17–29. If super roles are defined in the processed model, the first one is chosen as a super class, which is similar to the COMPOST case. Otherwise, `Person` becomes the super class. After parametrization, fields and the corresponding field accessors are added to the fragment (cf. Lines 33–36). In the last line, the parametrized and extended result is printed back to a source file.

In the following, the extension of the `ICompilationUnit` fragment with `addFields` is described since it is the most complex extension. The others are implemented in comparable ways and are therefore omitted. Listing 4.11 provides the composition program for adding a `RoleDefinition`'s properties to the generated implementation class. Considering hooks, Reusew*air* has no support to address hooks in their context (e.g., using a regular path language or pattern matching). Especially, the C_{m}SL has no means for specifying hooks. Instead, any occurrence of a nonterminal in a list can be used as an implicit hook. For finding and extending such points, Reusew*air* provides its generated visitor API. The API has a `visit` stub implementation for each fragment type declared in the FCM. In the `addFields` composer, the corresponding visitor is provided as an inline class declaration between Lines 7–37. It traverses a given Java^{-} fragment until a method declaration is reached and iterates over all `PropertyDefintions` of the given role. During each iteration, it converts a property to a corresponding field fragment which is finally added to the class before its first `MethodDeclaration`.

Depending on a given property default value, the algorithm uses different variants of the field declaration—with or without a `Value` slot (cf. Lines 18– 27). The variant with `Value` slot is provided as an alternative to the optional default constructor because Java^{-} does not consider constructor declarations in its grammar.

Finally, the *prepend* composer is called in Line 34 so that the field is inserted at the position before the visited `IMethodDeclaration`.

```
public void compositionProgram() {

   /* load the business model with EMFText */
   BusinessModel bm = loadBusinessModel(new File("baf/in/model.bm"));

   /* for each role definition, generate a Java class */
   for(RoleDefinition role: bm.getRoleDefinitions()){

      /* load the person fragment with EMFText and set up basic parameters */
      ICompilationUnit fgmt = ICompilationUnitImpl.load("file:baf/in/person.rjava");
      String currentName = role.getName();
      IIdentifier ident = createIdentifier(currentName);
      IStringValue sIdent = IStringValueImpl.load("\"" + currentName + "\"");
      IStringValue sPfx = IStringValueImpl.load("\"\\n\"");

      /* bind type-signature related slots and the Pfx slot */
      fgmt.bind("Type",ident);
      fgmt.bind("TypeName", sIdent);
      fgmt.bind("Pfx",sPfx);

      /* bind the super type slot according to super roles declaration */
      if(role.getSuperRoles().isEmpty()){
         IQualifiedName sprIdent = createQualifiedName("Person");
         fgmt.bind("Super",sprIdent);}
      else {
         String sprRoleName = role.getSuperRoles().get(0).getName();
         IQualifiedName sprIdent = createQualifiedName(sprRoleName);
         fgmt.bind("Super",sprIdent);
      }

      /* extend the parametrized person fragment with fields, field accessors
         and print statements*/
      addGetters(fgmt,role);
      addSetters(fgmt,role);
      addFields(fgmt,role);
      addPrintStmts(fgmt,role)

      /* print the results back with EMFText */
      fgmt.print("baf/out/" + currentName + ".rjava");
   }
}
```

Listing 4.10: The BAF code generator for business roles as Reuse*wair* composition program.

```java
private void addFields(ICompilationUnit fgmt, RoleDefinition role){

   /* for each property definition, we add a field to the given class */
   for(final PropertyDefinition prop: role.getProperties()){

      /* creating a new inline JavaVisitor implementation */
      JavaVisitor extender = new JavaVisitor() {

         /* visiting a member method of the given class */
         public boolean visit(IMethodDeclaration decl){

            /* initializing fragments to be bound */
            IIdentifier propIdent = createIdentifier(prop.getName());
            IQualifiedName propType = createQualifiedName(prop.getType());
            IAttributeDeclaration toBeInserted = null;

            /* no default value in the model: load fragment without value slot */
            if(prop.getType().getDefault() == null)
               toBeInserted = IAttributeDeclarationImpl
                     .load("private [[Type]] [[FieldName]];");
            /* default value in the model: load fragment with Value slot */
            else {
               toBeInserted = IAttributeDeclarationImpl
                     .load("private [[Type]] [[FieldName]] = [i:Value:i];");
               IIntegerValue iValue = createIntegerValue(prop.getType().getDefault());
               toBeInserted.bind("Value",iValue);
            }

            /* bind type and field names */
            toBeInserted.bind("Type", propType);
            toBeInserted.bind("FieldName",propIdent);

            /* insert the new field before visited member method and stop visiting */
            decl.prepend(toBeInserted);
            return false;
         }
      };

      /* trigger the composition by passing the visitor object to the fragment */
      fgmt.accept(extender);
   }
}
```

Listing 4.11: Adding fields for each property of a given `RoleDefinition`.

4.3.3. Evaluation

In this section it has been shown, how Reusew*air* can be used to implement the BAF backend. The generator implemented with Reusewair nearly corresponds to what was sketched in Chapter 2. The emitted code is nearly equivalent to the expected code—except support for slots within strings and mixin-based inheritance. While the former is due to the lacking support of value slots in Reusew*air*, the latter is due to the lacking support of interface declarations in Java^{-}. Although the Reusew*air* implementation was improved by this author towards a support of the current version of EMFText for parsing and printing, and the implementation still has some flaws, U-ISC could be a viable approach for implementing the code generator in reasonable time and with reasonable effort. The C$_{\mathrm{m}}$SL is a small language with a few language concepts that have to be learned. Also, the concept of slotification of AST grammars is understandable by developers. It should also be mentioned that [Henriksson 2009] also introduced *embedded invasive software composition (E-ISC)* as an extension to U-ISC with support for adding complex composers (e.g., *import module*) as language concepts to the base grammar. This enables ISC macros with a visitor-based implementation.

On the downside, the approach has some weaknesses. Most obviously, it lacks opportunities to specify hook candidates in the fragment component model (cf. Definition 4.1 in Section 4.1.2). While the sets of slot nonterminals $\mathcal{S}$ and fragment nonterminals $\mathcal{F}$ are well-defined by the `slotify` and `fragtypes` operators in the C$_{\mathrm{m}}$SL, the set of hook nonterminals $\mathcal{H}$ always contains all list nonterminals in G_{base}, since C$_{\mathrm{m}}$SL lacks means to specify it.

Furthermore, the compartments of the point identification function $\int$ are not accessible—especially $\int_{\mathcal{S}}$ and $\int_{\mathcal{H}}$. This has several implications. Slotification is the only way to identify slots in U-ISC. However, this requires the grammar to be transformed (cf. Section 4.3.1) and can easily break existing tooling that relies on the G_{base} metamodel and consequently does not work with instances of G'_{base} and G_{reuse}. In the BAF example, this is not of importance. Because no existing tool implementation is used, everything can be generated from the specifications at hand. But consider the user may have wanted to use some code analysis algorithm during composition which originally was written for the base grammar and is broken for G_{reuse}. Conflicts between productions in the reuse grammar are a second problem due to the missing $\int_{\mathcal{S}}$ and slotification. For example, in the concrete syntax grammar in Listing 4.9 different hedge symbols have been specified to distinguish slots for integers and strings, where it would be good to have the same syntax for all slot declarations.[7]

Due to the lack of means to specify the hook identification function $\int_{\mathcal{H}}$, hooks cannot be identified easily via the generated component model API. The identification has to be implemented manually by the developer of the composition visitors. The realization of context-dependent hooks via $\int_{\mathcal{H}}$ is also an issue that has to be handled manually in the visitors. Since Reusew*air* only supports a depth-first traversal style and the access to context information and to the AST under composition is somewhat limited by the generated API, it seems very likely that the API does not

[7]A solution to this is replacing both with a single value slot nonterminal in the C$_{\mathrm{m}}$SL. However, this indirectly introduces slots for any kind of value instead of string and integer values only.

help to realize more complex composition systems with complex FCMs such as COMPOST. *The approach presented in this thesis overcomes these problems by using RAGs to model complex context and by composition strategies driven by attribute dependencies.*

4.4. Universal Invasive Software Composition for Graph Fragments

In recent research [Johannes 2011], the CFG-based approach of U-ISC was extended to also support the creation of composition systems for visual and textual modeling languages. The approach is called U-ISC/Graph and supports fragment languages based on the EMF [Eclipse Foundation 2013b] and the EMF standard metamodeling language *Ecore*, which is frequently used as an abstract-syntax specification language in the modeling domain. In the thesis [Johannes 2011], it is claimed that U-ISC/Graph enables component-based model-driven software development (MDSD) by introducing three novel concepts to ISC:

Fragment collaborations. The first contribution are *fragment collaborations* (*FraCols*). FraCols define component models on an abstract, language-independent level. The main concepts are *fragment roles* which declare abstract component interfaces based on the notion of *port types*, and *fragment collaborations* which declare directed collaborations between port types in the abstract fragment role definitions (cf. [Johannes 2011, p. 39]).

Graph fragments. The second important concept of U-ISC/Graph is the support of *graph fragments*. Although the naming suggests that general graphs can be handled by U-ISC/Graph, only a subset of the class of directed graphs with special properties is considered. Most importantly, graph fragments need to have a *designated spanning tree* making all of its nodes reachable from the fragment root. As the approach is restricted to component languages based on EMF/Ecore, the designated spanning-tree property is ensured by an adequate use of the Ecore's containment reference type. Other edges in graph fragments that do not belong to the spanning tree are called *non-containment references* and may connect nodes in arbitrary ways restricted by the component-language metamodel. For the specification of graph-fragment-based component models over such data structures, the *Reuse EXtension language for Component Model Configuration (REX$_{CM}$)* is provided. The basic concepts of REX$_{CM}$ will be explained briefly in Section 4.4.1.

Universal composition language. Finally, U-ISC/Graph provides the *Universal Composition Language (UCL)* for the composition of EMF-based fragments. UCL is a declarative language with a graphical diagrammatic notation (cf. [Johannes 2011, p. 104]). Fragments are denoted as named boxes with ports represented by circles with different line and fill styles depending on the port type. Composition steps are declared by drawing composition links between contributing and receiving ports or between configuring ports. In contrast to the other ISC approaches discussed so far, the composition operators are selected automatically by the type of link and the kind of points grouped by the port.

The following section explains the ISC refinements by U-ISC/Graph to support graph fragments. Afterwards, the *Reuseware* composition framework is applied to implement the composition system of the BAF example. Reuseware is the standard implementation of U-ISC/Graph.

4.4.1. Compositional Points

To provide a better integration with graph fragments, U-ISC/Graph refines the model of ISC by modifying the concepts of slots and hooks, and introducing additional kinds of compositional points. It can be distinguished between *tree-compositional* and *graph-compositional* points. It follows a short description of the tree-compositional points available in REX_{CM}.

Hook. The concept of hooks in U-ISC/Graph is equivalent to the concept of hooks in COMPOST: hooks can be explicit/declared in a fragment or they can be implicit for list-like concepts.

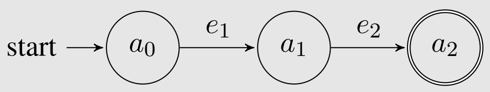
Prototype. A prototype is a designated node within a fragment whose contained subtree is meant to be copied—perhaps multiple times—and to be inserted into list hooks or to replace declared hooks during composition.

Value hook. While general hooks are nodes or lists, value hooks are string properties of nodes that can be manipulated by string replacement or transformation.

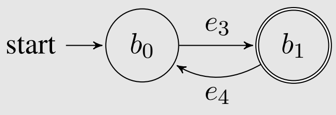
Value prototype. These special kinds of prototypes are values of node properties that can be used for composing them with value hooks.

The following use case exemplifies the basic principle of the *hook<–prototype* composition relation in U-ISC/Graph.

Example 4.6 (*hook<–prototype* composition of finite-state machines.).
Consider the list of states and edges in a finite-state machine SM_{A} (cf. Figure 4.5(a)) as implicit *hooks* and consider the states and edges of a finite-state machine SM_{B} (cf. Figure 4.5(b)) as *prototypes*.



(a) SM_{A} with edge and state-list hooks.



(b) SM_{B} with b_0 and e_3 prototypes.

Figure 4.5.: The state machine fragments SM_{A} and SM_{B} before composition

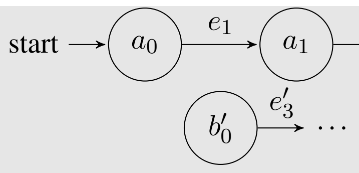


Figure 4.6.: SM_{A} after insertion of b'_0 (a copy of prototype b_0) and e'_3 (a copy of e_3).

A composition program can replicate arbitrary states of SM_{B} in SM_{A} by creating copies of these states and edges, and bind/extend the hooks of SM_{A} with these copies. Figure 4.6 shows the result of composition steps which replicated the state b_0 of SM_{B} and its outgoing edge e_3 in SM_{A} (it is assumed that the composition system also replicates the relation of e'_3 as an outgoing edge of b'_0). Observe that the above figures are graphical representations of Ecore-based models. Thus, in the implementation states and edges are objects that are child nodes of a state-machine object which is the distinct root node of the model. ◇

In the following, the graph-compositional points that go beyond the original concepts of ISC are described.

Slot. The meaning of slots in U-ISC/Graph is different from the notion of slots in U-ISC or in BETA. A slot is the endpoint of non-containment (reference) edge, which can be changed during composition so that the edge points to a different target object after composition.

Anchor. An anchor is a designated node within a fragment that is allowed to become an endpoint of a reference edge during composition, i.e., binding an anchor node to a slot node is equivalent to switching the endpoint of a reference which points from some node to a slot node to an anchor node.

Example 4.7 extends the above U-ISC/Graph composition example on finite-state machines with graph-compositional elements.

Example 4.7 (*slot<–anchor* composition of finite-state machines.).
As an example for the *slot<–anchor* composition relation (also called *configuration*) reconsider the state-machine composition system of Example 4.6. After the extensions of the state and edge hooks of SM_{A}, b'_0 and e'_1 are still not connected with any other state in SM_{A}. Thus, b'_0 is not reachable from any other state in SM_{A} and e'_1 is a dangling edge.

To change this, a U-ISC/Graph composition system can provide *slots* for outgoing edges of each state of a finite-state machine. Hence, since e_1 is an edge between a_0 and a_1, and e_2 is an edge between a_1 and a_2 in SM_{A} as depicted in Figure 4.5(a), the endpoints of e_1 and e_2 are recognized as slots and could be bound to the anchor b'_0. Figure 4.7 below shows the result of the binding of anchor b'_0 to the endpoint slot of e_1.

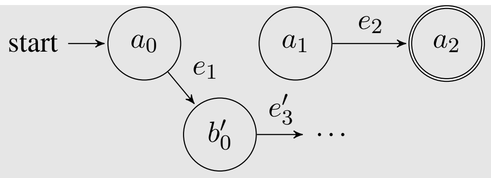


Figure 4.7.: SM_{A} after binding the endpoint slot of e_1 to anchor b'_0.

Now transition e_1 points to b'_0. However, this is still problematic since a_1 and a_2 are no longer reachable. To correct this, in a next composition step the endpoint slot of e'_3 can be bound to the anchor state a_1. The result of this composition is shown in Figure 4.8.

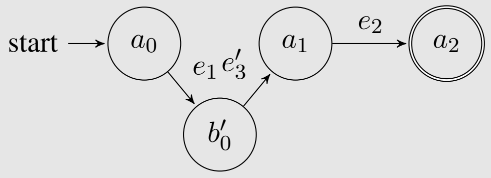


Figure 4.8.: SM_{A} after binding the endpoint slot of e'_3 to anchor a'_1.

After the last composition step, SM_{A} is a correct state machine without dangling edges and all states reachable. Observe that since well-formedness is not considered during composition, also incorrect state machines could be constructed by Reuseware. ◇

Subsequently, a U-ISC/Graph system for the BAF example is developed using Reuseware.

4.4.2. The Business Application Generator in Reuseware

To implement the BAF composition scenario in Reuseware, three substeps need to be conducted. First, a specification for the required fragment collaborations has to be developed. Second, the actual FCMs for the business DSL and Java have to be developed. Third, the composition program has to be implemented. Subsequently, the steps are explained in detail.

Fragment Collaborations

Initially, to implement the code generator with Reuseware [Software Technology Group and DevBoost GmbH 2013b], a fragment-collaboration specification has to be provided using the `fracol` language. It can be regarded as a manifestation of abstract component roles and their interplay, and thus declares the most basic collaboration relations occurring in a composition system. In Reuseware, fragment collaborations can be specified by the developer himself or, alternatively, an existing `fracol` specification can be reused. In Listing 4.12, a simple `fracol` specification supporting the BAF example is shown. It declares three fragment roles:

```
 1  fracol Simple.Fragment.Collaboration {
 2     fragment role Receiver {
 3        dynamic port type Reception;
 4     }
 5     fragment role Contributor {
 6        dynamic port type Contribution;
 7     }
 8     fragment role Configurable {
 9        dynamic port type Config;
10     }
11     contributing association Contribution {
12        Contributor.Contribution --> Receiver.Reception
13     }
14     configuration association Config {
15        Configurable.Config --> Configurable.Config
16     }
17  }
```

Listing 4.12: `fracol` specification declaring three fragment roles.

`Receiver`, `Contributor` and `Configurable`. Each of them provides a named *port type*[8] which denotes an abstract aggregation of compositional points (e.g., hooks and anchors) of a fragment. Reuseware supports two kinds of port types. *Dynamic ports* may be instantiated arbitrarily often while *static ports* exactly occur once in a fragment. In the lower part of the specification (Lines 11–16) *association links* are introduced. *Contributing associations* specify in which direction fragments can be composed. Above, Line 16 declares a link from the `Contribution` port of the `Contributor` fragment role to the `Reception` port of the `Receiver` fragment role. In a concrete composition scenario, this restricts possible compositions as follows: `Receiver` fragments can only be extended with content from `Contributor` fragments via the respective ports. Still, a fragment can be `Contributor` and `Receiver` both at the same time. Finally, *configuration links* define the direction of configuration actions during composition.

Component Model

Since fragment collaborations are abstract models of the composition interface, concrete FCMs in Reuseware have to be specified as realizations of these interfaces. For this purpose, the REX_{CM} is provided. A REX_{CM} specification declares compositional points of the component language and provides a mapping between points and `fracol` ports. The BAF has two component languages—the business DSL and Java. Hence, two REX_{CM} specifications have to be developed for the composition system. Listing 4.13 shows the component model of the Business DSL. Lines 1–4 declare the name of the component model specified, the `fracol` model that is realized

[8]Although the keyword `type` is used, port-type declarations do not have a fragment- or value type. A port-type declaration only specifies that fragment components “playing” the respective fragment role may or must provide a port or a group of ports mapped to the declaration’s name.

```
 1 componentmodel BusinessWeaving.BusinessComponentModel
 2 implements BusinessWeaving.collaborations.Simple
 3 epackages <http://www.emftext.org/language/businessmodel>
 4 rootclass businessmodel::BusinessModel {
 5
 6 fragment role Receiver {
 7    port type Reception {
 8       businessmodel::BusinessModel.roleDefinitions is hook {
 9          port = $'roles'$
10 }}}
11
12 fragment role Contributor {
13    port type Contribution {
14       businessmodel::RoleDefinition is value prototype {
15          port = $name$
16          point = $'name'$
17          value = $name$
18       }
19       businessmodel::PropertyDefinition is value prototype {
20          port = $eContainer().oclAsType(businessmodel::RoleDefinition).name+'->'+name$
21          point = $'Getter'$
22          value = $'get' +
23             name.toUpperCase().substring(1,1) + name.substring(2,name.length())$
24       }
25       businessmodel::PropertyDefinition is value prototype {
26          port = $eContainer().oclAsType(businessmodel::RoleDefinition).name+'->'+name$
27          point = $'Setter'$
28          value = $'set' +
29             name.toUpperCase().substring(1,1) + name.substring(2,name.length())$
30       }
31       businessmodel::PropertyDefinition is value prototype {
32          port = $eContainer().oclAsType(businessmodel::RoleDefinition).name+'->'+name$
33          point = $'Field'$
34          value = $name$
35       }
36       businessmodel::PropertyDefinition is value prototype {
37          port = $eContainer().oclAsType(businessmodel::RoleDefinition).name+'->'+name$
38          point = $'Type'$
39          value = $type.getTargetType()$
40       }
41 }}}
```

Listing 4.13: REX_{CM} specification declaring compositional points in the Business DSL and connecting them to the fragment roles of the `fracol` given in Listing 4.12.

(`implements` keyword), the namespace of the component language and a class in the language metamodel[9] that is used as `rootclass` in the context of the specification. Since the `fracol` from Listing 4.12 is implemented, the `Contributor` and `Receiver` fragment roles are reused. In Lines 6–10, a `Receiver.Reception` port is specified. As defined by an expression base on the Object Constraint Language (OCL) [Object Management Group (OMG) 2006], the port is named `roles` and provides a hook on the `roleDefinitions` list which enables the

[9]The interested reader may inspect the metamodel of the Business DSL in Appendix A.1.2.

composition-system users to extend the business model with new `RoleDefinitions`. In terms of the generic FCM introduced by Definition 4.1 in Section 4.1.2, the OCL is used as a specification language for the hook-identification function $\int_{\mathcal{H}}$. Generally, in REX$_{\text{CM}}$, OCL expressions can be used to determine the name of points and to determine the point-candidate sets $\mathcal{S}$ and $\mathcal{H}$. Since the OCL expressions are evaluated w.r.t. the underlying Ecore metamodel, $\int$ can be context sensitive and point names can be derived from the context. The remaining point types supported by the U-ISC/Graph model are basically declared in similar ways. The declaration of value prototypes is explained below.

Lines 12–41 specify multiple prototype ports as members of the port type `Contributor.-Contribution`. In Reuseware, prototypes are the only way to provide the components or values that are used to vary or extend other fragments at their hooks. Hence, in the BAF example they are required to access the content of the business model and make it available to the composition program (i.e., the BAF code generator). The first prototype definition in Lines 14–18 defines `RoleDefinitions` (e.g., "Employee") as *value prototypes*. The representative name of the port is declared via the `port` keyword and the respective OCL expression at the right-hand side. In the case of `RoleDefinition` prototypes, the `RoleDefinition.name` property is used to compute the name. The same property is also used to compute actual `value` of the value prototype. The point name used for matching the points of the source port with the points of the target port of the target fragment is specified using the `point` keyword and the OCL expression on the right-hand side. In the case of the `RoleDefinition` prototype, the name is "name".

The four remaining definitions declare `PropertyDefinitions` as value prototypes (according to the BAF metamodel, each `RoleDefinition` contains an arbitrary number of `PropertyDefinitions`). These are required to access and transform property values of `PropertyDefinitions` which are themselves needed to parametrize the Java templates used by the BAF code generator. The value of `PropertyDefinition.name` is provided twice as a value prototype (cf. Lines 19–24 and 25–30). The declared point names "Getter" and "Setter" anticipate their later use in the composition program to parametrize the names of the corresponding Java accessor methods. Hence, the prototype values are derived by converting the value of `PropertyDefinition.name` to start with an upper-case letter prepending "get" or "set" respectively. Additionally, `PropertyDefinition.name` is mapped to the point name "Field" (cf. Lines 31–35). It provides the value of `PropertyDefinition.name` directly as it is declared in the BAF model and will be bound to the field declarations and accesses by the composition program. The value of `PropertyDefinition.type` is provided as a value prototype with the according point name "Type" (cf. Lines 36–40). The value of the "Type" point is directly obtained from the BAF model without transformation. All the four `PropertyDefinition` prototypes are grouped by a port named by using the scheme

```
[RoleDefinition.name]->[PropertyDefinition.name],
```

as declared by the `port` OCL expressions.

Reuseware is an interpretative approach to fragment composition. Hence, the REX$_{\text{CM}}$ specification in Listing 4.13 is interpreted by the composition engine each time a business-model fragment

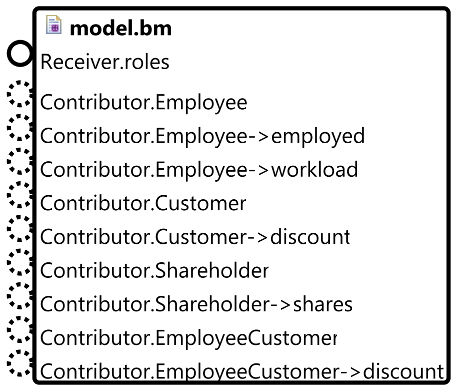

model.bm
Receiver.roles
Contributor.Employee
Contributor.Employee->employed
Contributor.Employee->workload
Contributor.Customer
Contributor.Customer->discount
Contributor.Shareholder
Contributor.Shareholder->shares
Contributor.EmployeeCustomer
Contributor.EmployeeCustomer->discount

Figure 4.9.: Graph-fragment box for the business model in Listing 2.2 based on the UCL notation of Reuseware.

is used in a composition program to find points and ports of the fragment. Since Reuseware provides a graphical composition language—the UCL—fragments are visualized using graphical metaphors, which is reminiscent of the classical box-port notation of UML component diagrams (cf. [Object Management Group (OMG) 2011d]).

Figure 4.9 shows the UCL representation of the textual business model `model.bm` from Listing 2.2 which contains the four `RoleDefinitions` `Employee`, `Customer`, `Shareholder` and `EmployeeCustomer`. The diagram can be read as follows. The business model is represented by the round-cornered box. Dotted circles attached to the box denote ports with value prototypes (e.g., the `Contributor.Employee` value prototype) or mixed kinds of ports while continuously drawn circles denote receiving ports with normal hooks (e.g., the `Receiver.roles` hook).

Since the code target of the generator is Java, in the next step, a REX_{CM}-based component model for this language has to be provided. Therefore, as component-language implementation, the Java Model Parser and Printer (JaMoPP) [Software Technology Group and DevBoost GmbH 2013a] is used—an EMFText-based Java metamodel for the EMF. It can be imported by a REX_{CM} specification from the according JaMoPP Eclipse plug-in. The FCM for JaMoPP can be inspected in Listing 4.14. The above-defined `fracol` specification is reused as a collaboration model, while the root concept the component model is associated with is `JavaCompilationUnit`—a container for Java classes and interfaces.

The first definition of a `Receiver`-role binding in Lines 6–28 declares two ports. The first `Reception` port provides a *hook* for the "members" list in class fragments and therefore enables the extension of the class body with new method and field declarations. The second `Reception` port provides two *value hooks*. First, the `NamedElement.name` property is declared conditionally as a value hook if the property ends with `"_Slot"`. If, for example, a class declaration `class A_Slot {...}` is encountered by Reuseware, its name is interpreted as a

```
componentmodel BusinessWeaving.JavaComponentModel
implements BusinessWeaving.collaborations.Simple
epackages <http://www.emftext.org/java>
rootclass java::containers::CompilationUnit{
   // Common Receiver ports & points supported by all Java CompilationUnits
   fragment role Receiver {
      port type Reception {
         java::classifiers::Class.members is hook {
            port = $'members'$
         }
      }
      port type Reception {
        java::commons::NamedElement.name is value hook
         if $name.endsWith('_Slot')$ {
            port = $'parameters'$
            point = $name.substring(1,name.indexOf('_Slot'))$
            begin idx = $0$
            end idx = $name.length()-1$
         }
        java::references::StringReference._value is value hook
         if $value.contains('[[')$ {
            port = $'parameters'$
            point = $value.substring(value.indexOf('[[')+3,value.indexOf(']]'))$
            begin idx = $value.indexOf('[[')$
            end idx = $value.indexOf(']]')+1$
         }
      }
   }
   // Contribution ports & points for CompilationUnits containing a Mixin-Class
   // providing prototypes for the BAF getters, setters and fields
   fragment role Contributor
     if $classifiers->asSequence()->first().name.startsWith('Mixin')${
      port type Contribution {
         java::members::Method is prototype {
            port = $ufi.trimExtension()$
         }
         java::members::Field is prototype {
            port = $ufi.trimExtension()$
         }
         java::expressions::Expression is prototype {
            port = $ufi.trimExtension()$
         }
         java::statements::Statement is prototype
          if $not oclIsKindOf(java::classifiers::ConcreteClassifier)${
            port = $ufi.trimExtension()$ }
      }
   }
}
```

Listing 4.14: REX$_{CM}$ specification declaring compositional points in Java and connecting them to the fragment roles of the `fracol` given in Listing 4.12.

value hook.[10] The actual `point` name is derived by removing the "`_Slot`" suffix from the value. Hence, the name "`A_Slot`" of a `NamedElement` provides a value hook "`A`". A replacement rule can be specified using the `begin idx` and `end idx` rules, which states that the hook can be bound to a value by replacing the characters between these indexes. Consequently, "`A_-Slot`" will be replaced entirely. The second point declaration of the `Reception` port begins in Line 20. It provides conditional value hooks that may occur in Java strings and is declared on the `StringReference.value` property. According to the `point` declaration, a string hook sticks to the format "`...[[A]]...`", where `A` denotes its name. During composition, according to the `begin idx` and `end idx` values, the hook declaration is replaced.

In Lines 31–47 the contributing parts—the prototypes—of Java fragments in the BAF example are declared. The binding to the `Contributor` fragment role is conditional and depends on if the `CompilationUnit`'s first classifier name starts with "`Mixin`" (cf. Line 32). It declares `Fields`, `Methods`, `Expressions` and `Statements` (excepted `ConcreteClassifier` statements) as *prototypes* grouped by a port named like the fragment file of the enclosing `CompilationUnit`. In this Reuseware-based implementation of the BAF example, mixins are templates of Java fields and methods grouped by a class which are meant to be parametrized with field and method names provided by the value prototypes of `RoleDefinitions` and `PropertyDefinitions`, and that should extend the basic `Person` fragment.

Developing the Composition Program

After having specified the required component models for the BAF and JaMoPP metamodels, the composition system is ready to be used. As a first step, the parametrization of the mixin fragment shall be investigated, since the REX_{CM} component model differs from the one that was assumed in the original BAF composition scenario in Chapter 2. The mixin template is shown in Listing 4.15. It contains a generic declaration of a private field and according generic set and get operations for that field.[11]

Figure 4.10 shows a simple UCL composition program which binds the *value prototypes* of the discount `PropertyDefinition` to the corresponding *value hooks* in the Java fragment in Listing 4.15. A binding of the `Contributor.EmployeeCustomer->discount` port (source) to the `Receiver.parameters` port (target) in `Mixin_frgmt.java` is specified by simply connecting the respective ports by drawing an arrow in the UCL editor. The Reuseware

[10] In the original BAF example introduced in Section 2 syntactic hedges were used to mark up slots. However, in the Reuseware-based implementation suffix recognition was chosen instead for technical reasons.

[11] The code shown in Listing 4.15 was modified by the author and would not work with the developed scenario. Due to the realization of the type analysis in JaMoPP, it was technically not possible to declare `Type_Slot` as a value hook, because JaMoPP resolves the type names of methods and fields as *non-containment* references. However, in the business model, Java types are not resolved, but represented by qualified names. Consequently, the *slot<–anchor* composer was not used and not considered in the component model (cf. Listing 4.14). A workaround with a nested class declaration with a type name value hook (`private class Type_Slot {}`) was employed instead to perform the example. However, this produces incorrect code because the composition system prints fully qualified type signatures, which, for example, results in the type name `Mixin.int` instead of `int` in Listing 4.16.

```
class Mixin {
   private Type_Slot Field_Slot;

   public Type_Slot Getter_Slot() {
      return Field_Slot;
   }
   public void Setter_Slot
      (Type_Slot Field_Slot) {
      this.Field_Slot = Field_Slot;
   }
}
```

Listing 4.15: Template `Mixin_frgmt.java` prepared for composition with Reuseware.

```
class Mixin {
 private int discount;

 public int getDiscount() {
   return discount;
 }
 public void
   setDiscount( int discount) {
   this.discount = discount;
 }
}
```

Listing 4.16: `Mixin_frgmt.java` parametrized with the `discount` port.

```
public class name_Slot extends Person {

   public String asString(){
      String v = "[[name]]";
      v+= "[[Pfx]]id:" + getID();
      v+= "[[Pfx]]name:" + getName();
      return v;
   }
}
```

Listing 4.17: JaMoPP-based `Person_frgmt.java` with value hooks.

composition engine then automatically binds the points grouped by the source port to the points of the target port by matching their names and evaluating the respective `value` OCL expressions (cf. the REX$_{\text{CM}}$ specification in Listing 4.13). The result of the single composition step of Figure 4.10 is shown in Listing 4.16 where the field and method name hooks have been bound to the corresponding property name ("discount").

While the UCL diagram in Figure 4.10 shows how the parametrization of accessory methods and fields basically works in Reuseware, it only stands as a single step of the integrated code generation picture. In Figure 4.11, the full UCL composition program is shown (the contents of gray boxes are printed after composition). `Mixin_frgmt.java` is replicated and instantiated five times—one for each individual `PropertyDefinition`. Furthermore, for each `RoleDefinition`, a copy of the fragment `Person_frgmt.java` (cf. Listing 4.17) has been added manually to the composition program. The `Person_frgmt.java` copies are the targets that should be printed to the file system after the composition has been finished. They are parametrized with the value prototypes provided by the name prototype of the `RoleDefinitions` which will be mapped to the class name, and extended with the parametrized member declarations of the `Mixin_frgmt.java` copies.

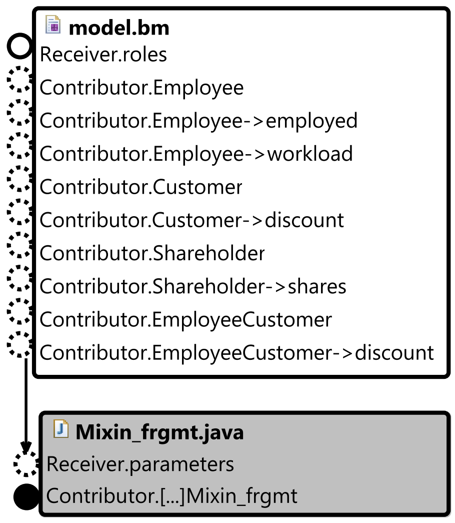

model.bm
Receiver.roles
Contributor.Employee
Contributor.Employee->employed
Contributor.Employee->workload
Contributor.Customer
Contributor.Customer->discount
Contributor.Shareholder
Contributor.Shareholder->shares
Contributor.EmployeeCustomer
Contributor.EmployeeCustomer->discount
Mixin_frgmt.java
Receiver.parameters
Contributor.[...]Mixin_frgmt

Figure 4.10.: A UCL composition program instantiating the template of Listing 4.15.

Operationally, the composition diagram in Figure 4.11 can be read as follows:

1. For each `PropertyDefinition` in the BAF model, parametrize a different `Mixin_-frgmt` copy with the *value prototypes* for field and method names.

2. For each `RoleDefinition` in the BAF model, parametrize a `Person_core` copy with the name *value prototype* from the BAF model and also set the `Pfx` slot to the according string-prefix value.

3. Extend the respective `members` *hooks* in the parametrized `Person_core` fragments of Step 2 with the resulting *prototypes* of Step 1.

4. Finally, print the resulting `Person_core` Java files with the according file names, e.g., `EmployeeCustomer.java`.

The interested reader can find the final result of the composition described in this section in Listing A.6 in Appendix A.2.

4.4.3. Evaluation

In this section, it has been shown how U-ISC/Graph and Reuseware can be used to implement a template-based code generator. Because of the complexity of the approach, the final evaluation discussion has been split into four subparagraphs on the technical evaluation, the support of ISC features, the application on the case study and a general analysis of the involved specification DSLs and their interplay.

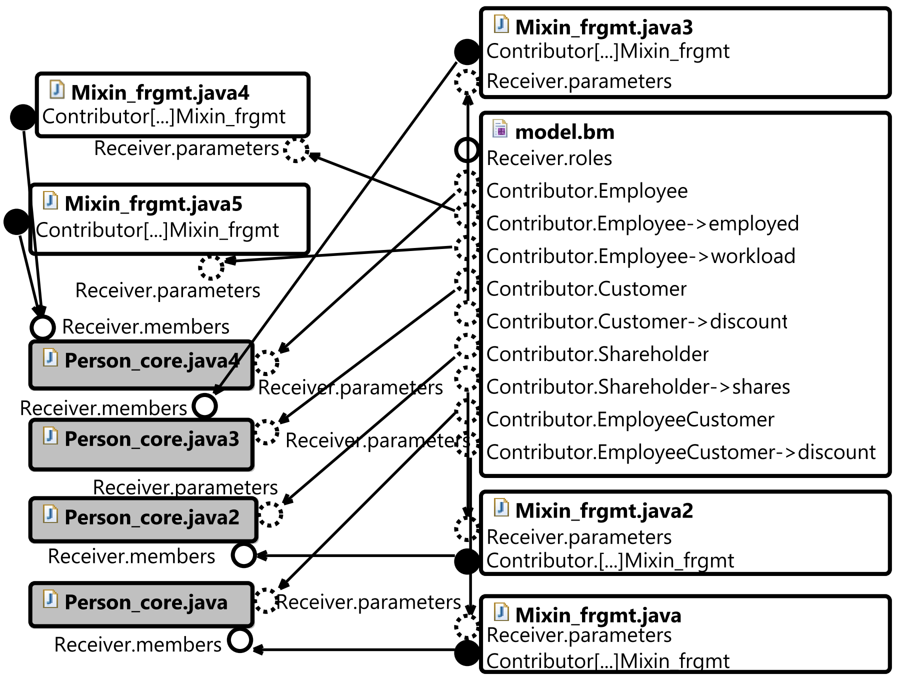


Figure 4.11.: The full composition program in UCL.

Technical Evaluation

The Reuseware tooling makes a technically mature impression, the provided editors—textual or graphical—are quite stable and usable. Due to its strong bundling with the EMF, with Reuseware it is easy to reuse metamodels of graphical and textual languages from EMF-based applications, and to provide reuse-extensions for them. In comparison with the COMPOST and Reusew*air* approaches discussed in Section 4.2 and 4.3, Reusware by far provides the most comprehensive set of composition tooling and complex specification languages. Hence, from this perspective, Reuseware could have been the best choice to build upon.

However, depending on the application scenario, some of these strengths are also weaknesses. Its tight integration with the EMF has the downside of inheriting sometimes undesirable technical properties. For example, in a packaged application that should be delivered to a customer, the whole EMF core infrastructure and its dependencies have to be delivered too, which can be problematic if the composition tooling should run in headless mode or in heterogeneous and automatized tool chains—a typical use case for template and macro processors. Furthermore, directly reusing metamodels from deployed language plug-ins can introduce direct dependencies to the applications that are built around the imported metamodel and transitive dependencies to other plug-ins causing unforeseen interactions between the dependent plug-ins and the composition system, which carries the potential to make the whole system fragile.

Support of Composition Features

Also conceptually there are points in favor and against the concepts of the specification and composition languages provided by Reuseware. In comparison to the C_mSL of U-ISC/Reuse*wair*, the REX_CM language provides composition-system developers with the full freedom to define slot-, hook- and fragment-identification functions—$\int_{\mathcal{S}}$, $\int_{\mathcal{H}}$, $\int_{\mathcal{F}}$—using the OCL. Moreover, it introduces concepts that are not in the standard model of ISC. The prototype concept balances the relation between source and target points (i.e., hooks) by supporting subtree extraction from a fragment, where Reusew*air* does not provide such a concept. However, COMPOST provides *composite fragment boxes* [Heuzeroth et al. 2006], a concept for hierarchically grouping and organizing fragment components. Composite boxes are similar but not equivalent to prototypes, since the composites can be arbitrarily nested. In contrast, the concepts of fragment roles and reference-anchor points do not have a counterpart in any of the two other approaches. Especially anchors help to *embed* an inserted fragment in its new context, which is essential if graphical models are composed in interactive usage scenarios and thus resemble manual edit operations like drawing an edge from one node to another. In typical composition scenarios with textual languages, such context-sensitive computations are usually realized by the compiler. The compiler employs reasonably complex code-analysis algorithms (e.g., name, type and control-flow analyses), which also cannot be reused unless a reusable technology like RAGs is used and available via an interface through the composition system. In general, it seems difficult to consider complex semantic rules which go beyond a key–value (i.e., *slot<–anchor*) relation, for example if scoping rules have to be considered.

Application on the BAF Use Case

For the implementation of general template languages and the corresponding template engines, it becomes obvious from the case study in this section that Reusware and U-ISC/Graph are not a suitable option for the following main reasons:

- In Reuseware, a matching of compositional points (e.g., hooks and prototypes) is done implicitly by name while in template engines this is done explicitly in the templates.

- A template engine should be only aware of its target language, but not of the model language. However, in U-ISC/Graph a REX_CM component model for all participating languages has to be provided making it problem specific on the language level.

- The UCL composition language lacks any kind of recursion, which makes the U-ISC/Graph approach even problem-specific on the instance level (i.e., every single component and composition step has to be enumerated explicitly).

Adequateness of the Provided Specification Languages

Creating the Reuseware-based BAF code generator was not an easy task. It turned out that the approach seems to have an inherently complex learning curve—it took more than a whole month to create the system. As discussed above, technically there are no problems that hinder an effective implementation process. However, the approach has some general problematic issues w.r.t. adequateness of language abstractions and a mixture of concerns among the different specification languages. These issues raise the question if the provided abstractions of U-ISC/Graph have been chosen right to ease and improve the complex and highly specialized task of creating fragment composition systems, which is one of the main objectives when engineering a domain-specific approach and tool. *Of course, this question cannot be solved generally in this section, since only one case study has been conducted by the author of this thesis is not as comprehensive as an empirical study or qualitative study of several Reuseware applications.*

Evaluation of the REX$_{CM}$. The main issues of the U-ISC/Graph approach as it is realized in Reuseware can be distinguished into two groups. The first group is related to the restricted expressiveness of the provided specification and composition languages. While providing 47 keywords[12] for specifying FCMs etc., the REX$_{CM}$ turned out to be very restrictive w.r.t. potential ways of organizing a component model. The port concept for grouping compositional points is poor in the sense that it destroys the composite structure of points which, however, is inherent to the inner fragment AST and also relevant for the organization of composition programs. The port concept flattens the nested structure into a list structure, losing the context. As a result, one has to define redundant ports that somehow resemble that context. For example, reconsider the ports which are derived for the `Employee RoleDefinition` and its `PropertyDefinitions` in Figure 4.11. Naturally, there should be only one port for the `RoleDefinition` and two subports for the properties replicating their physical grouping within the fragment. With flat ports however, the context of the `PropertyDefinition` prototypes is lost, which is unpleasant but nonproblematic as long as the composition program is a finite enumeration of single ports and composition steps. However, if fragment trees have an arbitrary depth (e.g., blocks in a programming language or composite states in a finite-state machine), the context information is essential to compute composition results—as is general recursion which is complementary also not supported by the UCL. There are further minor issues with the REX$_{CM}$ which are related to the binding of value hooks and point-name computation. The binding of value hooks is specified via the keywords `begin idx` and `end idx`, which is fine as long one wants to do string a replacement at one position. However, what if the value should be transformed before the composition or be combined with context-dependent information before composition? Furthermore, the OCL-based point-name computation should provide some support for libraries or at least reuse of OCL expressions as its standard-repertoire of string operations is rather small.

[12]Leaving the embedded OCL uncounted.

Evaluation of the UCL. For fragment composition in general, the composition language UCL also seems not to be an adequate approach since it only describes directed data flow between ports. It does not provide typical features one would expect from a universal composition language, like means for recursion or conditional branches. Moreover, UCL graphs have to be acyclic, otherwise the system may not terminate, since the language does not provide means to specify termination conditions. Also, it does not provide a fixpoint check which could automatically terminate the system if the lattice over sets of prototypes added to a hook is finite (i.e., the systems runs as long as *fresh* content is added to any fragment and terminates otherwise). Additionally, it should be possible to create "portable" composition programs with fragments not enumerated explicitly (i.e., complex composers). As it seems, the user of the UCL has to enumerate fragments participating in the composition by manually dragging them to the graphical editor. Typically, the number of fragments and their individual size is unknown in advance. For instance, in the BAF example, it would be nice to have means to specify that for each BAF model instance (n) and each `RoleDefinition` in there (m), $n * m$ Java classes should be emitted by the composition program. Instead, the author could only realize an explicit enumeration of fragments and composition steps, which works for the specifically provided inputs. A potential way to solve this problem in Reuseware is to specify a dedicated composition language which generates UCL programs depending on a set of input fragments. This approach is supported by the Reuse EXtension language for Composition Language Integration (REX_{CL}) in Reuseware. REX_{CL} allows composition-system developers to declare a language-mapping from the dedicated, user-defined composition DSL to the UCL. However, this has not been further investigated in context of the BAF study.

Evaluation of fragment collaborations. Another secondary issue comes from the concept of fragment collaborations. Devil's advocate could ask if there could ever be different fragment roles than those which are subject to extension or variation and those providing contents for that extension, or fragments playing these roles at the same time? Because of the data-flow semantics of the composition language UCL, it seems likely that composition systems based on an acyclic data-flow architecture consist of data sources (i.e., fragments only contributing their content), data sinks (i.e., fragment only receiving content and that are printed after the composition) and intermediate fragments which consume and produce new content. Of course, fragment collaborations could make sense to specify a top-level coarse-grained architecture of the compositional transformation process, e.g., for decoupling complex composers and connect them via well-defined fragment-interfaces. Reuseware's `fracols` are not expressive enough to achieve this objective because they cannot be typed explicitly with fragment- or value types, and because fragment roles cannot be annotated with binding constraints. Nevertheless, such concepts could be a valuable extension to the `fracols` language.

Evaluation of the language interplay. The second kind of U-ISC/Graph/Reuseware drawbacks comes from the interplay of composition and specification languages. The balancing of some compositional concepts—especially between REX$_{\text{CM}}$ and UCL—seems to be out of position. Due to the lacking expressiveness of the UCL, a composition diagram cannot be "programmed" without also having the corresponding component-model specification(s) available. When specifying an FCM, component-model developers have to have very concrete composition scenarios at hand. Contrary, composition-system users and developers creating composition programs are only allowed to draw arrows between ports without any means to specify explicit mappings between ports or complex connectors. This leads to the observation that REX$_{\text{CM}}$ component models are rather a "strangulating corset" than a user-guiding boundary for safe composition: it appears that the FCM typically has to be developed together with one or more concrete composition programs leading to a small class of composition problems being tackled by the developed system. Thus, if changes in a composition scenario or composition program are required, it is even very likely that the FCM has to be adapted too, e.g., simply because a fragment has to be provided as a prototype or a port name has to be changed just to define a different mapping. Value extraction, transformation and insertion have to be developed in the REX$_{\text{CM}}$ language—there is no other option. Language concepts which are actually concerned with the orchestration of composition are intertwined with concerns dealing with the specification of FCMs. Hence, the roles of component-model developer, composition-system developer and composition-system user are not clearly distinguishable in all probability.

4.5. Summary and Conclusions

In this chapter, the concepts of ISC and a general algebraic model for FCM have been introduced. The model precisely defines the compartments of an FCM and thus implies requirements for approaches to model or generate ISC systems. Accordingly, three concrete approaches to ISC have been investigated by applying them as implementation frameworks for the BAF generator.

First, COMPOST was investigated. As it already provides a built-in FCM for Java 1.4, it was applicable to the example with only small additional technical effort including small changes in the component model. Compared to what should have been expected from a code-generator implementation for the BAF, in comparison with the other two approaches, the solution based on COMPOST is closest to the expected outcome. Also, the "entrance barrier" of using it productively was the lowest among the compared systems. This was to be expected because the other systems do not provide built-in FCMs for Java. The architecture of COMPOST turned out to be difficult to extend and provide it with new FCMs, because it does not provide a declarative and structured approach for specifying such models. Still, in general, it could be used to adopt new FCMs—this requires some knowledge in compiler construction and language engineering, but would not be efficient.

Reusew*air* is the second approach to ISC that has been investigated in this chapter. The BAF generator implementation is based on a small subset of Java 1.4 (Java^{-}) and consequently is not really practical because any change to the source components could require an extension or change of the Java^{-} implementation. However, the implemented code generator and the FCM still demonstrate well the concepts and realization of the U-ISC approach which underpins Reusew*air*. The code emitted by the code generator is close to the expected outcome, but does not support the extension of the class body with constructors as constructors are not supported by Java$^-$. In comparison to COMPOST, Reusew*air* provides a simple FCM specification language and a component-model generator. As only slots are supported by the corresponding DSL and hooks have to be realized by implementing the generated visitor stubs, the approach is semi-automatic, leaving huge parts of the composition system incomplete (e.g., the look-up of hooks, the hook naming and the fragment traversal). Consequently, the resulting composition programs are significantly larger than COMPOST programs.

The last implementation based on Reuseware and U-ISC/Graph showed the best support for declarative FCM specifications so far. Unfortunately, it turned out that the approach is not as suitable as its predecessors to implement the BAF code generator. This is mainly due to the mixture of FCM specification and composition languages, which is not adequate for specifying fragment composition systems generally. The languages are difficult to use, because their concepts and features overlap such that FCM specifications are hardly developed in separation of concrete composition programs at hand. Since the composition language is data-flow-based and has no complex control-flow operators, it cannot express arbitrary compositions over a given FCM. Due to these restrictions, the approach of Reuseware is not suitable to implement code generators.

To summarize, none of the discussed approaches can be recommended as a general solution to ISC. The COMPOST system is the most flexible. Because of its Boxology approach, it can adapt any existing language implementation in Java if it has an adequate API. Reusew*air* is a very light-weight and grammar-aware approach, but is not very flexible if the slotify operator is not suitable if implicit hooks should be considered in the component model or if general language implementations beyond EMF/EMFText should be targeted. Moreover, none of the approaches has a support for context-sensitive properties of the language. For example, if a class fragment should be extended with a method, non of the approaches checks if there already is a method with the same signature contained in the fragment. In the COMPOST implementation of the BAF example, a redundant method is removed manually after it has been mixed-in by the composition system.[13] A semantics-aware composition system could avoid this by automatically checking such constraints.

The approach for ISC developed in this thesis is based on RAGs as implementation and specification framework. It overcomes many of the drawbacks of the approaches discussed above. The following list summarizes the advantages of an RAG-based approach to ISC:

A.1 RAGs enable a uniform and modular way of specifying FCMs. Thus, if RAGs are understood by the composition-system developer the specification of FCMs becomes easy without restricting the expressivity of the compartments of the FCM.

A.2 The core approach is not restricted to a certain host programming language or modeling framework, and can be implemented in any modern RAG tool.

[13]Compare Listing 4.6, `removeDuplicateMethods`.

A.3 Fragment language implementations which already exist as a RAG can be reused and augmented with fragment composition features. If a reusable RAG-based implementation does not exist, RAGs are a natural framework to create one.

A.4 Since RAGs are a declarative approach to name analysis, they are also a natural approach to specify and perform the look-up of compositional points.

A.5 With RAGs it is easy to develop stand-alone and embedded DSLs. Consequently, dedicated composition languages can be developed with RAG-based ISC.

A.6 RAGs can be combined with AST rewrites [Ekman and Hedin 2004], consider attribute dependencies and efficient caching mechanisms [Bürger 2012].

A.7 Also, FCMs for model-based DSLs can be specified since RAGs are well-suited for specifying the static semantics of metamodels (cf. [Bürger et al. 2011]). Thus, a RAG-based approach to ISC can support the application areas of U-ISC and U-ISC/Graph.

A.8 Attribute dependencies can be used to track dependencies between composition steps enabling a composition-dependency analysis.

A.9 Static language semantics can be considered during composition by reusing the respective attributes in predicate attributes—*fragment contracts*—which guard composition steps. This leads to the notion of *well-formed ISC*.

The basics of RAG-based ISC and well-formed ISC will be introduced in the next chapter.

5 Well-Formed Invasive Software Composition Based on RAGs

This chapter is an extended and improved version of the paper by Sven Karol, Christoff Bürger, and Uwe Aßmann [2012]. "Towards Well-Formed Fragment Composition with Reference Attribute Grammars." In: Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE '12). *New York, NY, USA: ACM, pp. 109–114.* ISBN*: 978-1-4503-1345-2.* DOI*:* `10.1145/2304736.2304755`*.*

Finding an adequate approach for developing ISC systems for arbitrary fragment languages has turned out to be an ambitious undertaking. It seems difficult to hit the sweet spot between a predefined closed set of domain abstractions provided as a DSL (Reuseware) and an implementation with a loosely coupled set of domain abstractions provided as a framework in a general purpose language (COMPOST). A closed approach is fine if the use case fits to the class of problems that can be handled by the system, but is inflexible when it comes to unintended usage scenarios. An open and flexible framework can typically support use cases not anticipated by the framework developer, but because of its nature provides less built-in support for tooling like heterogeneous code generation, interpretation and IDE integration. The approach for ISC-system development discussed in this chapter lies between both edges. It suggests RAGs as a declarative formalism for FCM specifications and introduces a collection of attributes and default equations (semantic functions) to realize the ISC abstractions. The attributions developed in this chapter are language independent *patterns* that can be implemented by arbitrary RAG systems. The corresponding RAG modules can be added as a *composition concern* to existing RAG-based language specifications or can be used to develop a composition system from scratch. Based on the FCM attributes, the chapter also discusses the implementation and interpretation of the ISC composition operators using attributes and rewrites based on ReRAGs. Additionally, it discusses several approaches for composition-program evaluation and augments the FCM RAG with *composer declarations* and composition strategies. Moreover, the usage of RAGs gives access to a well-established reasoning formalism on context-sensitive language properties, which are essential for accessing the type system of the CnL and for coordinating complex compositions. Based on context-sensitive information, complex constraints for the presence and naming of hooks can be developed. Information on fragments can be made explicit by reusing attributes of the underlying RAG and supplying them as *fragment assertions* to the composition system. Based on these assertions, the composition can be guarded by *fragment contracts*.

5.1. RAG-Based Fragment Component Models

In this section, RAGs are used as a specification formalism for FCMs. To be able to combine ISC and RAGs so that static semantics can be reused beneficially for fragment composition, it first needs to be investigated if RAGs are actually an adequate formalism for specifying FCMs. Therefore, the concept of a *composition environment* is introduced. The composition environment defines the set of fragment candidates and provides an integrated tree structure of the CnL and the augmenting composition concern. Second, attributes and equations for slot and hook identification in the composition environment are discussed. Finally, additional attributes for context-dependent point names and a concept for deletable points are introduced as extensions to the original FCM definition. As a notational framework, the SimpAG language introduced in Section 3.3.5 and EBNF are used.

5.1.1. Composition Environments

As explained in Section 3.3, RAGs are essentially CFGs (and practically EBNF grammars) enriched with semantic functions. Hence, to be able to use attributes for specifying FCMs, a fragment language which shall be augmented with composition abstractions needs to be integrated with the nonterminals required by the composition system to make fragments and compositional points accessible through attributes. Instances (i.e., trees) of that integrated grammar then provide a distinct spanning tree used by the RAG to evaluate attributes specified with respect to that grammar, and to access and reuse parts of the fragment language.

A *composition environment* fulfills the above requirements by integrating the CnL specification with the grammar that defines compositional abstractions. It provides a unifying root of the integrated AST structure which can be attributed and evaluated by the FCM RAG. Additionally, it adds access functionality to the composition system to look up fragments in environment and to find slots and hooks in fragments. Consequently, the composition environment provides a similar functionality as the *Boxology* of COMPOST (cf. Section 4.2).

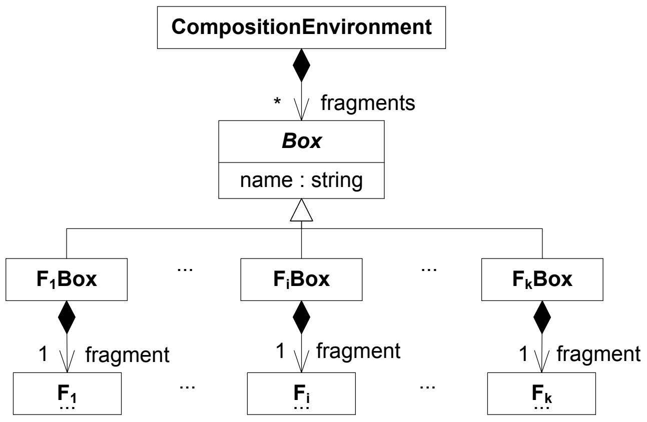


Figure 5.1.: Representation of the EBNF grammar of the composition environment as a UML class diagram.

Assume that the CnL is given as a reduced EBNF grammar with a set of nonterminals N. The set of fragment candidates $\mathcal{F} \subseteq N$ consists of $k \geq 1$ fragment candidates $F_1, \ldots, F_k \in \mathcal{F}$. A composition environment of the CnL is an EBNF grammar which defines the `CompositionEnvironment`, the abstract nonterminal `Box` and one inheriting `Box` nonterminal per each of the k fragment candidates:

```
external F1,...,Fi,...,Fk
CompositionEnvironment ::= fragments:Box*
@Box ::= name:<string>
F1Box▷Box ::=  fragment:F1
... (productions of F2 to Fi-1) ...
FiBox▷Box ::=  fragment:Fi
... (productions of Fi+1 to Fk-1) ...
FkBox▷Box ::= fragment:Fk
```

The class-diagrammatic representation of the grammar is shown in Figure 5.1. The `CompositionEnvironment` holds all fragments of a composition system via the `Box` list (for the interpretation of EBNF lists see Section 3.1.5). The `Box` nonterminal provides a `string` terminal to represent the `names` of fragment components. Based on the integrated grammar and the productions of the fragment language, a generative RAG tool like JastAdd can generate an AST class hierarchy whose runtime instances are composition-environment trees. An example composition-environment tree T is shown in Figure 5.2. T's root node is accordingly labeled `CompositionEnvironment` and has a list node labeled `BoxList` as a child. The `BoxList` has an indicated unbound number of `Fi Box` children each holding a fragment subtree of the component language rooted with an `Fi` node.

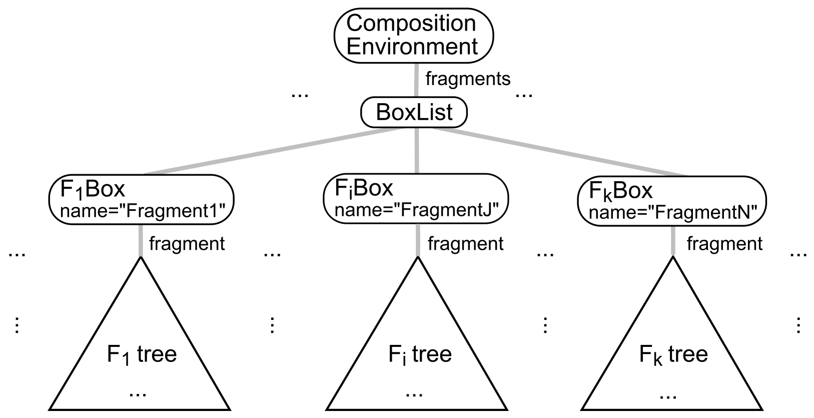


Figure 5.2.: An exemplary composition-environment tree.

Relating the above model to the FCMs notion given by Definition 4.1, the set of `Box`-inheriting productions defines the set of fragment candidates $\mathcal{F}$. The fragment-identification function $\int_{\mathcal{F}}$ is given by the `fragment` child of a `Box` and its name is given by the value of the `name` terminal of the box.

The next subsection shows how compositional points can be defined with attributes.

5.1.2. Compositional Points

To specify a complete FCM, the AST grammar alone is not sufficient since it does not provide enough information to derive the sets of slot and hook candidates ($\mathcal{S}$ and $\mathcal{H}$) and identification functions ($\int_{\mathcal{S}}$ and $\int_{\mathcal{H}}$) according to Definition 4.1. As described in the previous chapter, there are several potential approaches to achieve this. The first is to invasively extend the abstract syntax of the fragment language by mixing nonterminals (i.e., node types) for slots and hooks into the grammars as proposed in [Henriksson et al. 2008] (cf. Section 4.3). This can cause problems with existing language tooling such as parsers or editors since it changes the grammar as model of the language. A second non-invasive approach leaves the grammar intact and captures existing nonterminals as slots and hooks by specifying predicates over ASTs, e.g., as Boolean functions as initially proposed by [Johannes 2011] (cf. Section 4.4) who uses OCL constraints to define points. Generally, the approaches complement each other because it depends on the use case if a predicate is sufficient or if an extension of the grammar is needed. In RAGs, it is natural to use attributes and equations for declaring slot and hook candidates, and grammar extensions using EBNF productions.

Point Identification with Attributes

To adopt RAGs for FCM specification, attributes for the CnL nonterminals need to be defined. The attributes decide if a nonterminal can be a compositional point and compute the point's names. The definitions in this section suggest what such attributes can look like by using the SimpAG notation.

Consider any nonterminal $n \in N$ of the component language. Two predicate attributes (`isSlot` and `isHook`) and two string attributes (`hookName` and `slotName`) constitute the slot- and hook-identification functions and are declared in the FCM's RAG as follows:

$$\textbf{syn}\, \texttt{bool}_\perp\ \{n \,|\, n \in N\}\texttt{.isSlot} \tag{5.1}$$
$$\textbf{syn}\, \texttt{string}_\perp\ \{n \,|\, n \in N\}\texttt{.slotName} \tag{5.2}$$
$$\textbf{inh}\, \texttt{bool}_\perp\ \{n \,|\, n \in N\}\texttt{.isHook} \tag{5.3}$$
$$\textbf{inh}\, \texttt{string}_\perp\ \{n \,|\, n \in N\}\texttt{.hookName} \tag{5.4}$$

The rationale behind declaring `isSlot` as a synthesized (Declaration 5.1) and `isHook` as an inherited attribute (Declaration 5.3) follows from their different purposes in fragment composition systems. Typically, slots are declared explicitly in a fragment component, e.g., the `[[Type]]` and `[[Name]]` slots in the BAF example's Java fragment components (cf. Chapter 2). Consequently, if a parser instantiates a node v labeled with a nonterminal n in the set of slot candidates $\mathcal{S}$, the decision if the node actually is recognized as a slot and the slot name mostly depend on the children of v. In the BAF case, a slot node simply has some terminal child v`.name` with a token value surrounded by double square brackets. Hence, in the normal case, slots are *context-independent*. Therefore, a synthesized attribute equation can be used to decide whether a nonterminal n represents a slot or not. Equations 5.5 and 5.6 reflect that all nonterminals of the integrated grammar which are no slot candidates w.r.t. the CnL are never slots in a composition environment:

$$\textbf{fun}\, \{n \,|\, n \in N \setminus \mathcal{S}\}\texttt{.isSlot} \quad = \textit{false} \tag{5.5}$$
$$\textbf{fun}\, \{n \,|\, n \in N \setminus \mathcal{S}\}\texttt{.slotName} = \perp \tag{5.6}$$

For each occurrence n_i of a slot candidate $n \in \mathcal{S}$ on the left-hand side of a production of the component-language grammar, a distinct equation decides if a slot-candidate node actually is a slot node according to a recognition `pattern` which may only use attributes and children local to that node. Equation 5.7 below implements the matching pattern while Equation 5.8 computes the name of the slot from attributes in the local context:

$$\textbf{fun}\, n_i\texttt{.isSlot} \quad = \begin{cases} \textit{true} & \textbf{if}\, \textit{node matches pattern}, \\ \textit{false} & \textbf{else}. \end{cases} \tag{5.7}$$

$$\textbf{fun}\, n_i\texttt{.slotName} = \begin{cases} \textit{name from } n_i\textit{-context} & \textbf{if}\, \textit{node is a slot}, \\ \perp & \textbf{else}. \end{cases} \tag{5.8}$$

In contrast to slots, the existence and naming of hooks is *context-dependent* so that inherited attributes are the natural choice for their specification. According to Definition 4.1, hooks have to be recursive list nonterminals. As an example, consider an arbitrary list nonterminal `List` and two other nonterminals `M` and `N` of the CnL as well as the following productions:

```
N ::= ... List ...
M ::= ... List ...
```

Each production defines a distinct context for `List`-labeled nodes whose parent could be an `M`- or `N`-labeled node and which would also provide different siblings of `List` potentially influencing the decision about the hook status of `List` nodes. These contexts can be easily distinguished using inherited attributes and broadcasting. Equation 5.9 below specifies the value *false* for the `isHook` attribute while Equation 5.10 specifies $\perp$ as the value of the `hookName` attribute for all children of `CompositionEnvironment` nodes:

$$\textbf{fun}\downarrow \texttt{CompositionEnvironment.child}_{\texttt{all}}\texttt{.isHook} \quad = \textit{false} \tag{5.9}$$

$$\textbf{fun}\downarrow \texttt{CompositionEnvironment.child}_{\texttt{all}}\texttt{.hookName} = \perp \tag{5.10}$$

Due to broadcasting, the values of both equations are passed downwards the tree, unless a node is labeled with a nonterminal which redefines the attributes in its own context. Then, the equation of the redefinition is used to compute the attribute value and, if the equation is a broadcast equation, it is distributed down the tree replacing the original value. Consequently, the hook-identification function can be implemented by "overriding" the broadcasting equation of the `CompositionEnvironment`. Hence, for each production $p = n_i ::= \alpha$ where n_i is the i-th production of n in the desugared CnL grammar, and each l_j in $\alpha = \beta\, l_j\, \gamma$, where $l \in \mathcal{H}$ and l_j is the j-th occurrence of l in α, an attribute equation is provided to decide if the j-th l node in a context n_i of a concrete AST is a hook. Equations 5.11 and 5.12 below show how this is achieved:

$$\textbf{fun}\; n_i\texttt{.}l_j\texttt{.isHook} \quad = \begin{cases} \textit{true} & \textbf{if}\; l\texttt{-node matches pattern}, \\ \textit{false} & \textbf{else}. \end{cases} \tag{5.11}$$

$$\textbf{fun}\; n_i\texttt{.}l_j\texttt{.hookName} = \begin{cases} \texttt{name from context} & \textbf{if}\; \texttt{node is hook}, \\ \perp & \textbf{else}. \end{cases} \tag{5.12}$$

The matching pattern in Equation 5.11 must be given in the context of n provided by p in such a way that the corresponding expressions may use synthesized attributes of the right-hand side symbols in α and inherited attributes of n itself. Similarly, the name of the hook must be computed w.r.t. the current context n_i. In contrast to the naming of slots, which is typically derived directly from the slot declarations in fragments, hook names normally depend on the parental context. As an example, reconsider the two above productions of the nonterminals `M` and `N` having a list nonterminal `List` on their right-hand sides. Both contexts need to compute different values of the `hookName` attribute to make the hooks distinguishable.

Furthermore, `M` and `N` may themselves appear in recursive contexts. Such contexts need to provide point names reflecting that recursion by introducing scoped prefixes for hook names. Inherited attributes with broadcasting are predestined to provide path-based prefixes as a `point-Prefix` attribute. The equations below exemplify its declaration and evaluation functions:

$$\textbf{inh}\; \texttt{string}_{\perp}\; \{n \,|\, n \in N\}\texttt{.pointPrefix} \tag{5.13}$$

$$\textbf{fun} \downarrow \texttt{CompositionEnvironment.child}_{\texttt{all}}\texttt{.pointPrefix} = \texttt{""} \tag{5.14}$$
$$\textbf{fun} \downarrow \texttt{Box.child}_{\texttt{all}}\texttt{.pointPrefix} = \texttt{name} \tag{5.15}$$

Declaration 5.13 defines `pointPrefix` as an inherited attribute. Equation 5.14 provides an empty prefix, which is broadcasted through the environment by default. Equation 5.15 shadows the default hook prefix and broadcasts the value of the `name` terminal of `Box` nodes to its subtree. Nodes within the subtree can add new scope names to a given prefix by appending a local name to the inherited `pointPrefix`. This can be realized by providing an additional equation for `pointPrefix`. Let n_i be the i-th context of a nonterminal n, Equation 5.16 shows how a new hook prefix can be constructed:

$$\textbf{fun} \downarrow n_i\texttt{.child}_{\texttt{all}}\texttt{.pointPrefix} = \texttt{pointPrefix} + \texttt{"."} + \textit{\texttt{scope name}} \tag{5.16}$$

The `pointPrefix` attribute can then be used in equations of the `hookName` attribute (Equation 5.12) to provide context-dependent names, e.g., `pointPrefix` + `"."` + *`local name`*.

Earlier work on ISC showed that a single hook may have multiple names: its default name and an arbitrary number of aliases. For example, in COMPOST a method of a Java class provides a *statements* hook but also a *method-entry* as well as one or more *method-exit* hooks. The statements hook is the top-level block of the method and can be extended by composition. The method-entry hook refers to the same top-level block and consequently is an alias of the statements hook. Also, at least one method-exit hook is an alias of the statements hook. To support aliases, an attribute declaration and corresponding equations can be added to the component model RAG. Declaration 5.17 below adds the `hookAliases` attribute to the FCM's RAG:

$$\textbf{inh}\ \texttt{string}_{\perp}\star\ \{n \mid n \in N\}\texttt{.hookAliases} \tag{5.17}$$

By default, the alias list of a hook is empty as shown in the broadcasting Equation 5.18 below and has no relevance to the component model. As in the `hookName` case, if a hook shall have aliases to be considered by the composition system, a corresponding equation needs to be provided, as shown in Equation 5.19:

$$\textbf{fun} \downarrow \texttt{CompositionEnvironment.child}_{\texttt{all}}\texttt{.hookAliases} = \texttt{[ ]} \tag{5.18}$$
$$\textbf{fun}\ n_i\texttt{.}l_j\texttt{.hookAliases} = \begin{cases} \textit{\texttt{the aliases}} & \textbf{if}\ \textit{\texttt{child is hook}}, \\ \texttt{[ ]} & \textbf{else}. \end{cases} \tag{5.19}$$

Besides a support of multiple names of a hook, a second advantage of aliases is that each alias can provide a different default position for fragment insertion. Moreover, the position may depend on the contents of the list node itself as well as accessible attributes in the hook's context. To support name-dependent default positions, it is suggested to use a synthesized attribute with a parameter such as `hookPosition` in Declaration 5.20:

$$\textbf{syn}\ \texttt{int}_{\perp}\ \{n \mid n \in N\}\texttt{.hookPosition(name)} \tag{5.20}$$

The reason for choosing a synthesized attribute are experiments which showed that, while hook identification and names mostly depend on their context, the default position usually depends on the provided names and the hook's contents. Similar to the attributes before, `hookPosition` has a general default equation for non-hook nodes evaluating $\perp$, as shown in Equation 5.21. For each hook candidate n and occurrence i, an equation has to be provided which evaluates the `hookPosition`'s value depending on its `name` parameter, as exemplified in Equation 5.22:

$$\textbf{fun}\,\{n\,|\,n \in N \setminus \mathcal{H}\}\texttt{.hookPosition} = \perp \tag{5.21}$$

$$\textbf{fun}\,n_i\texttt{.hookPosition(name)} = \begin{cases} \textit{index for}\ \texttt{name} & \textbf{if}\ \texttt{isHook} = \textit{true}, \\ \perp & \textbf{else}. \end{cases} \tag{5.22}$$

Coming back to the above Java example, the presented attribution patterns allow composition-system developers to predefine distinct default positions. The statements hook could be associated with the last position in the list, method entry would provide the first position and method exit could provide last position of the hook or the position before, depending on whether the last statement is a return or not.

Collecting Points

Up to this point, only FCM specification attributes have been discussed. There still remains the task of collecting and looking up slot and hook nodes in the composition environment, which are scattered arbitrarily over the fragments in the environment AST. Without a support for these tasks in the environment, the trees would have to be traversed manually in composition programs to find matching points for composers. Fortunately, tasks such as look-up and collection of nodes in ASTs are standard applications of RAGs. Reference attributes supporting these tasks are easy to specify while the traversal of the tree necessary to compute them is handled by the attribute evaluator, and the attribute evaluation and caching strategies it supports. The attribute declarations below declare the reference attributes `slots` and `hooks`, where N again denotes the set of nonterminals in the CnL including the `Box` nonterminals and the `CompositionEnvironment`:

$$\textbf{syn}\ \texttt{Node}\star\ \{n\,|\,n \in N\}\texttt{.slots} \tag{5.23}$$

$$\textbf{syn}\ \texttt{Node}\star\ \{n\,|\,n \in N\}\texttt{.hooks} \tag{5.24}$$

A simple and mostly sufficient implementation of the attributes in Declarations 5.23 and 5.24 is shown by the Equations 5.25 and 5.26. It relies on the bottom-up evaluation of synthesized attributes and collects lists of references to compositional-point nodes in a depth-first style from their occurrences to the upper nodes in the environment tree.

$$\textbf{fun}\,\{n\,|\,n \in N\}\texttt{.slots} = \begin{cases} \{\textit{node}\} & \textbf{if}\ \texttt{isSlot} = \textit{true}, \\ \bigcup\limits_{c \in \texttt{child}_{\texttt{all}}} c\texttt{.slots} & \textbf{else}. \end{cases} \tag{5.25}$$

$$\textbf{fun}\ \{n\,|\,n \in N\}.\texttt{hooks} = \begin{cases} \bigcup\limits_{c\in\texttt{child}_{\texttt{all}}} c.\texttt{hooks} \cup \{\mathit{node}\} & \textbf{if}\ \texttt{isHook} = \mathit{true}, \\ \bigcup\limits_{c\in\texttt{child}_{\texttt{all}}} c.\texttt{hooks} & \textbf{else}. \end{cases} \tag{5.26}$$

Figure 5.3 shows the initial composition-environment tree of Figure 5.2 augmented with attribute instances and a runtime snapshot of the RAG-induced data-flow graph connecting them. The displayed instances cover a small subset of all attributes in the environment—sufficient to explain the general interplay of the component-model attributes when they are evaluated. The three fragment trees shown in the figure contain a specific number of compositional points, of which four are actually displayed. "Fragment1" in the $\texttt{F}_1\texttt{Box}$ contains two of them—$\texttt{v}_{1,1}$ (the 1st point of $\texttt{F}_1\texttt{Box}$) with the label $\texttt{N}_{1,1}$, and $\texttt{v}_{1,q}$ (the q-th point of $\texttt{F}_1\texttt{Box}$) with the label $\texttt{N}_{1,q}$. The value of isHook depends on attribute instances in the enclosing fragment tree and values inherited from the composition environment. The latter is indicated by the symbolic inherited-attribute boxes with dotted lines whose information carried flows into the fragment tree. In contrast, the evaluation of isSlot only depends on the nodes themselves. Depending on the values of the isPoint and isHook predicates, the slots and hooks attributes collect *references* to $\texttt{v}_{1,1}$ and $\texttt{v}_{1,q}$, first locally at the nodes themselves and then upwards to the Box nodes so that each fragment box holds two lists of references to its own slots and hooks. The $\texttt{F}_1\texttt{Box}$ has $\texttt{v}_{1,q}$ in its slot list and $\texttt{v}_{1,1}$ in its hook list, the $\texttt{F}_i\texttt{Box}$ has $\texttt{v}_{i,1}$ in its hook list, and the $\texttt{F}_k\texttt{Box}$ contains a reference to $\texttt{v}_{k,r}$ in the local slot list. Finally, the fragment-specific lists are integrated at the BoxList node and transferred to the CompositionEnvironment root, where it can be accessed by external clients. The final state of the attributed environment AST is shown in Figure 5.4. The reference edges induced by the slots and hooks attributes constitute a reference overlay graph on the AST which creates an environment ASG where the AST is a spanning tree.

The attribution patterns presented in this subsection can be easily extended with new attributes and RAG modules defining new kinds of compositional points. Based on this observation, the next subsection introduces *rudiments* as points that can be *extracted* from a fragment component.

5.1.3. Subtractive Composition with Rudiments

The standard model of ISC—ISC_{core} (cf. Section 4.1.3)—is *additive* in nature. That is, the compositions possible in that model always enrich the content of fragment components by adding new subtrees. In the general case, this kind of transformation is sufficient to model arbitrary decompositions using fragment components. However, a strict predetermined decomposition is not always desirable or even viable: consider a set of fragment components and composition programs which have been designed and implemented for a specific purpose. Moreover, assume that the program has already been compiled and delivered. A new use case is hard to incorporate into the system, if the delivered decomposition is not compatible with the new use case. That means the original decomposition is not suitable for the new problem and needs to be changed so that the original composition programs also would need to be rewritten. A subtractive composition

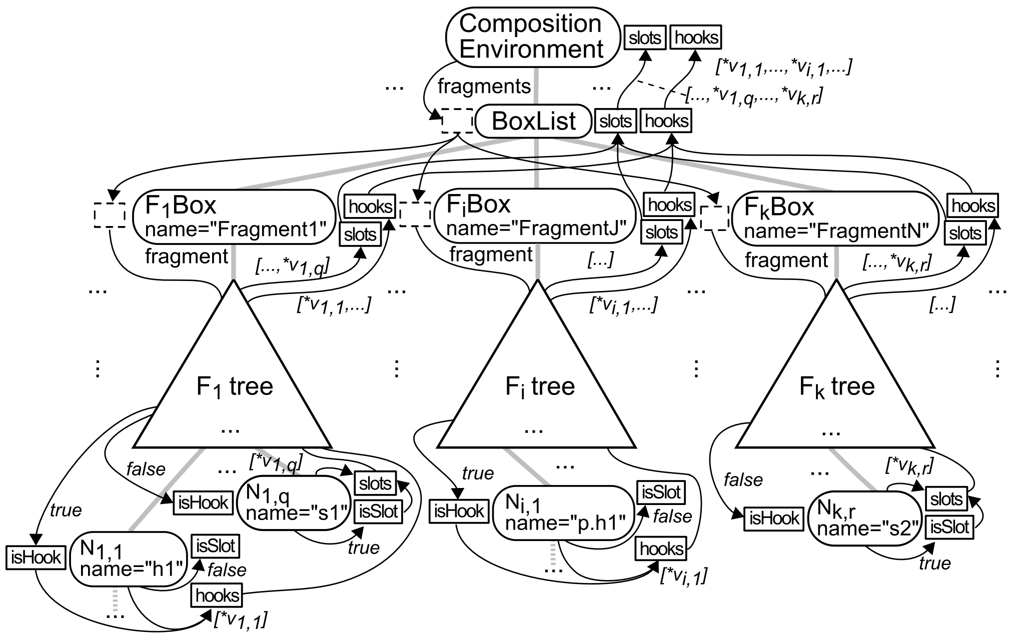


Figure 5.3.: The environment tree of Figure 5.2 with component-model attributes and data flow between these attributes.

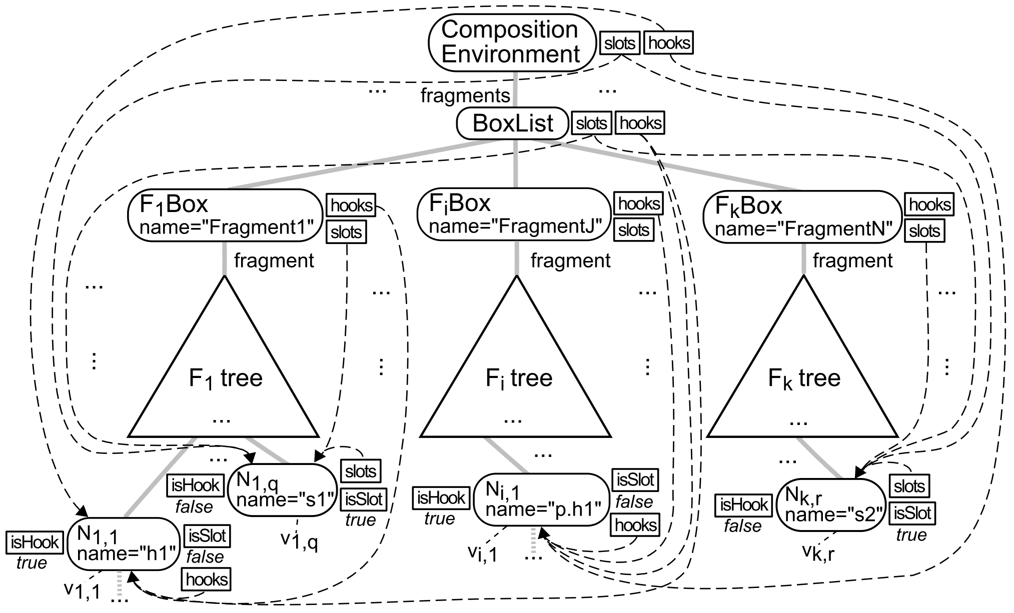


Figure 5.4.: The environment tree of Figure 5.2 with evaluated component-model attributes and reference overlay graph.

operator can support unintended use cases by introducing decompositions that are specific to a composition program. For example, the aspect-weaving model of AspectJ [Kiczales et al. 2001] supports this by the *around* advice which can be used to remove, replace or wrap joinpoints (hooks) of a fragment. Consequently, ISC systems should support such *subtractive* compositions.

To include subtractive compositions, a new category of points besides slots and hooks is established below. The new category is called *rudiments* and provides the original FCM definition with a notion for *fragment extraction*. Creating an extension of the original Definition 4.1 is straightforward and therefore omitted here in favor of a RAG-only specification. However, for interested readers, Appendix A.3 provides Definition A.1—a corresponding extension of Definition 4.1. To declare the point-identification and naming attributes for rudiments, the corresponding attribute declarations for hooks (Equation 5.3 and 5.4) can be replicated:

$$\textbf{inh}\,\texttt{bool}_{\perp}\,\{n\,|\,n \in N\}\texttt{.isRudiment} \tag{5.27}$$

$$\textbf{inh}\,\texttt{string}_{\perp}\,\{n\,|\,n \in N\}\texttt{.rudimentName} \tag{5.28}$$

Declaration 5.27 declares the `isRudiment` predicate for arbitrary nodes of the CnL grammar and the environment nodes as an inherited attribute. The choice of an inheriting declaration is due to the fact that a node can only be deleted from the AST if this is supported by its context, i.e., like the composition of slots and hooks, also deletion must preserve the integrity of the AST w.r.t. the CnL grammar. Hence, a rudiment can only be deleted if this is allowed by the context of the node, i.e., if the context is still valid after the rudiment has been deleted. In general, this is the case if the context of the node is a list or an optional nonterminal. The equations below specify the defaults of the rudiment declarations above as broadcasting attributes like in the corresponding hook definitions (cf. Equations 5.9 and 5.10):

$$\textbf{fun}\downarrow\texttt{CompositionEnvironment.child}_{\texttt{all}}\texttt{.isRudiment} \quad = \textit{false} \tag{5.29}$$

$$\textbf{fun}\downarrow\texttt{CompositionEnvironment.child}_{\texttt{all}}\texttt{.rudimentName} = \perp \tag{5.30}$$

Consequently, by default, the FCM does not provide any rudiment points, so that for the specific optional and list contexts redefining equations can be specified. For any list node $l \in N$ of the CnL, a non-broadcast equation can be provided to define if a child of l shall be a rudiment and which point name is derived from the context:

$$\textbf{fun}\,l\texttt{.child}_{\texttt{all}}\texttt{.isRudiment} = \begin{cases} \textit{true} & \textbf{if}\,\textit{child matches pattern}, \\ \textit{false} & \textbf{else}. \end{cases} \tag{5.31}$$

$$\textbf{fun}\,l\texttt{.child}_{\texttt{all}}\texttt{.rudimentName} = \begin{cases} \textit{name} & \textbf{if}\,\texttt{child.isRudiment} = \textit{true}, \\ \perp & \textbf{else}. \end{cases} \tag{5.32}$$

Equations 5.31 and 5.32 specify the equation for all entries of a list node via $\texttt{child}_{\texttt{all}}$, which is—for lists—equivalent to having an equation in the context of each recursion of a right-recursive nonterminal in the desugared CnL grammar. For each member of the list, a matching *`pattern`* decides if it is allowed to be extracted from the list and, depending on the result of that decision,

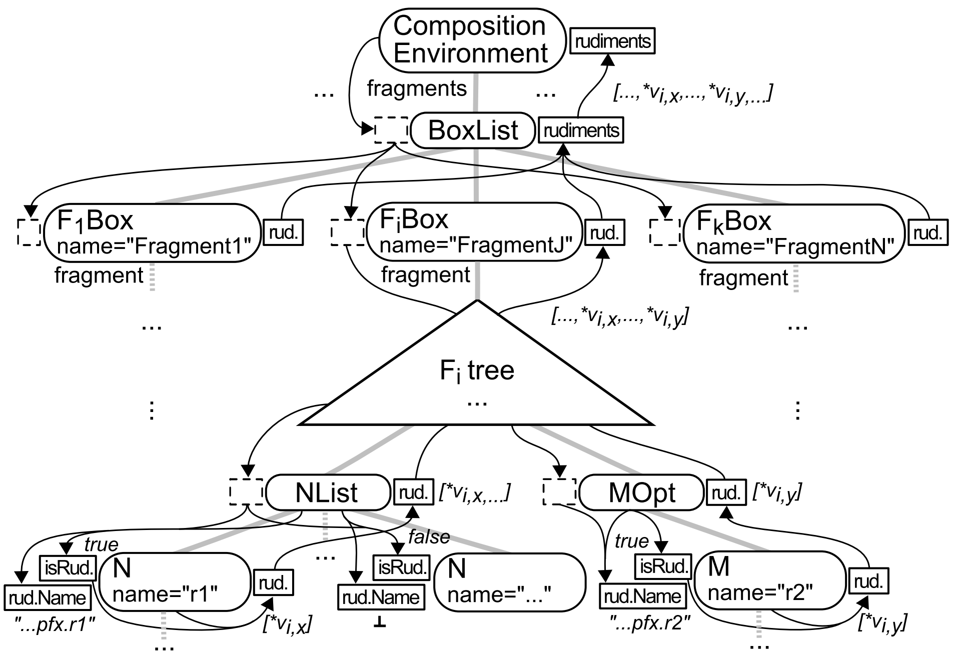


Figure 5.5.: The environment tree of Figure 5.2 with attribute instances related to rudiments and sketched data flow between them. Observe that `rud.` is used as a shorthand for `rudiment` in the Figure.

provides a *name* for the rudiment which may be derived from information in the upper tree, e.g., reusing the `pointPrefix` attribute.

For optional nonterminals the equations look similar. Consider any optional nonterminal $o \in N$ of the CnL with two alternatives $o_1 ::= n$ and $o_2 ::= \epsilon$ in the desugared CnL grammar. For the first child carrying alternative o_1, a rudiment can be defined by providing equations in the form of Equation 5.33 and Equation 5.34:

$$\textbf{fun}\ o_1\texttt{.child}_1\texttt{.isRudiment} = \begin{cases} \mathit{true} & \textbf{if}\ \mathit{entry\ matches\ pattern}, \\ \mathit{false} & \textbf{else}. \end{cases} \tag{5.33}$$

$$\textbf{fun}\ o_1\texttt{.child}_1\texttt{.rudimentName} = \begin{cases} \mathit{name} & \textbf{if}\ \texttt{child.isRudiment} = \mathit{true}, \\ \bot & \textbf{else}. \end{cases} \tag{5.34}$$

As in the list case, a matching *pattern* decides the `isRudiment` equation and a *name* is provided if the node is a rudiment.

To make the collection of rudiments available to the composition environment, a `rudiments` attribute is provided. The declaration and equation of that attribute correspond to the definition of the `hooks` attribute in 5.24 and 5.26 and are therefore omitted here.

Figure 5.5 shows a view on the attributed exemplary environment tree in Figure 5.2 with a focus on the rudiment-defining attributes and data-flow dependencies between their instances in the composition environment. The figure shows the identification of two rudiments in the subtree of the central $\mathtt{F_iBox}$, while the contents of the $\mathtt{F_1Box}$ and $\mathtt{F_kBox}$ are not represented in the figure. In the context of the `NList` node, the list's entry v_{ix} is a rudiment. The evaluation depends on the parental context of the `NList` node as well as the contents of the node itself. Similarly, the `rudimentName` is computed from the direct and indirect parental context of v_{ix} by prepending a prefix to the assumed `name` terminal child of the node (e.g., using an attribute like `pointPrefix` or something else). Considering the second entry of the `NList`, `isRudiment` evaluates *false* and, thus, the `rudimentName` is $\perp$. Moreover, in the context of the `MOpt` node, the `M`-labeled node v_{ix} is recognized as a rudiment. Like in the `NList` case, the corresponding decisions depend on the `MOpt` node and its parental context. Similar to the `hooks` and `slots` attributes, the `rudiments` attribute makes the rudiment nodes available at the root nodes of the `Box` subtrees and the `CompositionEnvironment` top-level root node.

5.1.4. LogProg Example

In the following example, the RAG-based method to specify FCMs will be used to provide the LogProg FCM developed in Example 4.2.

Example 5.1 (SimpAG-based fragment component model).
Reconsider the formal fragment component model developed in Example 4.2 and the EBNF grammar G_{Log} of the LogProg language defined in Example 3.13. The following composition-environment grammar G_{LogFCM} declares the fragment candidates `Program`, `Stmt` and `Expr`.

```
external Program,Stmt,Expr
CompositionEnvironment ::= fragments:Box*
@Box ::= name:<string>
ProgramBox▷Box ::= fragment:Program
StmtBox▷Box ::= fragment:Stmt
ExprBox▷Box ::= fragment:Expr
```

The above grammar is a short form and only valid in combination with G_{Log} by composing the two EBNF grammars, where the productions of G_{Log} are simply prepended to G_{LogFCM}.

To make the G_{LogFCM} a RAG-based FCM AG_{LogFCM}, the SimpAG attribute declarations and equations introduced previously in this section need to be imported, and the patterns of the equations of the point-identification attributes have to be instantiated. For slots, this includes the `isSlot`, `slotName` and `slots` attributes. The default equations 5.5 and 5.6 for non-slot nodes are instantiated w.r.t. the of set nonterminals N of G_{Log} and the set of slot candidates $\mathcal{S} = \{\mathit{Expr}\}$ of the formal LogProg FCM in Example 4.2. Since *Expr* is a slot candidate, two equations for `isSlot` and `slotName` according to the attribution patterns of Equation 5.7 and

Equation 5.8 have to be provided as shown below:

$$\textbf{fun}\ \texttt{Expr.isSlot} = \begin{cases} \mathit{true} & \textbf{if}\ \texttt{termCount} = 1 \\ & \quad \textbf{and}\ \texttt{ident} \neq \bot \\ & \quad \textbf{and}\ \texttt{matches(ident, ".+Slot")}, \\ \mathit{false} & \textbf{else}. \end{cases}$$

$$\textbf{fun}\ \texttt{Expr.slotName} = \begin{cases} \texttt{pfx(ident)} & \textbf{if}\ \texttt{isSlot} = \mathit{true}, \\ \bot & \textbf{else}. \end{cases}$$

In the first equation, `termCount` shall be a synthesized attribute counting the number of occurrences of `Term` nodes in the `Expr` subtree. The `ident` attribute shall provide the value of the `ident` terminal within the subtree if such exists. The function `matches` checks if the given `ident` value in the first parameter matches the *regular expression* on the right-hand side, i.e, it checks if the `ident` value has a "`Slot`" suffix. In the second equation, the slot's name is derived by extracting a value from the `ident` string which stands in front of the suffix.

To define hooks, equations for `isHook` and `hookName` have to be provided.[a] The default realizations with broadcasting definitions in the Equations 5.9 and 5.10 can simply be reused, only requiring that specific equations must be provided for hook candidates. In the formal FCM of Example 4.2, $\mathcal{H} = \{\mathit{StmtList}\}$ defines `StmtList` as a candidate nonterminal for hooks. The two equations below define `StmtList` as a hook in the context of program:

$$\textbf{fun}\ \texttt{Program.StmtList.isHook} = \mathit{true}$$
$$\textbf{fun}\ \texttt{Program.StmtList.hookName} = \texttt{pointPrefix} + \texttt{"Stmts"}$$

The first equation specifies that `StmtList` always is a hook if it is a child of a `Program` node. Hence, since `StmtList` in G_{Log} only occurs in that context, any `StmtList` in LogProg is a hook. The second equation derives the name of the hook by using the value of the inherited `pointPrefix` attribute and concatenates "`Stmts`". For `pointPrefix`, the equation below provides a value based on the name of the enclosing fragment box and appends "`#`" as a separating character:[b]

$$\textbf{fun} \downarrow \texttt{Box.Fragment.pointPrefix} = \texttt{name} + \texttt{"\#"}$$

Consequently, the name of the `StmtList` hook of a `ProgramBox` with an assumed name "`simple.fgmt`" is "`simple.fgmt#Stmts`".

Besides slots and hooks, the RAG-based FCM for LogProg shall also support `Stmt` nodes as rudiments so that statements can be extracted from a LogProg program. Therefore, the broadcasting Equations 5.29 and 5.30 are reused as defaults for the values of the `isRudiment` and `rudimentName` attributes. For `Stmt` nodes which are child of a `StmtList` node, specific equations are provided:

$$\textbf{fun}\ \texttt{StmtList.child}_{\texttt{all}}\texttt{.isRudiment} = \textit{true}$$

$$\textbf{fun}\ \texttt{StmtList.child}_{\texttt{all}}\texttt{.rudimentName} = \texttt{pointPrefix} + \texttt{"Stmt"} + \texttt{listPos}$$

The first equation specifies that any child of a `StmtList` is a rudiment. Hence, all `Stmt` nodes of a list can potentially be extracted. The second equation provides the names of a `Stmt` rudiment by constructing the specific name using the `pointPrefix` attribute as defined before and by using the node's position (`listPos`) in a `StmtList`.

Figure 5.6 shows a runtime instance of the above defined RAG-based FCM with the most important attribute instances contributing to the respective point sets. Instances of other attributes have been omitted to simplify the figure. The environment tree contains three fragment components. The leftmost component is an expression box containing an `Expr` tree with a primitive Boolean value "`t`" (cf. T_{ex} in Example 4.3). According to the above-defined FCM, it does not provide any compositional point. The rightmost component is a statement box containing an `Stmt` tree representing the LogProg statement `c = t;` (cf. T_{st} in Example 4.4). It also does not provide any compositional points. The central fragment component is a complete LogProg program with a `Program` tree. The textual representation of that fragment corresponds to Listing 4.1 in Example 4.2. The `StmtList` in the fragment is recognized as a hook by the RAG and its name is computed to "`simple.fgmt#Stmts`". Furthermore, the `Expr` node of the second statement of the list is recognized as a slot with the name "`init`" which is derived from the value of the `ident` token at the leaf of the corresponding subtree. Finally, the LogProg program contains three rudiments—the `Stmt` nodes of the `StmtList`. The rudiment names `stmt.fgmt#Stmt1`, `stmt.fgmt#Stmt2` and `stmt.fgmt#Stmt3` are derived from the `pointPrefix` and the list position of the respective node. ◇

[a]Because the LogProg FCM is simple, aliases and predefined positions are not required and the corresponding attributes, `hookAliases` and `hookPosition`, are not considered in this example.
[b]In practice, it is usually better to use a dedicated data structure to represent names or naming patterns.

In combination with the slot- and hook-identification attributes, the RAG-based approach on component-model specification can model arbitrary FCMs with support for additive and subtractive composition. However, the integration of the compositional attribute grammar parts with the original specifications of the CnL is not always a trivial task, especially if the original attribution shall be reused during composition. For example, to make the composition more reliable by incorporating parts of the language semantics as described later in this chapter, or simply to reuse CnL attributes to compute point names. The next subsection investigates some issues that may occur.

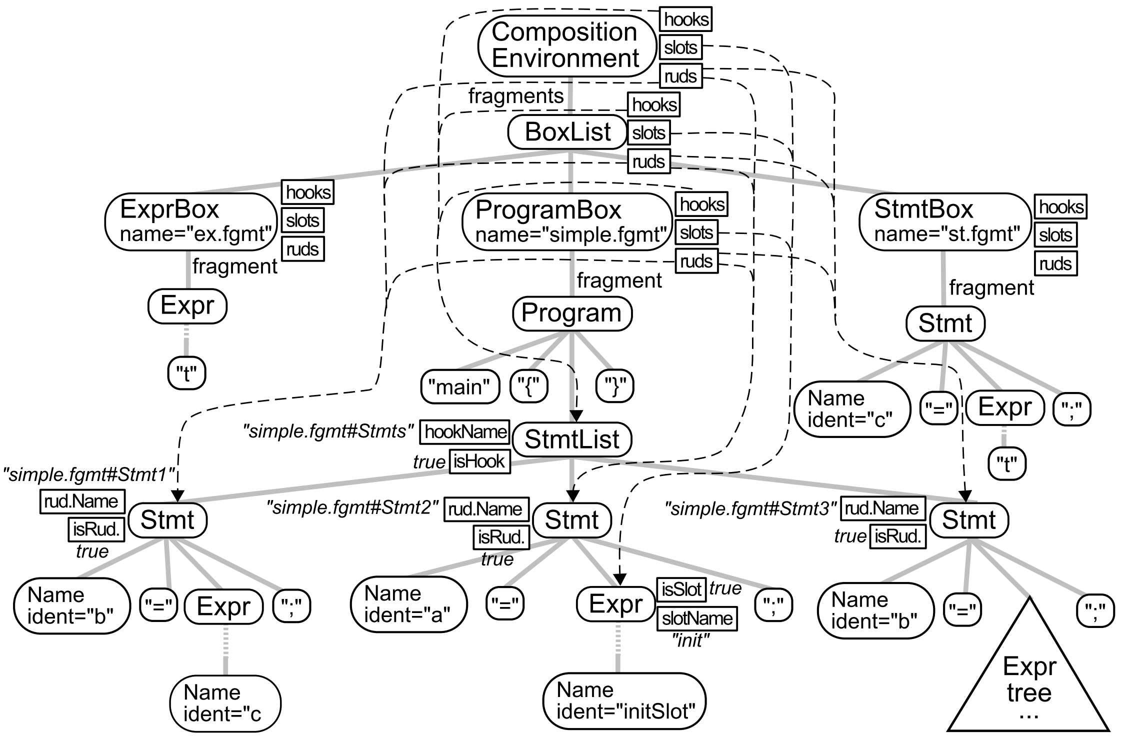


Figure 5.6.: Excerpt of a LogProg composition environment at runtime with reference edges computed by an attribute evaluator.

5.1.5. Organizational Aspects

Logically, the attributes involved in the specification of FCMs using RAGs can be distinguished into three groups of RAG and grammar modules—the *fragment RAG*, the *component-model RAG* and the *glue RAG*. The modules of the fragment RAG contain the grammar and semantics of the CnL. Typically, the fragment RAG provides an integral part or complete frontend of a CnL compiler, including CFGs, semantics analyses and optimizations.

The modules of the component-model RAG which are described in this section suggest attribute declarations and some attribute equations to specify a component model. In contrast to the CnL's RAG modules, the component model RAG presented in this chapter does not have a stand-alone meaning as it is a collection of attribution patterns which can be reused by composition-system developers or even be generated by a tool like Reusew*air*/Reuseware. The component-model RAG could be used as an extensible intermediate representation from a high-level DSL-based component-model specification. Hence, the component-model RAG can be further distinguished in a reusable "core" part providing the basic infrastructure and CnL-specific parts which define concrete slot and hook concepts in the CnL. Moreover, it may be necessary, to extend or adopt the CnL grammar with additional nonterminals and productions. The support of custom language extensions for composition is essential and practically relevant for the acceptance of FCMs and invasive composition systems in general—fortunately, specifying language extensions modularly is a typical use case for RAGs. For example, custom language extensions to the CnL are required to introduce specific markup for compositional points (e.g., the slotification operator of Reuse*wair*), or to support specific composition syntax embedded in the CnL as it will be demonstrated later in the thesis.

Finally, the parts of the CnL that have been extended or have been declared as fragment candidates using `Box` nonterminals may require additional *glue RAG* specifications. These may become necessary, because the CnL nonterminals are embedded into potentially unforeseen contexts. The new contexts become problematic, if the CnL specifies attributes in the fragment tree which depend on information provided directly or indirectly by inherited attributes. In this thesis, the problem is referred to as the *open-context problem* of language composition.

To solve the open-context problem, three strategies become emergent. The first is to syntactically close the contexts by adding attribute equations to the glue RAG, which provide some context, but may cause the semantics-analysis algorithms of the CnL to compute unintended or erroneous results. As an example, consider a method-declaration box of some programming language CnL. The standard name-analysis implementation of the CnL may expect some preinitialized environment containing type information and system-dependent values. A plain syntactical provision of such an environment would not include this information making the dependent attributes less reliable.

The second strategy to solve the open-context problem produces more reliable results by essentially also semantically restore the expected context information as far as possible. In the above example, it could provide the expected environment values "simulating" the original context. However, while the semantics restoration leads to better results, it typically requires a much deeper involvement with the CnL: the composition-system developer needs to investigate and understand the semantics specification of the CnL—a time consuming and complex task if the CnL is developed independently from the FCM.

A third variant only applies if the composition system should *not* be semantics-aware. In that case, the attributes of the CnL may be excluded from the FCM. However, this also should be done with care: RAG systems like JastAdd frequently support AST rewrites based on higher-order attributes to normalize and disambiguate the AST after the initial parse. The composition-system developer should be aware of this and define the FCM w.r.t. the most adequate transformation stage of the AST.

In general, if a CnL should not only be used as a fragment specification but also be extended with dedicated compositional constructs itself, composition-system developers have to make sure that these do not "disturb" the original attribution. For example, this may happen with dedicated slot declarations, embedded composition operators or macro calls. The problem is referred to as the *incomplete embedding problem* of language composition in this thesis and is mostly caused by synthesized attributes expected by the context of new compositional constructs embedded into the CnL. Consider the slotification of arithmetic expressions in some general-purpose language for

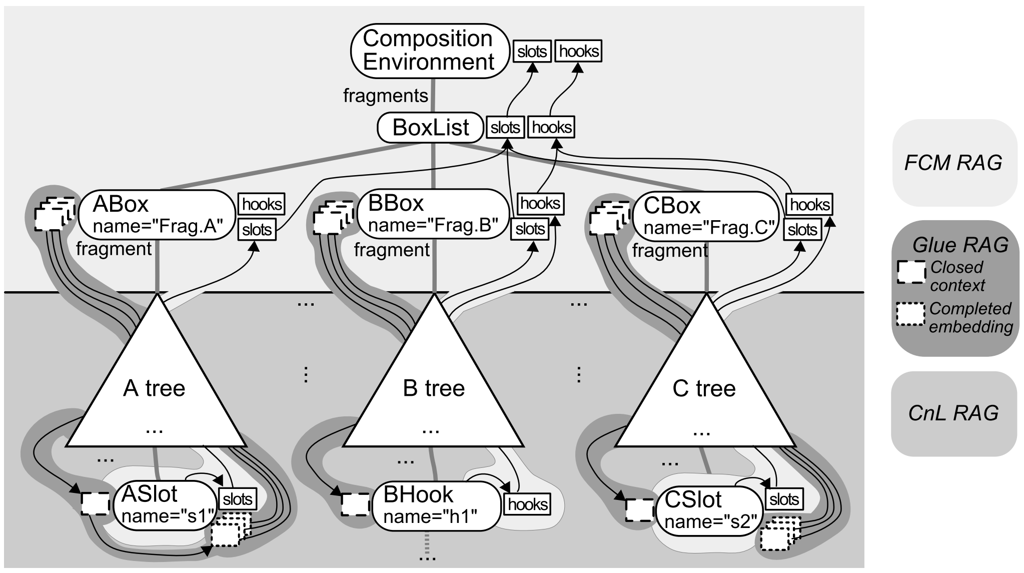


Figure 5.7.: A composition environment with gluing attributes as well as CnL and FCM attributes emphasized.

instance. Typically, type- and name-analysis-related attributes compute the type of an expression. Now, if the expression is slotified in such a way that a slot may occur instead of or within the expression subtree, the type analysis attribution needs to be completed with gluing attributes and equations, so that the whole analysis provides a reasonable or at least any useful result.

Figure 5.7 visualizes the interplay between attributes of the FCM RAG, the CnL RAG and glue RAG in an exemplary fragment environment. It is assumed that the FCM uses attributes of the CnL, e.g., to identify hooks or using CnL-semantics in other ways. Each fragment has a set of instances of inherited context attributes which need to be provided by the `Box` contexts. Some provided attribute values are then used in the contained trees while others are used at compositional points. For example, slot `s1` (transitively) depends on one of these attributes. Its value is then used to provide an adequate embedding of `s1` into its containing tree, e.g., it may provide a name or a type signature.

Depending on the degree of integration of CnL, FCM and CsL, the specification of gluing attributes can be cumbersome and error-prone. Unfortunately, the author of this thesis is not aware of a general solution to the open-context and incomplete-embedding problems. *However, Chapter 7 proposes scalable ISC—an agile and incremental style of composition-system implementation to cope with different degrees of complexity in their development.*

5.2. Composition Operators

An FCM is sufficient to model fragment components and compositional points. A primitive composition system can already use instances of a specific fragment environment to manage fragments and to find points in the components. However, the composition system would still have to implement the point look-up and the composition operations in separation of the environment. Fortunately, RAGs can also support the composition system in these tasks. The look-up of composition operators only requires a few additional attributes as name analysis is a standard use case of RAGs. Moreover, the works on HOAGs [Vogt et al. 1989], ReRAGs [Ekman and Hedin 2004] and, recently, RAG-controlled rewriting [Bürger 2012] have shown that RAGs can be beneficially combined with transformations on the AST. Based on these techniques, this section discusses the extension of the previously introduced RAG-based fragment environment with a support for composition operators. The discussion first focuses on the primitive composers—*bind*, *extend* and *extract*—and their conceptual integration into the fragment environment. Based on this integration, potential strategies of rewrite-based composer execution are presented and evaluated with respect to their basic properties.

5.2.1. Composer Declarations

Adding a support for composition operators is straight-forward by simply introducing composer declarations as nonterminals and productions to the EBNF grammar of the composition environment, and using attributes to resolve the respective point and fragment names. The following EBNF grammar “module” specifies the extension of the basic ISC model with composer declarations:

```
external Box
CompositionEnvironment ::=
             fragments:Box* composers:Composer*
@Composer ::= pointName:<string> fragmentName:<string>
Bind▷Composer ::= ϵ
Extend▷Composer ::= position:<int>
Extract▷Composer ::= ϵ
```

Box and its depending definitions correspond to those introduced in the basic environment grammar in Section 5.1.1. CompositionEnvironment has been redefined to support a list of Composers denoting the composition program besides the different fragment types supported by the system. A Composer declaration is represented as an abstract nonterminal predefining pointName and fragmentName terminals. Its inheriting nonterminals—Bind, Extend and Extract—represent the respective composition operations. The declaration of Extend has an additional terminal child position to address a specific position in a hook. Figure 5.8 shows an example CompositionEnvironment tree with a ComposerList subtree. The list contains three exemplary Composer nodes, one for each of the respective composer types.

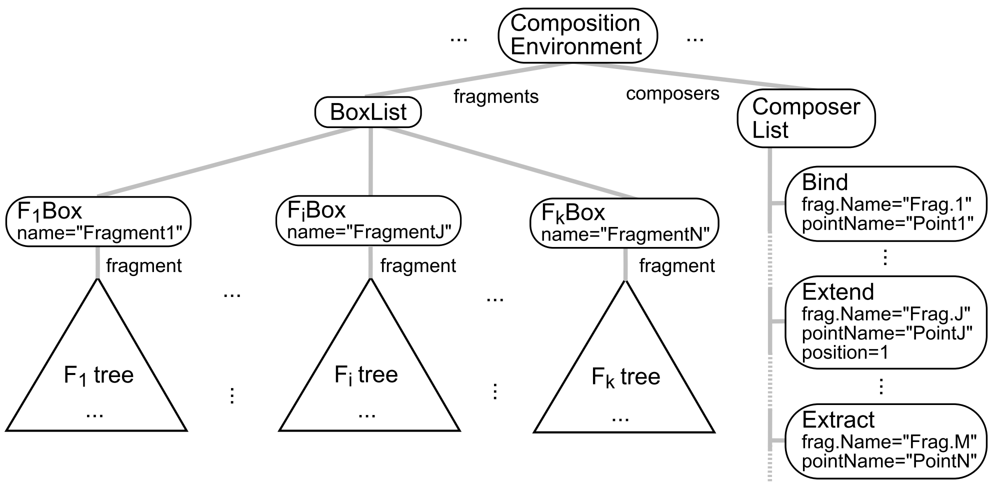


Figure 5.8.: An example composition environment with composers.

Associated with the nonterminals, attributes can be used to compute references to compositional points from the `pointName` terminal and to the argument fragment from the `fragmentName`. As a basic interface for this, two synthesized attributes can be declared:

$$\textbf{syn}\ \texttt{Node}\ \{\texttt{Bind},\texttt{Extend}\}.\texttt{srcFragment} \tag{5.35}$$

$$\textbf{syn}\ \texttt{Node}\star\ \texttt{Composer.points} \tag{5.36}$$

The argument fragment of a composer declaration is constructed or resolved by the `srcFragment` attribute declared in Declaration 5.35 while the target points of the composition are to be resolved by the `points` attribute introduced in Declaration 5.36. Being decompositional in nature, the `Extract` composer is not required to be provided with a source fragment. In general, both attributes can be regarded as interfaces to provide arbitrary RAG-based matching functions for compositional points and fragments. However, for now and for the basic model of ISC, it is sufficient to realize the look-up of the respective fragment and compositional points using the attributes at hand and a `Pattern` to abstractly represent different matching schemes. `Pattern`s are abstract representations of a name and thus constructed from the respective point or fragment name associated with a `Composer` node. Using the binary operator $\simeq : \texttt{Node} \times \texttt{Pattern} \rightarrow \texttt{bool}_\perp$, where $v \in \texttt{Node}$ is an AST node according to the CnL grammar, the RAG can determine if the pattern applies to a given node. In practice, common types of patterns in fragment composition systems are *string-based names*, *regular paths* (e.g., common to XML-based applications) and *pointcut expressions* (e.g., common to AOP).

Four inherited parametrized look-up attributes—`lookUpF` for fragments and `lookUpS`, `lookUpH`, `lookUpR` for the respective point categories—are provided to find the nodes in the environmental context:

$$\textbf{inh}\ \texttt{Node}\ \{\texttt{Composer},\texttt{ComposerList}\}.\texttt{lookUpF}(\texttt{Pattern}) \tag{5.37}$$

$$\textbf{inh}\ \texttt{Node}\star\ \{\texttt{Composer},\texttt{ComposerList}\}.\texttt{lookUpS(Pattern)} \tag{5.38a}$$

$$\textbf{inh}\ \texttt{Node}\star\ \{\texttt{Composer},\texttt{ComposerList}\}.\texttt{lookUpH(Pattern)} \tag{5.38b}$$

$$\textbf{inh}\ \texttt{Node}\star\ \{\texttt{Composer},\texttt{ComposerList}\}.\texttt{lookUpR(Pattern)} \tag{5.38c}$$

A `Pattern`-based fragment look-up is easily realized using a broadcasting equation as shown in Equation 5.39 below.

$$\textbf{fun} \downarrow \texttt{CompositionEnvironment.composers.lookUpF(name)} \\ = \begin{cases} f\texttt{.fragment} & \textbf{if}\ f \in \texttt{fragments}\ \textbf{and}\ f \simeq \texttt{name} \\ \perp & \textbf{else}. \end{cases} \tag{5.39}$$

The attribute inspects the fragment boxes in the `BoxList` and provides a reference to that fragment matching the `Pattern` passed to the evaluation function via the parameter `name`. In the systems presented this thesis, fragments in the composition environment are typically looked up via string-based names, i.e., the semantics of $f \simeq$ `name` boils down to f`.name` $=$ `name`.

Looking up compositional points is also specified straightforwardly using the point-collecting attributes introduced in the previous section. Since the basic implementation of Declarations 5.38a–5.38c is similar, Equation 5.40 below specifies the slot look-up function as a representative:

$$\textbf{fun} \downarrow \texttt{CompositionEnvironment.composers.lookUpS(name)} \\ = \{v \,|\, v \in \texttt{slots}\ \textbf{and}\ v \simeq \texttt{name}\} \tag{5.40}$$

The equation constructs a set of references to slot nodes whose value of the `slotName` attributes matches the pattern passed via the parameter `name`. In the most simple case, the matching can just be string equivalence so that $v \simeq$ `name` in fact is v`.slotName` $=$ `name`. Contrary, point names which have been constructed using the `pointPrefix` attribute and are qualified by a prefix of arbitrary length benefit from a regular-expression-based matching with wildcards. In that case, the parametrized pattern is a DFA A constructed from a path-based name so that $v \simeq$ `name` holds if $A.\mathit{accept}(v\texttt{.name}) = \mathit{true}$ where *accept* checks if a given string is accepted by the automaton. For hooks, potential aliases provided by the `hookAliases` attribute have to be checked. Hence, if v is a hook, then $v \simeq$ `name` holds if there exists a $\mathit{name} \in v\texttt{.hookAliasesU}\{v\texttt{.hookName}\}$ so that $A.\mathit{accept}(\mathit{name}) = \mathit{true}$.

Finally, the look-up attributes can be used in equations of the `srcFragment` and `points` attributes of Declarations 5.35 and 5.36. Depending on the requirements of a composition system, different matching patterns may be used in general. In this section, it is assumed that fragments and slots are matched by name (Equations 5.41 and 5.42a), while hooks and rudiments are matched using prefixes and wildcards in the matching names (Equations 5.42b and 5.42c):

$$\textbf{fun}\ \{\texttt{Bind, Extend}\}\texttt{.srcFragment} = \texttt{lookUpF(pointName)} \tag{5.41}$$

$$\textbf{fun}\ \texttt{Bind.points} = \texttt{lookUpS(pointName)} \tag{5.42a}$$

$$\textbf{fun}\ \texttt{Extend.points} = \texttt{lookUpH(}\mathit{DFA}\texttt{(pointName))} \tag{5.42b}$$

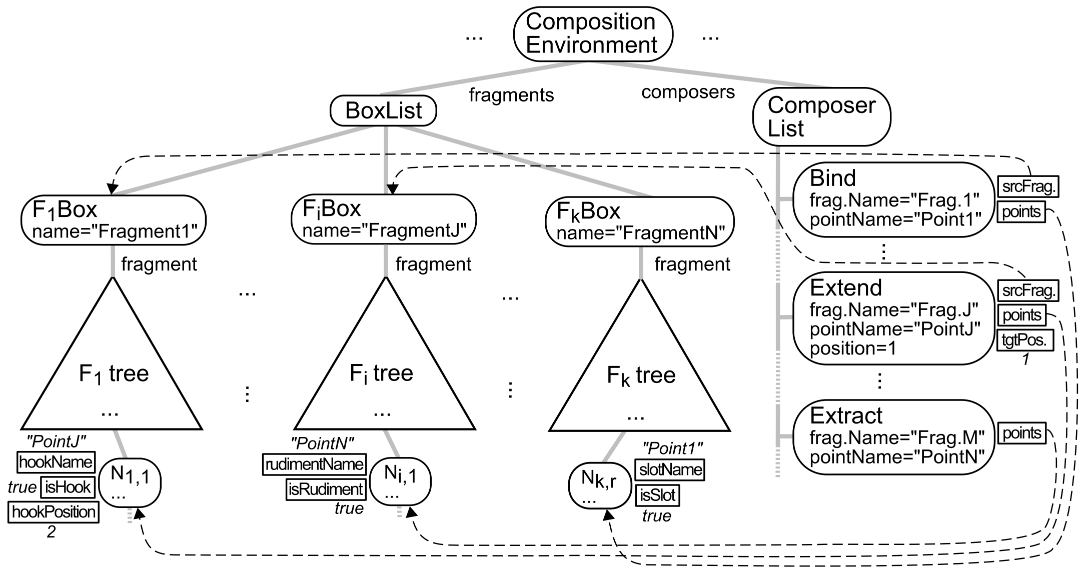


Figure 5.9.: An example composition-environment tree with three composers and evaluated reference attributes.

$$\textbf{fun}\ \texttt{Extract.points} = \texttt{lookUpR(}\mathit{DFA}\texttt{(pointName))} \quad (5.42c)$$

For hooks it is additionally suggested specifying an attribute to compute the hook's insertion position depending on its default or the position declared by users via the `position` terminal of `Extend`. The attribute `tgtPosition` defined in Declaration 5.43 and Equation 5.44 below provides the value of `position` terminal if it is declared, or otherwise the default `hookPosition` associated with the matched names of the matched hooks.

$$\textbf{syn}\ \texttt{int}_\perp\ \texttt{Extend.tgtPosition} \quad (5.43)$$

$$\textbf{fun}\ \texttt{Extend.tgtPosition} = \begin{cases} \texttt{position} & \textbf{if}\ \texttt{position} \neq \perp, \\ \texttt{hookPosition}\ \mathit{\texttt{for}} & \textbf{else}. \\ \mathit{\texttt{matched name}} & \end{cases} \quad (5.44)$$

Figure 5.9 augments the example composition-environment tree of Figure 5.8 with instances of the `srcFragment` and `points` attributes as well as three nodes representing points of the supported categories. The figure selectively shows the values of instances of point-identification attributes, and references from composers to points and fragments. Also, an instance of the `tgtPosition` attribute according to the above definition is shown.

It is important to notice that the equations of the look-up attributes support the classical box metaphor introduced by COMPOST (cf. Section 4.2) where fragments and composers are separate concepts. However, virtually `srcFragment` and `points` can implement arbitrary matching functions providing fragments from *anywhere* in the composition environment. This generalized view is discussed in the next section.

5.2.2. Generalized Composer Declaration

The approach of modeling primitive composers as explicit nonterminals in the `CompositionEnvironment` grammar can be generalized towards arbitrary composers embedded in a fragment component. The approach can be seen as a variant of *embedded invasive software composition (E-ISC)* [Henriksson 2009]. As for compositional points, it is straight-forward to use predicate attributes to determine composer signatures instead of using statically predefined declaration nonterminals like `Bind`, `Extend` or `Extract` as they were introduced in the `CompositionEnvironment` in the previous Section 5.2.1. Given the set N of nonterminals of the integrated composition-environment grammar, Declarations 5.45–5.47 declare a predicate attribute per each type of basic composition operator: `isBind`, `isExtend` and `isExtract`.

$$\textbf{syn}\ \texttt{bool}_{\perp}\ \{n \mid n \in N\}\texttt{.isBind} \tag{5.45}$$

$$\textbf{syn}\ \texttt{bool}_{\perp}\ \{n \mid n \in N\}\texttt{.isExtend} \tag{5.46}$$

$$\textbf{syn}\ \texttt{bool}_{\perp}\ \{n \mid n \in N\}\texttt{.isExtract} \tag{5.47}$$

For each nonterminal in N a corresponding equation has to be provided. By default and for non-composer nodes, the composer-identification attributes shall evaluate *false*, or *true* for composer nodes. The corresponding definitions are exemplified in Equations 5.48–5.53:

$$\textbf{fun}\ \{n \mid n \in N \setminus \mathcal{B}\}\texttt{.isBind} = \mathit{false} \tag{5.48}$$

$$\textbf{fun}\ \{n \mid n \in \mathcal{B}\}\texttt{.isBind} = \mathit{true} \tag{5.49}$$

$$\textbf{fun}\ \{n \mid n \in N \setminus \mathcal{E}\}\texttt{.isExtend} = \mathit{false} \tag{5.50}$$

$$\textbf{fun}\ \{n \mid n \in \mathcal{E}\}\texttt{.isExtend} = \mathit{true} \tag{5.51}$$

$$\textbf{fun}\ \{n \mid n \in N \setminus \mathcal{D}\}\texttt{.isExtract} = \mathit{false} \tag{5.52}$$

$$\textbf{fun}\ \{n \mid n \in \mathcal{D}\}\texttt{.isExtract} = \mathit{true} \tag{5.53}$$

Observe that $\mathcal{B}$ denotes a set of bind-composer nonterminals, $\mathcal{E}$ is a set of extend-composer nonterminals and $\mathcal{D}$ denotes a set of extract-composer nonterminals. Reconsidering the `CompositionEnvironment` grammar in Section 5.2.1, $\mathcal{B} = \{\texttt{Bind}\}$, $\mathcal{E} = \{\texttt{Extend}\}$ and $\mathcal{D} = \{\texttt{Extract}\}$.

Additionally, the other composer-related attributes need a generalization. In Declarations 5.54 to 5.56, the `points`, `srcFragment` and `tgtPosition` are redeclared in such a way that arbitrary nonterminals of the CnL can be provided with appropriate equations:

$$\textbf{syn}\ \texttt{Node*}\ \{n \mid n \in N\}\texttt{.points} \tag{5.54}$$

$$\textbf{syn}\ \texttt{Node}\ \{n \mid n \in N\}\texttt{.srcFragment} \tag{5.55}$$

$$\textbf{syn}\ \texttt{int}_\perp\ \{n\,|\,n \in N\}\texttt{.tgtPosition} \tag{5.56}$$

As before, for non-composer nodes default equations need to be provided yielding the bottom value ($\perp$). For composers, specific equations are provided. In the case of the `CompositionEn-vrironment` grammar and its dedicated composer nonterminals, Equations 5.41–5.42c apply for `points` and `srcFragment`, and Equation 5.44 applies for `tgtPosition`.

Based on the point-identification and composer-identification attributes, categories of embedded composers emerge. The categories classify composers according to their definition of point- and composer-identification attributes, and how they are integrated with the CnL, the CsL and the composition environment.

Primitive composers are part of the composition environment but not of the included CnL grammar. Thus, a primitive composer nonterminal *cop* never is a point so that $cop \in \mathcal{E} \cup \mathcal{B} \cup \mathcal{D} \implies cop \notin \mathcal{H} \cup \mathcal{S} \cup \mathcal{R}$. Moreover, `srcFragment` is a reference attribute depending on the fragment environment, i.e., the *cop* does not own or "produce" its argument fragment.

Primitive in-place composers are part of the composition environment *and* are part of the CnL grammar. Moreover, besides composer signatures they are also points. The `srcFragment` is provided by the composer itself and does not necessarily depend on fragment boxes in the environment. A typical primitive in-place composer is *include*, which binds itself to a loaded fragment.

Local in-place composers have the same properties as primitive composers, except that they perform multiple "local" composition operations such as binding slots in a loaded or copied fragment according to their own properties. Hence, local composers do not have further compositional side effects due to non-local compositions, except those on the CnL-semantics level, e.g., an imported declaration shadowing another one. A *macro call* is a typical local in-place composer which obtains and parametrizes a fragment from a macro declaration and replaces itself.

Non-local in-place composers have the same properties as local in-place composers, but their composition semantics has side effects, i.e., changes the fragment environment non-locally. For example, a *mixin composer* can extend a class with new methods and fields, finally extracting itself.

External composers are signatures of complex composers in an external, *dedicated* CsL with its own control flow. Hence, like primitive composers, they are never points in the CnL. However, they may reuse parts of the CnL to declare embedded fragments of the CnL and, internally, the composition environment's API to realize the compositional ingredients of the dedicated CsL. For example, *aspects* are an external CsL that uses a composition environment in its implementation.

Observe that composer signatures which are built into the CnL often are *syntactic sugar* which is transformed—i.e., *desugared*—by primitive or complex composers to lower level language constructs. Examples of in-place composers will be discussed in the case studies of minimal ISC in Chapter 7. In the remainder of this chapter, the model of Section 5.2.1 is further investigated. The next section discusses the semantics of composer declarations.

5.2.3. Composer Semantics

The semantics of `Composer` declarations is provided by the respective definitions of the primitive composition operators $\dot{\pi}$, $\ddot{\pi}$ and $\bar{\pi}$ given previously (Definitions 4.2 and 4.3), or in the appendix (Definition A.2). Thus, before considering the composers in a larger context of a composition program, it will shortly be discussed how the declarations of the composers are actually interpreted by mapping them to the defined composition operators. Therefore, an interpretation function $\|\mathcal{T}\| : \mathcal{T} \rightsquigarrow \mathit{dom}(\dot{\pi}) \cup \mathit{dom}(\ddot{\pi}) \cup \mathit{dom}(\bar{\pi})$ is proposed which defines the reduction of `Bind`, `Extend` and `Extract` nodes to an application of the corresponding operator functions. Rather than giving a complete definition of $\|\mathcal{T}\|$, the next few paragraphs only sketch out its basic working principles. The equations 5.57a–5.57f below describe how an `Extend` declaration is transformed step by step with $\|\mathcal{T}\|$ yielding an application of $\ddot{\pi}$:

$$\|\texttt{Extend[...]}\|$$
$$= \ddot{\pi}_{\|env\|}(\|\texttt{tgtFragment}\|, \|\texttt{srcFragment}\|, \|\texttt{points}\|, \|\texttt{tgtPosition}\|) \tag{5.57a}$$
$$= \ddot{\pi}_{FCM}(\|\texttt{tgtFragment}\|, \|\texttt{srcFragment}\|, \|\texttt{points}\|, \|\texttt{tgtPosition}\|) \tag{5.57b}$$
$$= \ddot{\pi}_{FCM}(T_1, \|\texttt{srcFragment}\|, \|\texttt{points}\|, \|\texttt{tgtPosition}\|) \tag{5.57c}$$
$$= \ddot{\pi}_{FCM}(T_1, T_2, \|\texttt{points}\|, \|\texttt{tgtPosition}\|) \tag{5.57d}$$
$$= \ddot{\pi}_{FCM}(T_1, T_2, \|\texttt{points}\|, x) \tag{5.57e}$$
$$= \ddot{\pi}_{FCM}(T_1, T_2, \{(T_1, \{v \,|\, v \in V_1 \wedge v = \|p\| \wedge p \in \texttt{points}\})\}, x) \tag{5.57f}$$

Observe that $T_1 = (V_1, E_1, Lab_1, \mathcal{L}_\Sigma)$ and $T_2 = (V_2, E_2, Lab_2, \mathcal{L}_\Sigma)$, $T_1, T_2 \in \mathcal{T}(\mathcal{L}_\Sigma)$, and $x \in \mathbb{N}_0$.

First, an FCM is derived from the current environment env. Since the compartments of an FCM in the `CompositionEnvironment` have been specified in Section 5.1.1, here details are omitted. Next, `tgtFragment` is transformed into the parameter T_1. The `tgtFragment` attribute is a helper attribute, which provides a reference to the fragment component containing all the hooks matched by `points`. In conformance with Definition 4.3, $\ddot{\pi}$ requires the fragment tree containing all compositional points as an argument. Therefore, it is assumed that all points in `points` are contained in the same fragment box and that `tgtFragment` provides a reference to the corresponding fragment. The fragment tree itself is then just interpreted as T_1. Similarly, `srcFragment` provides the second argument T_2, which shall extend the hooks of T_1. The respective target position in the list node is given by the `tgtPosition` attribute. The value x of the attribute is then used as a third parameter to $\ddot{\pi}$. The slots provided by `points` are then interpreted as a set of nodes v in T_1 and a corresponding mapping form T_1 to the set of mapped points.

As the single steps of interpreting `Bind` composer declarations are nearly equivalent to the interpretation of `Extend`, Equation 5.58a and 5.58b below only show the first and the last part of the transformation.

$$\|\texttt{Bind[...]}\| = \dot{\pi}_{\|env\|}(\|\texttt{tgtFragment}\|, \|\texttt{srcFragment}\|, \|\texttt{points}\|) \tag{5.58a}$$

$$\vdots$$

$$= \dot{\pi}_{FCM}(T_1, T_2, \{(T_1, \{v \,|\, v \in V_1 \wedge v = \|p\| \wedge p \in \texttt{points}\})\}) \tag{5.58b}$$

Also, the interpretation of `Extract` is similar to that of `Bind` and `Extend`, except that `srcFragment` and T_2 are not considered as they are not required by the operation. Equation 5.59a and 5.59b show begin and end of the respective interpretation with $\|\mathcal{T}\|$, where $T \in \mathcal{T}(\mathcal{L}_\Sigma)$ and $T = (V, E, Lab, \mathcal{L}_\Sigma)$:

$$\|\texttt{Extract[...]}\| = \bar{\pi}_{\|env\|}(\|\texttt{tgtFragment}\|, \|\texttt{points}\|) \tag{5.59a}$$

$$\vdots$$

$$= \bar{\pi}_{FCM}(T, \{(T, \{v \,|\, v \in V \wedge v = \|p\| \wedge p \in \texttt{points}\})\}) \tag{5.59b}$$

The next section puts the semantics of single composition steps into the context of composition programs and potential interpretation strategies.

5.3. Interpretation of Composition Programs

A set of `Composer` declarations has a complex rewrite semantics. In particular, semantics is determined by the primitive operators, the fragment component model provided by a `CompositionEnvironment` and an algorithm which decides when to apply a composer depending on a given situation.

There are several opportunities for writing composition programs. An *external* composition program keeps the composition semantically in separation of the composition environment using control-flow and recursion features of the composition language (CsL). To support external CsLs, the composition environment should support a *command* mode which allows clients to instruct compositions directly. For example, COMPOST uses Java as an external composition language and provides an API to instruct composition operations. Moreover, early versions of Reuse*air* provided dedicated languages for composition with a restricted API and typical imperative constructs like if-statements and loops (cf. [Karol 2008]). The advantage of an external—and sufficiently expressive—composition language is that arbitrary compositions can be expressed and, if a general-purpose language like Java is instrumented, the features and libraries of the host language are available. On the downside, the responsibility for the composition and all related tasks is up to the composition-program writer.

Composition approaches which are integrated with the composition environment help to reduce the number of tasks and problems the programmer has to deal with by providing declarative abstractions and strategies. Typically, composition steps are not independent of each other, i.e., each of them is part of a larger context in which one composition step may induce or prohibit others. In AOP, the general situation where one aspect influences another is called *aspect interaction* [Douence et al. 2002]. In [Kniesel and Bardey 2006; Kniesel 2009], aspect interactions are further distinguished into *semantic interactions* and *weaving interactions*. Semantic interactions occur if extensions added by an aspect A influence extensions woven by some aspect B "indirectly" at runtime, i.e., when the composed fragments are executed or interpreted according to the CnL semantics. Hence, A writes to the program state read by B. Complementary, weaving interactions are related to the semantics of weaving itself, i.e., to the semantics of the composition program. Weaving interactions between an aspect A and an aspect B occur if A changes one of the pointcuts of B or vice versa. For example, the weaving operations of A may provide new compositional points to a pointcut of B or remove points from it. [Kniesel and Bardey 2006] coin the terms *triggering* and *inhibition* to describe both situations: "A triggers or enables B if it adds, removes or modifies elements in such a way that B's predicate becomes true" while "A inhibits or disables B if it adds, removes or modifies elements in such a way that B's predicate becomes false."

This definition can be transferred to ISC terminology. Let F be an arbitrary fragment component of a FCM, and X and Y be composers matching the sets p and q of compositional points in F. Moreover, let F_x be the fragment tree resulting from an application of X to F and F_y be the fragment tree resulting from the application of Y to F. If q_x is the set of compositional points matched by Y in F_x and if there exists a compositional point v so that $v \notin q$ and $v \in q_x$, then X *triggers* Y at v. On the other hand, if $v \in q$ and $v \notin q_x$, then X *inhibits* Y at v. Interactions between composition steps are normality in ISC. For example, fragments regularly need to be parametrized before or after composition, or a hook needs an additional extension after its enclosing fragment itself has been used for extending another one.

While composition interactions are generally harmless, users and developers of a composition system still need to be aware of them and *how* the system evaluates composition programs—especially if the composition program is managed by the composition environment. Otherwise, unintended interactions between parts of a composition program may occur. Such unintended interactions are also called *composition interferences* in this thesis.

In general, an evaluator of composition programs has the following general problem to solve:

As long as compositional points, composers and matching points are available in the environment, and as long as no error occurs, choose a point, a corresponding composer and apply it.

For the concrete implementation of a composition-program evaluator solving the above problem, several approaches are viable. The first is *operator-determined composition* where the set of composer declarations in the environment is interpreted as a collection of ISC-specific rewrite rules which are evaluated by a fixpoint algorithm. The second approach is *point-determined* *composition*, where the composition depends on the occurrence of compositional points. This is related to aspect-weaving where aspects are woven at a joinpoint according to the advices matching that joinpoint. The third approach is *attribute-determined composition* where the composition is interleaved lazily with the attribute evaluation. The subsequent paragraphs discuss the three approaches.

5.3.1. Operator-Determined Composition

In this approach to composer evaluation, composer declarations in a composition environment are interpreted as a set of rules matching nodes in the composition environment and applying local AST rewrite operations to them. In contrast to rewrite systems in general [Ehrig et al. 2010; Baader and Nipkow 1999], the types of rewrites are given by the primitive composers of ISC—bind, extend and (optionally) extract—and by the FCM defined by the composition environment. Given a set of fragment components and a set of composer declarations, the basic idea is to apply the corresponding composition operators as long as compositional points can be matched by any declaration and, thus, fragments can be changed by composer application.

Algorithm 1 describes operator-determined composition in detail. As an input it takes a composition-environment tree env which is transformed and becomes the output of the algorithm. There are four global sets maintained by the outer `while` loop. *Op* is first initialized with the composer declarations of env—the "rules" to be applied. App is initially empty and contains composers that have been applied successfully at some compositional point. Complementary to App, Dy is used to maintain composer declarations which have been chosen for application but did not match any compositional point where it could have been applied to. Ex is a global set necessary to avoid unintended reapplications of composers. This occurs if a composer has an extension semantics such that the compositional point still exists after the composer has been applied to it. Therefore, Ex has a map-like structure containing pairs of AST nodes (v_1, v_2) where v_1 is a compositional point and v_2 is a composer node.

An iteration of the outer loop works as follows. After a composer cop has been chosen for application by the customizable function `nextComposer`, the set of applicable points *Pts* is computed from the points referenced by the cop's `points` attribute modulo the exhausted points associated with cop. The inner loop then traverses over the elements in *Pts* and applies cop at each point. After a successful evaluation of the inner loop, cop is removed from the operator set *Op*, which avoids the accidental reapplication of composers and ensures termination if *Op* does not grow. Afterwards, it is checked if the composer has been applied to any point by checking if it is an element of App. If not, cop is added to the set of delayed composers Dy. Finally, if *Op* is empty, a condition checks if any composer has been applied during the last *pass* (a complete traversal of *Op*) and decides which of the applied, exhausted or delayed composers could be reapplied in the next pass. The decision about this is made by the `reapply` function.

The inner loop traverses the elements of *Pts* and triggers the composition rewrite according to cop's type. First, a pt is chosen from the set by the `nextPoint` function. Next, it is checked if cop is one of the additive composer declarations `Bind` or `Extend`, or if it is a subtractive

Algorithm 1: The basic composer evaluation algorithm of ISC in pseudo code.

```
input   : a composition environment $env$
output  : the composed environment
$Op \leftarrow env.$composers – the set of composer declarations
$App \leftarrow \emptyset$ – the set of applied composers
$Dy \leftarrow \emptyset$ – the set of delayed composers
$Ex \leftarrow \emptyset$ – the set of exhausted $(point, composer)$ pairs
while $Op \neq \emptyset$ do                                        /* begin outer loop */
    $cop \leftarrow$ nextComposer($Op, App, Dy, Ex$)                 /* choose composer */
    $Pts \leftarrow \{v \mid v \in cop.$points $\wedge (v, cop) \notin Ex\}$       /* eval point set */
    $res \leftarrow \bot$
    while $Pts \neq \emptyset$ do                                   /* begin inner loop */
        $pt \leftarrow$ nextPoint($Pts$)                                /* choose point */
        if isBind($cop$) then
            $src \leftarrow$ pull($cop.$srcFragment,$|Pts|$)           /* pull fragment */
            $res \leftarrow$ doBind($env, src, pt$)
            if $res = OK$ then                                 /* mark as exhausted */
                $Ex \leftarrow Ex \cup \{(p, cop) \mid (q, cop) \in Ex \wedge$ isCopy($p, q$)$\}$
        else if isExtend($cop$) then
            $src \leftarrow$ pull($cop.$srcFragment,$|Pts|$)           /* pull fragment */
            $pos \leftarrow cop.$tgtPosition
            $res \leftarrow$ doExtend($env, src, pos, pt$)
            if $res = OK$ then                                 /* mark as exhausted */
                $Ex \leftarrow Ex \cup \{(pt, cop)\} \cup \{(p, cop) \mid (q, cop) \in Ex \wedge$ isCopy($p, q$)$\}$
        else // isExtract($cop$)
            $res \leftarrow$ doExtract($env, pt$)
        if $res = OK$ then                              /* composition successful */
            $App \leftarrow App \cup \{cop\}$
            $Pts \leftarrow Pts \setminus \{pt\}$
            if isExtract($cop$) then
                $Pts \leftarrow \{v \mid v \in Pts \wedge v \in env\}$            /* repair point set */
        else reportProblem($res$) and stop              /* composition failed */
    $Op \leftarrow Op \setminus \{cop\}$
    if $cop \notin App$ then $Dy \leftarrow Dy \cup \{cop\}$
    if $Op = \emptyset \wedge App \neq \emptyset$ then                 /* initialize next pass */
        $Op \leftarrow$ reapply($Dy, App, Ex$)   /* what should be reapplied?  */
        $App, Dy \leftarrow \emptyset$
```

`Extract`. In the case *cop* is additive and at least one slot or hook is in Pts, a fresh fragment tree is provided in *src* by the `pull` operation. In the case of an extend operation, the target list position *pos* is requested from the `tgtPosition` attribute of *cop*. A composition is executed by the `do[Composer]` operations which perform the rewrites of the fragments in *env* according to the declarations' semantics as discussed in the previous section. As a result, the operations return a status object which can be `OK`, if the composition was successful (i.e., syntactic integrity is ensured), or otherwise a problem-specific descriptor. To ensure that hooks are not extended more than once by the same composition operator, the exhausted set *Ex* is updated after each successful additive composition. Thereby, the `isCp` function evaluates *true* if its first argument node is a copy of the second argument, otherwise it is *false*. Finally, the inner loop checks the status object in *res*. If the status is `OK`, it performs an update of Pts and App. In the case the subtree rooted by the current *pt* was extracted from its enclosing fragment component, Pts is "repaired" in such a way that compositional points in that subtree are skipped by the loop. Otherwise, in the case of a problem occurring during composer application, the algorithm reports the status object via the `reportProblem` operation and stops.

The basic algorithm can be tailored towards different evaluation strategies determined by the concrete realizations of the functions and operations left open in the explanations above. Variations of the algorithm can be easily realized in object-oriented languages using the strategy design pattern [Gamma et al. 1995] or higher-order functions in functional programming. Of course, variations of the algorithm only have any importance if there are at least two interacting composer declarations in the composition environment, i.e., a composer which triggers or inhibits another one.

The outer loop can be varied in the implementations of `nextComposer`, `pull` and `reapply`. An implementation of `nextComposer` provides a precedence order of composition operations depending on the Op, App, Dy and Ex sets—the state of the outer loop's current pass. The following global composer-selection strategies are immediately applicable:

- **Non-deterministic**: the decision is random and independent of any property of any composer (e.g., its type) or the loop state.
- **Op-ordered**: the decision depends on the precedence order implied by the composer list in the composition environment. Hence, this strategy is determined by the composition system user.
- **Type-determined**: composer selection depends on the type of the composer and a precedence relation over the three composer types yielding six possible orderings. Which ordering is finally adequate depends on the kind of *pattern* used for matching points (cf. Section 5.2.1) and of course the specific requirements of the composition system user. For example, in case points are named using simple path expressions as suggested previously, it is reasonable to use the order `Extract` > `Bind` > `Extend`. Hence, inhibitions caused by extractions are preferred over any bindings are preferred over extensions, so that redundant matches of additive composers in later deleted subtrees are avoided (cf. [Karol

et al. 2011]). On the other hand, it can be reasonable to postpone extractions if complex patterns are used for matching [Aßmann 2000].

- **Type-op-combined**: the strategies above are combinable. A preselection may choose an Op subset depending on the composer type, the selection from this subset then may depend on the user-provided precedences.
- **Analysis-based**: the system can analyze the composer declarations at hand. For example, it can analyze the matching patterns of compositional points and derive a data-flow-based ordering of the composition operators so that composer declarations using source fragments without compositional dependencies (i.e., the fragments not manipulated by a composer themselves). This kind of analysis may also detect cyclic composition chains or conflicts (cf. [Karol et al. 2011]).

After a pass has been finished successfully by the outer loop (i.e., Op is empty) and at least one composer has been applied, `reapply` decides which composers should be available in the next pass. In principle, three options of varying `reapply` become emergent:

- **Single-pass:** the algorithm may terminate after the first pass by leaving Op empty. This is suitable if no composition interactions occur or, for example, if the composer declarations are preordered in such a way that triggering effects are guaranteed to be applied *before* the triggered compositions.
- **Multi-pass-delayed:** only delayed composers may be applied during the next pass, i.e., $Op = Dy$.
- **Multi-pass-all:** all composers may be applied in the next pass, i.e., $Op = Dy \cup App$. This is the most general reapplication strategy since all composers are checked for matching new points.

Also the inner loop has some complementing variation opportunities: the implementations of the `nextPoint` function as well as the `doExtract` and `pull` operation have variable realizations. While for additive composers the order of the single point-wise applications is not important, for extractions it may prefer upper-level rudiments over lower-level rudiments to avoid redundant deletions. Moreover, by default, the extraction operation only performs the deletion of complete subtrees rooted by the referenced rudiment. A variant of `doExtract` can construct a fresh fragment box from the deleted subtree using the `fragmentName` terminal child of `Extract` inherited from `Composer` (cf. Section 5.2.1). Using this advanced semantics, the `Extract` composer becomes an inverse of `Bind` and `Extend`—cutting of rudiment subtrees and providing them as fragment boxes for additive compositions. If $|Pts| > 1$, the advanced `Extract` creates more than one fragment by using `fragmentName` as the first part of the new fragment box and a unique index number as a suffix, e.g., $|Pts|$. Of course in that case, `nextPoint` needs to select rudiments in a deterministic order.

Considering the `pull` function, two basic realization options exist. First, `pull` can provide a fresh copy of the source fragment of *cop* at the cost of traversing the fragment's subtree and cloning it. The second option has a *move* semantics: the source fragment is cut off the fragment list avoiding the costs of creating an extra copy. Of course, this is only applicable if only one point remains to be composed and if there are no other composer declarations requiring that source fragment.

5.3.2. Point-Determined Composition

In contrast to the previous composer-determined approach to ISC composer interpretation, the approach discussed in the subsequent paragraph is based on a point-wise traversal algorithm. This view on fragment composition is closer to the notion of aspect weaving where aspects are woven at specific joinpoints in the execution of a program. Given a set of fragments and a set of composer declarations, the algorithm visits the compositional points of each fragment by traversing the underlying AST, looking up composers associated with each point and applying them in a specific order.

Algorithm 2 describes point-determined composition in more detail. It takes a `Compositio-nEnvironment` env as an input parameter and emits it after the composition is finished. To maintain the traversal steps, the algorithm uses a $stack$, which is initially empty (the bottom of the stack is #). The outer foreach loop inspects each fragment box registered in the environment pushing the root node of the encapsulated fragment AST to the $stack$ so that it becomes the top-level element on the stack. In each iteration, the inner while loop then pops the top-level element from the $stack$. For the obtained node pt it is then checked if the node is a point (i.e., if one of pt.`isSlot`, pt.`isHook` or pt.`isRudiment` evaluates *true*) and if it has matching `Composer` declarations, which are obtained via the `composers` attribute. It computes the inverse of the `Composer`'s `points` attribute and has therefore a simple specification which is omitted here. If pt is not a compositional point or has no associated `Composer`, its child nodes are pushed to $stack$ before continuing with the next iteration. Otherwise, if composers are associated with pt, it is first checked if a `Bind` declaration is in the set. That is, bind operations are preferred over extractions and extensions. Consequently, if pt is a rudiment and a slot at the same time, a bind is executed first. In the main loop's next iteration, the `composers` attribute is reevaluated with respect to the new subtree *src* potentially revising the original rudiment status of the node at this position in the current fragment. In case more than one `Bind` declaration is associated with pt, `chooseBind` selects the one to be applied, e.g., based on the declarations order in the `composers` list of the environment. After a successful composition, the new subtree's root is pushed on the stack and the set of exhausted composers is updated in such a way that copies of exhausted nodes in the inserted subtree are also marked as exhausted to avoid unintended and redundant composer applications.

If pt is a rudiment with at least one associated `Extract` declaration and if it is not a slot with associated `Bind`, the extract composer is executed and the subtree with the root node pt is removed from the fragment. That is, in this variant of point-determined composition, deletions

Algorithm 2: The point-wise composition algorithm of ISC in pseudo code.

```
input  : a composition environment $env$
output : the composed environment
$Ex \leftarrow \emptyset$ – the set of exhausted ($point, composer$) pairs
$stack \leftarrow [\#]$ – the stack of points to be processed, initially empty
foreach $box \in env$.fragments do              /* outer box loop (optional) */
    $root \leftarrow box$.fragment
    $stack$.push($root$)                          /* push fragment root on stack */
    while $stack$.peek $\neq \#$ do                                 /* main loop */
        $pt \leftarrow stack.pop$                      /* get top node from stack */
        if $pt$.isPoint & $pt$.composers $\neq \emptyset$ then
            $Op \leftarrow pt$.composers         /* request associated composers */
            $res \leftarrow OK$
            if $pt$.isSlot & $\exists v \in Op$ : isBind($v$) then             /* bind */
                $cop \leftarrow$ chooseBind($Op$)               /* choose composer */
                $src \leftarrow$ pull($cop$.srcFragment)          /* pull fragment */
                $res \leftarrow$ doBind($env, src, pt$)
                if $res = OK$ then                           /* mark as exhausted */
                    $Ex \leftarrow Ex \cup \{(p, cop) \mid (q, cop) \in Ex \wedge$ isCopy($p, q$)$\}$
                    $stack$.push($src$)         /* push subtree root on stack */
            else if $pt$.isRud. & $\exists v \in Op$ : isExtract($v$) then     /* extract */
                $res \leftarrow$ doExtract($env, pt$)
            else if $pt$.isHook & $\exists v \in Op$ : isExtend($v$) then       /* extend */
                while $res = OK$ & $\exists v \in Op$ : isExtend($v$) $\wedge (v, cop) \notin Ex$ do
                    $cop \leftarrow$ chooseExtend($Op, Ex$)      /* choose composer */
                    $src \leftarrow$ pull($cop$.srcFragment)      /* pull fragment */
                    $pos \leftarrow cop$.tgtPosition
                    $res \leftarrow$ doExtend($env, src, pos, pt$)
                    if $res = OK$ then                       /* mark as exhausted */
                        $Ex \leftarrow Ex \cup \{(pt, cop)\} \cup \{(p, cop) | (q, cop) \in Ex \wedge$ isCopy($p, q$)$\}$
                if $res = OK$ then                   /* push children on stack */
                    foreach $chi \in pt$.child$_{all}$ do  $stack$.push($chi$)
            else $res = NO\_COMPOSER$
            if $res \neq OK$ then                           /* composition failed */
                reportProblem($res$) and stop
        else                                         /* push children on stack */
            foreach $chi \in pt$.child$_{all}$ do  $stack$.push($chi$)
```

are preferred over extensions. Assuming that only simple path-based names are used for matching points and that extensions at pt or general compositions in its subtree do not revise pt's rudiment status, the additional compositions are redundant and should be avoided.

If neither bind nor extract have been applied, the algorithm interprets and applies all non-exhausted `Extend` declarations. As hooks are list nodes, the insertion order matters. Thus, `chooseExtend` selects the next composer to be applied from Op under consideration of exhausted entries in Ex. As already suggested before, a natural choice is a user-provided ordering in the composers list of env. After each successful extend composition step, Ex is updated. Finally, all child nodes of pt—including the freshly added ones—are pushed to the $stack$.

Figure 5.10 shows a snapshot view on a composition environment with at least four fragment components which is interpreted by the point-determined composition algorithm. Part 1 of the figure represents a visit situation of a node v with label N which is a slot and a rudiment at the same time. Via `composers`, v is associated with `Bind` and `Extract` declarations. According to Algorithm 2, the `Bind` composer is preferred over `Extract` so that the slot `"Point1"` is bound to the contents of the left-most $\texttt{F}_1\texttt{Box}$ named `"Fragment1"`. The result of the composition is shown in Part 2 of the figure. Due to the bind application, `isRudiment` and `isSlot` values change to *false* so that the new $\texttt{F}_1$ node is not recognized as a slot or rudiment by the composition system. However, in the new context, it is recognized as a hook with the name `"PointJ"`. As there is not yet an entry in the exhausted set Ex it is matched by the two `Extend` declarations. The algorithm applies both of them according to their occurrence. First (step 2a), the contents of the $\texttt{F}_k\texttt{Box}$ are appended at the end of the hook list which is denoted by the value n of the `tgtPosition` attribute. In the second step (2b), the contents of the $\texttt{F}_1\texttt{Box}$ are inserted at position 1 of the hook. The final result of the composition excerpt is shown in the bottom part of Figure 5.10. None of the displayed composers can be applied at the position of the original node v: `Bind` and `Extract` do not match while the `Extend` declarations are marked as exhausted in the internal state of the algorithm. Hence, the algorithm continues to interpret the left-most child of the new $\texttt{F}_1$ node.

5.3.3. Complexity of the Basic Algorithms

In this section, the efficiency of RAG-based composition is discussed, essentially with respect to Algorithm 1 and Algorithm 2. The time it takes to execute a composition program depends on the following parameters:

- n—the cumulative number of nodes in fragment components,
- p—the cumulative number of points in fragment components, in worst case $p = n$, and
- c—the number of composer declarations in the environment.

Three specific cases of algorithmic behaviour can be distinguished.

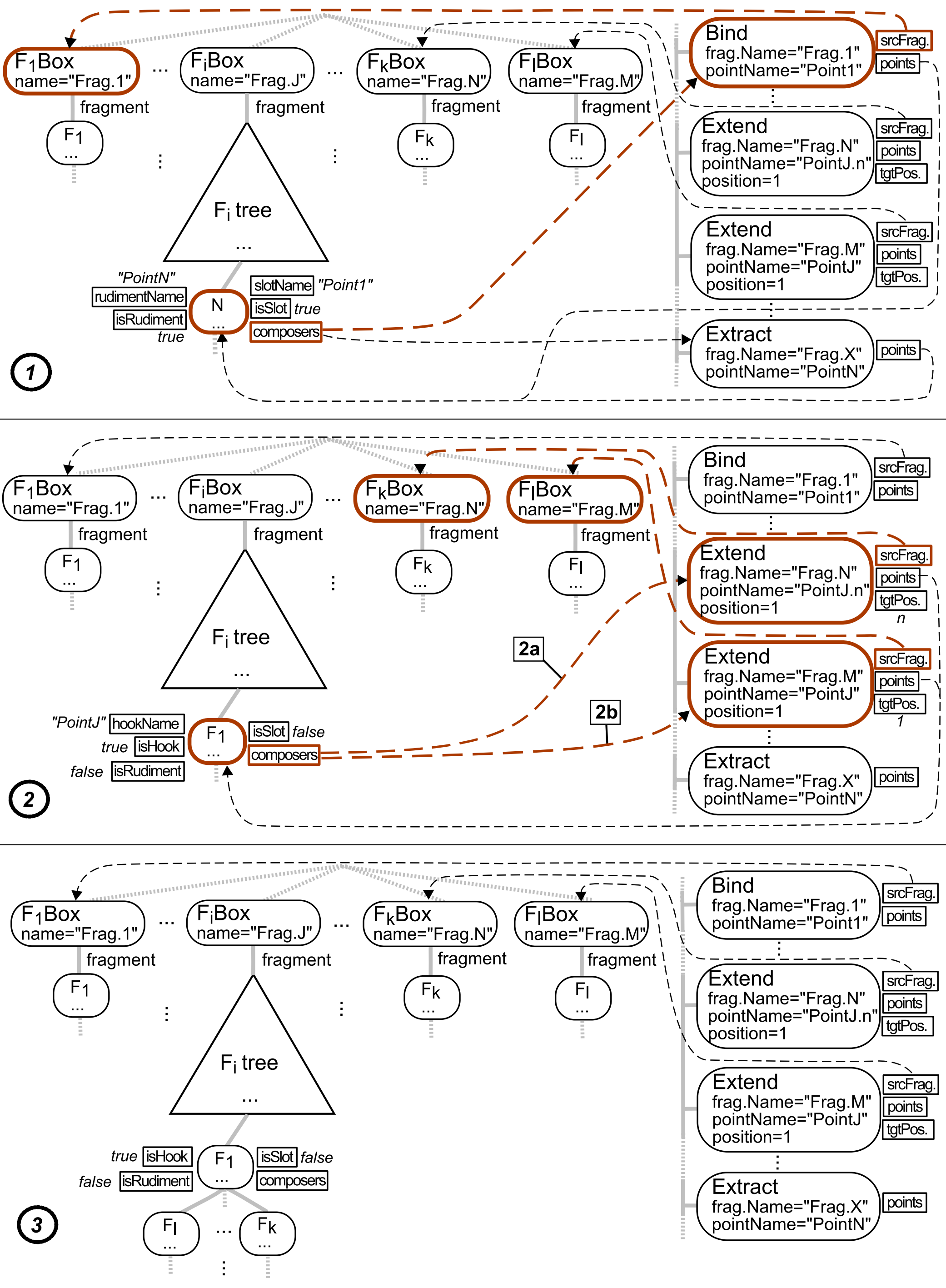


Figure 5.10.: An example point-wise composition in three steps.

Case 1: Composition in one pass, no triggerings occur. Cost for search and comparison without matching costs:

$$cost = n * c \tag{5.60}$$

Considering complex matching conditions:

$$cost = c * (\mathrm{matchCost}(n) + \mathrm{matchCost}(fgmt) + \mathrm{copyCost}(fgmt)) \tag{5.61}$$

–or–

$$cost = n * (\mathrm{matchCost}(c) + \mathrm{matchCost}(fgmt) + \mathrm{copyCost}(fgmt)) \tag{5.62}$$

where $\mathrm{matchCost}(n) = k * n$ if all nodes were iterated and matched in constant time, in the arbitrary case $\mathrm{matchCost}(n) >> k * n$, in the informed case (e.g., if paths and DFAs are used) $\mathrm{matchCost}(n) = k * \log(n)$. Considering fragment matching, $\mathrm{matchCost}(fgmt) = f$ where f is typically the number of fragments in the list of fragment boxes. In general, it can have an arbitrary complexity. Considering the costs of copying a fragment, $\mathrm{copyCost}(fgmt) = n$ is typically the number of nodes in the fragment (a partition of n). If a fragment is used only once, the cost of moving it is $n = 1$. Hence, in the case of operator-determined composition with DFA-based look-up (Algorithm 1):

$$cost = c * (k_1 * \log(n) + f + k_2 * n) \tag{5.63}$$

Case 2: Compositions are triggered, however the number of points monotonically decreases in each pass. In the case of composition scenarios which require multiple passes, termination can be guaranteed if the number of points associated with composers that can be found in the composition environment steadily decreases. Hence, in each pass at least one point is removed from the set, so that in the worst case another factor of n contributes to the cost function:

$$cost' = n * cost \tag{5.64}$$

Consequently, the problem has at least a quadratic complexity if c is considered as a constant. However, in the generalized composer case, any node in the fragment environment can be a composer so that with $c = n$ it results in an at least cubic complexity.

Case 3: The number of composition steps is unbound. In this case, the computation time is also unbound and the composition program may not terminate.

5.3.4. Attribute-Determined Composition

A third variant of composition-program evaluation is the integration of composition rewrites with the attribute evaluator. Hence, the traversal algorithms presented above are replaced by the attribute-evaluation algorithm and the composer interpretation is embedded into it. Demand-driven evaluators solve attribute equations on demand, i.e., given a set of “request” attribute instances, the evaluation function recursively visits the nodes in the AST solving the equations of the attribute instances the requested instances depend on. For non-circular AGs and RAGs, the ordering of evaluation steps results from a topological sort of attribute instances according to the attribute-dependency graph or data-flow graph of the current AST. The evaluator starts with the *minimum elements*—the attribute instances without dependencies—computing their values based on the equations specified in the AG (cf. [Wilhelm and Maurer 1997]). Solutions of the minimum elements provide values for the evaluation of depending instances. In the next iteration, the evaluator computes values of attribute instances depending only on instances already computed previously. This is successively repeated until all dependencies of the *maximum elements* are resolved and their values were computed. Finally, the evaluator returns the computed values of the requested instances.

Classical AG evaluators typically compute the complete attribution of the whole syntax tree based on a statically precomputed visiting order (cf. [Wilhelm and Maurer 1997]) while most modern RAG systems are demand-driven and use dynamic visitation strategies that depend on the given AST and the requested attribute instances.

For demand-driven RAG evaluators, [Ekman and Hedin 2004] proposed an algorithm and notation for *conditional rewrite* rules. A RAG with conditional rewrite rules is called a *rewritable reference attribute grammar (ReRAG)*. In contrast to classical higher-order attributes, which are unconditional, conditional rewrites can easily be adopted for the operators of ISC leading to a demand-driven, attribute-integrated interpretation of composition programs. Conditional rewrites are transparently integrated into the dynamic evaluation algorithm. “Transparently” means that, before an AST node or one of its attributes is accessed[1], the attribute evaluator checks if rewrite rules are available for that node, checks their Boolean conditions and successively applies those rewrites with a positively evaluated (*true*) precondition. Following [Ekman and Hedin 2004], a conditional rewrite rule can be specified using the format in Equation 5.65:

$$\texttt{\textbf{rewrite}}\ \texttt{nt}_1\ \texttt{\textbf{when}}\ \textit{\texttt{condition}}\ \texttt{\textbf{to}}\ \texttt{nt}_2\ \texttt{\textbf{with}}\ \textit{\texttt{expression}} \qquad (5.65)$$

A rewrite rule r is specified with respect to a node v labeled with a nonterminal $\texttt{nt}_1 \in N$, where N again is the set of nonterminals of the attributed grammar. Hence, if the attribute evaluator visits a node v with label $\texttt{nt}_1$, it checks if this rule is applicable—in other words, an $\texttt{nt}_1$-labeled node is a rewrite *candidate* and r is a *candidate rule*. The system rewrites any candidate at the very beginning of its visitation and before the actual attribute evaluation if there is at least one applicable candidate rule. The decision about a rule’s application is depends on `condition`, which is a Boolean expression over AST nodes, terminal values and attribute instances, type information as well as any other information provided by the RAG system.[2] Moreover, as a

[1]For instance, an attribute of that node shall be computed, child nodes or other properties of the node are requested.
[2]A “closed” system may only allow access to RAG-specific information via a well-defined API while an “open” system does not have such restrictions.

convention *condition* is scoped w.r.t. the current node, which can be accessed via SimpAG pseudo code *node*.

The nonterminal $\texttt{nt}_2 \in N$ is the label of subtree's root node which is constructed to replace the currently visited node and its subtree. According to [Ekman and Hedin 2004], the $\texttt{nt}_2$ subtree is only allowed to replace the $\texttt{nt}_1$ subtree if their types are "compatible" w.r.t. the type hierarchy in the AST grammar. This can be generalized to term-tree replacement. Let $T_1 = s[\ldots n_i[\ldots] \ldots]$ be a syntax tree w.r.t. $G = (N, \Sigma, P, s)$ where $s \Rrightarrow^*_G T_1$, $n_i[\ldots]$ is the subtree with root v_i to be rewritten and $T_2 = m[\ldots]$ is the replacement tree where the n_i correspond to $\texttt{nt}_1$ and $\texttt{nt}_2$ corresponds to m. For a valid rewrite, the following conditions must hold (cf. Section 4.1.3):

- if $T_1' = T_1[\{v_i\}/T_2] = s[\ldots m[\ldots] \ldots]$ is the rewritten tree, then $s \Rrightarrow^*_G T_1'$,
- the set of inherited attributes $\mathrm{Inh}_x(m)$ must be equivalent to $\mathrm{Inh}_x(n_i)$,
- the set of synthesized attributes $\mathrm{Syn}_x(m)$ must be a super set of the used attributes of $\mathrm{Syn}_x(n_i)$ which are actually required by equations in v_i's context p_i,
- the equations in the new context p_i' must be equivalent to the equations applicable in p_i.

The replacement subtree itself is created by a construction *expression* which yields an AST fragment with a root node labeled $\texttt{nt}_2$. Similar to *condition*, the *expression* can be created using arbitrary attributes, terminals and nonterminals available at the current node. For example, the replacement can be constructed by copying referenced subtrees or a programmed construction.

Since attributes are accessible in *condition* and *expression*, other rewrites may be triggered *on demand* by the attribute evaluator at distant candidate nodes somewhere in the AST. Consequently, before the current rewrite can be finished, dependent rewrites are conducted. That means, the rewrite order is declaratively induced by attribute dependencies: given a set of attribute instances $I_s \subseteq \mathrm{AttI}_x(v_s)$ of a node v_s (cf. Section 3.3.1), the ReRAG evaluator starts to visit the nodes and attribute instances of the AST according to its visitation scheme, typically beginning at v_s trying to compute attribute values there and successively traversing the tree to compute pending attribute values to finally solve the maximum equations. At each visited node v_r, it applies the rewrites with a positive precondition and, as the precondition and the construction expression can use attributes, it recursively triggers the evaluator, which then has the attribute instances $I_r \subseteq \mathrm{AttI}_x(v_r)$ of v_r used the current rewrite as maximum elements to be solved.

The procedures `eval` and `applyRewrites` sketch the ReRAG algorithm using pseudo code. Assuming the evaluation of only one attribute instance *att* at a time, `eval` maintains a set of local direct dependencies D of *att* at v and a set of solutions S of the equations of these direct dependencies. In a first step, `eval` checks, if v has associated rewrite rules and if it is not yet rewritten. If this is the case, it calls `applyRewrites` on v. Afterwards, it uses the local dependencies to other attribute instances to compute a value for each of them by calling `eval` with the dependencies as a parameter, which may also cause further rewrites. After all

Procedure eval(Node,Attribute) : a simplified demand-driven attribute-evaluation function with rewrites.

```
input  : an AST node $v$
input  : an attribute $att$ of $v$
output : the value of $att$ at $v$

$D \leftarrow \emptyset$ – the set of dependencies $(node, attribute)$ of $att$ at $v$
$S \leftarrow \emptyset$ – the set of solutions $(node, attribute, value)$ to the dependencies in $D$
                                  /* check if rewrites can be applied:  */
if inRewrite($v$) = $false$ & rulesOf($v$) $\neq \emptyset$ then
    $v \leftarrow$ applyRewrites($v$)           /* call rewrite procedure on $v$ */
$D \leftarrow$ depOf($v, att$)                  /* request direct dependencies */
foreach $(c, att) \in D$ do
    $res \leftarrow$ eval($c, att$)             /* recursive call to evaluator */
    $S \leftarrow S \cup (c, att, res)$
return evalEq($v, att, S$)      /* finally call equation interpreter */
```

dependencies have been resolved, evaluator interprets the equation of att in the current context and returns the computed result.

The procedure `applyRewrites` starts with an unconditional outer loop. It first looks up v's rewrite rules and sets its status flag to *false*. The inner loop then successively tries to apply the rewrite in the rule set as long as rules are available and none has been applied. In each iteration, `nextRule` chooses a rule from the set (e.g., according to a prioritizing order) and updates the rewrite status of v. Then, it checks if *`condition`* evaluates *true* and constructs the replacement tree by evaluating *`expression`*. Observe that calling an attribute `att` on a node v is equivalent to invoking `eval`(v, `att`). Infinite direct and indirect recursions of v rewrites are avoided by the status check so that the values are computed w.r.t. the not-rewritten node (cf. [Ekman and Hedin 2004]). After the replacement of v's subtree has been executed via `doReplace` and the status flag is set to *true* so that the inner loop terminates, the root of the new subtree becomes the node-under-rewrite in the next iteration of the outer loop.

To combine a ReRAG rewrite engine such as the one above with an ISC fragment environment, ISC-specific rewrite rules need to be specified in such a way that the engine can evaluate them. The following declarations specify ReRAG rules for each composition operator of ISC based on the notation introduced by Declaration 5.65. Declaration 5.66 below specifies a rule for the bind composition operator for arbitrary nonterminals n of the CnL.

$$\begin{aligned}&\texttt{\textbf{rewrite}}\ n\ \textbf{when}\ \texttt{isSlot} = \mathit{true} \wedge (\exists \mathit{cop} \in \texttt{composers}\!: \mathit{cop}.\texttt{isBind})\\ &\quad \texttt{\textbf{to}}\ n'\ \texttt{\textbf{with}}\ \texttt{pull(chooseBind(composers).srcFragment)}\end{aligned} \tag{5.66}$$

Like the previous composition algorithms, by its application condition, it is checked if a visited

Procedure applyRewrites(Node) : attribute-driven rewrites.

```
input   : an AST node $v$
output  : the rewritten node if rewrites apply or $v$ if not
repeat                                                    /* start outer loop */
    $R \leftarrow$ rulesOf($v$)
    $applied \leftarrow false$                            /* update status flag */
    while $R \neq \emptyset$ & $applied = false$ do       /* start inner loop */
        $rule \leftarrow$ nextRule($R$)                   /* choose next rule */
        $R \leftarrow R \setminus \{rule\}$
        setInRewrite($v, true$)              /* mark $v$ as being rewritten */
        if eval($v, rule.condition$) $= true$ then
            $t \leftarrow$ eval($v, rule.expression$)     /* evaluate expression */
            setInRewrite($v, false$)
            $v \leftarrow$ doReplace($v, t$)              /* replace subtree */
            $applied \leftarrow true$                     /* update status flag */
        else
            setInRewrite($v, false$)                      /* unmark $v$ */
until $applied = false$
return $v$
```

node labeled with n is a slot according to the FCM and has a matching `Bind` composer declaration in the composition program. If this is the case, the rewrite is executed by choosing an applicable `Bind` declaration (e.g., the first one in the composer list) and `pulls` the fragment referenced by the `srcFragment` attribute of `Bind`. The rewrite engine then replaces the current node and its subtree with the new fragment. Observe that the root of the fragment must be labeled with a nonterminal n' and must retain the AST's integrity w.r.t the CnL grammar.

Declaration 5.67 specifies an extend rewrite (*node* refers to the current node):

```
rewrite $n$ when
    isHook $\wedge$ ($\exists cop \in$ composers: $cop$.isExtend $\wedge$ ($node, cop$) $\notin Ex$)
to $n$ with {
    $cop \leftarrow$ chooseExtend(composers, $Ex$)                         (5.67)
    doExtend(pull($cop$.srcFragment), $cop$.tgtPosition, $node$)
    $Ex \leftarrow Ex \cup \{(node, cop)\}$
    return $node$}
```

In its application condition, it checks if the current node is a hook and if a non-exhausted `Extend` declaration exists in its `composers` list. If the condition is *true*, the rewrite is executed by choosing an applicable `Extend` declaration and applying it to the current hook via `doExtend`. Like in the algorithms presented before, after the composition step is executed, Ex is updated to avoid accidental reapplications. Finally, the rewrite rule emits the extended hook.

Declaration 5.68 specifies a fragment-extraction rewrite:

```
**rewrite** $n$ **when** $\exists c \in \texttt{child}_{\texttt{all}}\, \exists cop \in c.\texttt{composers}$:
    $c.\texttt{isRudiment} \land cop.\texttt{isExtract}$
**to** $n$ **with** {                                                    (5.68)
    **foreach** $c \in \{m \mid m \in \texttt{child}_{\texttt{all}}$
      $\land\, m.\texttt{isRudiment} \land (\exists cop \in m.\texttt{composers}: cop.\texttt{isExtract})\}$
        doExtract($c$)
    **return** node}
```

The rewrite precondition checks if the currently visited node has children which are rudiments that have a matching `Extract` declaration. If this is the case, the rewrite extracts each rudiment child from the current node and emits the rewritten node as a result.

In comparison to the previous algorithm, the ReRAG implementation of ISC has some advantages. In contrast to Algorithm 1 and Algorithm 2, demand-driven composition is lazy—composing only fragments which are required by the evaluated attributes. Moreover, composition semantics and FCM declarations are seamlessly embedded into one formalism. However, there are some drawbacks in the approach which hinder a general superiority over the other algorithms. The composition rewrites may interfere with other rewrites of the CnL ReRAG which are not backed by the FCM. Such rewrites can destroy composition results or interfere with the matching of compositional points. Hence, besides the composition operators, composition-system developers have to take care of CnL-specific rewrites. This seems not to be a problem as long as the roles of the CnL developer, the FCM designer and the composition programmer are played by the same person or team. Otherwise, composition-program development becomes complicated as it is typically not transparent to the programmer which internal rewrite occurs when and how it would interfere with the composition. While [Ekman and Hedin 2004] assume confluency of rewrites as the normal case for ReRAGs, this is not the case for composition programs, where the interaction (i.e., triggering and inhibition) of composition steps is common. Even worse, interferences between CnL rewrites and composition rewrites can easily cause non-terminating behavior. For example, a CnL rewrite may add a new child node to a fragment assuming that the absence of that node is the rewrite condition. Now assume that an `Extract` composer subsequently removes that node making the rewrite condition true again causing an endless loop. Of course, this may also happen the other way around, if a CnL rewrite deletes a specific kind of node from a hook which is then readded via an `Extend` operation. Generally, composition rewrites should not be contradicting to the rewrites of the CnL because these belong to the static CnL semantics and, thus, contribute to its computation. FCMs should be designed in such a way that accidental interferences are avoided and composition is forbidden in these cases.

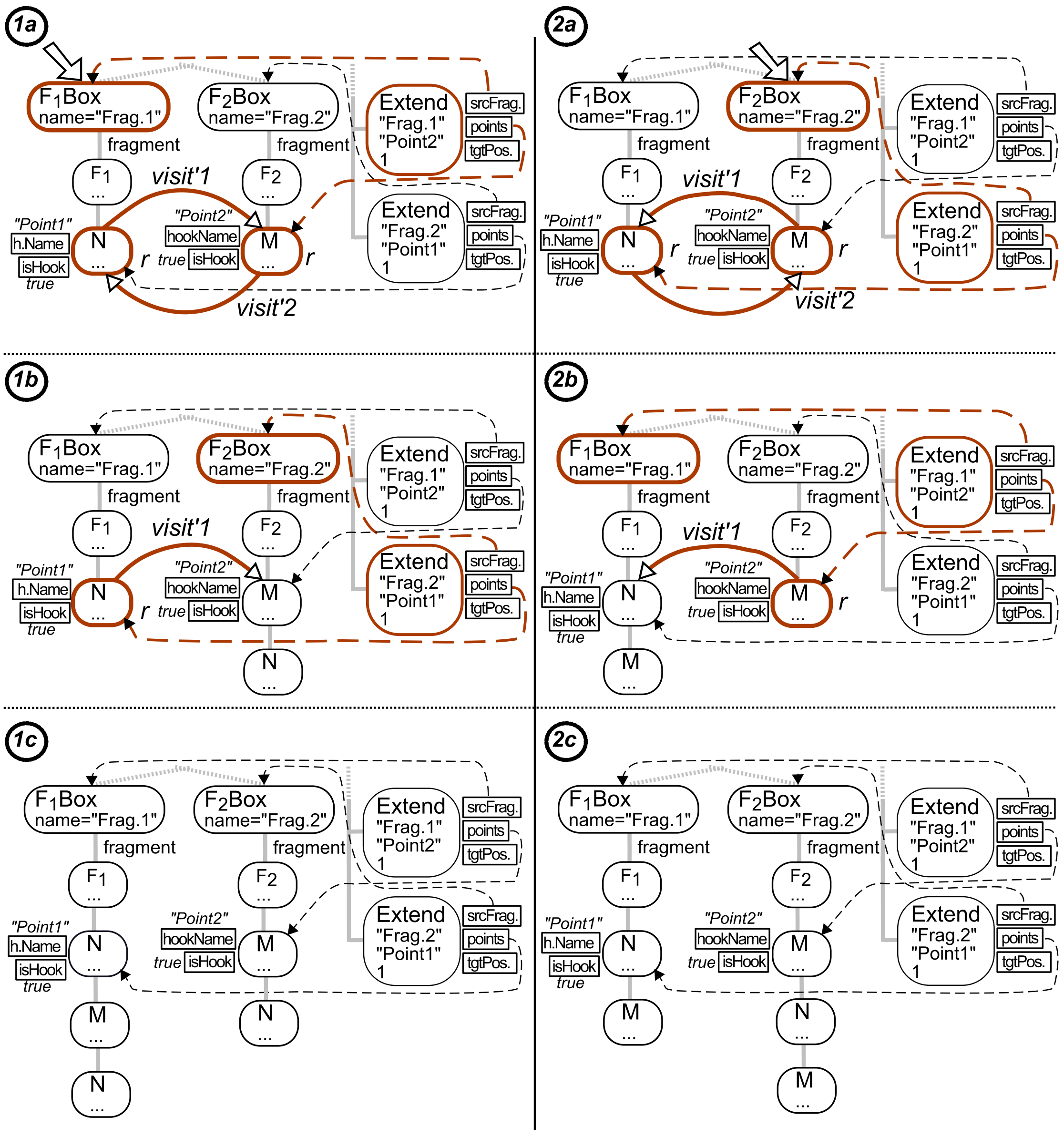


Figure 5.11.: Attribute-determined composition can yield different results depending on where the ReRAG evaluation starts.

But also if composition operations are isolated from CnL-specific rewrites, it is not easy for the composition-system user to predict the result of a composition program in the presence of attributes and demand-driven evaluation: depending on *where* in the composition environment the attribute evaluation starts (i.e., which attribute instances are requested for evaluation), the result can be different. This is due to cyclic rewrite (composer) dependencies. A simple composition scenario in Figure 5.11 exemplifies potential effects of circular composer dependencies. It shows composition phases of an example composition environment with two fragment components and two `Extend` composer declarations. On the figure's left-hand side, the ReRAG evaluator started its visit at the $\texttt{F}_1\texttt{Box}$. Situation `1a` describes the following: after reaching the `N`-labeled node v_1 with `hookName "Point1"`, the evaluator activated v_1's rewrite status (symbolized by annotating `r` to the respective node) and looked up the matching `Extend` declaration for `"Point1"` and the argument started to look up the `srcFragment` in the $\texttt{F}_2\texttt{Box}$. This look-up caused a visit of the `M`-labeled node v_2 which also is a hook and has $\texttt{F}_1\texttt{Box}$ as an argument. The second visit of v_1 now returns the contents of $\texttt{F}_1\texttt{Box}$ because v_1 is already marked with the rewrite flag (cf. Procedure `applyRewrites`). Thus, the hook `"Point2"` is extended with v_1 and its subtree—as shown in Situation `1b`. Afterwards, the rewrite status is removed from v_2 and the hook `"Point1"` is extended with the contents of $\texttt{F}_2\texttt{Box}$—the extended v_2 and its subtree. The composition's final result is shown in Situation `1c`, where `"Point1"` was extended with an $M[\ldots N[\ldots]\ldots]$ tree and `"Point2"` with an $N[\ldots]$ tree. In the right part of Figure 5.11, the same composition scenario is shown with an evaluation starting at $\texttt{F}_2\texttt{Box}$. Hence, this time, the `M`-labeled node v_2 is visited first so that the rewrite order is mirrored (Situation `2a`). Consequently, v_1 is extended first (Situation `2b`) and v_2 is second (Situation `2c`) yielding a different composition result: `"Point1"` is extended with an $M[\ldots]$ tree while `"Point2"` is extended with an $N[\ldots M[\ldots]\ldots]$ tree.

The drawbacks of ReRAG-based composition can be tempered by avoiding cyclic rewrite dependencies such as those described above and by supporting distinct rewrite and evaluation phases. Also, a deterministically enforced starting point or a global evaluation strategy of the evaluator can make the ReRAG-based approach more reliable.

5.3.5. Static Attribute Dependencies

An alternative idea for AG-based ISC was investigated in a master thesis [Thiele 2011] supervised by the author of this thesis. The developed approach is based on classic static evaluation-order generation for ANCAGs—a proper subclass of non-circular attribute grammars [Kennedy and Warren 1976] *without* reference attributes. In contrast to the approach presented in this chapter, composition operators are built into the AG tool and specification language as first class language constructs and special kinds of inherited attributes for the bind and extend composition operators. Hence, in addition to Syn_x and Inh_x (cf. Section 3.3), $\mathrm{Bind}_x \subseteq \mathrm{Inh}_x$ and $\mathrm{Extend}_x \subseteq \mathrm{Inh}_x$ assign bind and extend attributes to each nonterminal of the underlying CFG. Associated to each production of the CFG, bind and extend equations have to be provided. As the ReRAG rewrites, the composition equations have an application condition that decides whether a composition should be applied, e.g., using FCM-specification attributes like `isSlot` and `isHook`. To statically determine the evaluation and composition order, the local dependencies induced by the composition attributes are included in the nonterminal-specific IS-graphs[3]. Based on these composition-aware IS-graphs, an ANCAG evaluation order is generated that detects potential cyclic dependencies between attributes and/or composition operators.

While having a statically predetermined composition order is clearly beneficial, as it provides some checks of the evaluation orders and avoids redundant attribute computations, the ANCAG-based composition approach has some disadvantages. The main issue is that modern AG tools typically support reference attributes which are not considered by ANCAG algorithms. Thus, from a practical point of view, ANCAGs are not as expressive as modern AGs tending to bloat the specifications of real world compiler frontends. Moreover, ANCAG use an *over-approximation* of the actual dependency relations that may occur during attribute evaluation at runtime. The over-approximation is due to the nonterminal-wise computation of the local dependency graphs and to the fact that all possible dependencies of an equation are considered—also those which may never occur at runtime, e.g., because there is an if-statement in the equation whose branches may induce different subsets of the approximated local dependency relation, but never all of them. Composition operators introduce additional dependencies which have to be approximated by the ANCAG algorithm in such a way that the accidental detection of cyclic dependencies becomes more likely so that an evaluator cannot be generated. Furthermore, the ISC approach developed in [Thiele 2011] requires that all potential compositions are fixed at generation time of the evaluation order because the composition operators are integrated into the AG tool. Hence, a composition environment and a composition API is not supported.

5.4. Fragment Contracts

The RAG-based FCM specification method so far discussed in this chapter is basically an implementation of the ISC$_{\text{core}}$ model given in Section 4.1.3. The resulting approach already is superior to the FCM-specification approaches of COMPOST, Reusew*air* and Reuseware because it is declarative, expressive and implementation-independent so that it can be ported to any platform with RAG support. However, RAGs can naturally provide more information about a fragment component than just the syntactic types of nonterminals and productions of the underlying CFG: using the semantic-analyses capabilities of RAG frameworks can further increase the quality of code generators, weavers and the generated code. The argumentation in favor of semantic checking is analogous to the argumentation in favor of syntactic checking over no checks: while a good compiler (or model well-formedness check) will detect all problems that may hinder an interpretation of the model or compilation of the program, the corresponding error messages are related to points where the analysis failed and provide associated information.

[3]The IS-graph on nodes $\mathrm{Att}_x(N)$ of a nonterminal N contains the transitive closure of *all* possible dependencies/data-flow edges from attributes in $\mathrm{Inh}_x(N)$ to attributes in $\mathrm{Syn}_x(N)$ induced by *any* possible subtree with an N-labeled root.

Hence, errors introduced by a composition program will not be associated to their real cause—e.g., a composer—but to the analysis problem that caused the error message. For example, if a fragment composition system was not aware of the CnL and generates code that cannot be parsed by the CnL parser, the error message will report the internal parser state and the currently processed position in the input stream. Hence, if a bracket is missing in the output, a good parser will report a missing bracket, but perhaps not in the right place and also not in the original fragment box. Context-sensitive problems detected in the semantic analysis phase can be much more subtle and confusing than parsing errors. For example, a wrongly declared type of a method in a generated class can make the compiler complain about an overriding method of a subclass, which may not even be generated itself and may be completely unrelated to the cause.

This section describes how RAG-based *fragment contracts* can make composition and code generation more reliable by reusing CnL semantics (attributes) during the execution of composers and composition programs. This contract-based extension to ISC is called *well-formed ISC*.

5.4.1. Local Characteristic Attributes

To make the CnL semantics available for guarding composition, CnL attributes need to be accessed from the FCM's RAG. Hence, each nonterminal $n \in N$, which is a slot candidate $n \in \mathcal{S}$, a hook candidate $n \in \mathcal{H}$, rudiment candidate $n \in \mathcal{R}$ or a fragment candidate $n \in \mathcal{F}$ has a set of *characteristic* attributes $\mathrm{Ass}_x(n) \subseteq \mathrm{Att}_x(n)$ in the integrated RAG. In some cases, it is sufficient to just divert original CnL attributes as characteristic attributes (e.g., a look-up attribute to find methods depending on their signature). However, usually CnL attributes contributing to larger computations are not prepared for such usages. Therefore, the equations of characteristic attributes of the integrated FCM may access the CnL's original attributes to adapt and prepare point- or fragment-type-specific information—*gluing* CnL and FCM RAGs. Hence, for each attribute $prop \in \mathrm{Ass}_x(n)$ of $n \in \mathcal{S} \cup \mathcal{H} \cup \mathcal{R} \cup \mathcal{F}$ of the composition environment, an attribute declaration and equation are included in the attribute grammar. Thereby, it is distinguished between *point* and *fragment characteristics*. Typically, point-characteristic attributes specify what is *provided* at a point—i.e., what can be expected by a fragment to be valid at a point (e.g., the set of readable fields). In contrast, fragment-characteristic attributes typically specify what is *required* by a fragment to be valid (e.g., a specific variable definition which is needed to compute an expression's value).

The declaration scheme of characteristic attributes $prop$ is shown in Declaration 5.69 below, where n is a point candidate and κ is an arbitrary type supported by SimpAG:

$$\textbf{syn}\ \kappa\ \{n \mid prop \in \mathrm{Ass}_x(n)\}\,.\,prop \tag{5.69}$$

For each of a point candidate's occurrences n_i on a CnL production's left-hand side, a characteristic attribute instance depends on the actual point-identification attributes:

$$\textbf{fun}\ n_i\,.\,prop = \begin{cases} \bot & \textbf{if}\ \texttt{isSlot} = \texttt{isHook} = \textit{false} \\ & \quad \textbf{and}\ \texttt{isRudiment} = \textit{false}, \\ \kappa\texttt{-expression} & \textbf{else}. \end{cases} \tag{5.70}$$

If `isSlot`, `isHook` or `isRudiment` evaluate *false*, *prop* does not need to provide a value since the current n-labeled node is not a point. Otherwise, an `expression` of type κ over locally available attribute occurrences and other properties provides the characteristic value.

Complementary, for each fragment candidate f and each characteristic attribute of f, a corresponding declaration must be provided as shown in Declaration 5.71:

$$\textbf{syn}\ \kappa\ \{f \mid \mathit{prop} \in \mathrm{Ass}_x(f)\}\,.\,\mathit{prop} \tag{5.71}$$

Equation 5.72 shows the specification pattern of *prop* for fragment boxes:

$$\textbf{fun}\ f\,.\,\mathit{prop} = \kappa\texttt{-expression} \tag{5.72}$$

Characteristic attributes can now be used to formulate predicate attributes as fragment assertions. This is explained in the next subsection.

5.4.2. Attributes as Assertions

Given characteristic attributes on top of an RAG-based FCM, well-formedness preconditions can be established and integrated with the invasive composition system. In connection with the *design-by-contract* principle [Meyer 1992] such conditions are called *static fragment assertions*, since they are designed to make fragment composition more reliable by reusing static semantics properties. Using RAGs, static fragment assertions can be realized as predicate attributes associated with compositional points that use characteristic attributes of points and fragments. Declarations 5.73a and 5.73b define a `check` attribute for slot, hook and rudiment candidates:

$$\textbf{syn}\ \texttt{bool}_\perp\ \{n \mid n \in \mathcal{S} \cup \mathcal{H}\}\,.\,\texttt{check(Node)} \tag{5.73a}$$

$$\textbf{syn}\ \texttt{bool}_\perp\ \{n \mid n \in \mathcal{R}\}\,.\,\texttt{check} \tag{5.73b}$$

The `check` attribute for slots and hooks is parametrized with a fragment parameter while for rudiments it is not, since in that case no fragment is passed to the composer.

Equations 5.74 and 5.75 show the specification scheme of rudiment candidates, and slot and hook candidates respectively.

$$\textbf{fun}\ n_i\,.\,\texttt{check} = \begin{cases} \perp & \textbf{if}\ \texttt{isRudiment} = \mathit{false}, \\ \mathit{true} & \textbf{if}\ \mathrm{Ass}_x(n) = \emptyset, \\ \texttt{bool-expr} & \textbf{if}\ \mathrm{Ass}_x(n) \neq \emptyset. \\ \mathit{on}\ \mathrm{Ass}_x(n) & \end{cases} \tag{5.74}$$

If n is a rudiment candidate (Equation 5.74), `check` evaluates the instance of `isRudiment` at the current n_i node. If this evaluates *false*, the assertion is undefined. Otherwise, a Boolean expression over available point characteristics provides the assertion's value. Similarly, if n is a slot or hook candidate and the current node is not a slot or hook, the assertion is undefined.

Otherwise, a Boolean expression over the available point and fragment characteristics provides the value of the assertion attribute.

$$\textbf{fun}\ n_i\texttt{.check(fr)} = \begin{cases} \perp & \textbf{if}\ \texttt{isSlot} = \texttt{isHook} = \textit{false}, \\ \textit{true} & \textbf{if}\ \mathrm{Ass}_x(n) \cup \mathrm{Ass}_x(\mathrm{Lab}(\texttt{fr})) = \emptyset, \\ \textit{\texttt{bool-expr on}} & \textbf{if}\ \mathrm{Ass}_x(n) \cup \mathrm{Ass}_x(\mathrm{Lab}(\texttt{fr})) \neq \emptyset. \\ \mathrm{Ass}_x(n) \cup \mathrm{Ass}_x(\mathrm{Lab}(\texttt{fr})) & \end{cases} \tag{5.75}$$

The following paragraph describes the usage of the contract-checking attributes during composition execution.

5.4.3. Contract Specification

To use the assertion specified by a `check` attribute, it must be integrated with the composition engine and/or the composition-evaluation algorithms described in Section 5.3. One option is to integrate contract checking with the composer implementations `doBind`, `doExtend` and `doExtract` used in Algorithms 1 and 2, so that *before* each composition the check attribute is consulted. Equation 5.76 exemplifies the reimplementation `doBind'` of `doBind` as a contract method in the Eiffel notation proposed by [Meyer 1992] (the realization of the other operators is similar and therefore omitted here):

```
doBind'($env, src, pt$) is
    require
        $pt$.isSlot $= true$ & $pt$.check($src$) $= true$
    do
        $ctx \leftarrow pt$.parent                                   (5.76)
        doBind($env, src, pt$)
    ensure
        $pt$.parent $= \perp$ & $src$.parent $= ctx$
    end
```

The contract has three sections. The part between the keywords `require` and `do` denotes the precondition of a composition asserting that *pt* is a slot and that *src* is a compatible fragment according to the local instance of the `check` attribute. In the `do` part, the actual bind implementation is provided. It requests the parent node of the current point and executes the bind. Below the `do` part, delimited by `ensure` and `end`, the postcondition of the composition is checked. In this case, it double-checks if *pt* has successfully been replaced by *src*. The modified implementations replace the original calls in the respective algorithms. By this modification, the CnL attributes are incorporated into the basic interpretation of composition programs without the risk of triggering unintended rewrites induced by attribute dependencies. For attribute-determined composition, it is straight-forward to add the contract check as an extra expression part to the rewrite application condition (cf. Equations 5.65 and 5.66). However, as discussed in Section 5.3.4, additional attribute evaluations induced by direct and indirect dependencies of the `check` attribute during rewrite execution will have an impact on the order of composer execution.

Besides pre- and postconditions, contracts regularly have *invariants*, i.e., assertions that hold before *and* after the contracted operation is executed. For fragment contracts, obvious candidates for invariants are the *semantic conditions* which define the validity of an ASTs based on specific attributes. Some authors (e.g., [Alblas 1991]) define semantic conditions as a finite set $\mathrm{Cond}(p)$ of Boolean attributes and semantic evaluation functions associated with the productions $p \in P$ of an AG. The evaluation of semantic conditions on a given AST determines its validity w.r.t. the current attribution and thus defines a subset of valid trees generated by the underlying CFG. The invariant specification of a contract can simply use semantic conditions to maintain a global consistency state of the attributed AST during composition. Let $T = (V, E, \mathrm{Lab}, \mathcal{L}_\Sigma)$ be the AST of a given composition environment and $G = (N, \Sigma, P, S)$ the corresponding CFG. Equation 5.77 extends the above contract specification with a global composition invariant:

`invariant`

$$\forall v \in V\, \forall att \in \mathrm{Att}_x(\mathrm{Lab}(v)) \cap \mathrm{Cond}(\mathrm{ProcOf}(v)) \; : \; v.att = true \tag{5.77}$$

`end`

$\mathrm{ProcOf}(v)$ provides the production associated with $\mathrm{Lab}(v)$ if $\mathrm{Lab}(v) \in N$ and otherwise the empty set. Hence, the invariant's condition is checked for each attribute instance at each node v in T which is labeled with a nonterminal and has a corresponding attribute occurrence in semantic conditions $\mathrm{Cond}(\mathrm{ProcOf}(v))$. By default, the invariant is checked before the composition starts and then regularly after any composition step is executed. If $true$ is obtained at all nodes, the invariant holds and the composition is valid. Otherwise, the composition is invalid w.r.t. the previous composition step.

Using pre- and postconditions of fragment contracts as well as invariants, the composition system can rely on its internal state and values of attribute instances to report cause-related error messages to the user, via log files or direct feedback in an IDE. In comparison to classic ISC, this is a clear advantage, since error detection is not delayed to a compiler which is not aware of any preceding composition, and thus cannot generate cause-related error messages.

5.4.4. LogProg Example

In this section, Examples 3.14 and 5.1 are extended with fragment contracts to define a well-formed FCM for LogProg using SimpAG. The main purpose of this example is to demonstrate the integration of contract-related attributes with the plain FCM and CnL attributes. More practical examples will be given in Chapter 6 in context of a composition system for Java fragments.

Example 5.2 (Well-formed fragment component model).
The first step in the development of a well-formed FCM for LogProg is combining the attributes and equations of AG_{LogFCM} developed in Example 3.14 and AG_{Log} developed in Example 5.1, which results in AG_{LogWCM}. Hence, the underlying CFG of AG_{LogWCM} contains the original productions of G_{Log} and FCM productions of AG_{LogFCM}. Due to the open-context problem, the `Box` declarations of the FCM introduce new contexts for inherited attributes. Consequently, AG_{LogWCM} provides two additional equations for the `env` attribute in the contexts of `StmtBox` and `ExprBox` as shown in the equations below:

$$\textbf{fun}\ \texttt{StmtBox.fragment.env} = [\,]$$
$$\textbf{fun}\ \texttt{ExprBox.fragment.env} = [\,]$$

Observe that the equations initialize `env` with an empty table, which seems the most obvious solution to this incarnation of the open-context problem.

To specify a fragment contract, characteristic attributes need to be declared. For `Expr`, which is a slot candidate, $\text{Ass}_x(\texttt{Expr})$ shall contain the `provides` attribute with the purpose of providing a list of visible declaration names at expression slots. To declare the attribute in SimpAG, Declaration 5.69 is instantiated:

$$\textbf{syn}\ \texttt{string}_{\perp}\texttt{*}\ \texttt{Expr.provides}$$

To implement `provides`, the pattern of Equation 5.70 is instantiated, reusing the inherited `env` attribute, converting it into a plain list of declaration names:

$$\textbf{fun}\ \texttt{Expr.provides} = \begin{cases} \bigsqcup_{i=1}^{|\texttt{env}|} \textit{[key}_i\ \textit{of env]} & \textbf{if}\ \texttt{isSlot} = \textit{true}, \\ \perp & \textbf{else}. \end{cases}$$

Furthermore, a complementary characteristic attribute `requires` $\in \text{Ass}_x(\texttt{Expr})$ computes a list of declaration names which are used in the expression and, thus, must be provided by the slot's context. According to Declaration 5.71, `requires` is associated also with `Expr`:

$$\textbf{syn}\ \texttt{string}_{\perp}\texttt{*}\ \texttt{Expr.requires}$$

Implementing this attribute, the pattern of Equation 5.72 is instantiated:

$$\textbf{fun}\ \texttt{Expr.requires} = \texttt{child}_1\texttt{.usages}$$

The equation simply delegates to the `usages` attribute at its owned `Or` fragment which is an extension of the original attribution and performs the actual collection of variable usages in `Expr` trees:

$$
\begin{aligned}
&\textbf{syn}\ \texttt{string}_\perp\texttt{*}\ \{\texttt{And,Or,Term}\}\texttt{.usages}\\
&\textbf{fun}\ \texttt{Or}_1\texttt{.usages} = \texttt{child}_1\texttt{.usages} \sqcup \texttt{child}_3\texttt{.usages}\\
&\textbf{fun}\ \texttt{Or}_2\texttt{.usages} = \texttt{child}_1\texttt{.usages}\\
&\textbf{fun}\ \texttt{And}_1\texttt{.usages} = \texttt{child}_1\texttt{.usages} \sqcup \texttt{child}_3\texttt{.usages}\\
&\textbf{fun}\ \texttt{And}_2\texttt{.usages} = \texttt{child}_1\texttt{.usages}\\
&\textbf{fun}\ \texttt{Term}_1\texttt{.usages} = [\,]\\
&\textbf{fun}\ \texttt{Term}_2\texttt{.usages} = [\,]\\
&\textbf{fun}\ \texttt{Term}_3\texttt{.usages} = [\texttt{child}_1\texttt{.ident}]\\
&\textbf{fun}\ \texttt{Term}_4\texttt{.usages} = [\texttt{child}_2\texttt{.ident}]
\end{aligned}
$$

Using the characteristic attributes, a fragment assertion can be specified corresponding to the patterns of Declaration 5.73a and Equation 5.75:

$$
\begin{aligned}
&\textbf{syn}\ \texttt{bool}_\perp\ \texttt{Expr.check(Node)}\\
&\textbf{fun}\ \texttt{Expr.check(f)} = \begin{cases} \texttt{provided} \subseteq \texttt{f.required} & \textbf{if}\ \texttt{isSlot} = \mathit{true},\\ \perp & \textbf{if}\ \texttt{isSlot} = \mathit{false}. \end{cases}
\end{aligned}
$$

Given the `check` attribute, the composition system can now validate a precondition before binding any `Expr` slot in a LogProg environment. Moreover, given a negative result of the contract checking, it can prompt cause-related and precise error messages about the specific problem. For instance, assume `requires` = {`"a"`, `"b"`, `"c"`} and `provides` = {`"b"`, `"c"`}. Since `a` is required, the system fails and can issue a message: “Cannot bind fragment ‘A’ to slot ‘B’ in line ‘x’, because ‘A’ uses variable ‘a’ which is not provided at ‘B’.” ◇

5.4.5. When to Check Contracts

If a well-formed invasive composition system is developed, contracts should be defined with certain care. Too complex characteristic attributes and preconditions can drastically slow down the composition process. In the worst case, the RAG evaluator would have to reevaluate the whole AST before each execution of a composer. Hence, especially invariant checking should not be conducted continuously as it probably depends on most if not all attributes of the CnL. Another issue are logical dependencies among a group of composers so that there is no total ordering of composition steps that fulfills all their contracts when they are applied. Instead, before and after the group’s composers are applied, the invariant is valid, which indicates a well-formed composition result. To support such scoped cases, the composition system must provide a concept for organizing composers and contracts in groups. The composer declarations of a

`CompositionEnvironment` developed in Section 5.2.1 can be extended easily to support an arbitrary nesting of `Composers` by replacing `ComposerList` by a `CompositeComposer` list nonterminal which allows an *arbitrary* nesting of primitive and composite `Composers`.

Regarding composites, the interpretation algorithms can be modified to support them. Given operator-determined composition, a composite-supporting variant of Algorithm 1 maintains a stack of `CompositeComposers` where the stack's top element denotes the currently active composer group. The currently active group is interpreted as long as its composers have associated points they can transform or until another `CompositeComposer` is selected by `nextComposer`, pushed onto the stack and becomes the active group. After a group is finished, i.e., no more compositional points can be transformed, it is popped from the stack and the new top level group becomes active until the stack is empty. In presence of `CompositeComposers`, contracts are specified w.r.t. groups and contracts are checked at the first activation of a group (preconditions and invariants) and when it is popped from the stack (postconditions and invariants).

A composite-supporting variant of point-wise composition (Algorithm 2) works slightly different. A new outer loop wraps the outer box loop, maintaining the currently active group using a variable. As before, the currently active group determines the set of primitive `Composers` available in the inner loops for composition. After all available points and composers of the group have been processed in compliance with the current strategy, the next group is selected by the outer loop. The selection can be determined by a user-defined strategy or may simply respect the order of occurrence in the `CompositionEnvironment`. As before, the checking of contracts occurs at activation time of a group and after a group is finished. Regarding termination criteria, each group can be interpreted once, leaving some compositions unsolved, or until a fixpoint is reached, i.e., none of the compositional points has an applicable composer in any group.

The next section summarizes this chapter's contributions and draws a conclusion.

5.5. Summary and Conclusions

In this chapter, the RAG-based specification method for FCM has been presented. It has been shown how composition environments are defined by integrating the CnL grammar and the FCM grammar. It turned out that attributes are a convenient way of specifying and finding compositional points in fragments. Extending the plain FCM RAG, support for composer declarations was added to the grammar, using reference attributes to perform the look-up of fragment components and compositional points. At evaluation time, the instances of these reference attributes form an overlay graph that describes the relations between components.

Based on the composer extension, two algorithms for evaluating composition programs were introduced. The operator-determined algorithm interprets the list of composers as a set of rewrite rules whose actual application mode is highly configurable depending on a collection of variable functions. Complementary, the point-determined algorithm implements a weaving metaphor: traversing the fragment components of a composition environment, it finds compositional points, uses attributes to find associated composers and applies the corresponding composition operators.

Additionally, two attribute-dependent approaches to composition-program interpretation were discussed. It was observed that ReRAGs in general are not suitable to implement composition operators because compositional rewrites easily interfere with CnL rewrites and attributes, and because the application order is not transparent to the user. Second, the ANCAG approach developed in [Thiele 2011] was reviewed observing that ANCAG only support a small class of AGs without reference attributes. Additionally, the idea of a statically precomputed evaluation and composition order is inconsistent with composition environments employed by users in arbitrary, unanticipated scenarios.

Finally, the novel notion of well-formed ISC was introduced by extending the classic model of ISC with characteristic attributes and fragment assertions. These can be used in pre- and postconditions of fragment contracts to detect context-sensitive composition problems and issue cause-related error messages which can be logged or displayed by interactive editors.

While this chapter introduced RAG-based ISC on a conceptual level, Chapter 6 discusses its implementation in the JastAdd RAG system and presents a fragment composition system for Java 1.5.

6 The SkAT Framework and a RAG-Based Composition System for Java

To verify the concepts developed in this thesis, a RAG-based framework for ISC has been implemented. The *Skeletons and Application Templates (SkAT)* framework uses JastAdd RAGs to provide a reusable and modularized implementation of the basic attribution patterns developed in Chapter 5. The original purpose of SkAT is to provide an expressive and reliable technology to model and implement composition systems supporting patterns of parallel programming, which are frequently called *skeletons* in literature (cf. [Cole 2004; Goswami et al. 2002]), by abstractions for template metaprogramming (TMP) and the separation of cross-cutting concerns (SoCC) (cf. Chapter 1) in arbitrary programming languages. Since skeletons are only a specific and complex use case of fragment composition, SkAT naturally supports FCMs for application-specific code generation ("application templates"). Figure 6.1 gives an overview of the main compartments of SkAT. As the JastAdd metacompiler is used as an implementation framework, it is represented in the basement layer. Second is the core-specification and component language (CnL) layer. Each box on that level represents a set of associated JastAdd RAG specifications. `JastAddJ` and `Language X` denote CnL specifications that are bundled with SkAT and can be reused, where `Language X` symbolizes an extension point for adding further built-in language support. JastAddJ is the *JastAdd extensible Java compiler* originally developed by [Ekman 2006; Ekman and Hedin 2007a]. It implements an extensible, full-fledged RAG-based compilation frontend of Java, including name and type analysis, and forms the heart of *SkAT for Java (SkAT4J)*—the composition system discussed in this chapter. The remaining items on the same layer are SkAT's implementation modules. *SkAT core implementation (SkAT/Core)* contains the basic composition API and reusable FCM attributes such as the point-identification attributes. Extending SkAT/Core,

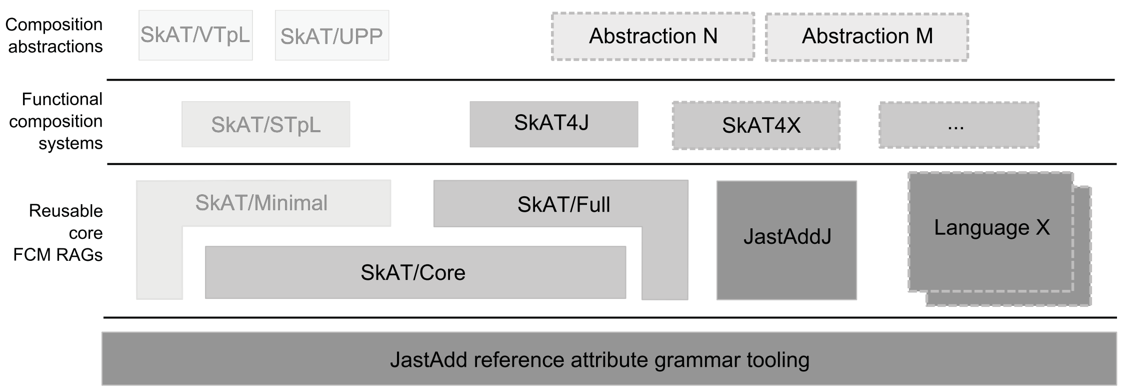


Figure 6.1.: An overview of SkAT for full ISC.

SkAT for full-fledged ISC (SkAT/Full) adds composition algorithms and composer declarations, while *SkAT for minimal ISC (SkAT/Minimal)* (grayed out) contains a minimal component model which is the basis of the other grayed out parts in the upper layer. These parts of SkAT constitute the foundations of *minimal* and *scalable* ISC, and are described in Chapter 7.

The third layer accommodates functional ISC systems with a typical COMPOST-like API, without complex dedicated CsL constructs such as in-place composers. Hence, SkAT4J is a classical ISC system. Reusing the basic APIs, systems with complex dedicated CsL constructs reside on the fourth layer. Since this chapter is focused on classical systems, it is not further investigated here. However, Chapter 7 discusses some examples of the fourth layer.

The remainder of this chapter is structured as follows. Section 6.1 shortly introduces JastAdd and JastAddJ. Section 6.2 discusses the SkAT/Core and SkAT/Full modules of SkAT which are relevant for this chapter. Afterwards, Section 6.3 presents SkAT4J as an application of SkAT/Full and JastAddJ. In Section 6.4, the resulting system is used to implement the BAF code generator of the use-case scenario given in Chapter 2. Moreover, the section introduces a library of reusable parallel skeletons for Java as a complex use case of ISC. Finally, Section 6.5 summarizes and concludes this chapter.

6.1. JastAdd and the JastAddJ Java Compiler

This section first introduces the main specification and implementation features of JastAdd. Afterwards, the components and structure of JastAddJ are described.

6.1.1. A Short Introduction to JastAdd

JastAdd is an object-oriented compiler-generator framework based on circular RAGs [Hedin 2000]. It supports rewrites that are integrated with attribute grammars [Ekman and Hedin 2004].

Due to its RAG features and the practical availability of extensible compiler frontends of practical languages such as Java, JastAdd is a suitable implementation framework for well-formed ISC. As JastAdd's specification language is very similar to that of SimpAG, only its relevant constituents are briefly described below.

Abstract syntax. As a language for AST grammars, JastAdd uses a form of flat EBNF with a maximum of one production per nonterminal and without choices. Similar to SimpAG, the language supports abstract nonterminals and nonterminal inheritance. In detail, a production has the following syntax:

```
[abstract] name[:super] ::= regexp ;
```

The optional keyword **abstract** denotes an abstract nonterminal. `name` is an identifier string and `super` is the optional name of a super nonterminal. On the right-hand side, `regexp` is a sequence of labeled terminals and nonterminals (normal, list or optional), where the types of terminals can be arbitrary Java types.

Attribute declarations. Inherited and synthesized attributes are declared by the keywords **syn** and **inh** using the following syntax:

```
[syn|inh] ret_type nt_type.name([params]) ;
```

Here, `nt_type` is a declared nonterminal of the AST grammar, `Opt`, if it is any optional nonterminal, `List`, if it is any list nonterminal or `ASTNode`, if the attribute shall be declared for all nonterminals in the AST grammar. `name` is a standard Java identifier string that denotes the attribute's name, which may have zero or more parameters (`params`) in Java syntax. The attribute's co-domain is determined by the return type—`ret_type`. It is not explicitly distinguished between reference attributes and normal attributes. However, if `ret_type` is an AST nonterminal, it is assumed that the attribute will compute an AST node which may be a fresh or referenced node. Otherwise, arbitrary types of Java can be used.

Semantic functions. Attribute implementations are specified using the keyword **eq** and the following syntax:

```
eq nt_type[.acc].name([params]) [= java_expr|java_block] ;
```

Here, `nt_type`, `name` and `params` are defined equivalently to JastAdd's attribute declarations. If `name` is an inherited attribute, a child access `acc` has to be specified in the context of `nt_-type`, evaluating one or more nodes associated with an instance of the `name` attribute. There are two ways for defining a nonterminal child's access. First, a child can be accessed directly via `get[label]`, where `label` denotes its label given by the AST grammar and `get` is a prefix. Alternatively, child nodes are generically retrievable via `getChild` by passing the corresponding list position. If `name` is declared as synthesized, `acc` is simply left out from the corresponding attribute equations. Since JastAdd's target language is Java, it is also used as

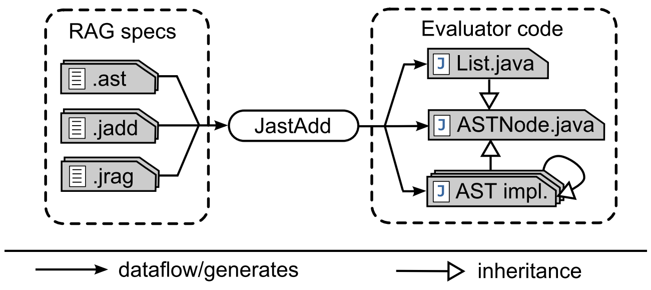


Figure 6.2.: JastAdd's evaluator-generation process.

equation language. Basically, an equation can be implemented by a Java expression or a Java block with according return statements. In both variants, an adequately typed value must be computed and returned.

Advanced features. Besides the regular RAG syntax, JastAdd supports inter-type declarations, i.e., it allows users to declare additional Java methods and fields which can be used in attribute equations or provide convenience API support. Moreover, attributes can be cached (*lazy* attributes) and be declared as circular, enabling a fixpoint-based computation of recursive attributes. The supported syntax of rewrite specifications was already anticipated in Section 5.3.4 (Equation 5.65), except that `condition` and `expression` are Java code in JastAdd's case.

Evaluator generation. Given a set of input files containing the above-described constructs, JastAdd checks the complete specification and generates an attribute evaluator. Figure 6.2 shows the code-generation process as a data-flow diagram. By convention, JastAdd uses the file endings `.ast` to denote the parts of the AST grammar, `.jrag` files contain semantic aspects, where each aspect contains a collection of attribute declarations, equations and rewrite specifications. While technically there are no differences to `.jrag`s, `.jadd` files typically contain aspects with inter-type declarations. After parsing and validating the specifications, JastAdd generates the evaluator as a bunch of Java classes: one per nonterminal of the integrated AST grammar with woven Java methods and fields that implement attributes, rewrites and inter-type declarations. All generated nonterminal classes inherit standard functionality and shared attributes from a common super class—`ASTNode`. List nonterminals and optional nonterminals are represented jointly by a `List` class and an `Opt` class. As the generated evaluator is demand-driven, an evaluation is started by invoking attribute methods on any object in the AST. Since nonterminals and attributes are mapped to Java classes, fields and methods, the resulting implementation makes heavy use of polymorphic dispatch to find and evaluate attributes associated with a specific node. For clients of JastAdd-based attribute evaluators starting the evaluation of attribute instance of an AST node is invoking a method of the respective Java object. Thus, evaluators can be transparently integrated into arbitrary Java applications, without any difference to other parts of an application.

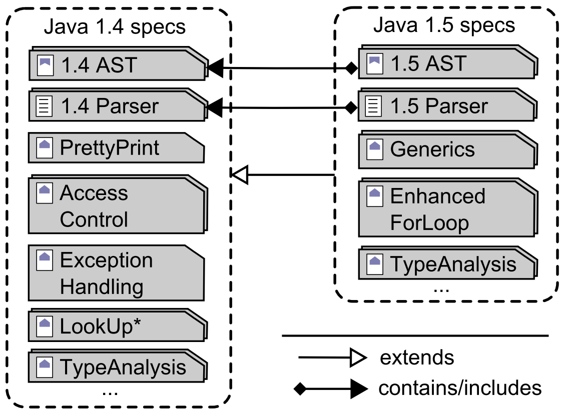


Figure 6.3.: An overview of JastAddJ's specifications (cf. [Ekman and Hedin 2007b]).

6.1.2. Ingredients of JastAddJ

JastAddJ is a Java 1.4 and 1.5 compiler with an extensible RAG-based frontend [Ekman and Hedin 2007b].[1] An overview of its specifications is given in Figure 6.3. Most specifications belong to the 1.4 version, while the extensions that make up the novel features introduces with Java 1.5 are symmetrically modelled as extending JastAdd RAG aspects.

In more detail, the Java 1.4 specifications include AST grammar modules (`.ast`), a parser grammar and the actual `.jrag` and `.jadd` files. The semantics specifications comprise a bunch of modules, whereas only a selection is shown in the figure. For pretty-printing, `PrettyPrint` defines an aspect which basically manages the unparsing of nodes to text. Thus, if a fragment composition system is developed as an extension of JastAddJ, the original pretty printer can simply be reused and extended if required. `AccessControl` encodes the visibility rules of types, methods and fields as given by the Java language specification, for example, it checks if a method may be called in a subclass depending on its visibility modifier. The `ExceptionHandling` aspect checks and validates the types of exceptions that may occur and, for example, computes which catch block applies for a given exception object. Modules prefixed with `LookUp` contain attributes contributing to the name analysis of JastAddJ, while `TypeAnalysis` attributes compute types of Java expressions and referenced variables.

The specification modules concerned with the semantics of Java 1.5 build on Java 1.4 by including, referencing and extending the corresponding grammars and attributions. Hence, the AST and parser grammars of Java 1.5 reuse most productions and nonterminals of Java 1.4, performing some redefinitions where necessary. Syntactic extensions include generic types, methods and the foreach loop. The respective `.jrag` modules provide extensions of the original analysis algorithms and add new aspects to name and type analysis. As a parser generator,

[1]Recently, JastAddJ was extended to support Java 7 and Java 8 (1.7 and 1.8 in the old nomenclature) [Hogeman 2014; Öqvist and Hedin 2013]. Unfortunately, this was too late to be considered by this work.

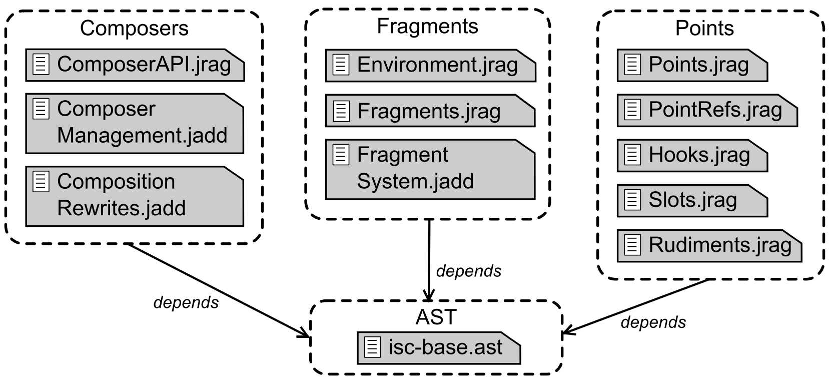


Figure 6.4.: Grouped specification modules of SkAT/Core.

JastAddJ uses Beaver [Demenchuk 2012], which implements the LALR bottom-up approach to parsing. The parser grammar is split into `.parser` modules which are composed to a Beaver specification during JastAddJ's code generation process. Thus, syntactic extensions to JastAddJ can easily add their own parser modules or replace existing ones.

6.2. Components of SkAT

This section discusses the basic components and specifications of SkAT/Core and SkAT/Full as well as their interplay. Because SkAT has been growing into a large system over time, only small excerpts from the code will be shown and discussed in detail. Other less important parts will be discussed briefly, some of them with more code excerpts in Appendix A.4.

6.2.1. Specifications of SkAT/Core

As sketched in Figure 6.1, SkAT has a set of generic reusable core specifications which define the basic ISC functionality and can be added as an add-on to any language implemented using JastAdd RAGs. Figure 6.4 gives an overview of the RAG modules provided by SkAT/Core. Organized along concerns, they can be distinguished into four groups of modules.

AST module group. The AST grammar defines the basic notion of a fragment `Box` and the `CompositionEnvironment` (cf. Listing 6.1). `Box`es have a name to represent them in the composition environment, and an output name which becomes their file name after composition. `TerminalSlot`s represent any points at the terminal level and thus cannot be modeled directly by the classic node-level composition operations. If the FCM shall support a parametrization of string values or other types of terminal values, `TerminalSlots` can be associated with these and a type-specific transformation can be provided. A corresponding example will be discussed in Section 6.3 in context of the Java FCM. Finally, the `Pattern` nonterminal introduces an abstract

```
CompositionEnvironment ::= Fragment:Box*;
abstract Box ::= <Name:String> <OutName:String> ;

abstract TerminalSlot ::=;

abstract Pattern ::=;
QRef:Pattern ::= <Name:String>;
```

Listing 6.1: JastAdd AST grammar of SkAT/Core.

```
1 aspect Environment{
2    // Passing down a reference to the environment root.
3    inh lazy CompositionEnvironment ASTNode.env();
4    eq CompositionEnvironment.getChild().env() = this;
5
6    // Provide the initial root context.
7    syn CompositionEnvironment CompositionEnvironment.env() = this;
8 }
```

Listing 6.2: `Environment.jrag` specification of SkAT/Core.

concept for realizing matching patterns of points and fragments. Inheriting from `Pattern`, `QRef` represents simple path-based names for DFA-based point-identification.

Fragments module group. The `fragments` group provides the basic fragment infrastructure. `FragmentSystem.jadd` contains the client API for loading and storing fragments, basically encapsulating the composition-environment tree and realizing the internal resource management. Composition systems derived from the core typically extend this API with system-specific functionality. Interested readers may look up the API in Listing A.7 of Appendix 6.1.

The actual fragment infrastructure is contained in `Environment.jrag` (Listing 6.2) and `Fragments.jrag` (Listing 6.3). The former specifies the `env` attribute passing down a reference to the environment root node. Observe that in JastAdd the initial value of inherited attributes is typically provided using a synthesized attribute at the root nonterminal (cf. Line 7). The second RAG module defines attributes concerned mainly with persistence (`shouldPersist` and `resourceName`), and look-up of fragments (`findFragment`). The look-up of fragment works by using a parametrized inherited attribute where the parameter is the name to search for. Moreover, `owningBox` passes down a reference to the top-level box to any node within a fragment box.

Points module group. The `points` group contains the point-identifying attributes and the point-collection infrastructure. The `Points.jrag` module provides convenience attributes for naming and point identification in the composition environment as shown in Listing 6.4. Lines 3–8 contain the most basic attributes. `isPoint` and `pointName` derive their values from the concrete identification attributes defined in the point-specific modules in the module group.

```
aspect Fragments{
   // Retrieve owned fragment of a Box (glue).
   syn ASTNode Box.fragment() = null;

   // By default, a Box is not persisted.
   syn boolean Box.shouldPersist() = false;
   syn String Box.resourceName() = getOutName() != null?getOutName():getName();

   // Fragment look-up from anywhere in the environment.
   inh Box ASTNode.findFragment(String fragmentName);
   syn Box CompositionEnvironment.findFragment(String fragmentName){
      ... // Iterate through box list and return box with fragmentName.
   }
   eq CompositionEnvironment.getChild(int index).findFragment(String fragmentName) =
      findFragment(fragmentName);
   eq Box.getChild(int index).findFragment(String fragmentName) =
      getName().equals(fragmentName)?this:findFragment(fragmentName);

   // From somwhere in a fragment, return the owning Box node/object.
   inh Box ASTNode.owningBox();
   eq CompositionEnvironment.getChild(int index).owningBox() = null;
   eq Box.getChild(int index).owningBox() = this;
}
```

Listing 6.3: `Fragments.jrag` specification of SkAT/Core.

```
aspect Points{
   // Identification and naming attributes.
   syn boolean ASTNode.isPoint()= isSlot() || isHook() || isRudiment();
   syn String ASTNode.pointName()= isSlot()?slotName():(
                          isHook()?((List)this).defaultHookName():(
                          isRudiment()?rudimentName():""));
   inh String ASTNode.pointPrefix();
   eq CompositionEnvironment.getChild().pointPrefix() = "";

   // Checks if a given pattern matches at this node.
   syn boolean ASTNode.hasQName(Pattern qName) = qName.isMatching(this);

   // AST type-checks using Java's type system.
   syn Class[] ASTNode.compatibleFragmentTypes();
   syn boolean ASTNode.isCompatibleInstance(Object fragment){
      ... // Check if 'fragment' is a compatible AST type.
   }

   // Check point-specific preconditions before executing a composition step.
   syn Object ASTNode.checkContractPre(Object fragment) = true;

   // Check point-specific postconditions after executing a composition step.
   syn Object ASTNode.checkContractPost(Object fragment) = true;
}...
```

Listing 6.4: Excerpt from the `Points.jrag` specification of SkAT/Core.

Moreover, the pointPrefix attribute provides context-dependent prefixes for point names. At the CompositionEnvironment root the prefix is empty (cf. Line 8). The matching of points is sketched in Line 11. Currently, the QRef pattern is the only point-matching implementation included in SkAT/Core. It is specified in the PointRefs.jrag module, implementing a DFA-based matcher to match point names with paths constructed from pointPrefix values. The matcher is invoked via Pattern.isMatching(ASTNode). Moreover, the module contains attributes concerned with the checking of node types during composition (cf. Lines 14–17). The attribute compatibleFragmentTypes computes a list of Java Class objects that represent the allowed syntactic types at a compositional point. Observe that reusing Java's runtime type system is possible because JastAdd generates a representative Class for each nonterminal and uses class-based inheritance as an implementation of nonterminal inheritance. Hence, isCompatibleInstance relies on Java's *instance-of* check.

As the last part of Points.jrag, Lines 20 and 23 declare the basic attributes for point-specific fragment contracts. checkContractPre is evaluated directly before a composition operation is executed. At that stage, the point is still a guaranteed member of the environment AST, while the fragment is not. Hence, context-dependent attribute instances at a slot or rudiment node can be evaluated while the fragment's attributes must not depend on the parent or any other predecessor in the AST. Complementary, checkContractPost is evaluated immediately after a composition step has been applied. At that stage, the point is no longer a member of the AST so that its attributes should not be evaluated, whereas the fragment is a member of the AST so that its attributes can safely be used because its context is available. Pre- and postconditions evaluate true if all assertions are fulfilled. Otherwise, any kind of object may be returned which can be used by SkAT to produce an error message. In case of a negative result of checkContractPost, the system roles back the composition and restores its original state. The semantic-error handling of SkAT supports two modes. In *normal mode*, the composition system stops immediately after an assertion's failure and reports the according message. In *recover mode*, the system skips the current composition step and continues processing.

The remaining specifications of the points group provide the specific point-identification attributes of the FCM. Listing 6.5 shows the slot-related attributes of Slot.jrag. Lines 3–5 comprise the basic FCM declarations of isSlot and slotName (cf. Section 5.1.1). Moreover, an extra isTerminalSlot declaration is given to mark up slots in terminal symbols (e.g., in string values). Complementary, Lines 8–10 contain default equations which define that List and Opt nodes are never recognized as slots, and that TerminalSlot nodes are always recognized as such. Moreover, Lines 13–15 contain the definition of isInSlot, which is a helper attribute to distinguish nodes within a slot's subtree from the others in the environment. By default, SkAT just ignores such nodes as a slot is basically an atomic replaceable unit. Lines 18 and 19 define a default type for slot bindings which is the type of the slot itself. If different fragment types shall be considered, this has to be defined as an extension. Finally, Lines 22–28 contain a small excerpt from the SlotCollection aspect. The implementations use a SkAT-specific collection API that performs a depth-first traversal of the environment tree, and collects all slots (collSlots), the first slot found according to a given Pattern or all slots matching the given pattern.

```
aspect Slots{
   // Determining slots in the RAG-based FCM.
   syn boolean ASTNode.isSlot() = false;
   syn boolean ASTNode.isTerminalSlot() = false;
   syn String ASTNode.slotName() = "";

   // Lists and Opts are never slots, a TerminalSlot always is a terminal slot.
   eq List.isSlot() = false;
   eq Opt.isSlot() = false;
   eq TerminalSlot.isTerminalSlot() = true;

   // Distributes, if a node is located within a slot.
   inh boolean ASTNode.isInSlot();
   eq CompositionEnvironment.getChild(int i).isInSlot() = false;
   eq ASTNode.getChild(int i).isInSlot() = isSlot();

   // Default replaceable AST type of a slot.
   eq ASTNode.compatibleFragmentTypes() =
         new Class[]{isSlot()?this.getClass():BottomFragmentType.class};
}

aspect SlotCollection{
   // Slot collecting attributes.
   syn java.util.List<ASTNode> ASTNode.collSlots();
   syn ASTNode ASTNode.findSlot(Pattern qName);
   syn java.util.List<ASTNode> ASTNode.findSlots(Pattern qName);
   ...
}
```

Listing 6.5: Excerpt from the `Slots.jrag` specification of SkAT/Core.

Since the contents of `Rudiments.jrag` and `Hooks.jrag` are very similar to what has been discussed above, they are not discussed in detail here. However, the interested reader may inspect them in Listing A.8 and Listing A.9 in Appendix A.4.

Composers module group. In the `composers` group has three different specifications. The contents of `ComposerAPI.jrag` are shown in Listing 6.6. It realizes SkAT's implementation of the generalized composer approach discussed in Section 5.2.2. Lines 3–10 comprise composer identification and related attributes. Additionally to `isBind`, `isBindRetain` describes a Bind composer declaration which *retains* the slot in the AST after binding it, which is only possible if the slot is a list entry. Moreover, `isComposerInplace` helps the system recognizing embedded in-place composer signatures. In Lines 13–16, attributes relating composers, fragments and compositional points are declared. Line 19 derives the actual syntactic type from source fragment, which is used to enforce syntactic correctness of compositions. Finally, `hasComposerFurtherSideEffects` helps the composition system to recognize composers which cause further side effects such as printing messages or requiring extra composition steps not covered by the composition program or the composition logic. Composers that produce such side effects are expected to provide a `composeSideEffects()` operation.

```
aspect ComposerAPI {
   // Generalized composer-identification attributes.
   syn boolean ASTNode.isBind() = false;
   syn boolean ASTNode.isBindRetain() = false;
   syn boolean ASTNode.isExtract() = false;
   syn boolean ASTNode.isExtend() = false;
   syn int ASTNode.extendTgtPosition() = -1;
   syn boolean ASTNode.isComposer() = isBind() || isExtend() || isExtract();
   syn boolean ASTNode.isComposerInplace() =
               isPoint() && isComposer() && composer() == this;

   // Reference edges between composers, fragments and points.
   syn ASTNode ASTNode.srcFragment(); // Composer --> Fragment
   syn ASTNode ASTNode.targetPoint(); // Composer --> Point
   syn ASTNode ASTNode.composer(); // Point --> Composer
   syn boolean ASTNode.hasAssociatedComposer() = isPoint()?composer() != null:false;

   // Actual type of a source fragment.
   syn Class ASTNode.fragmentType() = srcFragment().getClass();

   // Check if a composer has further side effects (none by default).
   syn boolean ASTNode.hasComposerFurtherSideEffects() = false;
}
```

Listing 6.6: Excerpt from the `ComposerAPI.jrag` specification of SkAT/Core.

The remaining specifications contain the operational implementations of the composition operators in Java as well as some additional convenience methods, e.g., a stack to report composition problems. Listing 6.7 shows an excerpt from `CompositionRewrites.jadd` including the Java signatures of the operator implementations. As the operator semantics has already been discussed in the previous chapters, their JastAdd-dependent implementation details are excluded from the listing. The implementation of `doBind` is sketched between Lines 3–6, `doExtend` can be found between Lines 12 and 15 while `doExtract` is located in Lines 18–21. The rewrite implementations uses JastAdd's inter-type declarations in such a way that the operations are woven into the emitted `ASTNode` class and can be directly invoked by clients or the built-in composition algorithms. The operations are always executed with respect to the owning object. Hence, after extract or bind execution in non-retaining mode, *this* node is no longer a valid part of the AST and the composition environment. After execution, the operations return a status object representing more detailed information on the composition result, which is used by the composition engine to provide precise log messages. While the standard AST-level composition operations are predefined, terminal slots can be transformed arbitrarily. Clients that define their own terminal slots need to provide specific strategies on transforming the corresponding values. Consequently, Line 9 provides an abstract signature of `doBind` which is expected to be overwritten in subtypes of `TerminalSlot`.

Since composition algorithms and composer interpretation have been discussed exhaustively in the previous chapters, the `ComposerManagement.jadd` module is only described briefly. `ASTNode.doCompose(model)` uses the SkAT/Core attributes to compute the composer

```
aspect CompositionRewrites {
   // Implementation of bind.
   public StepResult ASTNode.doBind(ASTNode fragment, boolean retain){
      if(isSlot()){
         // Check syntactic types, contracts and execute bind operation.
      } else return StepResult.POINT_CHECK_FAILED; }

   // Variation point for specific bind operations for terminal slots.
   public abstract StepResult TerminalSlot.doBind(ASTNode value, boolean retain);

   // Implementation of extend.
   public StepResult List.doExtend(ASTNode fragment, int index){
      if(isHook()){
         // Check syntactic types, contracts and execute extend operation.
      } else return StepResult.POINT_CHECK_FAILED; }

   // Implementation of extract
   public StepResult ASTNode.doExtract(){
      if(canExtract()){
         // Execute extract by removing self from parental context.
      } return StepResult.COMBINED_FAILURE; }

   ... // Further convenience and helper operations.
}
```

Listing 6.7: Excerpt from the `CompositionRewrites.jadd` of SkAT/Core.

arguments and checks if the arguments have been resolved accordingly. If the checks were successful, it dispatches to the composition rewrites, otherwise it provides according messages. In case several different composers are applicable at the current point, it prefers bind over extract and extract over extend. The `model` parameter is an optional argument which allows to parametrize the composition with a model object. It can be used to provide external system-dependent arguments to embedded composition operators and languages. For example, the *Universal Extensible Preprocessor (UPP)*, which will be discussed later in Section 7.4, uses this feature. Finally, `ComposerManagement.jadd` implements a composition-problem stack to maintain errors that occur during composition. The objects on this stack can be used to log error messages or to report them via error markers in interactive editors.

6.2.2. Specifications of SkAT/Full

SkAT/Full extends the core with modules for composer declaration, composition strategies and a full composition API. Figure 6.5 sketches the JastAdd specifications of SkAT/Full and their relations with SkAT/Core.

Basically, SkAT/Full comprises six additional JastAdd specifications. The AST module `isc-composers.ast` adds composer-declaration nonterminals and is shown in Listing 6.8. Corresponding to the SimpAG grammar of Section 5.2.1, it defines `Composer` as an abstract nonterminal which the concrete composer declarations inherit from. The composer declarations are integrated with SkAT/Core using the attributes in `ComposerIntegration.jrag` as

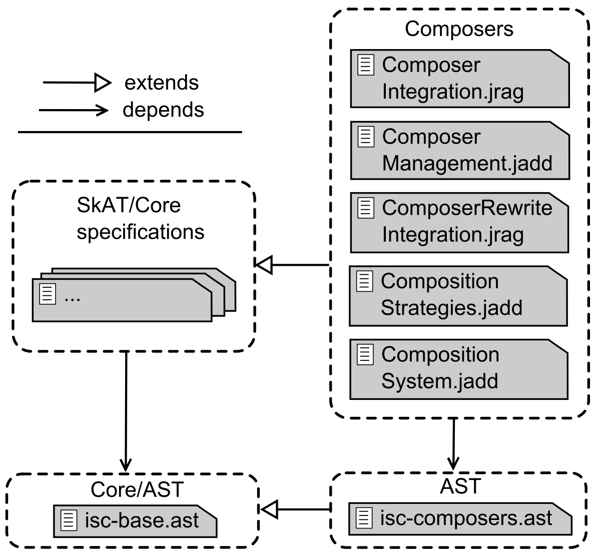


Figure 6.5.: Specifications of SkAT/Full.

```
CompositionProgram ::= Composer:Composer*;

abstract Composer ::= PointName:QRef <FragmentName:String>;
Bind:Composer ::=;
Extend:Composer ::= <Position:int>;
Extract:Composer ::=;
```

Listing 6.8: JastAdd AST grammar of SkAT/Full.

shown in Listing 6.9. Lines 3–5 announce the distinct composer nodes to the composition system. Lines 8–12 implement the `targetPoint`, the inverse `composer` and the `srcFragment` attributes by delegation to the SkAT/Full-specific look-up attributes. The respective declarations of the look-up attributes can be found between Lines 19 and 23. Some of the corresponding equations can be inspected in Lines 26–35.

The remaining specifications are only discussed briefly. `ComposerManagement.jadd` extends the core's composer management with an imperative API for marking composer–point pairs as exhausted (cf. Section 5.3). This *exhausted relation* is then used to prefilter points during the hook look-up (cf. `lookUpH` in Listing 6.9) to avoid redundant hook extensions.

In correspondence to the attribute-dependent composition rewrites discussed in Section 5.3.4, `ComposerRewriteIntegration.jrag` integrates the basic composition operations with JastAdd's rewrite API and its attribute evaluation algorithm. Hence, for each kind of a composer declaration, the module contains a rewrite specification which checks the syntactic integrity of compositional point and associated composer, and calls the respective operator implementation in SkAT/Core. If a problem occurs during composition, the environment's status is also updated.

```
aspect ComposerIntegration {
   // Announce composer declarations.
   eq Bind.isBind() = true;
   eq Extend.isExtend() = true;
   eq Extract.isExtract() = true;

   // Map to the specific look-up attributes of SkAT/Full.
   eq Bind.targetPoint() = lookUpS(getPointName());
   eq Extend.targetPoint() = lookUpH(getPointName());
   eq Extract.targetPoint() = lookUpR(getPointName());
   eq Composer.srcFragment() = lookUpF(getFragmentName()).fragment();
   eq ASTNode.composer() = LookUpComposer(this);

   // Compute a position from the declaration and the FCM's pre-defined index.
   eq Extend.extendTgtPosition() = getPosition()<0?((List)targetPoint())
                  .hookIndex(simpleName()):super.extendTgtPosition();

   // Declarations of look-up attributes.
   inh Box Composer.lookUpF(String fragmentName);
   inh ASTNode Composer.lookUpS(QRef pointName);
   inh List Composer.lookUpH(QRef pointName);
   inh ASTNode Composer.lookUpR(QRef pointName);
   inh Composer ASTNode.LookUpComposer(ASTNode point);

   // Implementations of look-up attributes.
   eq CompositionProgram.getComposer(int index).lookUpF(String fragmentName)
                                        = findFragment(fragmentName);
   eq CompositionProgram.getComposer(int index).lookUpH(QRef pointName){
      for(ASTNode hook:env().findHooks(pointName)){
         if(!getComposer(index).isExhausted(hook))
            return hook;
      }
      return null;
   }
   ... // The remaining look-up equations are implemented likewise.
}
```

Listing 6.9: Excerpt from the `ComposerIntegration.jrag` specification as a part of SkAT/Full.

SkAT/Full systems support several global composition strategies based on the composition algorithms discussed in Chapter 5. Thus, `CompositionStrategies.jadd` provides a Java interface which has to be implemented by any concrete strategy included in SkAT/Full and by any custom strategy. The included strategies are described below.

AGDepthFirstStrategy. This strategy combines the integrated demand-driven composition rewrites (cf. Section 5.3.4) with a recursive depth-first traversal of the composition environment. The traversal starts at the root of the leftmost fragment box in the environment. If a compositional point is visited, JastAdd's demand-driven evaluator invokes automatically and triggers composition rewrites defined by the associated composers.

AGBreadthFirstStrategy. This strategy combines the demand-driven composition rewrites with an iterative breadth-first traversal of the composition environment. If a compositional point is visited during the traversal, JastAdd's demand-driven evaluator invokes composition rewrites automatically.

OpOrderedStrategy (single pass). This strategy is an implementation of Algorithm 1 discussed in Chapter 5. It traverses the list of composer declarations *once* in order of appearance and applies the corresponding composition operators to each point matched by the qualified reference of a composer declaration. Assuming that none of the composers directly triggers itself, this strategy always terminates.

OpOrderedStrategy (multi pass). The multi-pass mode of OpOrderedStrategy tries to re-evaluate composer declarations in n passes. During pass 1 to $n-1$ at least one composition is executed while in the last pass no composition occurs. The number of passes n is typically unknown in advance.

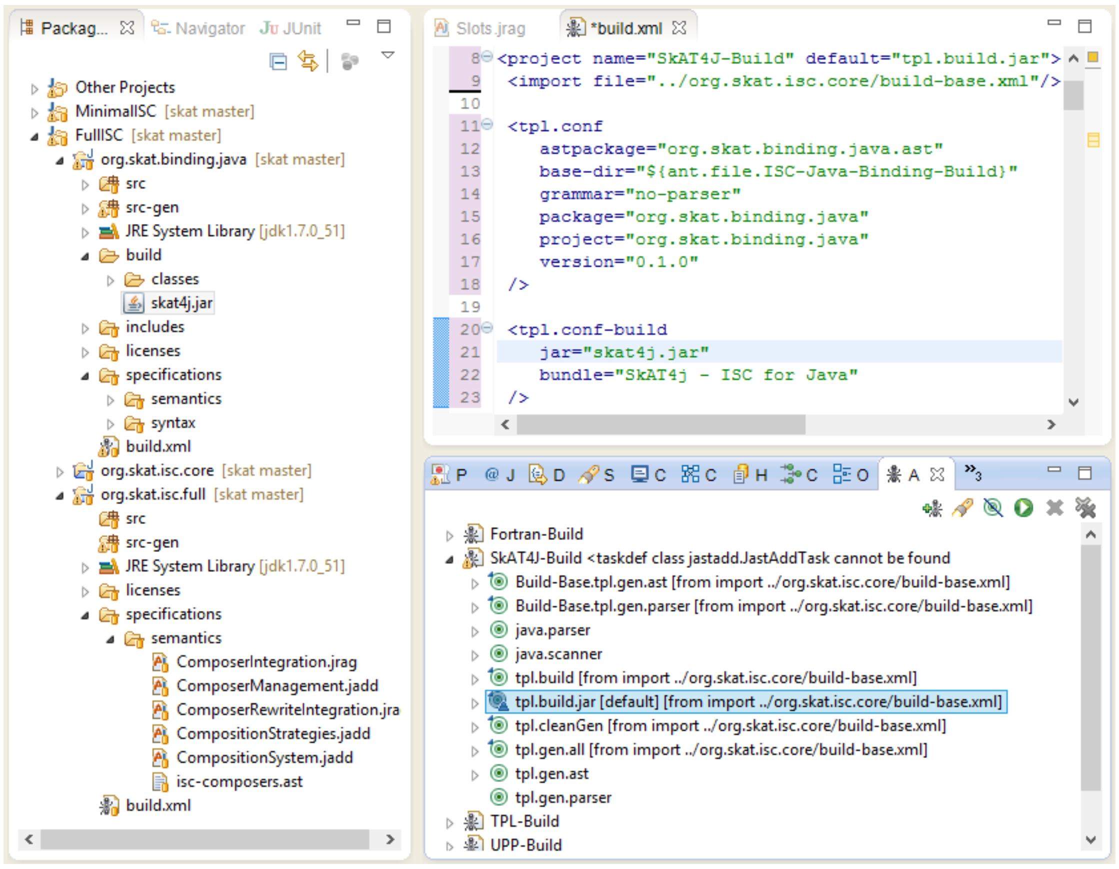
OpOrderedStrategy (strict). The strict mode of OpOrderedStrategy enforces a decreasing number of matched compositional points after each pass. Hence, if there are n points in the composition environment at the start of the evaluation algorithm, there are $n+1$ passes at maximum. In other words, if there is a strict solution to the composition problem, the system finds it, otherwise it will terminate and report that the composition is not strict.

PointOrderedStrategy. This strategy is a Java implementation of Algorithm 2 discussed in Chapter 5. It performs a depth-first traversal of the composition environment, applying composition operations *before* descending to its children. Moreover, it gives precedence to points of source fragments so that before a fragment component is copied its points are visited and evaluated first.

Extending the minimalistic fragment system of SkAT/Core, `CompositionSystem.jadd` provides a Java composition facade [Gamma et al. 1995] which eases composer creation for clients by providing factory methods for composers, the above-mentioned default strategies, and the basic API for loading and persisting fragment components. Moreover, SkAT/Full systems support a strategy-independent imperative composition mode. Hence, if a composition scenario cannot be solved reasonably using the strategy interface, Java can be used as an imperative composition language by enforcing an immediate composer execution.

6.2.3. Generating Composition Systems

To finally generate composition systems, SkAT/Core and SkAT/Full provide a compact build infrastructure based on the *Ant* build tool [The Apache Software Foundation 2014] for Java, reusing the JastAdd code generator. The infrastructure comprises the following tasks: it collects core, full and client-specific FCM specifications from the respective folders, it generates Java code from the set of collected RAG specifications and, optionally, it generates code for parsers and printers. After everything has been generated, code and manually implemented Java artifacts



```
Other Projects
MinimalISC [skat master]
FullISC [skat master]
  org.skat.binding.java [skat master]
    src
    src-gen
    JRE System Library [jdk1.7.0_51]
    build
      classes
      skat4j.jar
    includes
    licenses
    specifications
      semantics
      syntax
    build.xml
  org.skat.isc.core [skat master]
  org.skat.isc.full [skat master]
    src
    src-gen
    JRE System Library [jdk1.7.0_51]
    licenses
    specifications
      semantics
        ComposerIntegration.jrag
        ComposerManagement.jadd
        ComposerRewriteIntegration.jra
        CompositionStrategies.jadd
        CompositionSystem.jadd
        isc-composers.ast
    build.xml
```

```xml
<project name="SkAT4J-Build" default="tpl.build.jar">
 <import file="../org.skat.isc.core/build-base.xml"/>

 <tpl.conf
    astpackage="org.skat.binding.java.ast"
    base-dir="${ant.file.ISC-Java-Binding-Build}"
    grammar="no-parser"
    package="org.skat.binding.java"
    project="org.skat.binding.java"
    version="0.1.0"
 />

 <tpl.conf-build
    jar="skat4j.jar"
    bundle="SkAT4j - ISC for Java"
 />
```

```
Fortran-Build
SkAT4J-Build <taskdef class jastadd.JastAddTask cannot be found
  Build-Base.tpl.gen.ast [from import ../org.skat.isc.core/build-base.xml]
  Build-Base.tpl.gen.parser [from import ../org.skat.isc.core/build-base.xml]
  java.parser
  java.scanner
  tpl.build [from import ../org.skat.isc.core/build-base.xml]
  tpl.build.jar [default] [from import ../org.skat.isc.core/build-base.xml]
  tpl.cleanGen [from import ../org.skat.isc.core/build-base.xml]
  tpl.gen.all [from import ../org.skat.isc.core/build-base.xml]
  tpl.gen.ast
  tpl.gen.parser
TPL-Build
UPP-Build
```

Figure 6.6.: A typical SkAT setup in Eclipse.

are compiled and packaged into a self-contained Java archive. The archive file can be distributed as a normal Java library usable in arbitrary Java projects if added to the build path. Moreover, the build infrastructure can be easily configured to generate an executable application—assuming that an executable composition language is also available.

Finally, Figure 6.6 shows a typical setup of SkAT in Eclipse using the example of SkAT4J described in the next section. The workspace has three visible projects, two of them correspond to SkAT/Core and SkAT/Full. The third project contains the specifications of the Java composition system. It is configured in the displayed Ant build file that imports and reuses the core build infrastructure. The binary jar file (i.e., `skat4j.jar` in this example) is generated by invoking the `tpl.build.jar` target from command line or via the user interface of Eclipse.

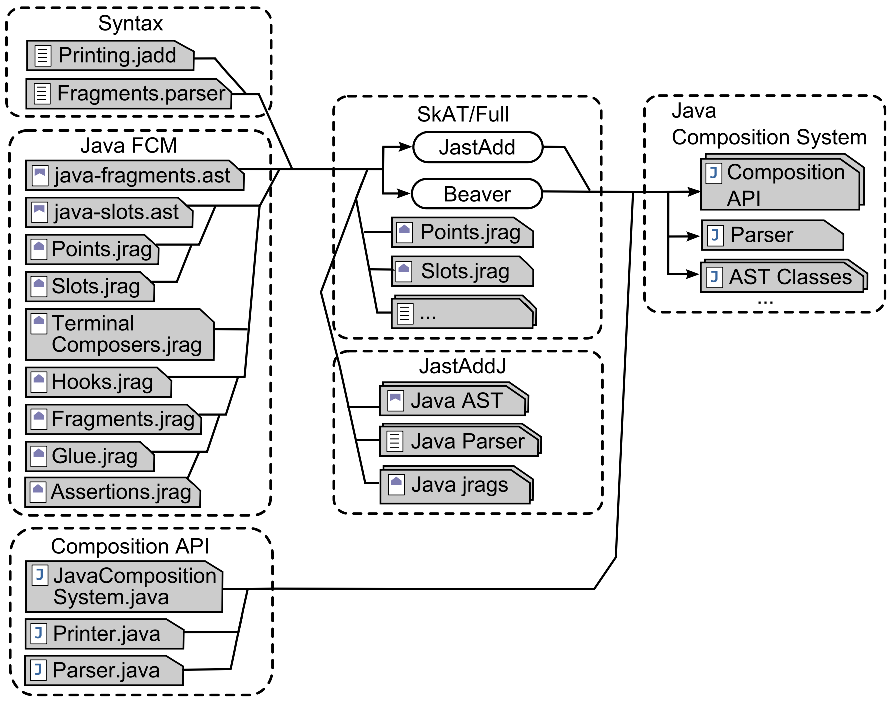


Figure 6.7.: Generating SkAT4J.

6.3. SkAT4J: A Composition System for Full Java

SkAT4J is a full-fledged invasive composition system for Java, based on SkAT/Full and JastAddJ. Its core consists of a set of JastAdd RAG modules which instantiate and extend the attributes and equations of SkAT/Full providing concrete point and fragment-identification attributes. Thereby, the RAG modules of JastAddJ are reused as CnL specifications to which the FCM attributes are related. Furthermore, JastAddJ's parser grammar based on the Beaver parser specification language [Demenchuk 2012] is reused and extended. Using the JastAdd evaluator generator and the Beaver parser generator, the SkAT/Full generator collects all specifications of the FCM and JastAddJ and generates the Java composition system. Figure 6.7 gives a more detailed view on this process and the participating specifications. On the left-hand side, the compartments of the Java FCM are shown. The `Syntax` module group extends JastAddJ's parsing and printing facilities with custom slot notations and new start symbols that provide an interface for parsing fragment boxes directly. While the module's contents are not discussed here in detail, interested readers may find the parser grammar in Appendix A.4.2, Listing A.13. Boxes, slots and the other parts of the FCM are contained in the `Java FCM` group. The API and composition facade to be used by clients is provided as plain Java classes in the `Composition API` module group. SkAT/Full, and the reused specifications of JastAddJ are sketched in the figure's center. Together with the

```
 1 abstract JavaFragmentBox:Box ::= ;
 2
 3 // Top-level boxes.
 4 CompilationUnitBox:JavaFragmentBox ::= Fragment:CompilationUnit;
 5 ClassBox:JavaFragmentBox ::= Fragment:ClassDecl;
 6 InterfaceBox:JavaFragmentBox ::= Fragment:InterfaceDecl;
 7
 8 ImportBox:JavaFragmentBox ::= Fragment:ImportDecl;
 9
10 // Member-level boxes.
11 MethodBox:JavaFragmentBox ::= Fragment:MethodDecl;
12 ConstructorBox:JavaFragmentBox ::= Fragment:ConstructorDecl;
13 FieldBox:JavaFragmentBox ::= Fragment:FieldDeclaration;
14 MemberBox:JavaFragmentBox ::= Fragment:MemberDecl;
15
16 // Block-level boxes.
17 StatementBox:JavaFragmentBox ::= Fragment:Stmt;
18 ExpressionBox:JavaFragmentBox ::= Fragment:Expr;
19 BlockBox:JavaFragmentBox ::= Fragment:Block;
20
21 abstract JavaNameBox:JavaFragmentBox ::= ;
22 DotBox:JavaNameBox ::= Fragment:Dot;
23 TypeAccessBox:JavaNameBox ::= Fragment:TypeAccess;
24 ParTypeAccessBox:JavaNameBox ::= Fragment:ParTypeAccess;
25 AccessBox:JavaNameBox ::= Fragment:Access;
26
27 abstract JavaTerminalBox:Box ::= ;
28 StringBox:JavaTerminalBox ::= Fragment:StringValue;
29 StringValue ::= <Value:String>;
```

Listing 6.10: Fragment component candidates of SkAT4J.

Java FCM specifications, these specifications become the input to the SkAT/Full generator, which generates the actual Java composition system. The final ingredients of the invasive composition system are sketched on the figure's right side. As to be expected, it comprises the generated AST classes (approx. 350 Java classes), including the attribute evaluator of JastAddJ, the additional attribute implementations of SkAT4J, SkAT/Core and SkAT/Full, and the facade classes provided by the composition API. After compilation and packaging, the composition system can be delivered to clients as a Java archive file (cf. Figure 6.6).

6.3.1. A Java "Boxology"

SkAT4J supports a subset of Java nonterminals as fragment candidates. The set of fragment candidates $\mathcal{F}$ is determined by `java-fragments.ast`—a JastAdd AST grammar adopting CnL nonterminals by using fragment-box nonterminals. The contents of that grammar are shown in Listing 6.10 while a corresponding graphical representation can be found in Figure A.2 of Appendix A.4. The fragment candidates are organized in layers. `JavaFragmentBox` acts as an abstract common ancestor of all nonterminal fragments in the FCM. Top-level boxes (cf. Lines 4–6) are typically conceived as root elements of a program by programmers. As such, `TopLevelBox`es are often the main “sinks” of Java composition programs because their ASTs will be printed in the end to obtain an executable program. `TopLevelBox`es of SkAT4J are `CompilationUnitBox`es representing a potentially compilable Java source file, `Class-Box`es containing the AST of a Java class declaration (`ClassDecl`) and `InterfaceBox`es containing the AST of a Java interface declaration (`InterfaceDecl`). To be able to compose import statements into `CompilationUnits` for importing new types and packages, SkAT4J also supports `ImportDecls` as fragment components, as shown in Line 8. Member-level boxes (cf. Lines 11–14) denote fragment components which are first-class members of class or interface declarations. More specifically, a `MemberBox` contains an arbitrary Java `MemberDecl`, `FieldBox` provides a Java `FieldDeclaration` as a fragment component, `MethodBox`es provide `MethodDecls` as fragment components and `ConstructorBox`es provide constructor declarations (`ConstructorDecls`). Block-level boxes (cf. Lines 17–19) are components that are typically used at the level of blocks, e.g., in method bodies, or deeper in the AST. SkAT4J supports `StatementBox`es encapsulating `Stmt` trees as fragment components, `BlockBox`es may contain entire `Blocks` (basically lists of statements in Java) and `ExpressionBox`es may contain arbitrary `Expr` trees, e.g., arithmetic expressions, logical expressions or simple values such as Integer numbers or strings. To also support names and identifiers in the different contexts they can appear, SkAT4J provides `JavaNameBox`es (cf. Lines 21–25). In JastAddJ, names of types, variables, references to variables, type variables and others are represented as `Access` nonterminals. `DotBox`es may contain `Dot` ASTs, which is basically a qualified name with one or more segments. A `TypeAccessBox` contains a simple name that represents an access of a type declaration. Similarly, `ParTypeAccessBox` encapsulates an access object to a generic type. If the name’s final context is not known beforehand or if it is used in several contexts, a generic `AccessBox` can contain arbitrary not finally classified names. Since JastAddJ’s automatic context-dependent classification of parsed identifiers is an issue for FCM specifications, it is discussed in more detail in a later section. As an addition to nonterminal-based fragment components, SkAT4J also supports `JavaTerminalBox`es (cf. Lines 27–29). Fragments of terminal boxes can be composed with terminal slots by a custom terminal-specific bind operation. Currently, SkAT4J only supports `StringBox`es encapsulating plain Java strings to bind slots in terminal strings by substring replacement.

To make SkAT’s fragment system aware of the newly introduced fragment candidates, they must be glued using the respective attributes `fragment` and `shouldPersist`, which were declared for all `Box`-inheriting nonterminals (cf. Listing 6.3). The corresponding RAG module `fragments.jrag` of SkAT4J is shown in Listing 6.11. For `JavaFragmentBox`es, `StringBox`es and all inheriting nonterminals, the `fragment` attribute simply maps to the `getFragment()` child accessor. The default equations of the core suggest that none of the fragments should be persisted. This behavior is kept, except that `CompilationUnitBox`es as main compilable units of Java are marked do be persisted automatically (cf. Line 8). The resource names for persisting to the file system are computed in Lines 11–14. If the composition system user provided an output name explicitly, this one is used, otherwise it is derived from the fragment’s name in the composition environment.

```
aspect Fragments{
   // Mapping to the fragment system.
   eq JavaFragmentBox.fragment() = getFragment();
   eq StringBox.fragment() = getFragment();
   eq AccessBox.fragment() = getFragmentNoTransform();

   // By default, only compilation units should persist after composition.
   eq CompilationUnitBox.shouldPersist() = true;

   // Determining the box'es final resource names for persistence.
   eq JavaFragmentBox.resourceName()
         = hasOutName()?getOutName():getName() + ".jbx";
   eq CompilationUnitBox.resourceName()
         = hasOutName()?getOutName(): packageName() + getName() + ".java";
...}
```

Listing 6.11: `Fragments.jrag` specification of SkAT4J.

```
eq MethodDecl.isSlot() = name().endsWith("Slot");
eq MethodDecl.slotName() = isSlot()?name().substring(0,name().length()-4):"";
eq MethodDecl.getChild(int i).isInSlot() = isSlot();
```

Listing 6.12: Equations from the `Slots.jrag` specification of SkAT4J.

6.3.2. Slot Identification

SkAT4J uses three different patterns to define its slot-candidate set $\mathcal{S}$ and the slot-identification function $\int_{\mathcal{S}}$ (cf. Section 4.1.3). First, a COMPOST-like naming approach is used, which recognizes slot naming conventions in the signature of a node's name without modifying or extending the CnL syntax. Second, explicit slot nonterminals are introduced as a monotonic extension to the CnL's AST grammar providing their own specific syntax for slot signatures in the CnL. Third, explicit terminal slots are introduced, also with their own signature syntax, extending the original AST grammar. The three groups will be discussed subsequently.

Pattern-Based Slots

SkAT4J's FCM supports one slot candidate, whose identification is based on the recognition of a specific naming pattern. The specification of `MethodDecl` slots is shown in Listing 6.12. Given these equations, the generated composition system recognizes method declarations of a class whose names have the suffix `Slot` as slot nodes:

```
void methodSlot(){};
```

The name of such slots is given by the substring preceding the `Slot` suffix. Hence, the above slot's name is `method`.

```
eq StmtSlot.isSlot() = true;
eq StmtSlot.slotName() = isSlot()?extract(getSlotName(),"[[","]]"):"";
eq StmtSlot.compatibleFragmentTypes() = new Class[]{Stmt.class};
```

Listing 6.13: Equations from `Slots.jrag` related to `StmtSlot`.

Dedicated Slot Nonterminals

Dedicated slot nonterminals are introduced as extensions to the nonterminals of the AST grammar of JastAddJ. These extensions should not modify the original language specifications of JastAddJ and, in the best case, should not cause interferences with its static semantics. To achieve this, the following basic AST extension scheme is employed:

```
XSlot:X-Or-Super ::= <SlotName:String> ;
```

`X` is the actual nonterminal of JastAddJ a slot candidate shall be provided for. `XSlot` inherits from `X-Or-Super` which can be `X` itself or a "more adequate" ancestor. An adequate ancestor is a nonterminal which can safely occur in the places of `X`, but has less or no problems with the consistent embedding into the CnL RAG (cf. the *incomplete embedding problem*, Section 5.1.5). As an example, consider the slot nonterminal `StmtSlot`. It is defined as follows in `java-slots.ast`:

```
StmtSlot:Stmt ::= <SlotName:String> ;
```

Still, the composition system has to be notified about `StmtSlot` as a slot candidate and needs to be supplied with a slot-identification function based on the corresponding FCM attributes. Listing 6.13 shows the according equations. Verbally, a `StmtSlot` node always is a slot and can be bound to any nonterminal inheriting from `Stmt`, e.g, assignments, variable declarations, blocks, expression statements or loops. As a conventional naming scheme, SkAT4J uses double square brackets as slot markup for dedicated terminal and nonterminal slots. This ensures that the JastAddJ parser grammar can be extended safely without causing ambiguities.

Table 6.1 summarizes the slot candidates which have been introduced in SkAT4J by the two approaches discussed above. Their full specification can be found in Listing A.10 of Appendix A.4.2.

Dedicated Terminal Slots

The core model of ISC is based on nonterminals as slot candidates and subtree replacement as bind composition operation. However, in some practically relevant situations, this model is not immediately applicable. As an example, consider the identifier of a class declaration. In an ideal case, it would be modeled as a separate nonterminal with a terminal child in the AST grammar so that defining a class-name slot would be trivial. However, typically the identifier of a class declaration is simply added as a terminal to the AST grammar. In theory, this would not be a problem since terminals can be lifted as nonterminals. Unfortunately, JastAdd does not support lifting neither on the specification level nor in the generated Java evaluator, where terminals are

Slot nonterminal	Example in context	Explanation

`MethodDecl`	`public class A {`
	`void MethodSlot(){};`
	`}`
		A slot for method decls.

`MemberDeclSlot`	`public class A {`
	`[[MemberSlot]];`
	`}`
		A class-members slot, e.g., fields, methods.

`StmtSlot`	`public void m(){`
	`[[StmtSlot]];`
	`}`
		A slot for statements, e.g., declarations, blocks.

`ExprSlot`	`Object[] o =`
	`new Object[[[ExprSlot]]];`
		A slot for expressions, e.g., initializations.

`VarAccessSlot`	`int a = this.[[VarName]];`	A slot for accessing fields of an object.

`TypeAccessSlot`	`[[TypeSlot]] t =`
	`new [[TypeSlot]]();`
		A slot for “real” generic types.

`ArrayTypeAccessSlot`	`[[TypeSlot]][] t =`
	`new [[TypeSlot]][i];`
		A slot for generic arrays.

`GenericTypeAccessSlot`	`public void`
	`m(A<[[ParSlot]]> arg){}`
		Convenience slot for type parameters.

`TypeVariableSlot`	`public class`
	`A <[[ParSlot]]>{}`
		Convenience slot for type parameter decls.

Table 6.1.: Nonterminal slots of SkAT4J.

realized as normal fields. Moreover, changing JastAddJ's specifications manually is also not an option since this is a considerable effort and would have to be repeated with each follow-up version of JastAddJ. To circumvent this issue, the already mentioned concept of terminal slots was introduced in SkAT/Full and is used by SkAT4J.

Concrete terminal slots inherit from the `TerminalSlot` nonterminal and are allowed to access their parent node modifying any of its terminal values. A corresponding concrete situation is shown in Figure 6.8. Each terminal slot is provided with an inherited attribute `slotOwner` pointing to the node which shall be modified with a terminal-specific composition operation. From the composition system's perspective, terminal slots are transparently integrated into the environment tree: look-up and slot-identification attributes have the same signatures. To integrate terminal slots into SkAT4J's FCM, the following AST extension scheme is employed:

```
SlotableX:X ::= [YSlot] ...further terminal slots... ;
YSlot:TerminalSlot ::= ;
```

Here, `X` is the CnL nonterminal which should be provided with one or more terminal slots, whereas `Y` is derived from the corresponding terminal symbol's name. Consequently, for each terminal slot of `X`, a slot nonterminal `YSlot` is introduced that inherits from `TerminalSlot`.

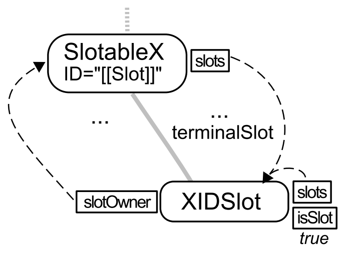


Figure 6.8.: A typical subtree with a terminal slot.

```
inh TypeDecl TypeDeclNameSlot.slotOwner();
eq SlotableClassDecl.getTypeDeclNameSlot().slotOwner() = this;
eq SlotableInterfaceDecl.TypeDeclNameSlot().slotOwner() = this;

eq TypeDeclNameSlot.isSlot() = isHedged(slotOwner().getID(),"[[","]]");
eq TypeDeclNameSlot.slotName()= isTerminalSlot()?
            extract(slotOwner().getID(),"[[","]]"):"";
eq TypeDeclNameSlot.compatibleFragmentTypes() = new Class[]{StringValue.class};
```

Listing 6.14: Equations from `Slots.jrag` related to `TypeDeclNameSlot`.

To glue them to the AST grammar and a nonterminal `SlotableX` is added which inherits from `X` and has for each terminal slot of `X` has the corresponding optional `YSlot` nonterminal on the right-hand side. As an example, consider the terminal slot `TypeDeclNameSlot` of a `ClassDecl`:

```
SlotableClassDecl:ClassDecl ::= [TypeDeclNameSlot];
TypeDeclNameSlot:TerminalSlot ::= ;
```

Listing 6.14 shows the associated slot-identification attributes. The equations are nearly equivalent to normal slots, except that `isSlot` and `slotName` by convention depend on the terminal value of the `slotOwner`. Moreover, Listing 6.15 shows the specific implementation of `doBind`. It takes a compatible `StringValue` as an argument and simply sets the `ID` value of the `slotOwner`.

Similar to dedicated slots, the slot markup syntax with double square brackets helps to distinguish between `X` nonterminals and `SlotableX` nonterminals in the parser grammar. Hence, the parser simply constructs a `SlotableX` node with `TerminalSlot` children instead of an

```
public StepResult TypeDeclNameSlot.doBind(ASTNode value, boolean retain){
   if(!isCompatibleInstance(value))
      return StepResult.TYPE_CHECK_FAILED;
   slotOwner().setID(((StringValue)value).getValue());
   return StepResult.OK;
}
```

Listing 6.15: Basic implementation of `TypeDeclNameSlot.doBind()`.

Slot nonterminal	Example in context	Explanation
`TypeDeclNameSlot`	`public interface [[NameSlot]] {}`	A slot for interface and class-declaration names.
`MethodDeclNameSlot`	`public void [[NameSlot]](){}`	A slot supporting generic method names.
`VarDeclNameSlot`, `VariableDeclaration-NameSlot`, `FieldDeclNameSlot`	`private String [[Name]];`	Slots supporting generic field and variable names.
`MethodAccessNameSlot`	`String a = [[NameSlot]]();`	A slot for generic method calls.
`ConstructorDeclNameSlot`	`public [[NameSlot]](){}`	A slot supporting generic constructor names.
`StringValueSlot`	`String s = "SkAT: [[Msg]] [[Msg]]."`	A slot in string literals.
`ParameterDeclNameSlot`	`public void m(String [[Name]]){}`	A slot for method-parameter names.

Table 6.2.: Terminal slots of SkAT4J.

`X` node in the AST if it recognizes one or more slot signatures. Another option to introduce `SlotableX` nodes into the AST is using JastAdd's rewrites. A rewrite condition of `X` nodes can check the presence or absence of slot signatures and create a fresh `SlotableX` node with the original `X` child nodes additionally adding fresh `TerminalSlot` children. Depending on what is more adequate in the specific case, SkAT4J employs both options.

Table 6.2 summarizes the supported terminal slots in SkAT4J which have been defined using the above discussed approach.

6.3.3. Hook Identification

Besides the slots presented in the previous subsection, the FCM of SkAT4J also supports extensibility via some implicit hooks. The attribute equations specifying $\mathcal{H}$ and $\int_{\mathcal{H}}$ (cf. Section 4.1.3) are provided by a corresponding `Hook.jrag` module. As a representative from that specification, Listing 6.16 shows the five hook-identification attributes of the `statements` hook, which exists in context of any block, except of those which are also in context of a slot (cf. Line 2). Observe that instead of inherited attributes as proposed in the theoretical RAG approach, SkAT apparently uses synthesized attributes for hook-identification instead. This results from the fact that JastAdd does not support list nonterminals in ideal ways. Essentially, inherited attributes for lists are not supported. Therefore, SkAT "simulates" inherited attributes in these cases by using synthesized attributes at the list nonterminal which delegate to a corresponding parametrized attribute instance at the parent node (`Block` nodes in case of `statement` hooks). Hence, above

```
 1  // Equivalent to inh 'eq Block.getStmtList().isHook() = ...;'.
 2  eq Block.isHook(List hook) = hook == getStmtList() && !isInSlot();
 3
 4  // Equivalent to inh 'eq Block.getStatementList().hookName() = ...;'.
 5  eq Block.hookName(List hook) = "statements";
 6
 7  // Equivalent to inh 'eq Block.getStmtList().hookAliases() = ...;'.
 8  eq Block.hookAliases(List hook) {
 9     if(isMethodRootBlock()){
10        return new String[] {"methodEntry","methodExit","statementsEnd"};
11     }
12     else if(!isMethodRootBlock() && endsWithReturn()){
13        return new String[] {"methodExit" + numReturns(),"statementsEnd"};
14     }
15     else /* All other situations, e.g., constructors, static blocks. */
16        return new String[] {"statementsEnd"};
17  }
18
19  // Equivalent to 'eq Block.getStmtList().hookIndex(String hookName) = ...;'.
20  eq Block.hookIndex(List hook, String hookName) {
21     if("methodEntry".equals(hookName) || "statements".equals(hookName)){
22        return 0;
23     }
24     else if (hookName.startsWith("methodExit") && endsWithReturn()){
25        return hook.numChildren()-1;
26     }
27     else /* All other situations: length of the list. */
28        return hook.numChildren();
29  }
30
31  // Equivalent to 'eq Block.getStmtList().compatibleFragmentTypes() = ...;'.
32  eq Block.compatibleFragmentTypes(List hook) = new Class[]{Stmt.class};
```

Listing 6.16: Equations from `Hooks.jrag`, determining statement list hooks.

of each “simulating” equation in Listing 6.16 an equivalent inherited signature is annotated as a comment. The hook’s name is defined in Line 5, i.e., `"statements"`. Its aliases are specified in Lines 8–17. Aliases have two different purposes. If a hook appears in multiple contexts, its aliases may assign a more adequate name for a specific context. For example, the `statements` hook alternatively has the name `methodEntry` if it is the root block of a method declaration. This helps clients of the composition system to address the `statements` hooks of all method root blocks of a class. Hence, in terms of aspect-oriented programming (AOP), hook names and matching patterns constitute a pointcut language. Moreover, observe that each `statements` hook which also has a `return` statement at the end can be addresses via `methodExitX`, where `X` is the number of appearance of the `return` in the control flow of its enclosing method. The hook’s default index position is determined by the equation of the `hookIndex` attribute in Lines 20–29. In the case the hook is addressed as `methodEntry` or `statements`, its default is 0, if it is a `methodExit`, the insertion is before the return statement and for all other names, it is the position after the last element.

The other hooks of SkAT4J are specified in similar ways and are therefore omitted. However,

Hook list	Hook name and aliases	Explanation
`Block.stmts`	`<scope>.statements` `.statementsEnd` `.methodEntry` `.methodExit`	Hooks to extend the statement list of any block.
`CompilationUnit .importDecls`	`<scope>.imports`	A hook to extend the list of import declarations.
`TypeDecl.bodyDecls`	`<scope>.members` `.membersEntry` `.membersExit`	Hooks to extend the members list of a class or interface declaration.
`ClassDecl.implements`	`<scope>.implements`	A hook to add new interfaces to a class.
`MethodDecl.parameters`	`<scope>.parameters`	A hook to add new parameters to a method.

Table 6.3.: Hooks supported by SkAT4J.

Table 6.3 gives an overview of all hooks currently supported by the component model of SkAT4J. Moreover, interested readers may inspect the full specification in Listing A.11 of Appendix A.4.2.

6.3.4. Language Glue

As SkAT4J basically extends JastAddJ and parts of its AST definitions and attributes, the extensions need to be *glued* with the JastAddJ RAG. In Section 5.1.5 of Chapter 5, the *open context problem* and the *incomplete embedding problems* of language composition were discussed. The majority of gluing attributes and equations of SkAT4J are contained in the `Glue.jrag` specification module. In the following, parts of this specification are presented and explained.

Listing 6.17 shows an excerpt of the `TreeCompleteness` aspect of `Glue.jrag`. This aspect provides some of the context required by the inherited attributes `compilationUnit` and `hostPackage`, which are relevant for name and type analyses in JastAddJ. The `compilationUnit` attribute passes a reference to the root `CompilationUnit` through the AST which is used by dependent attributes. Since all fragment-component candidates are typically descendants of `CompilationUnit`, they must be provided with a compensatory context. The corresponding equation is shown in Lines 5–6 of Listing 6.17. It provides a mockup `CompilationUnit` as a context object. Of course, this completion is *syntactic* in nature so that semantic analyses depending on that attribute may not return a practical result. However, other parts of the RAG remain intact and can be used in contract attributes. The inherited attribute `hostPackage` provides the current Java package—the default namespace to resolve names within a `CompilationUnit`. As fragment boxes (except `CompilationUnitBox`) do not belong to any package, the default package is provided as context (cf. Line 12 for the corresponding equation for `StatementBox`). Similarly, the `hostType` attribute needs to be provided with an equation

```
 1  aspect TreeCompleteness {
 2     // Original JastAddJ attribute:
 3     //  inh CompilationUnit TypeDecl.compilationUnit();
 4     // eq CompilationUnit.getChild().compilationUnit() = this;
 5     eq StatementBox.getFragment().compilationUnit() =
 6                 env().unknownType().compilationUnit();
 7     ... // Similar equations for non compilation-unit boxes where required.
 8
 9     // Original JastAddJ attribute:
10     // inh String Expr.hostPackage();
11     // eq CompilationUnit.getChild().hostPackage() = packageName();
12     eq StatementBox.getFragment().hostPackage() = "";
13     ... // Similar equations for non compilation-unit boxes where required.
14
15     // Original JastAddJ attribute:
16     // inh TypeDecl Stmt.hostType();
17     // syn TypeDecl TypeDecl.hostType() = this;
18     eq StatementBox.getFragment().hostType() = env().unknownType();
19     ... // Similar equations for lover-level boxes where required.
20
21     // Original JastAddJ attribute:
22     // syn lazy TypeDecl Expr.type();
23     eq ExprSlot.type() = unknownType();
24     ... // Similar equations where required.
25  }
```

Listing 6.17: Slice from the `TreeCompleteness` aspect of `Glue.jrag`.

for any fragment contributing to type analysis. The reference value of `hostType` is used for looking up names and types accessed from expressions and declarations within a box. Again, a reference to a mockup `TypeDecl` provides context (cf. in Line 18, the corresponding equation of `StatementBox`). Finally, the synthesized `type` attribute is provided for slots which may occur in place of expressions or names involved in type analysis. Since a slot's semantic type is unknown, a reference to the mockup type is used again to complete the RAG (cf. Line 23, a corresponding equation of `ExprSlot`).

Another more subtle issue is the classification of parsed names in JastAddJ. As already mentioned before, reference names such as variable or type names are parsed and stored unclassified as `ParseName` nodes in the AST. The final classification is then implemented by AST rewrites using a context-dependent reclassification operation. If an unclassified name is visited due to an ongoing attribute evaluation, a composition or creating a copy, the node is classified by the reclassification object provided by the context attribute `nameType`. Since FCM, composition and contract attributes can trigger reclassification rewrites, fragment boxes must mimic the context correctly, otherwise the composed AST may contain wrongly classified names. For example, `ParseName` could be classified as `TypeAccess` by an `AccessBox` representing a reference to a type declaration. However, if used in context of a variable assignment or expression, it may correctly be a reference to a variable declaration, which is represented by the nonterminal `VariableAccess` in JastAddJ. Listing 6.18 shows the `SyntacticClassification` aspect providing default name classification for most Java fragment boxes mimicking

```
 1  aspect SyntacticClassifiation{
 2     // Original JastAddJ attributes:
 3     // inh NameType Expr.nameType();
 4     // inh NameType BodyDecl.nameType();
 5     // Mimics context of CompilationUnit.
 6     eq ImportBox.getFragment().nameType() = NameType.PACKAGE_NAME;
 7
 8     // Mimics context of TypeDecl.
 9     eq FieldBox.getFragment().nameType() = NameType.EXPRESSION_NAME;
10     eq MemberBox.getFragment().nameType() = NameType.EXPRESSION_NAME;
11     eq ConstructorBox.getFragment().nameType() = NameType.EXPRESSION_NAME;
12     eq MethodBox.getFragment().nameType() = NameType.EXPRESSION_NAME;
13
14     // Mimics context of Block.
15     eq StatementBox.getFragment().nameType() = NameType.EXPRESSION_NAME;
16     eq ExpressionBox.getFragment().nameType() = NameType.EXPRESSION_NAME;
17
18     // TypeAccess boxes contain type names.
19     eq ParTypeAccessBox.getFragment().nameType() = NameType.TYPE_NAME;
20     eq TypeAccessBox.getFragment().nameType() = NameType.TYPE_NAME;
21
22     // Special case: AccessBoxes are classified as ambiguous names.
23     eq AccessBox.getFragment().nameType() = NameType.AMBIGUOUS_NAME;
24  }
```

Listing 6.18: The `SyntacticClassification` aspect of `Glue.jrag`.

the original equations of JastAddJ. Names in `ImportBox`es are classified as package names (cf. Line 6) by default. Names within fragments which are children of `TypeDecl` nodes in a normal JastAddJ AST are classified accordingly as variable names (`EXPRESSION_NAME`, cf. Lines 9–12). `StatementBox` and `ExpressionBox` use the same schemes. Boxes definitely containing type references provide an according classification object for type names (cf. Lines 19–20), while general `AccessBox`es classify their contained names as ambiguous names (`AMBIGUOUS_NAME`) delaying the final decision since they can only be classified correctly in their composed context (cf. Line 23). Observe that the remaining fragment boxes not considered in the attributions of Listing 6.18 do not require gluing equations because they already provide an adequate context inherently.

The glue module also provides an adaptation of JastAddJ's inherited `lookupType` attribute. In the standard case, this attribute searches through all source `CompilationUnits` in the input path and already compiled classes on the classpath, and compares to a given type signature consisting of package and name of the type. The results of calls to `lookupType` are cached. To make `lookupType` available as a characteristic attribute that can be used to specify fragment assertions, the original distinct start symbol `Program` of JastAddJ and the root nonterminal `CompositionEnvironment` are glued via nonterminal inheritance where `CompositionEnvironment` extends `Program`. This way, `lookupType` can be refined transparently to other attributes and also consider the contents of fragment boxes for finding type declarations. This is important if `CompilationUnits` are composed and shall be recognized

Assertions and Characteristics	Explanation

`syn TypeDecl.assertNotDeclared(MethodDecl)`
`// Characteristic Attributes:`
`syn MethodDecl.signature()`
`syn TypeDecl.localMethodsSignatureMap()`
	Checks if the signature of a given method is already contained in a type and reports a redeclaration.

`syn TypeDecl.assertNotDeclared(FieldDeclaration)`
`// Characteristic Attributes:`
`syn TypeDecl.localFieldsMap()`
	Checks if a field with the same name is already a member of a type and reports a redeclaration.

`syn TypeDecl.assertVariablesProvided(ASTNode)`
`// Characteristic Attributes:`
`*syn ASTNode.danglingVars()`
`syn TypeDecl.memberFields(String)`
`inh TypeDecl.lookupVariable(String)`
	Checks if all variables used in the given fragment are declared and visible in scope of the current type declaration, so that no dangling references remain after composition.

`syn TypeDecl.assertMethodsProvided(ASTNode)`
`// Characteristic Attributes:`
`*syn MethodDecl.danglingCalls()`
`syn TypeDecl.memberMethods(String)`
`inh TypeDecl.lookupMethod(String)`
	Asserts that all method calls in the given fragment have an according method declaration visible in the scope of the current type declaration.

`syn ExprSlot.assertCompatibleType(Expr expr)`
`// Characteristic Attributes:`
`syn Expr.type()`
`syn TypeDecl.wideningConversionTo(TypeDecl)`
	Asserts that expressions evaluate a compatible type w.r.t. an enclosing assignment, i.e., left-hand side and right-hand sides are compared.

Table 6.4.: Fragment assertions supported by SkAT4J.

by name and type analysis during composition. Hence, the modified version first uses the original implementation to search for types in the configured paths, if the type with the requested name is not found the new implementation searches and compares all `CompilationUnitBox`ex in the fragment list of the `CompositionEnvironment`.

Thanks to the integration with the JastAddJ RAG and the language glue described in this section, SkAT4J can support fragment assertions and contracts—the concepts constituting *well-formed ISC*. The next subsection discusses the contracts currently supported by SkAT4J.

6.3.5. Fragment Assertions and Contracts

Based on the concepts developed in Section 5.4, SkAT4J has some built-in fragment assertions and contracts, which are presented in this section. Fragment assertions and the custom characteristic attributes are specified in `Assertions.jrag`. Its contents are summarized in Table 6.4 while the full specification can be inspected in Listing A.12 of Appendix A.4.2. Currently, five assertion attributes are supported by SkAT4J, which can be used in fragment contracts

```
 1  // The contract of ExprSlot has no preconditions, but a post condition.
 2  eq ExprSlot.checkContractPre(Object fragment) = true;
 3  eq ExprSlot.checkContractPost(Object fragment) {
 4     if(fragment instanceof Expr){
 5        Object result = assertCompatibleType((Expr)fragment);
 6        if(result!=Boolean.TRUE)
 7           return result;
 8     }
 9     return true;
10  }
```

Listing 6.19: `ExprSlot` contract equations from the `Slots.jrag` specification of SkAT4J.

to ensure well-formed composition results. An attribute `assertNotDeclared` is specified for `MethodDecl` and `FieldDeclaration` parameters. It checks if the given parameter is already a member of the `TypeDecl` owning the attribute instance. For methods, the attribute returns *true* if no conflict with the signature of an already contained method is detected, otherwise the assertion fails. Similarly, for `FieldDeclarations` it is checked if a field with the given field's name is already a member of the host `TypeDecl`. The assertion attributes `assertMethodsProvided` and `assertFieldsProvided` compute the "dangling" usages of methods and variables in the given fragment and compare it to what is provided by the `TypeDecl` node owning the attribute instance. Observe that while all other characteristic attributes used in this section's assertions are existing attributes of JastAddJ, `danglingVars` and `danglingCalls` were added as custom attributes to `Assertions.jrag`. Both traverse the fragment AST and collect the dangling variable and method identifiers. The names found in the fragment are then checked against the members and scope of the `TypeDecl` owning the instance of the assertion attribute. Finally, `assertCompatibleType` compares the semantic types of the left-hand and right-hand sides of a variable assignment delivered by the JastAddJ type system. Therefore, it uses the type system's attribute `type`, which is defined on any expression, and `wideningConversionTo` to check compatibility of the types on both sides of an assignment.

Based on the assertion attributes, fragment contracts can finally be specified. Listing 6.19 shows the specification of a contract associated with SkAT4J's slot candidate for Java expressions (`ExprSlot`). Always evaluating *true*, the contract has no precondition (cf. Line 2). Hence, as long as `fragment` is a compatible subtree, a bind composition operation would be executed. However, after composition, a postcondition is evaluable (cf. Lines 3–10). It uses `assertCompatibleType` to verify if the fragment bound to the `ExprSlot` node is correctly typed w.r.t. the context. In case it is, the system recognizes the contract as fulfilled, otherwise it inverts the composition operation and emits a detailed problem-cause related status message about the failed contract check. As an example, consider the following members of an arbitrary class declaration:

```
int bar = 1;
public void foo() { bar = [[FooSlot]];}
```

Now assume `FooSlot` shall be bound to an expression fragment `x*42/33`. Depending on the type of `x`, the type system derives the type of the expression according to the Java language

```
 1 // The contract the members hook has a preconditions, but no postcondition.
 2 eq TypeDecl.checkContractPre(List hook, Object fragment){
 3    Object contractValue = Boolean.TRUE;
 4    if(isMembersHook(hook)){
 5       if(fragment instanceof MethodDecl)
 6          contractValue = assertNotDeclared((MethodDecl)fragment);
 7       if(contractValue == Boolean.TRUE && fragment instanceof FieldDeclaration)
 8          contractValue = assertNotDeclared((FieldDeclaration)fragment);
 9       if(contractValue == Boolean.TRUE)
10          contractValue = assertVariablesProvided((ASTNode)fragment);
11    }
12    return contractValue;
13 }
14 eq TypeDecl.checkContractPost(List hook, Object fragment) = true;
```

Listing 6.20: `Members` hook contract equations from the `Hooks.jrag` spec. of SkAT4J.

specification. If `x` was an integer variable, the expressions type would be `int`. Hence, after composition, the assignment's left-hand side and the right-hand side both would be typed as `int` so that the contract is fulfilled. However, if `x` would be a `double`, the type system computes `double` as the expression's type. Since a `double` value is not allowed to be assigned to `int` variables, the composition system would report an error message containing information about fragment, slot and failed assertion.

Complementary, Listing 6.20 shows the fragment contract of the `members` hook. In this case, the contract's precondition relies on two assertions (cf. Lines 2–13) while its postcondition constantly evaluates *true* (cf. Line 14) so that the composition persists if the precondition is fulfilled. The precondition first evaluates `assertNotDeclared` in case the hook is extended with a `MethodDecl` or `FieldDeclaration` fragment. Hence, if a method with the same signature or a field with the same name is already declared within the `TypeDecl` owning the hook, the contract fails and a composition specific message is produced. Finally, the contract checks, if all dangling variables required by the fragment are provided in the hook's context. As an example, consider the following declaration of a Java class:

```
public class Foo { private int bar = 1; }
```

Now assume a `FieldDeclaration` fragment `private String bar;`. The `members` hook cannot be extended with the fragment because it already has a declaration with the same name. This is recognized by the contract, avoiding the composition step's execution. Furthermore, assume the following `MethodDecl` fragment:

```
public int getBar(){ return bar; }
```

In this case, the hook can be extended since the method uses a variable `bar` which is declared in the hook's context. However, if the method would use other variables, the contract check would detect this and reject execution.

In complex composition scenarios, a fragment contract might fail at an early stage of the composition, but hold in a later stage. For example, a statement may use an undeclared variable which is meant to be added by the composition system. If the statement composition is applied first, the system would recognize the missing declaration and stop by default. However, if the composition system is in *recover mode*, it can delay the statement's insertion until the required declarations are composed. In combination with the *OpOrderedStrategy* in multi-pass or strict mode, contracts and composition specification denote a "composition problem" which is "solved" when a stable state is reached.

The usage of SkAT4J and several composition case studies are presented in the next section.

6.4. Example Applications

This section presents several fragment composition case studies of using SkAT4J . The first example in Section 6.4.2 demonstrates a simple composition program which realizes mixin-based inheritance [Bracha and Cook 1990] for Java classes. Afterwards, Section 6.4.3 shows how the code generator of the business application framework (BAF) is implemented in SkAT4J. The basic ideas and requirements of the BAF code generator were already discussed in Chapter 2. Finally, the prototype of a fragment and composition library for parallel programming is presented. It comprises fragment-based realizations of the well-known parallelization patterns *map*, *reduce* and *map-reduce* [Dean and Ghemawat 2008].

The next subsection documents the basic usage and client API of SkAT4J.

6.4.1. Using SkAT4J

The SkAT4J library provides a low-level API to create and execute composition programs mimicking the actual composition environment. In the following, the composition process of SkAT4J and its phases are described, and the usage of the API to realize them is explained.

Initialization and Configuration

Before SkAT4J can be used for compositions, it has to be initialized. This is typically done by creating a fresh `JavaCompositionSystem` object with a base `uri` as well as `in`put and `out`put directories:

```
JavaCompositionSystem cSys = new JavaCompositionSystem("<uri>","<in>","<out>");
```

During the object initialization, the system scans the directory for supported fragment files with endings `.java` or `.jbx` and adds them to the environment.

Optionally, a custom composition strategy can be configured as shown below. The available strategies have been introduced in section 6.2.2.

```
cSys.setCompositionStrategy(CompositionSystem.<strategy>);
```

Accordingly, SkAT's recovery mode can be enabled or disabled. By default, it is disabled so that the system stops when a fragment type check or contract fails.

```
cSys.setRecoverMode(true|false);
```

Fragment Management

Although fragments from the file system are already loaded during the initialization of the composition system, SkAT4J still has some possibilities of adding additional components to system. The following statement parses a given fragment string and creates and adds a new fragment box to the environment:

```
Box box = cSys.addFragment("<name>","<content>");
```

Alternatively, fragment boxes can be added directly:

```
Box box = cSys.addFragment(<box>);
```

Moreover, boxes can be copied. The following statement internally copies and registers a fragment with the given name, and returns the freshly created box:

```
Box newBox = box.copyBox("<name>","<newName>");
```

Composer Management

To create a composition program, composer nodes have to be added to the composition environment. Observe that adding a composer declaration to the environment does not automatically execute the corresponding operation. The statements below create and add `Extend` declarations, where `hookName` is the qualified name of the point or a name with wildcards. `fragmentName` denotes the name of a fragment in the environment, while `content` is a fragment string which is parsed and added to the environment.

```
cSys.addExtend("<hookName>","<fragmentName>");
cSys.addExtendContent("<hookName>","<content>");
```

Similarly, the statements below add `Bind` declarations, where `slotName` is the point's name, `fragmentName` is the name of a fragment in the environment and `content` is a string that represents a fragment.

```
cSys.addBind("<slotName>","<fragmentName>");
cSys.addBindContent("<slotName>","<content>");
```

The following statement adds an `Extract` declaration to the environment, where `rudimentName` is the point's qualified name:

```
cSys.addExtract("<rudimentName>");
```

Some compositional problems require staged solutions. To support staging, SkAT4J allows clearing the complete composition program and reset the state of the composition system with the statement below. Staging allows composition-program developers to decompose larger compositions in subprograms avoiding unintended interferences among compositional concerns.

```
cSys.clearCompositionProgram();
```

Transformation

The transformation phase covers the actual application of composition operators. In SkAT/Full, the composition engine is invoked via `triggerComposition`. This activates the interpreter, which evaluates the composition program with the chosen strategy.

```
cSys.triggerComposition();
```

In cases where the provided composition strategies and a linearly ordered composition program are inconvenient to efficiently solve the problem at hand, SkAT supports an *imperative mode* of composer execution with direct invocation. The statements below show how composers can be applied immediately:

```
cSys.doExtend("<hookName>","<fragmentName>");
cSys.doBind("<slotName>","<fragmentName>");
cSys.doExtract("<rudimentName>");
```

Error Inspection

During composition, SkAT internally maintains its health status. Hence, if problems occur, the status is updated. The statement below checks the status of the environment, i.e., in the case a severe problem occurred it is marked as unhealthy, which means that the composition program interpretation was not successful because a syntactic type check or a contract failed.

```
boolean health = cSys.isEnvHealthy();
```

Detailed errors and other status messages can be reported via the following statement, which emits the composition environment status:

```
cSys.printStatus();
```

Fragment Printing

After a successful execution of the composition engine, fragments can be pretty-printed to the file system via the statement below. By default, only `CompilationUnitBox`es are emitted since these can be compiled by the Java compiler. To print other fragments, a regular pattern can be passed as a parameter, matching custom box names.

```
cSys.persistFragments(<optional pattern>);
```

Having discussed SkAT4J's API and basic usage patterns, the following subsection exemplifies the implementation of a class-mixin composer.

6.4.2. A Class-Mixin Composer

In object-oriented programming, a mixin is a class which adds additional functionality to another class by composition [Bracha and Cook 1990; Aßmann 2003]. It is typically not meant to be instantiated itself and can be incomplete, i.e., the mixin may require the target class to provide

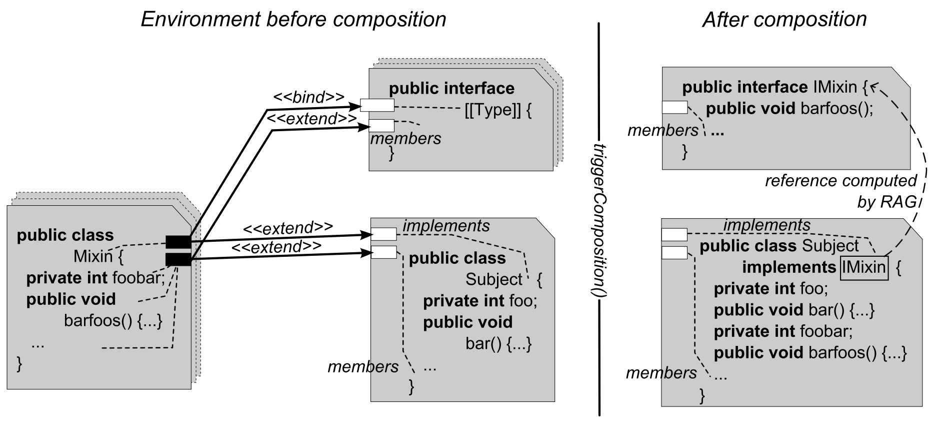


Figure 6.9.: Invasive composition of mixins.

certain methods, which it uses in its own implementation. Therefore, mixins are sometimes called *abstract subclasses* [Bracha and Cook 1990].

In ISC, mixin composition is basically “putting all members of the mixin class into corresponding boxes and writing a composition program which uses the *extend* composition operator to extend the members hook of a subject class with the contents of these boxes”. Figure 6.9 illustrates such a specific mixin composition for Java using ISC, where the figure’s left-hand side shows the environment before composition while the right-hand side shows the result of an application of mixin. On the left-hand side, `Mixin` provides a field `foobar`, a method `barfoos` and its own name as well as an arbitrary number of other fields and methods. The `Subject` class, which is the target of the composition, has at least one field `foo` and a method `bar`. Besides the `Subject`’s extension, an interface declaration with method signatures and the name of `Mixin` is derived. The interface is then referenced as an implemented interface of the `Subject` class by extending it at the `implements` hook with the corresponding name. Hence, if multiple classes are sequentially mixed into `Subject`, their derived interfaces are appended to the hook, which provides a simple form of multiple inheritance based on fragment composition. The result of a single mixin application is shown on the figure’s right-hand side. The member declarations of `Mixin` have been appended to the `members` hook of `Subject`, which now also implements the derived interface. Observe that the reference edge from the interface name in `Subject` to the `Mixin` interface declaration is computed automatically in SkAT4J. Classic ISC systems such as COMPOST and Reusew*air* this is not considered at all, in Reuseware this would need to be specified as a part of the FCM.

A SkAT4J composition program that realizes the discussed operation is shown in Listing 6.21, where it is assumed that the SkAT4J library has been added to the classpath and the composition system has been initialized in `cSys`. The composition program is defined in scope of a Java method which takes the names of the subject and mixin boxes as an input (parameters `mixin` and `subject`). In a first step, the mixin’s `CompilationUnixBox` is requested from the

```java
public void mixin(String subject, String mixin) throws IOException{

   CompilationUnitBox box = (CompilationUnitBox) cSys.findFragment(mixin);
   if(box==null)
      return;

   // First an interface is derived from the mixin class.
   ClassBox mixinBox = box.extractClassBox();
   String mixinName = mixinBox.getFragment().getID();
   String interfaceName = "I"+ mixinName;
   if(cSys.findFragment(interfaceName)==null)
      cSys.addFragment(interfaceName, "public interface " + interfaceName + " {}");

   // The 'implements' hook of the subject is extended with the new name.
   cSys.addExtendContent(subject + "#*.implements", interfaceName);

   // Generate extend declarations:
   for(BodyDecl decl : mixinBox.getFragment().getBodyDeclsNoTransform()){

      // Extend subject class and interface 'members' hook with mixin methods.
      if(decl instanceof MethodDecl){
         MethodDecl m = (MethodDecl)decl;
         MethodDecl m_interface = new MethodDecl(m,new Opt());
         String id = cSys.genID();
         String i_id = cSys.genID();
         cSys.addFragment(new MethodBox(id,id,m));
         cSys.addFragment(new MethodBox(i_id,i_id,m_interface));
         cSys.addExtend(subject+"#*.members",id);
         cSys.addExtend(interfaceName+"#*.members", i_id);
      }
      // Extend subject class with field declarations of the mixin.
      else if(decl instanceof FieldDecl || decl instanceof FieldDeclaration){
         MemberDecl f = (MemberDecl)decl;
         String id = cSys.genID();
         cSys.addFragment(new MemberBox(id,id,f));
         cSys.addExtend(subject+"#*.members",id);
      }
   }

   // Configuration & Transformation phase:
   cSys.setRecoverMode(true);
   cSys.setCompositionStrategy(CompositionSystem.OP_ORDERED_COMPOSITION_FP);
   cSys.triggerComposition();
   if(cSys.isEnvHealthy())
      cSys.clearCompositionProgram();
   else
      cSys.printStatus();
}
```

Listing 6.21: Implementation of a mixin composition operator in SkAT4J.

environment. Next, the mixinBox is requested from the compilation unit and an empty interface declaration is added to the environment of cSys (cf. Lines 8–12). Observe that, for convenience, the interface fragment is instantiated directly with the derived name instead of using an explicit interface template as suggested by Figure 6.9. Since both ways yield equivalent results, it is the programmers choice which one to use. The statement in Line 15 adds the extension of the subject's implements hook to the composition program. The loop between Lines 18–38 adds extensions of the members hook of the subject by traversing all BodyDecls in mixinBox, adding MethodDecls including their bodies and FieldDecls to the subject, and MethodDecls without implementation to the derived interface. Finally, the actual composition execution is configured and invoked in Lines 41–47. Before triggerComposition is called, recovery is enabled and the *OpOrderedStrategy* in multi-pass mode (OP_ORDERED_COMPOSITION_FP) activated. Hence, even if a contract failed, the system uses fragment contracts for validation and continues processing. The contracts and assertions currently available in SkAT4J were discussed in Section 6.3.5. Establishing a conflict-resolution strategy, contracts ensure that in case of equivalently named field or method declarations in subject and mixin, the subject's declarations are preferred while the others are discarded. After composition, the environment's status is checked, in the healthy case the composition program is cleared. Otherwise, the status is printed to console using a human-readable format.

In the next section, the mixin composer is employed in the larger context of a SkAT4J-based implementation of the BAF code generator.

6.4.3. The Business Application Generator in SkAT4J

The BAF example has been introduced in Chapter 2. Like in the COMPOST implementation developed in Section 4.2 and the Reuse*wair* implementation presented in Section 4.3, the composition program is provided as a Java method. Since SkAT4J already supports the same markup for slots as used in Chapter 2, the fragment components of Listing 2.3 and Listing 2.5 can be used.

The code generator itself is implemented in the method compositionProgram of Listing 6.22. Recall that the BAF uses a textual DSL based on EMFText as a specification language for business models. Hence, first a model instance is loaded using the provided file handle in Line 5. Next, the system is configured with a multi-pass *OpOrderedStrategy* and recovery mode enabled. Thus, contracts are enabled for intermediate validation. The generator's main loop ranges from Line 12 to Line 54 and iterates through the RoleDefinitions of the BAF model. Thereby, it is assumed that the RoleDefinitions are partially ordered according to role inheritance where definitions without a super role are the smallest elements. Within the loop, in the first block of statements (Lines 14–22), the Person.jbx box is cloned and parametrized with the basic values of the current RoleDefinition. Besides the naming of the new type, the super type is set to Person by binding the ImplicitSuperClass slot. Moreover, a default constructor is added via extending the membersEntry hook and the literal slot Pfx is bound to the newline character. Afterwards, composition is triggered to create the role-specific

```
private JavaCompositionSystem cSys = new JavaCompositionSystem("baf/","in","out");

public void compositionProgram(File modelFile) throws IOException {
  // Load the BusinessModel object with EMFText.
  BusinessModel bm = loadBusinessModel(modelFile);

  // Configure the composition system.
  cSys.setCompositionStrategy(CompositionSystem.OP_ORDERED_COMPOSITION_FP);
  cSys.setRecoverMode(true);

  // For each role definition, generate a Java class.
  for(RoleDefinition role: preOrder(bm.getRoleDefinitions())){
    // Instantiate template Person.jbx for each role.
    String cuName = role.getName() + ".java";
    cSys.copyBox("Person.jbx",cuName);
    cSys.addBindContent(cuName+"#Type",role.getName());
    cSys.addBindContent(cuName+"#TypeName","\"" + role.getName() + "\"");
    cSys.addBindContent(cuName+"#ImplicitSuperClass","Person");
    cSys.addExtendContent(cuName+"#*.membersEntry","public [[Type]](){super();}");
    cSys.addBindTerminal(cuName+"#Pfx","\n");
    cSys.triggerComposition();
    cSys.clearCompositionProgram();

    // Mix in code of super roles.
    for(RoleDefinition superRole:role.getSuperRoles()){
      mixin(role.getName() + ".java",superRole.getName() + ".java");
    }

    // Generate code for PropertyDefinitions of the current role.
    for(PropertyDefinition def:concat(role.getProperties(),getSuperProps(role))){
      // If there's not already a mixed-in implementation, add members.
      if(def.eContainer()==role && !isShadowed(def)){
        cSys.addExtend(cuName+"#*.members","Setter.jbx");
        cSys.addExtend(cuName+"#*.members","Getter.jbx");
        cSys.addExtend(cuName+"#*.members","Field.jbx");
        cSys.addBindContent(cuName+"#Type",def.getType().getTargetType());
        cSys.addBindContent(cuName+"#SetSfx","set" + toFirstUpper(def.getName()));
      }
      // Extend asString().
      if(def.eContainer()==role && !isShadowed(def) || def.eContainer()!=role){
        String stmt = "v += \"\\n [[Field]]:\" + [[GetSfx]]();";
        cSys.addExtendContent(cuName+"#*.asString.methodExit",stmt);
        cSys.addBindContent(cuName+"#GetSfx","get" + toFirstUpper(def.getName()));
        cSys.addBindContent(cuName+"#Field",def.getName());
      }
      // Extend constructor with defaults.
      if(def.eContainer()==role && def.getType().getDefault()!=null){
        String stmt = def.getName() + "=" + def.getType().getDefault() + ";";
        cSys.addExtendContent(cuName+"#*." + role.getName() + ".statements",stmt);
      }
      cSys.triggerComposition();
      cSys.clearCompositionProgram();
    }
  }
  cSys.persistFragments();}
```

Listing 6.22: SkAT4J-based implementation of the BAF code generator.

box. Next, between Lines 25 and 27, a mixin loop iterates through all super `RoleDefinitions` of the current role and calls the mixin composer which has been discussed previously. Consequently, the members of all super `RoleDefinitions` are added to the role-specific box, and corresponding Java interfaces are generated and added to the environment. Accessor methods and fields are added by the loop between Lines 30 and 53. It traverses the collection of `PropertyDefinitions` of the current role and its super `RoleDefinitions`. If the current property (`def`) is declared by the current role and has not already been declared by a super definition (and thus have been added via the mixin operation), the `members` hook is extended with the fragments `Setter.jbx`, `Getter.jbx` and `Field.jbx`. The `Type` slot in these fragments is bound to the `targetType` given by the current `PropertyDefinition` and the name of the `set` method is constructed from the declared `name`. The `set` method name and the field name are parametrized in the second block of the property loop (cf. Lines 40–45). Moreover, the `asString` method of the `Person.jbx` box is extended with a property-specific statement (cf. `stmt` in Line 41), which is constructed from a string and parametrized "on-the-fly". The extension of `asString` via its `methodExit` hook happens in case `def` is a child of the current `RoleDefinition` or if it is a child of any super `RoleDefinition`, since the original `asString` of `Person.jbx` is retained because of active fragment contracts. In the loop's last conditional block of statements in Lines 47-50, a statement to set the properties default value during object construction is added to the previously inserted default constructor via its `statements` hook if a default is declared in the currently processed `PropertyDefinition`. Finally, the specified composition is again executed via `triggerComposition`. Before exiting `compositionProgram`, the generated Java classes are printed to the output directory of the composition system.

Listing 6.23 contains an original result of the composition, generated from the BAF model in Listing 2.2 of Chapter 2. As a fully valid and usable implementation, the emitted code has all desired properties discussed in Chapter 2. Members have been added as intended, the string of `asString` considers prefixes and all members, and the constructor sets the default of `discount` automatically. Inheritance of `RoleDefinitions` is realized via mixin-based inheritance and direct inheritance from `Person` (cf. Listing 2.1). An alternative generator as also suggested in Chapter 2 could refer the `Employee` as a super type, only mixing in `Customer` or vice versa, which can be easily achieved by modifying two lines of the code generator: in Line 18 the `ImplicitSuperClass` slot has to be bound to the name of the first direct super `RoleDefinition` and the mixin loop should skip this definition in Line 25.

Similar to the mixin composer, contracts are checked during the execution of the BAF code generator. For example, before inserting parametrized copies of `setter.jbx` and `getter.jbx`, the system checks if the corresponding field has already been added to the respective class declaration. Thus, in Listing 6.23, `setDiscount` and `getDiscount` are only inserted if the `discount` field declaration has been inserted correctly before (cf. the `Members` hook contract in Listing 6.20). Further checks are possible. For example, the system could check if the expression statements in `asString` can yield string values as required by the method.

```java
public class EmployeeCustomer extends Person implements IEmployee, ICustomer {
  public EmployeeCustomer() {
    super();
    discount = 20;
  }
  public String asString() {
    String v = "EmployeeCustomer";
    v += "\n id:" + getID();
    v += "\n name:" + getName();
    v += "\n employed:" + getEmployed();
    v += "\n workload:" + getWorkload();
    v += "\n discount:" + getDiscount();
    return v;
  }
  private java.util.Date employed;
  private int workload;
  public void setEmployed(java.util.Date employed) {
    this.employed = employed;
  }
  public java.util.Date getEmployed() {
    return employed;
  }
  public void setWorkload(int workload) {
    this.workload = workload;
  }
  public int getWorkload() {
    return workload;
  }
  private int discount;
  public void setDiscount(int discount) {
    this.discount = discount;
  }
  public int getDiscount() {
    return discount;
  }
}
```

Listing 6.23: `EmployeeCustomer.java` emitted by the SkAT4J-based BAF generator.

6.4.4. Taming Parallel Algorithms with ISC-Based Skeletons

Parallel algorithmic skeletons (short: *skeletons*) are basic units of abstraction capturing typical parallel programming patterns as reusable building blocks in standard programming languages. Ideally, these building blocks completely hide the code dealing with managing parallelism and distribution. After they have been conceptualized by [Cole 1989], there have been various approaches in academia to make skeletons available for scientific programmers as well as for standard application developers. Due to the focus on single-core performance in the chip industry during the 1990s and early 2000s, most of the early skeleton approaches have been discontinued.[2] However, thanks to the end of the GHz race in the mid 2000s, the shift to multi-core processors

[2]For a comprehensive and continuously maintained overview of skeleton libraries and implementations see [Wikipedia contributors 2014a].

in desktop and mobile computing and the current trends in *heterogeneous computing* and *green computing*, skeleton research regained momentum since programming such systems efficiently is considered a big challenge [Benkner et al. 2011].

A common approach to implement libraries of reusable skeleton patterns is using TMP abstractions (e.g., *Muesli*—a skeleton library using C++ templates [Ciechanowicz et al. 2009]). However, it is still questionable if plain template metaprogramming is sufficient to express *any* parallel programming pattern or skeleton and cross-cutting dependencies, as it, for example, does not provide a Turing-complete composition language and also is not available in all important languages for parallel programming [Chalabine and Kessler 2006]. Therefore, the authors of [Chalabine and Kessler 2006] suggest ISC as a more practical implementation technique for parallel patterns and concerns. More recently, [Kessler and Löwe 2012] discuss performance-aware gray-box components. They use an ISC tool to produce and test sequential and parallel implementation variants of typical parallelizable algorithms. Thereby, ISC is employed to weave performance-aware scheduling code into parallel applications by static metaprogramming. Considering the applications proposed in [Chalabine and Kessler 2006] and [Kessler and Löwe 2012], SkAT would be a good choice of an implementation framework as it has several advantages over standard ISC approaches—namely well-formed composition, scalability w.r.t. language support (cf. Chapter 7), extensible FCMs and a solid formal basis on RAGs.

To support this argument, in the third case-study of this chapter SkAT4J is used to create an ISC-based implementation of a *MapReduce* application. MapReduce is a platform-independent programming model in distributed computing that has been applied for efficiently solving a variety of large-scale parallel programming tasks [Dean and Ghemawat 2008]. In the heart of MapReduce are *map* and *reduce* skeletons with a well-determined interface implemented for various target platforms, e.g., shared or distributed memory architectures.

A Simple MapReduce Application

In this example, the following conventions for map and reduce skeletons are used. A map skeleton expects a collection of input data chunks and applies a user-specified task to each of the input elements. The output of map is a collection of processed data which typically is *larger* than the input set, i.e., the result has the same or a larger number of elements. Like map, a reduce skeleton gets a collection of input data chunks and applies a user-specified task to it. The output of reduce is a collection of processed data, which is typically *smaller* than the input set, i.e., the result has fewer elements or at least one or zero elements. MapReduce composes map and reduce operations sequentially to realize complex distributed computations. Thereby, the skeletons' infrastructure encapsulates and hides program code which is responsible for allocating resources, distribution of data and safe parallel execution of map and reduce tasks. Map and reduce tasks are executed by *workers*. A worker is basically a lightweight process (shared address space, thread) or heavyweight process (own address space, distinct memory) which waits for computation tasks assigned by a designated *master* process. The number of worker processes is typically limited, depending on the resources available.

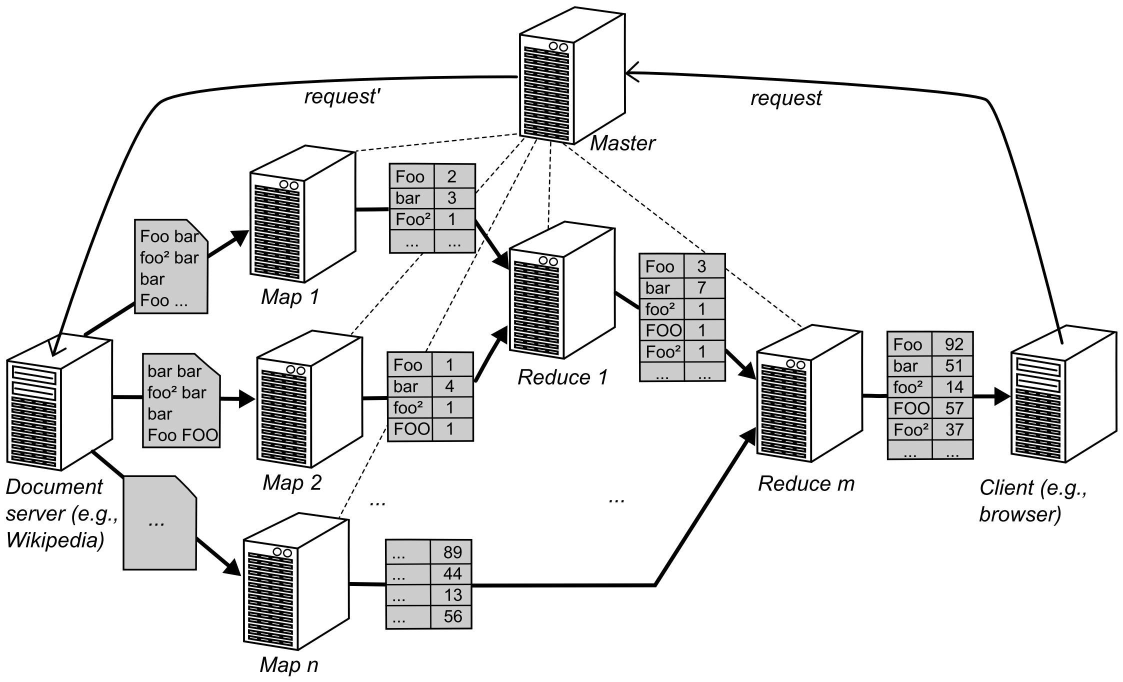


Figure 6.10.: MapReduce on a large set of documents.

An example of a MapReduce application used by [Dean and Ghemawat 2008] is counting occurrences of single words in large collections of documents in the web, which plays a role in estimating the relevance of a document w.r.t. a search query. Figure 6.10 contains the data-flow view on a narrowed variant of this kind of MapReduce application counting the occurrences of words in documents requested by a client from a document server. The document-search `request` sent by the `Client` triggers the `Master` to allocate k `Map` and `Reduce` workers which may be processes or threads running on the `Master`'s machine itself or on remote machines. Observe that $1 \leq k \leq n$, where n is the number of required map operations (document chunks). The documents requested from the `Document server` are split into n nearly equivalently sized chunks of "document data", where each chunk is delivered to a `Map` worker. Since at maximum k tasks are processed in parallel, the `Master` constantly assigns new `Map` work to the workers. In each `Map` execution, chunks are tokenized into their single words counting the occurrences of each word. Locally, each process maintains a list of hash tables with recognized words as keys and number of occurrences as values and finally provides these tables in its output port. This example's `Reduce` tasks reduces hash tables of `Map` and `Reduce` tasks into a single hash table which is also provided as an output value. `Reduce` task m computes the combined hash table of all requested documents, which is finally delivered to the `Client`.

A Fragment Library with Map and Reduce Skeletons

The implementation of applications as in Figure 6.10 can be eased by a ISC-based skeleton library where map and reduce skeletons are deposited as fragment components to be parametrized with corresponding application-specific tasks. The small fragment library developed in this case study comprises four Java map and reduce skeletons (to avoid a too disruptive text fragmentation, interested readers may look up the skeleton code in Appendix A.4.3):

- `simple_map.jbx` is a map skeleton that runs in a single master thread and may be used if only a single process can be allocated or the input-data size does not justify an allocation of more than one thread. It has the following slots (cf. Table 6.1):
  - `IN_TYPE`: a `TypeAccessSlot` determining the input type of the map task,
  - `OUT_TYPE`: a `TypeAccessSlot` determining the output type of map,
  - `MAP_OP`: a `MethodAccessSlot` determining the name of the map operation with `IN_TYPE` $\rightarrow$ `OUT_TYPE` as a signature,
  - `IN_PORT_INIT`: an `ExprSlot` to be filled with a first request of data chunk (expression according to `IN_TYPE`),
  - `IN_PORT_CONT`: an `ExprSlot` to be filled with a repeated request of data (may be the same as the expression assigned to `IN_PORT_INIT`),
  - `OUT_PORT`: a `VarAccessSlot` determining the output sink where the skeleton's computed results are written to.
- `simple_reduce.jbx` is a reduce skeleton that runs in a single thread if only a single process can be allocated or the input-data size is small. Except `MAP_OP`, it has the same slots as `simple_map.jbx` plus:
  - `OUT_INIT`: an `ExprSlot` which can be bound to an expression yielding a default reduction value in case no reduction is performed,
  - `REDUCE_OP`: a `MethodAccessSlot` determining the name of a reduce operation with two parameters of type `IN_TYPE` and `OUT_TYPE` where the reduced results are integrated.
- `concurrent_map.jbx` contains a shared memory variant of a map skeleton based on the `java.util.concurrent` package. A master thread maintains a thread pool and allocates workers with map tasks on demand. It has the same slots as `simple_map.jbx`, but additionally:
  - `WORKERS`: an `ExprSlot` to determine the maximum number of worker threads managed by the skeleton.
- `concurrent_mapreduce.jbx` is a combined map and reduce skeleton that uses the same thread pool for map and reduce operations. It corresponds to the MapReduce

application in Figure 6.10: a master thread assigns map tasks to workers of the thread pool, the outputs of map and reduce workers are input to reduce workers, each taking two input data sets. The set of slots supported by this skeleton derives from a unification of the slots supported by the three other skeletons.

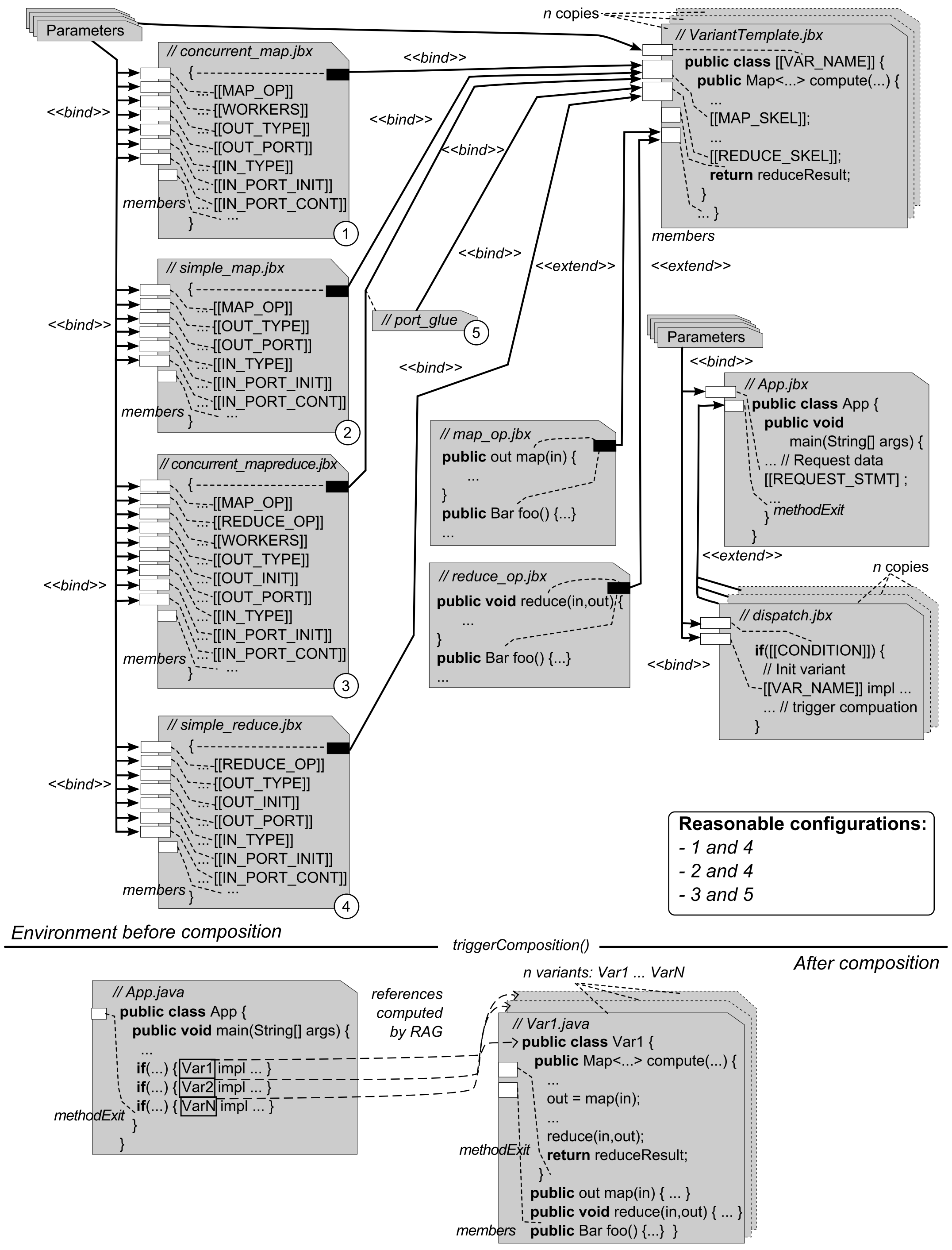
Above fragment components are Java blocks. Since blocks are statements, they can be bound to any `StmtSlot` or `statements` hook in SkAT4J. As blocks have a local scope, the input and output to the tasks must be glued with the surrounding program code via the `IN_PORT*` and `OUT_PORT` slots.

Using the Fragment Library

Figure 6.11 exemplifies a composition scenario where different variants of a MapReduce computation are generated. The variants are used by an application with a simple dispatcher which delegates calls of the application to specific variants depending on a dispatch condition, e.g., according to the current system resources and processed data. `VariantTemplate.jbx` in the upper-right corner of Figure 6.11 contains a template of a variant. After parametrization and extension, it becomes a single MapReduce application that can be instantiated as an object and executed via a call to `compute`. The template provides a `MAP_SKEL` and a `REDUCE_SKEL` slot which can be parametrized with map and reduce blocks. With the four skeletons discussed above and assuming the same set of slot parameters, a composition program can derive three obvious MapReduce variants. It can combine `simple_map.jbx` with `simple_reduce.jbx` producing a single-threaded variant, it can combine `concurrent_map.jbx` with `simple_-reduce.jbx` to obtain a variant with a parallel map and a single reduce (e.g., in cases map emits rather small data sets), or it binds `concurrent_mapreduce.jbx` to `MAP_SKEL` using some `port_glue` that is inserted in `REDUCE_SKEL` chaining the output port of map with the reduce operation's result.[3] The actual task implementations are sketched as `map_op.jbx` and `reduce_op.jbx` in the center of Figure 6.11. Each contains a set of Java member declarations to be woven into each concrete variant by extending its `members` hook. The entry point of the map task is `map` while the reduce task is invoked by calling `reduce`. Hence, these names are bound to the respective slots of the skeletons (`MAP_OP` and `REDUCE_OP`). As in the examples before, fragment contracts check the validity of the composition: a method name not accessible from within the generated variant will not be inserted into the skeleton ignoring the corresponding composer declaration and reporting a problem. Something similar holds for the method `foo` of both task implementations: the conflict will be recognized by a contract check and solved by preferring the first inserted one.

Besides n implementation classes, the scenario also aims at generating an `App` class (fragment box `App.jbx`) which is executable from command line and evaluates parameters from a simplified model of the current system to choose one of the generated variants for execution. The

[3]Instances of map and reduce skeletons can be combined in arbitrary ways by chaining their `IN_PORT`s and `OUT_PORT`s, yielding arbitrary parallel applications.



```
// concurrent_map.jbx
{
  [[MAP_OP]]
  [[WORKERS]]
  [[OUT_TYPE]]
  [[OUT_PORT]]
  [[IN_TYPE]]
  [[IN_PORT_INIT]]
  [[IN_PORT_CONT]]
  ...
}
```

```
// simple_map.jbx
{
  [[MAP_OP]]
  [[OUT_TYPE]]
  [[OUT_PORT]]
  [[IN_TYPE]]
  [[IN_PORT_INIT]]
  [[IN_PORT_CONT]]
  ...
}
```

```
// concurrent_mapreduce.jbx
{
  [[MAP_OP]]
  [[REDUCE_OP]]
  [[WORKERS]]
  [[OUT_TYPE]]
  [[OUT_INIT]]
  [[OUT_PORT]]
  [[IN_TYPE]]
  [[IN_PORT_INIT]]
  [[IN_PORT_CONT]]
  ...
}
```

```
// simple_reduce.jbx
{
  [[REDUCE_OP]]
  [[OUT_TYPE]]
  [[OUT_INIT]]
  [[OUT_PORT]]
  [[IN_TYPE]]
  [[IN_PORT_INIT]]
  [[IN_PORT_CONT]]
  ...
}
```

```
// port_glue
```

```
// VariantTemplate.jbx
public class [[VAR_NAME]] {
  public Map<...> compute(...) {
    ...
    [[MAP_SKEL]];
    ...
    [[REDUCE_SKEL]];
    return reduceResult;
  }
... }
```

```
// map_op.jbx
public out map(in) {
    ...
}
public Bar foo() {...}
...
```

```
// reduce_op.jbx
public void reduce(in,out) {
    ...
}
public Bar foo() {...}
...
```

```
// App.jbx
public class App {
  public void
        main(String[] args) {
    ... // Request data
    [[REQUEST_STMT] ;
    ...
  }
}
```

```
// dispatch.jbx
if([[CONDITION]]) {
  // Init variant
  [[VAR_NAME]] impl ...
  ... // trigger compuation
}
```

Reasonable configurations:
- 1 and 4
- 2 and 4
- 3 and 5

```
// App.java
public class App {
  public void main(String[] args) {
    ...
    if(...) {Var1 impl ... }
    if(...) {Var2 impl ... }
    if(...) {VarN impl ... }
  }
}
```

```
// Var1.java
public class Var1 {
  public Map<...> compute(...) {
    ...
    out = map(in);
    ...
    reduce(in,out);
    return reduceResult;
  }
  public out map(in) { ... }
  public void reduce(in,out) { ... }
  public Bar foo() {...} }
```

Figure 6.11.: Invasive composition of MapReduce skeletons.

```
 1  public class [[VAR_NAME]] {
 2     public Map<String,Integer> compute(Collection<String> in) throws Exception{
 3        // 1. Query Data.
 4        Iterator<String> it = in.iterator(); // the IN_PORT source
 5
 6        // 2. Split and initialise Map.
 7        Collection<Map<String,Integer>> mapResult = null; // the map OUT_PORT sink
 8        [[MAP_SKEL]]; // the slot for map operations
 9
10        // 3. Do reduce of mapped Data.
11        Iterator<Map<String,Integer>> mIt = mapResult.iterator();
12        Map<String,Integer> reduceResult = null; // the reduce OUT_PORT sink
13        [[REDUCE_SKEL]]; // the slot for reduce operations
14
15        // 4. Return the result.
16        return reduceResult;
17     }
18  }
```

Listing 6.24: Detailed variant template of the document crawler.

application's data is obtained via a request statement expression to be bound to REQUEST_STMT. One dispatch statement (fragment box dispatch.jbx) per variant is added via the method-Exit hook to the main method (the application's entry point). Each of the n required dispatch statements needs to be parametrized with a Boolean dispatch condition via the CONDITION slot and the name of the specific variant via VAR_NAME. Finally, after composition execution, the composition system emits the n variant Java classes Var1–VarN as shown in the lower part of Figure 6.11. The generated App class holds references to each variant in its dispatch blocks within the main method. After compilation with a Java compiler, the application can be deployed on a target machine and executed by a Java 1.5 or better runtime.

Consecutively, the composition scheme is realized as a SkAT4J composition program to compose a self-tuning document-crawling application.

Composing a MapReduce-Based Document Crawler

The MapReduce generator described in this section is an instance of the composition scenario of Figure 6.11. Therefore, the application-specific fragments participating the MapReduce composition are discussed before any composition-program part is be presented.

Listing 6.24 contains the basic variant template providing the according slots for naming a variant (VAR_NAME) and parametrizing it with map (MAP_SKEL) and reduce (REDUCE_-SKEL) implementations. As an input parameter it expects a Collection of strings where each string represents a document in Wikipedia's Wiki syntax (not discussed here, see [Wikipedia contributors 2014b]). This is expected to be a *lazy* Collection requesting and caching data from the document server on demand via an Iterator object. Later bound to the IN_-PORT of the map skeleton, the Iterator is requested from the method input in Line 4. The Collection of key-value Maps declared in mapResult in Line 7 is later used in the OUT_-

```
public static void reduce(Map<String,Integer> in1, Map<String,Integer> inout2){
   for(String string:in1.keySet()){
      if(inout2.containsKey(string)){
         inout2.put(string,inout2.get(string) + in1.get(string));
      }
      else{
         inout2.put(string,in1.get(string));
      }
   }
}
```

Listing 6.25: Reduce implementation of `reduce_op.jbx`.

`PORT` of map and provides the input to the reduce skeleton by accessing its `Iterator` (cf. Line 11). While map provides a set of key-value `Maps`, reduce computes a single `Map` whose output variable `reduceResult` is declared in Line 12 and later accessed by the skeleton.

An implementation of the reduce task is provided as the Java method `reduce` shown in Listing 6.25. It takes two maps `in1` and `inout2` as an input, where `inout2` is also used as an output variable. The `reduce` algorithm is simple: iterating over the word-count pairs of `in1`, it checks for each pair if its word is already contained in `inout2`. If yes, then the values are totaled otherwise the pair is copied to `inout2`.

The implementation of the reduce task is provided by the Java method `map` and `isWS` shown in Listing 6.26. The algorithm is rather simple: it takes a document string (`text`) as an input and iterates through its characters. Hereby, a single word is recognized as a continuous sequence of one or more non white-space characters which is determined by `isWS` recognizing wiki markup and other characters such as blanks as white space. To store values, `map` uses a standard `HashMap` (cf. Line 2). If a `word` is not contained in this map, it will be added with value 1, otherwise the counter is simply increased by 1 (cf. Line 13). Hence, on a per document basis, the result is already reduced while the global reduction is performed by `reduce`.

The application's main template is shown in Listing 6.27. Besides its slot to request documents (`REQUEST_STMT`) and the `methodExit` hook of `main`, it declares the `result` variable which is later used as an output sink of the final reduction step.

Listing 6.28 shows the employed dispatch template. It provides the `CONDITION` slot to set a dispatch condition and `VAR_NAME` to determine the dispatched variant. In case of a positively evaluated dispatch condition, the variant is instantiated and its `compute` method is invoked on the documents stored in `in`.

Having presented all relevant fragment components, the MapReduce generator can now be discussed. The main composition program is shown in Listing 6.29 and is again implemented as a Java method. For demonstration purposes of this example, it specifies the generation of two variants: a completely single threaded implementation ("1") and a variant supporting two threads ("2"). The program has four logical concerns ("slices"). Like in the previous examples, there is code configuring and managing the composition, e.g., initialization of the composition system in the beginning and execution at the end of the program. In between, the composition arrows of

```
public static Map<String,Integer> map(String text){
   Map<String,Integer> storage = new HashMap<String,Integer>();
   int textSize = text.length();
   for(int i = 0; i < textSize;){
      while(i < textSize && isWS(text.charAt(i)))
         i++;
      int start = i;
      while(i < textSize && !isWS(text.charAt(i)))
         i++;
      if(start < i){
         String word = text.substring(start,i);
         if(storage.containsKey(word))
            storage.put(word,storage.get(word) + 1);
         else
            storage.put(word,1);
      }
   }
   return storage;
}

public static boolean isWS(char c) {
   if(c < ’!’ || c == ’]’ || c == ’|’ ... )
      return true;
   return false;
}
```

Listing 6.26: Map implementation of `map_op.jbx`.

```
public class App {
   public static void main(String[] args) throws Exception {
      // Request documents from source.
      Collection<String> in = [[REQUEST_STMT]];
      Map<String,Integer> result = null;
   }
}
```

Listing 6.27: Application class template `App.java`.

Figure 6.11 are set up as composer declarations in the composition environment. In Lines 4–5, a call to `composeVariant` adds the two variant implementations, which will be described in detail in the paragraph after this paragraph. The next block between Lines 8 and 15 instantiates the application class `App.jbx` as a full Java class `App.java`. Moreover, it binds `REQUEST_-STMT` to the external method `Query.getWikiDocuments(string,int)`. This method takes two parameters to be provided by the user as a command-line parameter to `App`. The first parameter is the prefix of the requested documents, e.g., “A” queries documents whose names start with “A”. The second parameter is the number of documents requested. From these two parameters, multiple `http` requests to the Wikipedia web service API [Wikimedia Foundation, Inc 2014] are constructed and posted. The queries’ results are provided in form of a collection of document strings, as already mentioned above. The next statements of the composition program provide extensions to the `methodExit` hook of `App`’s `main` method. These extensions are

```
if([[CONDITION]]){
   // Initialize variant.
   [[VAR_NAME]] impl = new [[VAR_NAME]]();

   // Perform operation.
   result = impl.compute(in);
}
```

Listing 6.28: Dispatch template `dispatch.jbx`.

declarations of the two variables used in the variant-dispatch conditions: `cores` denotes the number of available physical processors supported by the Java `Runtime` while `threshold` is a command-line parameter denoting the corner-case document number for switching between single-threaded and multi-threaded variants. Observe that some in-line fragments are prefixed by `STMT::=`, which gives a hint to SkAT4J's fragment parser to parse a statement component. The actual dispatch code is woven in Lines 17–18 via `weaveDispatch`, which will be explained in more detail in a later paragraph. Completing the composition environment, the next compositional statements in Lines 20–24 extend `App`'s `main` with network code initializing a `Socket` connection based on a client's network address provided by the third command-line parameter of `main`. The `result` computed by one of the two variants is then delivered to the client. Finally, the composition engine executes the specification and persists all `.java` fragments.

Listing 6.30 contains `composeVariant`, which is the composition subprogram called by `compositionProgram` discussed above. It takes two parameters: `variantName` is the variant's class name and `workers` is the number of worker threads the variant should handle. The program can be read as follows. First, `VariantTemplate.jbx` is copied and renamed according to `variantName` (cf. Line 3). Then, the newly created fragment box is prepared for parametrization and extension by adding required import declarations (cf. `weaveImports`, Line 7). The map skeletons are woven between Lines 10 and 26. If more than one worker is to be supported by the variant, the multi-threaded variant of `concurrent_mapreduce.jbx` is bound to `MAP_SKEL` and its `WORKERS` slot is constantly set to `workers` (cf. Line 11 and 12). In contrast, in the case only one thread is requested, `MAP_SKEL` is bound to the single-threaded variant of the map skeleton—`simple_map.jbx` (see Line 14). In the next block of composition statements, the previously inserted skeleton is parametrized (cf. the corresponding parts of Figure 6.11). The implementation of the map task (as provided by `map_op.jbx`) is added via the variant's `members` hook in Line 19 and Line 20, the call to `map` is bound to `MAP_OP` in Line 21. The `IN_PORT*` slots are filled with a call to the variant's document iterator on the collection of strings the variant is invoked on while the output sink `OUT_PORT` is bound to `mapResult`. Complementary, `IN_TYPE` is set to `String` (as documents are strings) and `OUT_TYPE` is `Map<String, Integer>` (as `map` computes a list of such `Maps`). The reduce skeletons are woven from Line 29 to Line 44. Again, it is distinguished between a multi-threaded and a single-threaded variant. Since in the concurrent case a combined map *and* reduce skeleton is used, a glue statement is composed in Line 31 which simply transfers the first

```
private JavaCompositionSystem cSys = new JavaCompositionSystem("skel/","in","out");
public void compositionProgram() throws IOException{
   // Parametrize and extend variant templates.
   composeVariant("WikiCrawler1", 1);
   composeVariant("WikiCrawler2", 2);

   // Compose main application class and dispatch code.
   cSys.copyBox("App.jbx","App.java");
   cSys.addBindContent("App.java#REQUEST_STMT",
      "Query.getWikiPages(args[0],Integer.parseInt(args[1])).values()");
   String mainHook = "App.java#App.main.methodExit";
   cSys.addExtendContent(mainHook,
      "STMT::= int cores = Runtime.getRuntime().availableProcessors();");
   cSys.addExtendContent(mainHook,
      "STMT::= int threshold = args.length >= 4?Integer.parseInt(args[3]):10000;");

   weaveDispatch("WikiCrawler1", 1);
   weaveDispatch("WikiCrawler2", 2);

   cSys.addExtendContent(mainHook,
      "STMT::= java.net.Socket sock = new java.net.Socket(args[2], 12345);");
   cSys.addExtendContent(mainHook,
   "(new java.io.ObjectOutputStream(sock.getOutputStream())).writeObject(result);");
   cSys.addExtendContent(mainHook, "sock.close();");

   // Execute composition and write to output directory.
   cSys.triggerComposition();
   cSys.persistFragments(".*\\.java");
}
```

Listing 6.29: MapReduce composition program.

and only entry of `map`'s output to `reduceResult`. In the single-threaded case, the contents of `simple_reduce.jbx` are bound to `REDUCE_SKEL` in Line 33. The next compositional statements extend and parametrize the reduce-related parts of the variant. The reduce operation is added by extending `members` in Line 37 while a corresponding method call is bound to `REDUCE_OP` in Line 38. In the remainder of `composeVariant`, all reduce-related slots are treated. `IN_TYPE` and `OUT_TYPE` are set to map's entry type `Map<String,Integer>`, since the reduce operation in this example takes a set of these values reducing it to a single value of the same type. The corresponding `IN_PORT*` slots are set to iterator calls on map's output `mIt`. Finally, `OUT_PORT` is bound to `reduceResult` whose default value is an empty `Map`.

As the last part of the composition program, the dispatcher generator is discussed in this paragraph. The corresponding method `weaveDispatch`, which is called by `composition-Program`, is shown in Listing 6.31. First, a variant-specific copy of the dispatch template in `dispatch.jbx` is created in Line 4. This copy's `CONDITION` slot is parametrized in Lines 7–14. The constructed Boolean expression uses the variables `core` and `threshold` previously provided as an extension to `App`'s main method. The composed conditions are simple: if more than one `core` is available and the input size is larger than the threshold value, use the multi-threaded variant, otherwise the single-threaded one. Finally, the dispatch copy's `VAR_NAME` slot

```
private void composeVariant(String variantName, int workers) throws IOException {
   // Create copy of variant template.
   cSys.copyBox("VariantTemplate.jbx", variantName + ".java");
   cSys.addBindContent(variantName + ".java#VAR_NAME", variantName);

   // Weave import statements.
   weaveImportComposers(variantName + ".java");

   // Parametrize map slot depending on worker count.
   if(workers > 1){
      cSys.addBind(variantName + ".java#MAP_SKEL", "concurrent_mapreduce.jbx");
      cSys.addBindContent(variantName + ".java#WORKERS", "(" + workers + ")");
   }
   else cSys.addBind(variantName + ".java#MAP_SKEL", "simple_map.jbx");

   // Bind and extend variant with map-dependent stuff.
   String mapPort = "it.hasNext()?it.next():null";
   String mapType = "Map<String, Integer>";
   cSys.addExtend(variantName + ".members", "map_op.jbx_0");
   cSys.addExtend(variantName + ".members", "map_op.jbx_1");
   cSys.addBindTerminal(variantName + ".java#MAP_OP", "map");
   cSys.addBindContent(variantName + ".java#IN_PORT_INIT", mapPort);
   cSys.addBindContent(variantName + ".java#IN_PORT_CONT", mapPort);
   cSys.addBindContent(variantName + ".java#IN_TYPE", "String");
   cSys.addBindContent(variantName + ".java#OUT_TYPE", mapType);
   cSys.addBindContent(variantName + ".java#OUT_PORT", "mapResult");

   // Parametrize reduce slot depending on worker count.
   if(workers > 1)
      cSys.addBindContent(variantName + ".java#REDUCE_SKEL",
                           "reduceResult = mIt.next();");
   else
      cSys.addBind(variantName + ".java#REDUCE_SKEL", "simple_reduce.jbx");

   // Bind and extend variant with reduce-dependent stuff.
   String reducePort = "mIt.hasNext()?mIt.next():null";
   cSys.addExtend(variantName + ".members", "reduce_op.jbx");
   cSys.addBindTerminal(variantName + ".java#REDUCE_OP", "reduce");
   cSys.addBindContent(variantName + ".java#IN_TYPE", mapType);
   cSys.addBindContent(variantName + ".java#OUT_TYPE", mapType);
   cSys.addBindContent(variantName + ".java#IN_PORT_INIT", reducePort);
   cSys.addBindContent(variantName + ".java#IN_PORT_CONT", reducePort);
   cSys.addBindContent(variantName + ".java#OUT_PORT", "reduceResult");
   cSys.addBindContent(variantName + ".java#OUT_INIT", "new Hash" + mapType + "()");
}
```

Listing 6.30: MapReduce composition program: variant composition.

is set to the name of the current variant and `App.main` is extended with the statement.

The composition system described above generates three output files. The `App.java` file emitted by `SkAT4J` is shown in Listing 6.32. It contains all the woven statements, the socket communication with the client and the dispatch code to the variants. Because of their lengths, interested readers may look up both components in Appendix A.4.3, Listing A.18 and Listing A.19. Regarding fragment contracts, the composition system steadily validates the conditions and

```java
private void weaveDispatch(String variantName, int workers) throws Exception {
   // Create dispatcher copy.
   String ifName = "dispatch" + workers + ".jbx";
   cSys.copyBox("dispatch.jbx", ifName);

   // Bind dispatcher condition depending on worker count.
   if(workers>1){
      cSys.addBindContent(ifName+"#CONDITION",
            "cores >= " + workers + "&& threshold < in.size()");
   }
   else {
      cSys.addBindContent(ifName+"#CONDITION",
            "cores == 1 || threshold >= in.size()");
   }

   // Bind dispatch target and extend App.main.
   cSys.addBindContent(ifName+"#VAR_NAME",variantName);
   cSys.addExtend("App.java#App.main.methodExit", ifName);
}
```

Listing 6.31: MapReduce composition program: dispatch weaver.

assertions described previously in Section 6.4. Hence, the system only composes user-defined map and reduce operations if those rely on variables and methods visible in the target scope. Moreover, it checks the type correctness of `IN_PORT` and `OUT_PORT` slots in context of assignment expressions. Of course, not all potential context-sensitive problems can be detected with the current contracts in SkAT4J. For instance, in the above example, types and variable uses of fragment bound to the `CONDITION` slot could be validated additionally while in more difficult cases, complex invariants could be applicable (cf. Section 5.4). Finally, assuming successful contract and invariant checks, the three source files can be compiled by the Java compiler and packaged as an executable `.jar` file.

This section only gave a small excerpt from what is possible with SkAT considering skeletal code generation. The Java skeleton library discussed above can easily be extended with additional variants, e.g., supporting distributed computations via Java's remote method invocation API, web service invocations for cloud-based applications or even computations on graphics processing units (GPUs). The latter has already been investigated in a student's thesis [Henadeera 2014], supervised by the author of this thesis, using the "Rootbeer" Java-to-GPU compiler [Pratt-Szeliga et al. 2012]. Although the approach looks promising, it is still in its very early stage and is not discussed here and therefore remains as future work.

6.5. Summary and Conclusions

In this chapter, the SkAT fragment-composition framework was introduced. Based on the JastAdd RAG system, it implements the RAG-based fragment-composition approach introduced in Chapter 5. SkAT/Core provides the basic fragment environment and the basic composition API.

```java
public class App {
  public static void main(String[] args) throws Exception {
    Collection<String> in
       = Query.getWikiPages(args[0], Integer.parseInt(args[1])).values();
    Map<String, Integer> result = null;
    int cores = Runtime.getRuntime().availableProcessors();
    int threshold = args.length >= 4 ? Integer.parseInt(args[3]) : 10000;
    if(cores == 1 || threshold >= in.size()) {
      WikiCrawler1 impl = new WikiCrawler1();
      result = impl.compute(in);
    }
    if(cores >= 2 && threshold < in.size()) {
      WikiCrawler2 impl = new WikiCrawler2();
      result = impl.compute(in);
    }
    java.net.Socket sock = new java.net.Socket(args[2], 12345);
    (new java.io.ObjectOutputStream(sock.getOutputStream())).writeObject(result);
    sock.close();
  }
}
```

Listing 6.32: The generated MapReduce application `App.java`.

It defines the point-identification attributes for slots, hooks and rudiments and an implementation of the corresponding composition operators bind, extend and extract. Moreover, the look-up attributes for points and composer declarations are specified as a part of SkAT/Core, so that points and composers in fragment components of a specific fragment environment can easily be found by using these attributes via SkAT/Core's Java-based API. Using attribute grammars as an implementation technique for static languages semantics, SkAT/Core defines a basic infrastructure for semantic fragment contracts supporting pre- and postconditions as well as invariant checks to make fragment composition more reliable. Thanks to these attributes,

SkAT is the first ISC framework which supports well-formed ISC.

As a layer on top of SkAT/Core, SkAT/Full adds composer declarations, composition strategies and a rewrite integration with JastAdd's demand-driven ReRAG algorithm. Hence, composition systems based on SkAT/Full support different execution modes. In imperative mode, composition is executed at declaration time while strategies are always evaluated on a set of composer declarations which were declared beforehand. In comparison with existing ISC systems,

SkAT is the first ISC system supporting different composition strategies with well-defined semantics.

Using JastAdd RAGs as FCM specification language adds other advantages over previous ISC specification and implementation approaches: RAG-based component models are extensible and can be composed with other RAG-based languages. Additionally, these FCMs can be organized along custom composition-related concerns in such a way that composition-system developers can organize their specifications freely according to their "mental model". Moreover, in direct comparison to DSL-based FCM specification approaches like Reuseware, JastAdd RAGs are as well declarative but in contrast to the plain DSLs they are also expressive enough to specify arbitrary FCMs.

As an example SkAT/Full composition system, SkAT4J has been specified and presented. SkAT4J is an extension of JastAddJ—a JastAdd implementation of a full Java compiler. To support the implementation of this system, the FCM of COMPOST was taken as a reference. Hence, hook names, fragment model and composition API are similar, but internally the implementation, specification and composition features are superior, since it has a declarative and extensible specification, fragment contracts, and composition strategies.

In the last section of this chapter, three example applications of SkAT4J were presented. The mixin example demonstrated the basic working principles of the composition system and showed how strategies and fragment contracts can drive composition and resolve conflicts. Afterwards, the mixin composer is used in the realization of a SkAT4J-based code generator for the introductory BAF example. Using SkAT4J, it was possible to cover *all* requirements of the BAF code generator without any compromises or modifications, which is fairly better than the other approaches that were discussed in Chapter 4. Please observe that the BAF example has been defined independently of any of the composition systems which were compared in its implementation and that the experiments were conducted as objective and fair as possible. A considerable amount of time has been invested to create the other tools' implementations, to fix bugs or implement workarounds in the other systems to achieve the best possible results. Hence, in comparison, to the best knowledge and experience of this thesis' author

SkAT4J is the most stable, reliable and advanced ISC system for Java that has been developed so far.

The last of the three case studies shows ISC as a practical and advanced implementation approach for algorithmic skeletons and static generation of variants of parallel implementations. However, in the future this claim should be approved in a larger scenario and a larger fragment library, which are not covered by this thesis.

To finalize the discussions of this chapter, Table 6.5 provides a direct comparison of the ISC framework presented in Chapter 4 and SkAT w.r.t. 15 important concerns of ISC. For details on the respective features it is kindly referred to the corresponding parts of this thesis. Obviously, SkAT has support for the most of these features, except the fragment role concept introduced by Reuseware and layout-preserving graphical composition. Considering FCM extensibility, in Reuseware additional FCM specifications can be added interpreted by the system. However, as composition and FCM are strongly coupled, this is not a "real" extension mechanism of the FCM, but of the "composition-model-program" as a whole (cf. the discussion in Section 4.4.2). Another interesting issue in the table is the support of graphical modeling languages like in Reuseware. In principle SkAT can also be applied to graphical modeling languages. This can be achieved by using the JastEMF Ecore backend for JastAdd, which makes SkAT compatible with such languages. However, composition of graphical artifacts involves additional information such as

	COMPOST	**Reuse*wair***	**Reuseware**	**SkAT**
Tree model	AST (Recoder)	EMOF (Ecore)	EMOF (Ecore)	AST (JastAdd)
Graph model	–	–	overlay graph, EMOF (Ecore)	overlay graph, RAG (JastAdd)
Tree FCM	slot, hook	slot	hook, prototype	slot, hook, rudiment
Tree operators	bind, extend, extract	bind, extend	hook ← prototype	bind, extend, extract
Graph FCM	–	–	slot, anchor	attribute declarations
Graph operators	–	–	slot ← anchor	attribute equations
Composition language	Turing complete (Java)	Turing complete (Java)	acyclic dataflow graph (DSL)	Turing complete (Java)
Composition strategies	–	–	–	fixpoint, pointwise, ordered, attributes
FCM specification	class hierarchy (Java) and inheritance	DSL	DSLs	attributes and equations (JastAdd RAG)
FCM extensibility	class-based inheritance	–	–*	RAG modules
Fragment roles	–	–	untyped	–
FCM manifestation	direct implementation	generative	interpreted	generative
Graphical language support	–	–	graphical EMOF languages	via JastEMF (not investigated)
Well-formedness	–	–	–	fragment assertions and contract attributes
Partial languages	–	–	–	island FCMs (next chapter)

Table 6.5.: A comparison of ISC frameworks.

layout information or positioning. Hence, a full support of graphical languages remains as future work for SkAT.

Having discussed the heavy-weight approach of well-formed ISC in this chapter, the next chapter on *scalable* ISC presents a complementary light-weight approach to ISC with *minimal* FCMs and partial language support.

7 Scaling Invasive Software Composition

ISC and well-formed ISC demand great skill from composition-system developers. Implementing fragment composition systems for programming languages requires knowledge about grammar-based language engineering and compiler construction. Although DSLs for component models such as Reuseware (cf. Section 4.4) can ease the development and lower the entrance barriers, complexity cannot be avoided in all cases. Abstract and concrete syntax of the component language (CnL) have to be provided as specifications or models that can be processed in the technological space of the composition tooling. For example, SkAT is based on JastAdd's RAG language and Java while Reuseware uses metamodels based on the Eclipse Modeling Framework (EMF). Moreover, a corresponding machine-processable specification of concrete syntax has to be provided. In the case of textual languages, this would be a parser grammar which is coupled with the language model and one or more target environments, e.g., Java. Consequently, if the CnL is not already available in the required form, the composition-system developer takes tasks of a compiler developer. Depending on the complexity of the CnL, this can be a time consuming and error-prone task which actually shifts the focus from composition-system development to compiler-frontend construction.

This chapter therefore suggests to complement the complex ISC approaches with a simpler one. *Minimal ISC* removes complexity from composition by initially excluding many of the features that make ISC reliable but also have high development costs. For fragment composition based on minimal ISC, no abstract syntax or language specifics are required since it supports composition at the lexical level, similar to language-independent preprocessors such as the C preprocessor (CPP). In contrast to the latter, minimal ISC provides an explicit fragment component model and can be extended easily. Due to RAGs, the extensibility of minimal ISC enables a more agile and flexible style of composition-system development which allows fragment composition

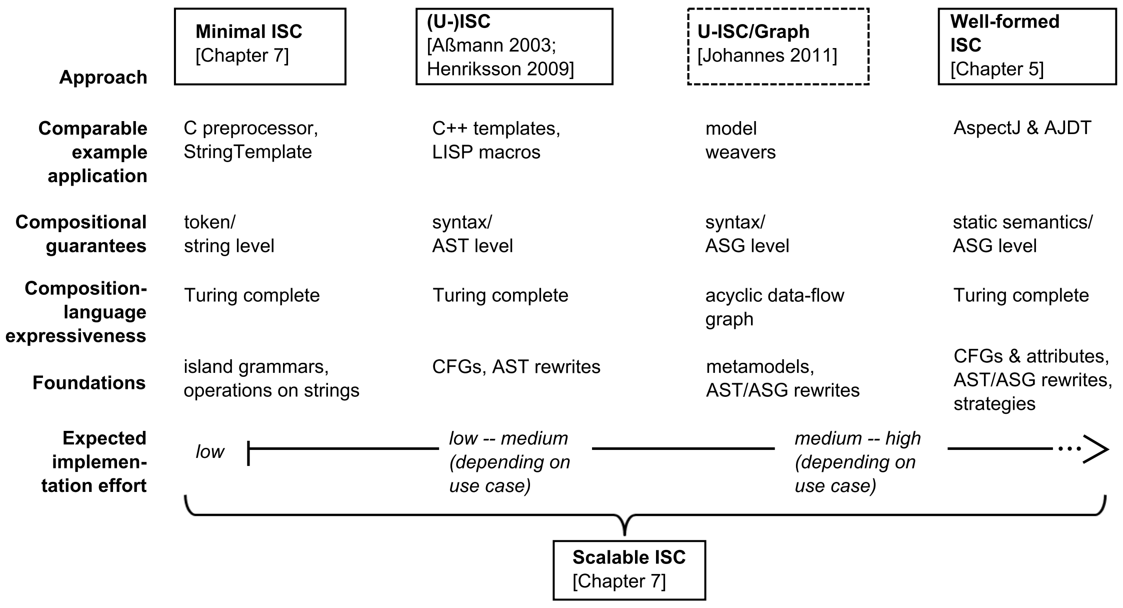

Approach
Minimal ISC [Chapter 7]
(U-)ISC [Aßmann 2003; Henriksson 2009]
U-ISC/Graph [Johannes 2011]
Well-formed ISC [Chapter 5]
Comparable example application
C preprocessor, StringTemplate
C++ templates, LISP macros
model weavers
AspectJ & AJDT
Compositional guarantees
token/ string level
syntax/ AST level
syntax/ ASG level
static semantics/ ASG level
Composition-language expressiveness
Turing complete
Turing complete
acyclic data-flow graph
Turing complete
Foundations
island grammars, operations on strings
CFGs, AST rewrites
metamodels, AST/ASG rewrites
CFGs & attributes, AST/ASG rewrites, strategies
Expected implemen-tation effort
low
low -- medium (depending on use case)
medium -- high (depending on use case)
Scalable ISC [Chapter 7]

Figure 7.1.: A comparison of the complexity of different ISC approaches.

systems to grow with the needs of an evolving software project. The basic idea of the resulting *scalable ISC* approach is sketched in Figure 7.1. Starting with a minimal FCM and composition operations on the string/token level, a full ISC system is developed step by step—adding features while requirements evolve and the system is kept operational. The full ISC model can then be enriched successively with well-formedness constraints of arbitrary complexity. Figure 7.1 also includes U-ISC/Graph as implemented in Reuseware. Since in comparison with the other approaches, operations on graphs are considered in the FCM and the composition, the effort of implementing such a system can be considered naturally between U-ISC and well-formed ISC. However, U-ISC/Graph is not supported directly by scalable ISC because model and terminology of U-ISC/Graph are fairly different from the other approaches (cf. Section 4.4).

The remainder of this chapter is structured as follows. Section 7.1 introduces the formal basics of minimal ISC and *island component models*. In Section 7.2, a SkAT-based realization of minimal ISC is presented. Section 7.3 discusses two composition systems implemented with SkAT-based minimal ISC and shows how they can be used to improve the code quality of an existing string-based code generator. Section 7.4 presents a CPP-like macro system based on minimal ISC and SkAT. Thanks to the declarative SkAT approach, the preprocessor is equipped with support for extensible syntax and semantics. Finally, Section 7.5 discusses the need of extensible fragment component models and composition systems and suggests an agile composition-system development process that supports scalable ISC.

7.1. Minimal Invasive Software Composition

Minimal ISC provides a language independent FCM with untyped slots. Specifically for textual languages, it builds upon *island grammars*—a special kind of context-free grammar (CFG), originally invented to robustly parse mixed languages with relevant (*islands*) and secondary (*water*) parts [Moonen 2001; Synytskyy et al. 2003]. Before diving into the details of FCMs based on island grammars and their specification using RAGs in Sections 7.1.3 and 7.1.4, the following Sections 7.1.1 and 7.1.2 recapitulate the island-grammar definition from the literature and discuss related grammar patterns.

7.1.1. Island Grammars

Definition 7.1 defines island grammars in relation to CFGs and formal languages. The definition corresponds to the one given in [Moonen 2001].

Definition 7.1 (Island Grammar):
Let $G = (N, \Sigma, P, S)$ be a context-free grammar and $L(G) = L_{\text{core}}$ the language generated by G. Further, let $I \subset \Sigma^*$ be a non-empty set of *constructs of interest* so that $\forall \gamma \in I\, \exists \alpha, \beta \in \Sigma^* : \alpha\gamma\beta \in L_{\text{core}}$. A context-free grammar $G_I = (N_I, \Sigma_I, P_I, S_I)$ with $L(G_I) = L_{\text{island}}$ is called an *island grammar* with respect to I and G under the following conditions:

- $L_{\text{core}} \subset L(G_I)$. G_I generates an extension of L_{core}, i.e., L_{island} has more sentences than L_{core} and all sentences of L_{core} are also in L_{island}.
- $\forall i \in I\, \exists n \in N_I\, \exists \alpha, \beta \in \Sigma^* : n \Rightarrow^*_{G_I} i \wedge \alpha i \beta \notin L_{\text{core}} \wedge \alpha i \beta \in L(G_I)$. Each construct of interest occurs in at least one context in L_{island} which is not in L_{core}.
- G is more "complex" than G_I. Complexity here means *descriptive complexity* which can be measured in multiple ways, e.g., graph complexity, number and length of productions etc. (cf. [Gruska 1976]). $\diamond$

The constructs of interest in I of the above definitions represent the relevant island parts of the mixed language while the other parts can be considered as water. Figure 7.2 visualizes the set relations between the constructs of interest, L_{core} and L_{island} as an Euler diagram. Thereby, I_Σ denotes the subset of Σ^* containing all sentences $\alpha\gamma\beta$ with $\gamma \in I$ and $\alpha, \beta \in \Sigma^*$, I_{core} denotes the subset of I_Σ which is also a subset of L_{core} and L_{island}, and I_{island} denotes the subset of Σ^* which is also a subset of L_{island} but not of L_{core}. Gently spoken, island-containing sentences of the core language have to be island-containing sentences of the language induced by the island grammar, but not vice versa. Hence, a corresponding island parser generated from an island grammar accepts *all* programs written in the core language and some which are not written in the core language, but may contain constructs of interest.

Example 7.1 illustrates the usage of Definition 7.1 by defining a simple island grammar of logical And-expressions.

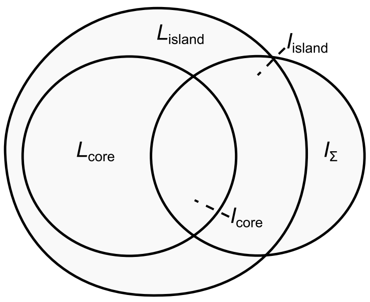


Figure 7.2.: Euler diagram of the relations between core language L_{core}, constructs of interest I and the language L_{island}.

Example 7.1 (island grammar (EBNF)).
Let G be the EBNF grammar from Example 3.2 with the following productions:

```
Expression ::= Or
Or ::= Or "|" Or | And
And ::= And "&" And | Term
Term ::= "t" | "f"
```

Thus, the language generated by G is $L(G) = \{$`"t"`, `"f"`, `"t|f"`, `"f|t"`, `"t|t"`, `"f|f"`, `"t&f"`, `"f&t"`, `"t&t"`, `"f&f"`, `"t|f|t"`, `"t|f&t"`, $\ldots\}$.

Now assume $L_{\text{core}} = L(G)$ and the following scenario: a legacy text format L_{legacy}, whose complete grammar is unknown, "mixes" some sentences from L_{core}—*islands*—with other sentences—*water*—in L_{legacy}. For example, `"t"` $\in L_{\text{core}}$ may reoccur in the sentence `"...a=t..."` $\in L_{\text{legacy}}$ where `t` is considered an island, and `...a=` and `...` are considered water. Moreover, all `&`-expressions over `t` should be recognized as islands in L_{legacy} sentences, i.e., $I = \{$`"t"`, `"t&t"`, `"t&t&t"`, `"t&t&t&t"`, $\ldots\}$.

The following EBNF grammar G_I is an island grammar w.r.t. G and I with the following productions (`<legacy_token>` shall be a token that represents arbitrary characters in the L_{legacy} alphabet):

```
Expression ::= ( And | Text )+
And ::= And "&" Term | Term
Term ::= "t"
Text ::= <legacy_token>+
```

G_I is an island grammar w.r.t. G and I because $L(G_I)$ contains all alternating sequences of `&`-expressions over `t` and arbitrary L_{legacy} text. This fulfills the second bullet of the island grammar definition—constructs of interest in I are not only recognized in context of logical expressions in L_{core} but arbitrary text of L_{legacy}. Furthermore, $L(G_I)$ is an extension of L_{core} and thus fulfills the first bullet of Definition 7.1. As G_I also contains fewer productions than G, the complexity criterion of the third bullet is also fulfilled. Thus, considering Figure 7.2, $L(G)$ is L_{core}, $L(G_I)$ is L_{island}, and `"...a=t..."` is an exemplary element of I_{island} and `"t|f&t"` is an exemplary element of I_{core}. ◇

Island grammars were originally developed with a focus on *robust parsing* to practically detect certain constructs of a certain language in unforeseen contexts. Hence, the constructs of interest of an island grammar are actually sentences that should be found in an unknown file format while the island grammar itself is a *context-free search pattern*.

In the standard application of ISC, robustness is not an issue since in an ideal case L_{core} and its compositional extensions would be recognized exactly. However, to recognize L_{core} exactly, it has to specified by its grammar, which is what actually should be avoided because of the assumed complexity of the language. Using an island grammar as CnL specification (plus compositional constructs like slots and fragment-component declarations) leads to small and simple grammars, and simple FCMs.

7.1.2. Parsing Patterns for Island Grammars

From a technical perspective, not all parsing approaches (cf. Section 3.2) are suitable for generating a parser from an island grammar. The widely-used classical parsing approach with sequential, uncoupled tokenization and syntactical analysis phases is not adequate—the unstructured textual water parts may overlap with tokens that belong to the concepts of interest, which may cause violations of island-grammar constraints or unresolvable ambiguities. Reconsider G_I of Example 7.1 for instance. `<legacy_token>` intersects with `"t"` and `"&"` so that more than one syntax tree can be derived for certain inputs. Consequently, one of the more advanced generalized or scannerless parsing approaches should be used. For example, [Moonen 2001] employs GLR parsing and the Syntax Definition Formalism (SDF) [Visser 1997] for specifying robust island grammars for certain COBOL constructs. In this chapter, PEGs and packrat parsing [Ford 2004; Grimm 2006] are used as implementation techniques for island grammars in the example composition systems.

Although Definition 7.1 does not require an island grammar to have a certain structural format or pattern, it is a good starting point to have one. The following incomplete EBNF grammar G_{island} suggests a basic structure for concrete syntax, where a fragment simply consists of `Island` and `Water` productions:

```
external I₁,...,Iₙ
Fragment ::= ( Island | Water )*
Island ::= I₁ | ... | Iₙ
Water ::=  <any_text>
```

The nonterminals I_i represent the constructs of interest (e.g., a method declaration and its dependent productions) which stem from the language definition of the core language L_{core}.

`Water` elements are just denoted to consist of an arbitrary piece of text. In practice, the text would just consist of a sequence of characters that can occur in the fragment files to be parsed.

However, while the above grammar fulfills Definition 7.1, it is difficult to be handled by parser generators because every string generated by `Island` can also be generated by `Water`, which makes the language ambiguous causing problems for classic LR and LL generators. Hence, a formalism to disambiguate grammars—like PEGs (cf. Section 3.2.1)—is required. Using predicate operators and ordered choices, G_{island} can be converted into a more precise PEG specification G'_{island} (“.” denotes any supported character):

```
external I1,...,In
Fragment <- ( Island / Water )*
Island <- I1 / ... / In
Water <-  ( !( Island ) . )+
```

This definition of an island grammar always prefers `Islands` over `Water` and defines `Water` as any string which is not an `Island` by using the *not* predicate. The following example demonstrates how the G'_{island} pattern is employed to define island PEGs.

Example 7.2 (Island Grammar (PEG)).
The EBNF-based island grammar of Example 7.1 can be reformulated as an island PEG:

```
external And
Fragment <- ( Island / Water )*
Island <- And
Water <-  ( !( Island ) . )+
```

where `And` is specified as an extra PEG module (if supported by the PEG tool employed) or simply appended to the specification, e.g.:

```
And <- And '&' Term / Term
Term <- 't'
```

However, there still are issues with the improved G'_{island}: the constructs of interest $\mathtt{I}_i$ may have a very complex structure in the original language L_{core}. Thus, one may want to keep them underspecified, containing water themselves. Furthermore, in the case of L_{core} it is not only a fragment language, but also a composition language, an embedded composition operator (or macro) may refer to water parts. To this problem, no general predefined solution using PEGs can be given. This becomes obvious, if, for example, an underspecified concept $\mathtt{I}_i$ with $\mathtt{I}_i$ `<-` α `Water` β is considered: `Water` will consume the content specified in β as long as β is not an island itself. Hence, each underspecified concept of interest requires its own water specifications, e.g., $\mathtt{I}_i$ `<-` α `( !( Island /` β `) . )+` β . In the special case the concept of interest is a part of the composition language taking water fragments as embedded arguments, syntactic hedges can be employed. In the following, $\mathtt{C}_i$ are compositional concepts of interest of the form $\mathtt{C}_i$ `<-` α `Water` β with $\beta = D\,\beta_1$, and D a delimiter. Based on these assumptions, a refined "composition-aware" island PEG G''_{island} can be derived from G'_{island}:

```
external I1,...,In,C1,...,Cm,D
Fragment <- ( Island / Water )*
Island <- I1 / ... / In / C1 / ... / Cm
Water <- ( !( D / Island ) . )+
```

The following example develops an island PEG based on the G''_{island} pattern.

Example 7.3 (island grammar (PEG) with slots and delimiters).
The island PEG of Example 7.2 can be reformulated as an island PEG with slots and slot delimiters according to the extended island grammar pattern:

```
external And, Slot, Delimiter
Fragment <- ( Island / Water )*
Island <- And / Slot
Water <-  ( !( Delimiter / Island ) . )+
```

where `Slot` and `Delimiter` are defined in a separate PEG module or simply appended to the grammar, e.g., slots surrounded by hash marks:

```
Slot <- Delimiter <slot_ident> Delimiter
Delimiter <- '#'
```

In the following sections, the notions of *island FCMs* and *minimal FCMs* are developed.

7.1.3. Island Fragment Component Models

As discussed above, an island grammar generates a mixed language with water and island text and subsumes the original core language (cf. the Euler diagram in Figure 7.2). However, for language processing tools such as fragment composition systems, the string-based view is not adequate because these tools work on syntax trees, and core and island grammars induce different trees. Accordingly, Figure 7.2 can be reinterpreted (where G is a grammar of L_{core} and G_I is a grammar of L_{island}): for each sentence w in L_{core} there is at least one syntax tree T generated by G and—since $L_{\text{core}} \subset L_{\text{island}}$—there also is at least one corresponding syntax tree T_I generated by G_I. If w does not contain an island ($w \notin I_{\text{core}}$), it generally holds that T is different from T_I (i.e., there is no isomorphism between T and T_I), basically because T_I represents water as a "list-like tree" while T is a "complex tree". However, if w contains an island, T and T_I are generally complex trees which may be identical (isomorphic) or may have common subtrees (subtree-isomorphic).

For island-grammar-based FCMs (*island FCMs*), compositional constructs such as slots, hooks, and composers are defined as island constructs or are defined with respect to island constructs in I_{core} and I_{island}. Hence, an island FCM is defined w.r.t. to a specific CnL, but using an island grammar instead of a full grammar (or model) of the CnL:

Definition 7.2 (island fragment component model):
Let $FCM = (G', \mathcal{S}, \mathcal{H}, \mathcal{F}, \int, \mathcal{L}_\int)$ be a fragment component model and G a grammar of the subjected component language $L(G)$. FCM is called an *island fragment component model* w.r.t. G under the following conditions:

- G' is an island grammar with respect to G,
- $\exists f' \in \mathcal{F}$ so that f' generates islands and water (i.e., the projected grammar $G'_{|f'}$ is an island grammar with respect to a projected grammar $G_{|f}$ with f—a nonterminal of G). $\diamond$

Hence, an island FCM is not aware of all fragment types and constructs of the fragment language but it is aware of some of them. Example 7.4 develops an island FCM for logical and-expressions.

Example 7.4 (island fragment component model).
Let G_I be the island grammar developed in Example 7.3 with slots and and-expressions as constructs of interest.

$FCM_I = (G_I, \mathcal{S}, \mathcal{H}, \mathcal{F}, \int, \mathcal{L}_\int)$ is a corresponding FCM with $\mathcal{S} = \{\texttt{Slot}\}$, $\mathcal{H} = \emptyset$, $\mathcal{F} = \{\texttt{Fragment}\}$, and $\int$ with $\int_\mathcal{S}$ assigning a slot name $w \in \mathcal{L}_\int$ to any `Slot` node of an G_I-tree, $\int_\mathcal{H}$ assigning $\perp$ to any node of an G_I-tree and $\int_\mathcal{F}$ assigning a name to any `Fragment`-rooted $G_{I|\texttt{Fragment}}$-tree. According to Definition 7.2, FCM_I is an island FCM since G_I is an island grammar w.r.t. the EBNF grammar G from Example 3.2 and $G_{I|\texttt{Fragment}} = G_I$ is an island grammar. $\diamond$

Fragment composition systems based on island FCMs can be developed with less implementation effort. This can be useful if no suitable grammar is available for the targeted technology space or the composition framework. As an example, consider a textual DSL L_D of a certain domain D that has been specified as a standard by an industrial consortium including a documentary CFG with some hundred productions for its concrete syntax. To realize a fragment composition system for L_D with a generative tool like SkAT or Reuseware (cf. Section 4.4), the CFG documented by the standard has to be translated into adequate grammar formalisms for these tools, i.e., a parser grammar, JastAdd AST grammars in the case of SkAT, or EMF models in the case of Reuseware. Also, the other compartments of the FCM have to be specified w.r.t. these grammars and become more complex. With an island FCM, this effort can be reduced because less language concepts of L_D have to be considered.

The usage of island FCMs comes with the downside of losing the full syntactic composition guarantees of ISC. However, since the language induced by the full grammar is a subset of the language given by the island grammar (cf. Definition 7.1), fragment components of the full

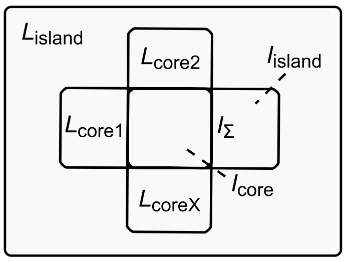


Figure 7.3.: Euler diagram of the relations between different core languages $L_{\text{core}*}$ with a common slot-signature, common constructs of interest I and the island language L_{island}.

language can still be managed by the composition system retaining at least "some" compositional guarantees based on the island constructs. The support of these guarantees depends on the complexity of the island grammar as well as on the actual component model specification. Concerning well-formed ISC, semantic fragment contracts may also be specified w.r.t. island FCMs. This gives a notion of *scaling* ISC across the dimensions of *specification complexity* and *quality of composition systems*, which will be discussed in more detail in Section 7.5.

Island FCMs can support an arbitrary number of constructs of interest from the core language as long as the FCM fulfills Definition 7.2. However, at the low end of the scale, what would be an adequate notion of a *minimal* (non-empty) island FCM and what kinds of implications would it have? If FCMs are considered generally, a reasonably "small" component model with respect to the respective CnL could be an FCM that only supports a small set of slot or hook candidates. It could be called "minimal", if it even only considers a single slot or hook type, and only one type of fragment. However, because FCMs are defined w.r.t. the CnL and there is no reasonable criterion to compare two FCMs of a CnL, there exists no minimal FCM for arbitrary CnLs, unless one is an extension of the other.

If island FCMs are considered, the situation is different. Reconsidering the relations between island language L_{island} and the core language L_{core} (cf. Figure 7.2), the set I_{core} should be "minimized" while L_{Island} is maximized in such a way that a common minimal slot language and grammar are obtained. Hence, $L_{\text{Island}} = \Sigma^*$ while $I_{\text{island}} \cup I_{\text{core}}$ consists of all sentences in Σ^* containing at least one slot. Figure 7.3 visualizes these relations as an alternative Euler diagram. It contains multiple core languages that share a common slot signature. The signature is supported by the corresponding island grammar G_I generating L_{Island}. Example 7.5 provides a minimal slot island grammar.

Example 7.5 (minimal island grammar (PEG) with slots and delimiters).
The island grammar of example 7.3 can be minimized to a slot island grammar by excluding any core-language specific productions.

```
    Fragment <- ( Island / Water )*
    Island <- Slot
    Water <- ( !( Delimiter / Island ) . )+
    Slot <- Delimiter <slot_ident> Delimiter
    Delimiter <- '#'
```

Since this grammar describes a regular language of alternating slots and arbitrary strings (i.e., `(#<slot_ident>#|.)*`), the language is called a *regular slot language*. ◇

Using a regular slot language as a basis of an FCM leads to a composition system which does not support any guarantees on the syntax of composed fragments. Also, the system misses a support for hooks—as there are no specific core language constructs considered in a minimal slot island grammar, hooks cannot be specified. The following definition introduces the notion of a *minimal FCM* based on the notion of minimal slot island grammars:

Definition 7.3 (minimal fragment component model):
Let $FCM = (G_I, \mathcal{S}, \mathcal{H}, \mathcal{F}, \int, \mathcal{L}_\int)$ be a fragment component model and $G_I = (N, \Sigma, P, S)$ an island grammar w.r.t. some CnL with slot candidates. FCM is called *minimal* if

- G_I is a minimal slot island grammar with $Slot \in N$, the slot nonterminal,
- $\mathcal{S} = \{Slot\}$, $\mathcal{H} = \emptyset$ and $\mathcal{F} = \{S\}$,
- $\int$ assigns a name $w \in \mathcal{L}_\int$ to each $Slot$-node and S-node of an arbitrary G_I-trees and $\perp$ to any other node. ◇

Example 7.6 defines a minimal FCM.

Example 7.6 (minimal fragment component model).
Reconsider the minimal slot island grammar of Example 7.5 as G_I. The corresponding minimal FCM is $FCM = (G_I, \mathcal{S}, \mathcal{H}, \mathcal{F}, \int, \mathcal{L}_\int)$ with $\mathcal{S} = \{\texttt{Slot}\}$, $\mathcal{F} = \{\texttt{Fragment}\}$ and the other compartments initialized according to Definition 7.3. ◇

While the slot language is regular, in a minimal FCM it is lifted to syntax trees with characters and compartments of slots as leaf nodes. For a corresponding ISC system, it suffices to support one AST grammar (model) to represent the minimal FCM, since all concrete slot grammars are equivalent except their delimiter symbols. Thus, in a composition system they can be represented by the same AST grammar (model).

Section 7.1.4 sketches out how the RAG-based approach of specifying invasive composition systems developed in Chapter 5 can be used to holistically specify a minimal FCM using an island AST-grammar in its core.

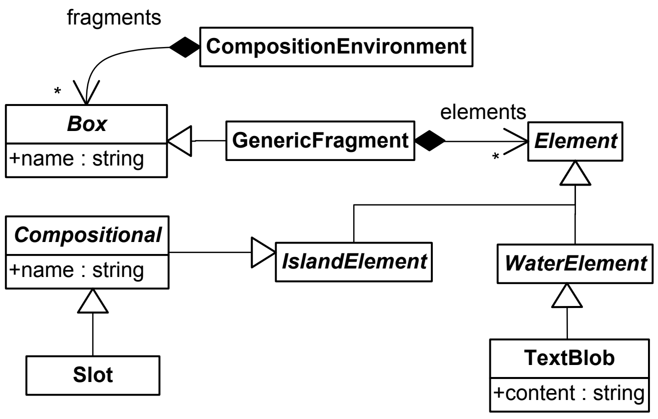


Figure 7.4.: Static structure of the minimal FCM model as a UML class diagram.

7.1.4. Minimal Fragment Component Model Specification

There are several reasonable ways of implementing a minimal ISC system. For instance, it is straight-forward to realize it using a standard regular expression engine or string operation library and wrapping a user friendly API around it. The result would be a fast template engine which is difficult to extend. However, in this thesis a holistic and extensible approach is favored to keep the number of involved technologies small and component models comparable. Extensibility is important to later be able to specialize and add language-specific constructs of interest to the FCM. As it has been discussed before in Section 3.3, current RAGs systems enable exactly this by providing concern-based extension mechanisms for attributes. Although island grammars are less complex than grammars of full-fledged languages, still RAGs can be used as a basic FCM specification formalism. Hence, first, the static structure of the minimal FCM is specified as an abstract EBNF grammar G_{MinCM}, which will later be decorated with attributes:

```
CompositionEnvironment ::= fragments:Box*
@Box ::= name:<string>
GenericFragment▷Box ::= elements:Element*
@Element ::= ϵ
@WaterElement▷Element ::= ϵ
TextBlob▷WaterElement ::= content:<string>
@IslandElement▷Element ::= ϵ
@Compositional▷IslandElement ::= name:<string>
Slot▷Compositional ::= ϵ
```

Figure 7.4 contains a diagrammatic representation of G_{MinCM} as a UML class diagram. Based on the generic fragment FCM presented in Chapter 5, the `CompositionEnvironment` contains a list of `GenericFragments` consisting of a list of `Elements`. There are two main classifiers for `Elements`, `WaterElements`, corresponding to water parts, and `IslandElements`, corresponding to constructs of interest in the language. `WaterElements` are typically `TextBlobs` which may contain unstructured text.[1]

Since minimal ISC shall be applicable to arbitrary fragment languages out of the box, G_{MinCM} does not include any `Island` construct of a core language. However, since RAGs are used, it can be extended with such. Denoted by `Compositional`, the component model supports island constructs which represent compositional elements such as slots or embedded composers. In the basic minimal ISC case, these are only `Slots` that can be used as placeholders for arbitrary unstructured pieces of `WaterElements`, i.e., text.

Finally, to make G_{MinCM} a minimal FCM, its compartments $\mathcal{H}$ and $\mathcal{F}$ are specified using attributes. Therefore, most parts of the generic component model RAG presented in Chapter 5 can be reused and must only be instantiated for the minimal FCM. To define $\int_{\mathcal{S}}$, the `isSlot` and `slotName` attributes have to be specified in such a way that any `Slot`-labeled node in any G_{MinCM}-tree are recognized as a slot while all other nodes are not. Hence, $\int_{\mathcal{S}}$ of the minimal FCM in SimpAG notation looks as follows:

$$\textbf{fun}\ \texttt{Slot.isSlot} = \textit{true} \tag{7.1}$$
$$\textbf{fun}\ \texttt{Slot.slotName} = \texttt{name} \tag{7.2}$$
$$\textbf{fun}\ \{n \mid n \in N \setminus \mathcal{S}\}\texttt{.isSlot} = \textit{false} \tag{7.3}$$
$$\textbf{fun}\ \{n \mid n \in N \setminus \mathcal{S}\}\texttt{.slotName} = \bot \tag{7.4}$$

To define $\int_{\mathcal{H}}$ using attributes, the inherited `isHook` always yields *false* independently of any context. In SimpAG notation, `isHook` and the corresponding `hookName` attribute derive as follows:

$$\textbf{fun}\ \{n \in N\}\texttt{.child}_{\texttt{all}}\texttt{.isHook} = \textit{false} \tag{7.5}$$
$$\textbf{fun}\ \{n \in N\}\texttt{.child}_{\texttt{all}}\texttt{.hookName} = \bot \tag{7.6}$$

Given the RAG-based specification of the minimal FCM and the composition-operator definitions developed in Chapter 5, attribute-grammar systems like JastAdd can derive an implementation of the minimal composition system. Figure 7.5 shows an example instance tree with overlay references of the fragment environment as it may occur in real composition scenarios. A corresponding tree with a superimposed data-flow graph can be found in Figure A.3 of Appendix A.5. Supporting the example, it is assumed that the FCM tree has been initialized by some parser component which accepts "[[MySlotName]]" as concrete syntax for slots. The figure contains three fragments with different alternating sequences of text and slots. Fragment `Tpl1` is the sequence "A [[S1]] [[S2]]" which is represented by a `TextBlob` and two `Slot` nodes in the composition environment. Fragment `Tpl2` is the sequence "B [[S3]] CD" which is represented by two `TextBlob` and one `Slot` nodes. Used as an argument to slot bindings, the third fragment

[1]It is possible that blobs may contain arbitrary kinds of data, not only characters. However, this is not investigated in this thesis.

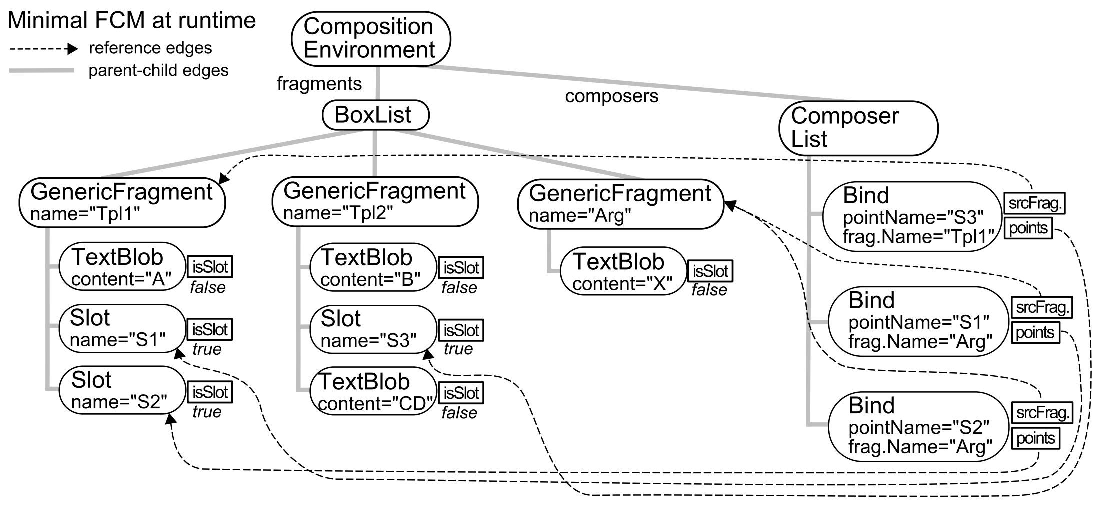

Minimal FCM at runtime
reference edges
parent-child edges
Composition Environment
fragments
composers
BoxList
Composer List
GenericFragment name="Tpl1"
GenericFragment name="Tpl2"
GenericFragment name="Arg"
TextBlob content="A" isSlot false
Slot name="S1" isSlot true
Slot name="S2" isSlot true
TextBlob content="B" isSlot false
Slot name="S3" isSlot true
TextBlob content="CD" isSlot false
TextBlob content="X" isSlot false
Bind pointName="S3" frag.Name="Tpl1" srcFrag. points
Bind pointName="S1" frag.Name="Arg" srcFrag. points
Bind pointName="S2" frag.Name="Arg" srcFrag. points

Figure 7.5.: Example instance of the minimal FCM with three fragments *Tpl1*, *Tpl2* and *Arg* including a simple composition program.

Arg only contains a `TextBlob` “X”. The composition program in the rightmost part of the figure contains three `Bind` nodes. The first declares a binding of `S3` in `Tpl2` with a copy of `Tpl1`. The remaining `Bind` nodes declare bindings of *S1* and *S2* in *Tpl1* with the `Arg` fragment. For composition execution, if a *fragment-depth-first* and *parametrize-before-copy* strategy is preferred, `S1` and `S2` of `Tpl1` are first bound with copies of `Arg` yielding the sequence “A X X”. Finally, `S3` in `Tpl2` is bound with the parametrized `Tpl1`, which yields the sequence “B A X X C D”.

7.2. Minimal Invasive Software Composition in SkAT

This section discusses the SkAT-based realization of a minimal invasive composition system and some composition abstractions that have been developed as an extension to this realization. SkAT/Minimal provides the infrastructure for minimal ISC applications including a JastAdd-based implementation of the minimal FCM presented in the previous section. Figure 7.6 provides a coarse-grained view on SkAT's systems architecture with the parts emphasized relevant for this chapter, while the faded-out parts actually belong to the SkAT-based realization of standard ISC and well-formed ISC already described in Chapters 5 and 6. On the base-level, the JastAdd tool is still used as a basic framework and RAG realization. Similar to SkAT/Full, which provides the basic attributes and functionality for well-formed ISC, SkAT/Minimal relies on the basic kernel—SkAT/Core. Thus, SkAT/Full and SkAT/Minimal share the basic attribute definitions for compositional points as well as the regular path language to address them.

By default, SkAT/Minimal only supports untyped slots and, thus, does not give any guarantees on composed fragments. Figure 7.7 shows the specifications involved as well as the SkAT

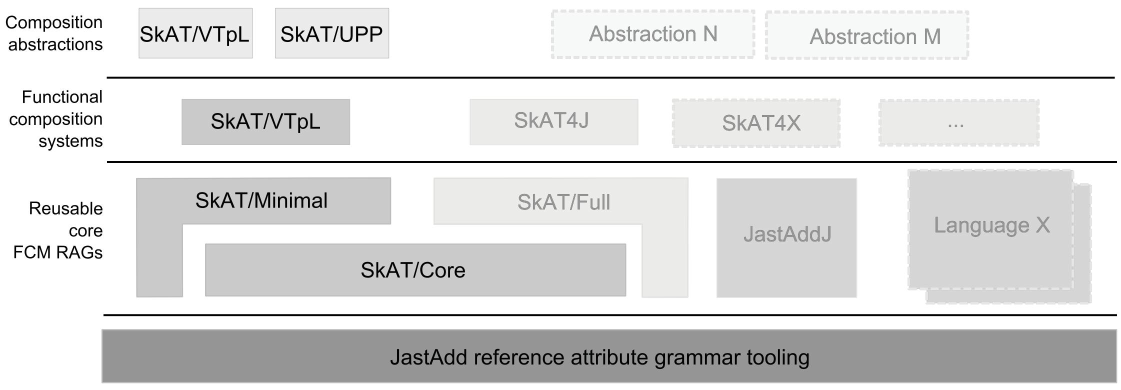


Figure 7.6.: Overview of the SkAT architecture with the minimal ISC parts emphasized.

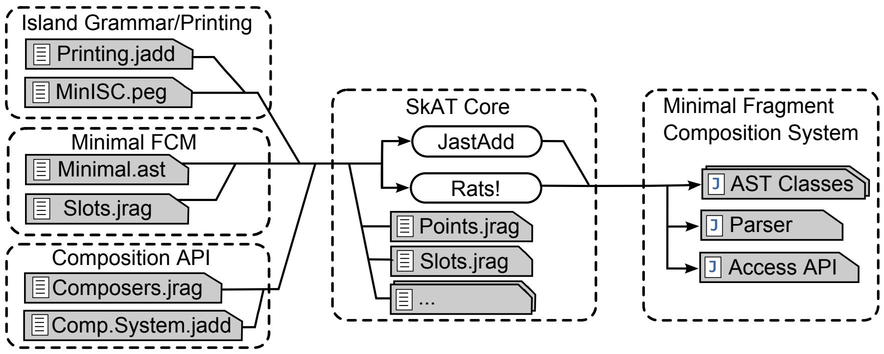


Figure 7.7.: Data-flow diagram of the SkAT/Minimal generation chain.

configuration for minimal ISC. The minimal FCM consists of two specifications. `Minimal.ast` contains the JastAdd AST grammar that corresponds to the SimpAG model presented above in Section 7.1.4. However, interested readers may find a complete specification in Listing A.20 in Appendix A.5. `Slots.jrag` is presented in Listing 7.7 and contains the JastAdd-based slot-identification attributes. Note that hooks need not to be considered explicitly since the defaults specified in the core RAG still hold and minimal ISC does not introduce any hooks by default. The specification also provides an equation for the `compatibleFragmentTypes` attribute stating arbitrary `Element` nodes as valid slot replacement. `Printing.jadd` provides the minimal FCM with a basic printing API to emit fragments as text at any point during composition. `Composers.jadd` and `CompositionSystem.jadd` contain a text-optimized API of the basic invasive composition operators.

Besides the JastAdd RAG tool, the SkAT configuration for minimal ISC uses the packrat-parser generator *Rats!* [Grimm 2006] to realize textual concrete syntax of island grammars. Using this tool has several advantages supporting minimal ISC:

```
aspect MinimalSlots {
   eq Slot.isSlot() = true;
   eq Slot.slotName() = getName();
   eq Compositional.compatibleFragmentTypes() = new Class[]{Element.class};
}
```

Listing 7.1: JastAdd Slots.jrag specification of the minimal FCM.

- *Modular and extensible Grammars.* Given a set of base grammars (syntax modules), new grammars can be derived via extension mechanisms. Existing productions in the base modules can be extended, refined or replaced as required. This can be seen complementary to the extensibility of JastAdd RAGs.
- *Unlimited lookahead at linear time.* Packrat parsers are not restricted with a maximum lookahead to make parsing decisions. This is achieved by combining a backtracking algorithm with an intelligent caching of intermediate results [Ford 2002].
- *State dependent token recognition.* Parsing without the need to independently create tokens streams in advance is essential to avoid overlaps and thus conflicts between terminals in island parts and text in water parts.
- *Repository independent creation of unique ASTs.* The AST can be constructed directly without the need of producing intermediate results with a parser-specific repository. The only requirement is to use Java, which is also the case for JastAdd RAGs.
- *Preemptive disambiguation.* By enforcing a strict ordering of grammar productions combined with the greedy packrat-parsing algorithm, and the possibility to specify syntactic predicates as custom lookahead checks, ambiguous grammars can be made deterministic in such a way that changes and extensions to fragment languages can be handled better than with standard parsing technologies.

Complementing the basic minimal FCM, SkAT/Minimal provides a generic *Rats!*-based island-grammar module which can be imported and reused in client applications (cf. Listing 7.2). While the grammar of the module generally corresponds to the abstract island PEG presented in Section 7.1.1, it already has a prepared production for slot declarations as island elements which can be parametrized with specific delimiters by clients reusing that module.

In the following two sections, three typical applications of SkAT/Minimal are presented. The examples have been chosen carefully to demonstrate different levels of implementation complexity of SkAT/Minimal applications. Furthermore, the interplay of grammarware technologies to implement *extensible fragment composition systems* is illustrated. Section 7.3 presents Slot Template Language (STpL) and Variant Template Language (VTpL), which have been used to reengineer parts of an existing code generator. The generator internally uses standard string-concatenation operations to produce the output code tangled with computations on the input model. To improve the quality of the code generator some parts have been refactored by making

```
// Defines the qualified module name (parser.MinISCBase) and required parameter
// module for delimiter syntax.
module parser.MinISCBase(Delimiters);

// Basic lexical syntax.
String Identifier = [a-zA-Z] [a-zA-Z0-9_]*;
void EndOfFile = !_;

// The start of minimal generic fragments.
public GenericFragment GenericFragmentDecl = elements:TplElements? EndOfFile {};
public List<Element> TplElements = content:(yyValue:Island / yyValue:Water)+ {};

// Syntax of water elements.
WaterElement Water = yyValue:TextElement {};
TextBlob TextElement = value:WaterContent {};
String WaterContent = ( (EscapeDelimiter Delimiters) / !( Delimiters / Island) _ )+ ;

// Hook of island elements syntax, provides a basic syntax pattern for slots.
IslandElement Island = yyValue:SlotDecl{};
Slot SlotDecl = SlotDelimiter name:Identifier SlotDelimiter { };
```

Listing 7.2: The SkAT/Minimal island-grammar module, except tree-construction actions.

patterns in the code explicit as an ISC system, which helps to reduce the tangling problem. Thereby, STpL just extends the SkAT/Minimal model with a concrete notation for slots. The VTpL further extends STpL and adds compositional constructs that model *fragment variants* and *fragment prototypes*. Section 7.4 introduces the Universal Extensible Preprocessor (UPP), a string-based template engine which uses many concepts of the CPP and provides extensions to its basic concepts.

7.3. Improving Code Generators with SkAT/Minimal Applications

Complex code-generation applications such as JastAdd's internal generator, which translates the attribute grammar into Java code, or textual modeling frameworks such as EMFText which generate a bunch of IDE plug-ins (cf. [Heidenreich et al. 2013]) emit vast amounts of lines of source code from—in relation—small input specifications. Regardless of the complexity of such tools, their developers frequently use string-concatenation operations in favor of template engines (e.g., [Parr 2006; Efftinge et al. 2008]), which would provide a holistic view on the input sources in contrast to fragmented and scattered string-concatenation operations. Concluding from the author's considerable experience in the development of generative applications (cf. [Bürger and Karol 2007; Heidenreich et al. 2009a; Heidenreich et al. 2009b; Niederhausen et al. 2009; Karol et al. 2010; Bürger and Karol 2010; Aßmann et al. 2012; Heidenreich et al. 2013]), an important reason why programmers of highly generative applications often avoid using template engines as the predominant tools is a discrepancy between the constrained local control flow of the template language and the global control flow imposed by the architectural design of the code-generating application as a whole. A template's constructs for sequential iteration and branching decide about how to visit the input model and emit fragment content sequentially on a per file basis. However, generative applications often employ architectures that do not produce their output in such straight-forward ways. Instead, the output is computed with a focus on problem decomposition, which cannot necessarily be superposed well to templates. For example, a program generator may concurrently produce several parts of the output depending on the source model and may not generate the parts in the same order as they finally occur in the emitted files. Even worse, it may generate in-memory intermediate representations in a multi-staged process as objects of further parametrization, extension or optimization before emitting it. Hence, developers in such cases often prefer string-concatenation-based construction of the output sources, or use programmed AST or model transformations if applicable. However, both approaches lack the holistic viewpoint of templates on the produced output.

ISC systems sustain the holistic template-based view on fragments while keeping the control flow in the code generator—as it already has been demonstrated with the BAF example introduced in Chapter 2 of this thesis. Unfortunately, if no ISC system for the involved fragment languages is available for immediate use in the application's implementation platform and if there are no capacities left to develop and test the required composition systems, full-featured ISC simply would not be an option. Fortunately, minimal ISC and the STpL can be used instead.

7.3.1. STpL: A Minimal Slot Mark-Up Template Language

The STpL is a minimal slot markup language that can be considered an incarnation of a frequently occurring string-replacement pattern in complex generative applications. In this pattern, a string object is constructed by iterating over elements of an input model recurrently appending new string parts which constitute element representations in some output fragment. Some of these parts contain placeholders (i.e., slots surrounded by syntactic hedges) that postpone decisions about their actual content to some later point in time when they are replaced with concrete strings.

In the following, a concrete example from JastAdd's code generator is discussed which, amongst others, has been investigated by the author with the objective of finding a concept to improve the tool architecture and code quality. The driving force behind this were efforts of creating an alternative backend for JastAdd that supports the infrastructure of the Eclipse Modeling Framework (EMF) to combine RAGs with state-of-the-art modeling tools (cf. [Bürger and Karol 2010; Bürger et al. 2011; Heidenreich et al. 2013]). However, the current implementation of the backend is a complex transformational approach that leaves the RAG and EMF code generators intact and instead transforms and merges the generated Java-based AST class hierarchies. While this demonstrated the practical feasibility of model-based RAGs, it turned out that the integration-based code-generation process is approximately $60\times$ *slower* than the unmodified process so that an assumed generation time of 1–2 seconds increases to 1–2 minutes. Since it is expected that a further improvement of the approach would lead to huge additional implementation efforts, it was decided to investigate the possibility of reengineering the original backend step by step.

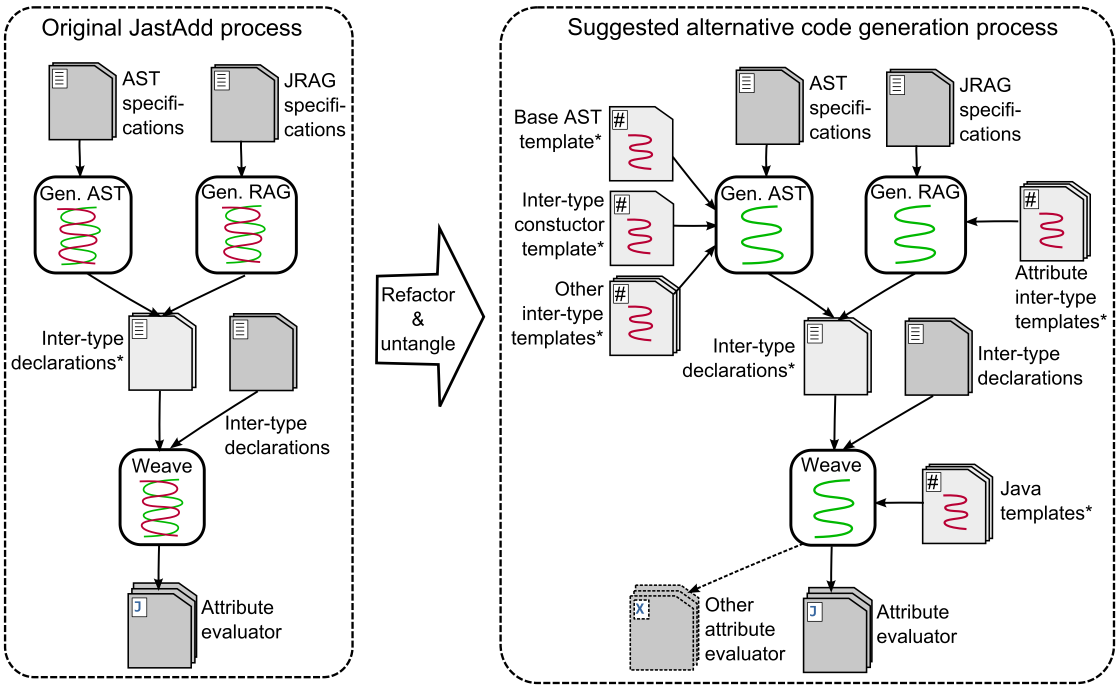


Figure 7.8.: A high-level view on the JastAdd2 code-generation process (version R20110506) and a suggested improvement using text-based templates. Tool-internal specifications and templates are marked with an extra asterisk (*).

As shown in Figure 7.8 on the left, the JastAdd generator works in multiple phases. It can be logically distinguished between three subprocesses. In the “Generate AST” phase, it reads, adds the built-in AST classes and validates the set of input specifications. If successful, the AST grammar is traversed, and an intermediate representation is generated, which is based on static aspects with inter-type declarations. Afterwards, in the “Generate RAG” phase the attribute declarations and specifications are validated, and also translated into inter-type declarations. Finally, the static aspect weaver is invoked to weave the intermediate aspects into Java compilation units, which are the actual implementation of the RAG evaluator. While the high-level logical view on the code-generation process seems reasonable, the implementation behind seems not. With the tool lifespan of over a decade now, several additions to the original RAG feature set have been made (e.g., [Magnusson 2007; Söderberg 2012]) and many user requests have been considered in the implementation. In particular, inconsistent coding styles and the massive tangling of string-building code with the code-generation logic make it hard to read, understand and maintain the implementation, especially if larger changes—like an integration with the EMF—shall be realized. Therefore, after an in-depth study of the backend, it was suggested to improve the code generator by refactoring and untangling the code using a template engine. A

```
...
public String ASTDecl.buildingConstructor() {
    ... // (initialization)
  s.append(" // Declared in " + getFileName() + " line " + getStartLine() + "\n");
  s.append(" public #ID#.#ID#(");
    ... // (append constructor parameters)
  s.append(") {\n");
    ... // (append constructor body)
  if( ... /* condition */) { ...
     s.append("    is$Final(true);\n");
  }
  s.append(" }\n\n");
  return s.toString().replaceAll("#ID#", name());
}
...
public void ASTDecl.jjtGenConstructor(...) {
    ... // (initialization)
  finalInit = "   is$Final(true);\n";
    ...
  s = "  public #ID#.#ID#() {\n" +
  "      super();\n" +
  "#NTA#";
   ... // (append parts of constructor body)
  s = s.replaceAll("#ID#", name());
  s = s.replaceAll("#NTA#", t);
   ... // (append parts of constructor body)
  stream.println(finalInit);
  stream.println(" }\n");
   ...
}
...
```

Listing 7.3: Constructor-building code slice from the JastAdd2 code generator (from production code of release R20110506; distributed under New BSD License).

logical view on the improved process is shown on the right side of Figure 7.8. In this process, large parts of the code emitted by the generator via string-concatenation operations are moved to generator-internal templates which are loaded and parametrized using an invasive composition system. These templates provide a holistic view on the implementation semantics of the generated RAG systems and allow JastAdd developers to read, and better understand, extend and maintain the code. The examples in this section are a small excerpt from the changes that have been prototyped to experiment with the untangling of the backend and the possibility of making it better exchangeable.[2]

Listing 7.3 shows a small excerpt from the JastAdd code generator emitting intermediate code for the default constructors of AST nodes. While both methods in the listing are contained in the same source file and both of them emit constructor code which is placed next to each other in the final Java files, there are approximately 700 lines between them. Markup for slots can be found

[2]The results of the experiments have been presented to the JastAdd team during an internal workshop in January 2013. Some ideas converged into the current 2.1.x releases of the tool.

```
// Defining the qualified module name.
module org.skat.isc.minimal.StplParser();

// Declare the import of the base grammar and parametrization with this grammar.
modify parser.MinISCBase(org.skat.isc.minimal.StplParser);

// Defining the JastAdd specific delimiters.
String SlotDelimiter = "#";
String Delimiters = SlotDelimiter;
String EscapeDelimiter = '\\';
```

Listing 7.4: The STpL island grammar module that extends the SkAT/Minimal base grammar of Listing 7.2.

in Lines 5, 20 (`#ID#`) and 22 (`#NTA#`). The bindings occur at the end of the methods via calls to the standard replacement API of Java strings (cf. Lines 13, 24 and 25). Besides these, there are other more implicit slots and bindings that do not follow the above pattern. In Line 4 and Line 27 slots are "bound" by direct string insertion. Obviously, it would be easy to also use slot markup in these places, e.g., `#FILENAME#`, `#LINE#`, `#FINIT#`. A full ISC system such as SkAT4J can help to improve the readability of the code before it is emitted. Simultaneously, it can increase the quality of the code generator by finding and avoiding syntax errors in relation with their causing composition steps instead of relying on compiler messages. The string-based composition of fragments in Listing 7.3 could hence be replaced by a fragment composition system to improve the quality of the generator. Code which is adjacent in the output would then also be adjacent in the input fragments, while the control flow in the code generators stays similar by converting the string-emitting code into composition programs.

However, as already sketched in the introduction of this section, a dependency of full ISC comes at the downside of a costly language and technology-dependent implementation. Considering the example of Listing 7.3, SkAT4J cannot be used directly because the generated intermediate code is specific to the language of the aspect weaver.[3] Instead, the STpL uses SkAT/Minimal to mimic the code generator as an unsafe fragment composition system and provides a string-based composition API. Listing 7.4 shows the STpL-specific parametrization of the SkAT/Minimal island grammar which declares the hash mark as slot delimiter and the backslash as escape symbol.

Besides that, STpL reuses SkAT/Minimal in an internal programming library for ISC-based string composition. Listing 7.5 shows a part of the STpL-based generic JastAdd aspect containing the constructor template with arguments. Listing 7.6 contains an exemplary composition result for the `Person` concept of BAF example. The corresponding STpL composition program can be inspected in Listing 7.7. In Line 5, the template API is initialized with the contents of Listing 7.5. In Lines 6–10, the island slots are bound via calls to the API. Internally, the SkAT/Minimal

[3]Of course, it would be possible to extend SkAT4J to support the aspect syntax. Moreover, since the aspect weaver of JastAdd actually is a fragment composition system, it would also be possible to use SkAT4J for implementing the weavings. However, these ideas have been postponed because the implementation and restructuring efforts have been considered too high for an initial investigation.

```
aspect #ID#Constructors {
   ...
   // Declared in #FILENAME# line #LINE#
   /**
    * Constructor for #ID#-ASTNodes.
    */
   public #ID#.#ID#( ... /* params */) {
      ... // (constructor body )
      #FINIT#
   }
}
```

Listing 7.5: Excerpt from the STpL template for JastAdd constructors.

```
aspect PersonConstructors {
   ...
   // Declared in BAF.ast line 42
   /**
    * Constructor for Person-ASTNodes.
    */
   public Person.Person( ... ) {
      ... // (constructor body )
      is$Final(true);
   }
}
```

Listing 7.6: Parametrized intermediate aspect derived from Listing 7.5.

```
...
public String ASTDecl.buildingConstructor() {
   ... // (initialization)
   // Initializing STpL API with content string:
   STPLTemplate tpl = new STPLTemplate("http://unique.ident",content);
   tpl.bind("ID", name());
   tpl.bind("FILENAME",getFileName());
   tpl.bind("LINE",String.valueOf(getStartLine()));
   if( ... /* assume condition = true */ )
      tpl.bind("FINIT","is$Final(true);");
   ... // (handle constructor body)
   // Emitting the generated code:
   return tpl.getGenCode();
}
...
public void ASTDecl.jjtGenConstructor(...) { ... /* further parametrizations */ }
...
```

Listing 7.7: STpL-based composition program which replaces the plain string concatenation-based fragment composition of `buildingConstructor` in Listing 7.3. If the STpL template of Listing 7.5 is the input fragment, Listing 7.6 appears as the corresponding output.

attributes are used to perform the slot look-up and to invoke the composers with the according parameters.

While the STpL provides sufficient support to parametrize text fragments, it lacks some support for modeling code variants which are embedded in the input fragments. The next subsection discusses an extension of the STpL which provides declarations of code variants and a variant selection mechanism via an extended composition API.

7.3.2. VTpL: A Text-Based Variability Template Language

Similar to normal programming languages, template engines typically provide constructs for branching and iteration for conditional (depending on some Boolean expression) and iterative (over a collection of similar parameter objects) fragment-code expansion. In passive template

```
1  public String ASTDecl.buildingConstructor() {
2    ... // appending constructor parameters:
3    for(Iterator iter = getComponents(); iter.hasNext(); ) {
4      Components c = (Components)iter.next();
5      if(!c.isNTA()) { //if not a nonterminal attribute (NTA)
6        if(i != 0) s.append(", ");
7        s.append(c.constrParmType() + " p" + i);
8        i++;
9      }
10   }
11   ... // appending constructor body:
12   for(Iterator iter = getComponents(); iter.hasNext(); ) {
13     Components c = (Components)iter.next();
14     if(!c.isNTA()) {
15       if(c instanceof TokenComponent) {
16         TokenComponent t = (TokenComponent)c;
17         s.append("    set" + t.getTokenId().getID() + "(p" + i + ");\n");
18       }
19       else {
20         s.append("    setChild(p" + String.valueOf(i) + ", " + j + ");\n");
21         j++;
22       }
23       i++;
24     }
25     ... //NTA variants of initialization (3 more)
26   }
27 }
```

Listing 7.8: Another code slice from the JastAdd2 code generator (from production code of release R20110506; distributed under New BSD License).

languages without such "active" concepts, branching and iteration have to be expressed with the means of the host language (e.g., *if-then-else* or *for-loop* statements). However, as the STpL only has support for slots, the variants would still have to be constructed via string operations or loaded from file, and then bound to slots. The VTpL as an extension to the STpL provides two additional constructs. *Variant* markup allows users to declare named in-line variants of source fragments that can be selected by clients. Complementarily, the *prototype* concept is introduced to mark up subfragments in the template that may be parametrized and instantiated arbitrarily.

As before, an example from the JastAdd code generator is used to document how prototypes and variants are realized using string-concatenation operations and control-flow concepts of the programming language. Focusing on the different strategies on handling slots in the code generator, Listing 7.8 contains constructor-generating code that first was excluded from the original Listing 7.3. The skipped parts emit the parameters and the actual body of the constructor. In Listing 7.8 they can be found in Lines 3–10 and Lines 12–26. For parameters, the generator iterates over the components of a node declaration (i.e., the right-hand side of a production in the AST grammar) and appends a parameter declaration to the parameter list in the case it is not a higher-order nonterminal attribute (NTA). During this process, a fragment representing parameter declarations (cf. Line 7) is instantiated and parametrized repeatedly and thus can be considered a prototype in the above terminology. Using STpL, the fragment can be rewritten to `"#ARGTYPE# p#IDX#"`. For the constructor's body, the generator also iterates the components. For different types of components, different variants of a fragment for setting the parameter values are emitted. Concrete non-NTA variants can be found in Lines 17 and 20. Using STpL, they can be rewritten to `"set#ARG#(p#IDX#);"` and `"setChild(p#IDX#,#IDX#);"`. Since each variant can be instantiated multiple times, they can also be considered as prototypes.

To support `VariantLists` and `Prototypes` in the composition system, the G_{MinCM} is extended by adding the corresponding nonterminals as `Compositionals` to the component model (an interested reader may find the full grammar in Appendix A.5.1):

```
import G_MinCM
VariantList▷Compositional ::=
          variants:Element* activeName:<string>?
Variant▷Compositional ::= content:Element*
Prototype▷Compositional ::= content:Element*
```

Thus, a `VariantList` has a list of `Variant` nodes as children, which themselves are named composites of island and water `Elements`. Similarly, `Prototypes` are named composites of water and island `Elements`. The implementation of those compositional concepts is realized as a direct extension to SkAT/Minimal and the STpL. Figure 7.9 gives a general overview of the

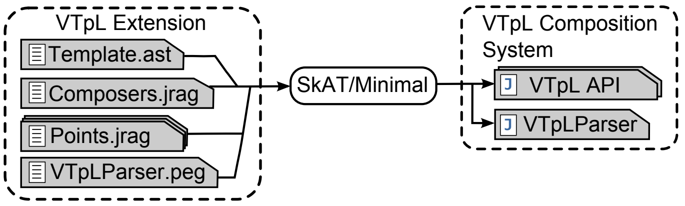


Figure 7.9.: Specification-setup and generation of the VTpL composition system.

involved SkAT specifications and the composition-system generation based on SkAT/Minimal. Basically, it adds four specification modules. The corresponding prototype and variant-AST definitions are contained in the `Template.ast` grammar (can be inspected in Appendix A.5, Listing A.21). Like in the STpL case, the textual representation is specified as a PEG that imports and extends the minimal island PEG of SkAT/Minimal. `Points.jrag` and `Composer.jadd` specify the semantics of prototypes and variants as compositional points and composers. Furthermore, some VTpL-specific extensions are added to the composition API. To generate the composition system, the SkAT/Minimal generator weaves the VTpL specifications and its own specifications (cf. also Figure 7.7), emits and compiles the Java-based composition-system implementation. In the following, relevant parts of the involved specifications are investigated.

```
 1 module org.skat.binding.vtpl.VtplParser();
 2 modify parser.MinISCBase(org.skat.binding.vtpl.VtplParser);
 3
 4 // Delimiters for the complex variants, prototype constructs and StPL slots.
 5 String LeftDelimiter = "[[";
 6 String RightDelimiter = "]]";
 7 String InfixDelimiter = "::";
 8 String SlotDelimiter = "#";
 9 String EscapeDelimiter = "\\" ;
10 String Delimiters = SlotDelimiter/LeftDelimiter/RightDelimiter;
11
12 // Overriding the imported island production to support the new constructs.
13 IslandElement Island :=
14    yyValue:VariantListDecl / yyValue:PrototypeDecl / yyValue:SlotDecl { };
15
16 // Syntax of VariantLists and single Variant objects.
17 String Variants_Open =
18    LeftDelimiter "VARIANTS" InfixDelimiter yyValue:Identifier RightDelimiter;
19 String Variants_Close = LeftDelimiter "VARIANTS" RightDelimiter;
20 VariantList VariantListDecl = name:Variants_Open
21    content:(yyValue:VariantDecl / yyValue:TextElement)* Variants_Close {};
22 String Var_Open =
23    LeftDelimiter "VAR" InfixDelimiter yyValue:Identifier RightDelimiter;
24 Variant VariantDecl = name:Var_Open content:TplElements {};
25
26 // Syntax of Prototypes.
27 String Prototype_Open =
28    LeftDelimiter "PROTOTYPE" InfixDelimiter yyValue:Identifier RightDelimiter;
29 String Prototype_Close = LeftDelimiter "PROTOTYPE" RightDelimiter;
30 Prototype PrototypeDecl =
31       name:Prototype_Open content:TplElements Prototype_Close {};
```

Listing 7.9: The VTpL island grammar, except AST construction.

7.3.3. In-Place Composers in VTpL

VTpL's PEG extension can be inspected in Listing 7.9. The slot delimiters are equivalent to those of the STpL.[4] For the new constructs, different delimiters have been chosen since using the same delimiter would make it harder to disambiguate the grammar, and to make them clearly distinguishable for users when looking at the code. Hence, double square brackets—`[[` and `]]`—are used as markup of variants and prototypes, and within the markup double colons—`::`—are employed to separate different parts of declarations. According to their definition in Lines 17–24, variant lists are surrounded by `[[VARIANTS::listName]]` as opening tag and `[[VARIANTS]]` as closing tag. Single variants are tagged by `[[VAR::varName]]`. Similarly, according to Lines 27–31 prototypes are delimited with `[[PROTOTYPE::protName]]` as opening tag and `[[PROTOTYPE]]` as closing tag. VTpL permits an arbitrary nesting of variant lists and prototypes. Hence, a prototype may contain a list of variants and a variant may contain a prototype. At each nesting level, slots may be defined and addressed.

[4]It is also possible to reuse and modify the STpL's PEG module.

```
aspect VTPLPoints{
   eq Prototype.isSlot() = true;
   eq Prototype.slotName() = getName();
   eq VariantList.isSlot() = true;
   eq VariantList.slotName() = getName();
}
```

Listing 7.10: `Slot.jrag` specification of the VTpL.

Conceptually, variant lists and prototypes are slots with *primitive in-place composer* semantics (cf. Section 5.2.2). In fact, extracting a variant from a variant list or instantiating a prototype “means” binding the respective points with a fragment contained in the subtree of themselves. Therefore, RAG modules for these points and composers are required to specify their properties accordingly. Listing 7.10 shows the JastAdd RAG module which declares `VariantList` and `Prototype` as slots so that they can be bound to island grammar `Elements`—similar to the basic STpL `Slots`.

The complementing composer RAG module can be inspected in Listing 7.11. Lines 3–6 for the `Prototype` nonterminal and Lines 18–20 for the `VariantList` nonterminal determine the composer properties. The `isBind()` equation identifies instances of both nonterminals as a bind composer. Furthermore, the instances of the `targetPoint()` attribute—linking composers to points—and the `composer()` attribute—referring from points to composers—point to the same `VariantList` or `Prototype` node if they are in the same context. Hence, the compositional point where the composer should be applied is the composer definition itself (in-place composer semantics). Derived from these declarations, at composition time, the SkAT composition engine automatically determines the correct composition modes and executes variant extraction and prototype instantiation as in-place bind composers. The replacing fragment is retrieved via the `srcFragment()` attribute (cf. Line 9 and 23 in Listing 7.11). In the case of `VariantList`, `srcFragment()` computes a reference to the `Element` nodes constituting the active variant, which realizes an *xor* semantics for variant extraction. An active variant is determined by a composer argument that can be passed to the `VariantList` node from a composition program via the VTpL composition API (cf. Line 24). For `Prototypes`, `srcFragment()` produces a fresh copy of its contained `Element` nodes which is optionally parametrized by the slot bindings given as optional argument model (cf. Line 11). In contrast to the extraction of variants, prototype instantiation copies and retains the `Prototype` fragment so that it can be instantiated arbitrarily often until it is finally removed from its parent list node by an according statement in the composition program.

Additionally, attributes for the look-up of variants and prototypes, which simply use the SkAT point look-up collection attributes, are specified in additional RAG modules. The VTpL tooling also comprises a composition facade as an extension to the basic SkAT/Minimal API that provides additional convenience methods for the newly introduced constructs. The API is used in the VTpL-based code generator presented in the next section.

```
aspect VTPLComposers{
   // Declaring Prototype objects as bind composers.
   eq Prototype.isBind() = true;
   eq Prototype.isBindRetain() = true;
   eq Prototype.targetPoint() = this;
   eq Prototype.composer() = this;

   // Prototype source fragment derivation.
   eq Prototype.srcFragment() {
      List<Element> content = getContentList().fullCopy();
      Object model = getOptComposerArg();
      if(model!=null){
         content.bindEach((Map<String,String>)model);
      }
      return content;}

   // Declaring VariantList objects as bind composers.
   eq VariantList.isBind() = true;
   eq VariantList.targetPoint() = this;
   eq VariantList.composer() = this;

   // Selecting inner variant as source fragment.
   eq VariantList.srcFragment() {
      Variant activeVariant = findVariantLocal(getOptComposerArg().toString());
      if(activeVariant!=null)
         return activeVariant.getContentList();
      return null;}
}
```

Listing 7.11: `Composers.jrag` specification of the VTpL.

```
aspect PersonConstructors{
   ...
   public Person.Person([[PROTOTYPE::ARGLIST]]#ARGTYPE# p#IDX#[[PROTOTYPE]]){
   [[PROTOTYPE::SETCHILDVARIANTS]]
      [[VARIANTS::SETCHILD]]
      [[VAR::TERMINAL]] set#ARG#(p#IDX#);
      [[VAR::NONTERMINAL]] setChild(p#IDX#,#IDX#);
      [[VAR::NTA1]] setChild(new #CONS#(), #IDX#);
      [[VAR::NTA2]] setChild(null, #IDX#);
      [[VARIANTS]]
   [[PROTOTYPE]]
      is$Final(true);
   }
}
```

Listing 7.12: Excerpt from the VTpL template for JastAdd constructors complementing the plain STpL template in Listing 7.5.

```
aspect PersonConstructors{
   ...
   public Person.Person(Name p0, int p1){
      setChild(p0,0);
      setage(p1);
      is$Final(true);
   }
}
```

Listing 7.13: Instantiated intermediate JastAdd aspect derived from Listing 7.12 (formatted).

7.3.4. Applying VTpL and STpL to JastAdd

As in the STpL example of the previous section, the VTpL composition system can be applied to the JastAdd code-generation scenario. The above Listing 7.12 shows another intermediate snapshot of a fragment in the modified JastAdd code generator, after having bound the basic slots for the `Person` concept from the BAF example (cf. Listing 7.6). The example contains two prototypes. `ARGLIST` in Line 3 provides a fragment for constructor parameters. The fragment can be parametrized using two slots—`#ARGTYPE#` for the type, and `#IDX#` for parameter's index. The `SETCHILDVARIANTS` prototype spans Lines 4–11 and provides a variant list—`SETCHILD`. After an instantiation of the prototype, `SETCHILD` provides several variants to set the initial values of the node's children. Variant `TERMINAL` sets terminal values according to their specific setter-operation's name, which can be bound via the `#ARG#` slot, and the parameter index `#IDX#` to replicate its name in the constructor arguments. The second variant (`NONTERMINAL`) provides a fragment for setting nonterminal child nodes via the index-based `setChild` method of JastAdd. The remaining variants provide fragments to initialize NTA attributes with default values. Since NTAs are computed by later attribute evaluation, the defaults are typically `null` or an empty list container which has a specific constructor name to be parametrized via the `#CONS#` slot.

An exemplary instantiation is shown in Listing 7.13 whereas the corresponding VTpL-based part of the modified JastAdd code generator is shown in Listing 7.14. After initializing the API of the VTpL composition system in Line 4 and binding the global slots according to the STpL example (e.g., `#ID#`), the composition program traverses the child components of the current node declaration and uses the composition system to generate the component-specific arguments and body code of the constructor. Line 9 creates an instance of the `SETCHILDVARIANTS` prototype. It should be noted that `instantiatePrototype()` internally performs a look-up of `PROTOTYPE` node with the provided prototype name using the look-up attributes provided by the composition environment, and then uses the standard composer rewrites, which themselves use the attributes specified in Listing 7.11 to retrieve the argument fragments and points. Afterwards, the `#IDX#` slot of the instantiated `SETCHILD` variant list is bound to the current index value (cf. Line 11). The constructor arguments are instantiated in Line 14. However, in this case, the values to be bound to the internal slots are passed directly to the composer, which then handles the binding internally. According to the embedded Java syntax, single arguments must be

```
...
public String ASTDecl.buildingConstructor() {
   ... // (initialization)
   VTPLTemplate tpl = new VTPLTemplate("http://my.location",content);
   ... // (slot bindings)
   for(Iterator it = getComponents(); it.hasNext();){
      Components component = (Components) it.next();
      // Instantiate SETCHILD variant list:
      tpl.instantiatePrototype("SETCHILDVARIANTS");
      // Bind the IDX slots in SETCHILD:
      tpl.bind("SETCHILD.*.IDX",index)
      // For normal children instantiate and parametrize argument:
      if(!component.isNTA()){
         tpl.instantiatePrototype("ARGLIST",
            "ARGLIST.ARGTYPE",sep+component.constrParmType(),
            "ARGLIST.IDX",String.valueOf(index));
         sep = ", ";
      }
      // Extract the corresponding SETCHILD variants
      if(component instanceof TokenComponent){
         tpl.extractVariant("SETCHILD", "TERMINAL");
         tpl.bind("ARG",component.name());
      } else if(component instanceof AggregateComponents){
         tpl.extractVariant("SETCHILD", "NONTERMINAL");
      }
      ...// (further NTA cases)
   }
   ... // Removing prototypes and emitting the generated code:
   return tpl.getGenCode();
} ...
```

Listing 7.14: VTpL-based composition program that replaces the string-based `building-Constructor` in Listing 7.8. If the VTpL template of Listing 7.12 is the input fragment, Listing 7.13 appears as a corresponding output.

separated by comma. Since minimal ISC does not incorporate language-specific constructs, this information has to be encoded in the code generator, e.g., using an extra variable `sep`. Finally, the composition program selects and extracts from the `SETCHILD` variant list an adequate variant of the initialization fragment variants, depending on the kind of child component (cf. Lines 21 and 24). Similar to prototype instantiation, `extractVariant` performs a look-up of the referred `VariantList` and executes the composition by delegation to the encapsulated composition environment. Afterwards, the composition result from the VTpL template is further processed by the integrated JastAdd aspect weaver.

To conclude the discussions on VTpL and STpL as examples for template languages without control flow, potentials and restrictions of the approach which were identified during the development of the case studies are discussed subsequently.

- In general, it can be discovered that the usage of the VTpL does not necessarily reduce the code size when comparing number of code lines in the respective Listings before and after VTpL is employed. However, subjectively the code looks cleaner and better structured due to the compositional abstractions, which make it more readable for developers who are not yet familiar with the code.

- Furthermore, it does not always seem ideal to put every fragment into a variant list or prototype. For instance, in the previous example, the varying setter calls of the constructor body could also be managed as separate fragment components from within the composition program and binding them (in retain mode) to a constructor-body slot, or extending a body hook if a full-fledged component model was available. Hence, for the success of VTpL—and fragment composition systems with embedded composers in general—the right reuse mechanisms and abstractions have to be found.

- As the VTpL composer markup is explicitly available as an AST structure, it would also be an option to use other approaches to specify compositions or configurations such as feature diagrams [Kang et al. 1990]—a formalism to specify valid variants and products of a software product line [Apel et al. 2008]. An instance of a feature tree would then specify which variants and prototypes need to be instantiated. However, since syntactic and semantic correctness of the composed programs is also a major issue in software product line research, it would be valuable to investigate lifting VTpL to language-specific implementations using well-formed ISC, and use fragment contracts and RAGs in general to determine configuration-correctness beforehand. Section 7.5 will discuss possibilities of extending and improving minimal ISC systems with better support for the actual language syntax and semantics.

- Depending on the actual fragment language, using language-unaware delimiters such as square brackets disturbs services of the language tooling, e.g., textual editors with language analysis functionality that are not aware of the component extension. Assuming a robust service implementation, if delimiters are chosen carefully so that the services are less disturbed, the existing language tooling can still be used to some extent. A common approach to achieve this is to mimic language extensions as comments. For example, consider the Open specifications for Multi Processing (OpenMP) API [OpenMP Architecture Review Board 2013] for the Fortran [ISO/IEC 1991] programming language. Fortran uses "`!`" as delimiter for comments. OpenMP embeds its commands into comments using a specialized prefix "`!$omp ...`". In Appendix A.5.2 of this thesis, a similar approach for the VTpL's syntax is sketched.

To complete the series of case studies for minimal ISC, the following section discusses the example of an *active* preprocessor language with its own control flow.

7.4. A Universal Extensible Preprocessor

The CPP is one of the most used macro languages as it is an integral part of the C language specification [ISO/IEC 1999; Harbison III and Steele 2002]. It is typically bundled with the compiler, but actually a self-contained tool that transforms the input source in a *preprocessing* phase before the actual compilation. During that phase, the CPP line by line, starting with the first line, scans the source file for preprocesser directives prefixed by the `#` delimiter. If a directive is found, it is immediately interpreted by the preprocessor. For example, `#include` is replaced by the content of the included file, `#define` defines a macro which can be called at all subsequent positions, and `#ifdef` can check if a macro is defined before it is used and therefore enables conditional compilation. In contrast to the compiler, which works on the syntax tree, the CPP operates at the token level, recognizing standard C tokens (e.g., literals) and special CPP tokens [Harbison III and Steele 2002]. Hence, the CPP is usable as a template engine for arbitrary target languages with C-like tokens.

In this section, it will be shown that the approach of minimal ISC can be used as an implementation technique to create a CPP-like macro engine—the *Universal Extensible Preprocessor (UPP)*. On the one hand, this demonstrates that the approach is capable of modeling real-world template and macro languages. On the other hand, the UPP also provides some advantages over the CPP. Due to the minimal ISC approach, it has an extensible component model so that new island elements can be added by providing an according RAG specification. Furthermore, the syntax of preprocessor directives can be adopted to the look and feel of the target language.

7.4.1. Preprocessor Directives as Points and Composers

From the perspective of a composition-system developer, most of the directives in a preprocessor template have a dual semantics as points and composers. All are compositional points where fragments should be bound to a slot or rudiments should be extracted. Example 7.7 illustrates this for the `#include` directive.

Example 7.7 (`#include` as an STpL composition program).
Consider the C program below which contains a `main` implementation with a call to `foo()`. The implementation of `foo()` shall be provided by including it from an external file.

```
#include "fooimpl.bar"
int main(void)
{
   foo();
}
```

Listing 7.15: A simple C program with include directive.

If the file with the implementation above is processed by the preprocessor, the `#include` directive in the first line is just replaced by the contents of `fooimpl.bar`. With standard ISC concepts, this can be reformulated as an STpL program:

```
public void fooInclude() {
   STPLTemplate tpl = new STPLTemplate(...);
   tpl.bind("fooimpl", load("fooimpl.bar"));
}
```

Listing 7.16: Simple STpL include code.

Note that the original `#include` must be replaced as `#fooimpl#` slot to make the STpL program work. ◇

In the following overview, the most important directives supported by UPP are described in short. To characterize the directives in ISC terminology, a 3-tuple $CHAR = (P, C, S)$ will be given with $P \in \{slot, hook, rudiment, \perp\}$, $C \in \{bind, extend, extract, \perp\}$ and S a side effect with $S \in \{fragment, \neg fragment, other, \perp\}$ where $fragment$ means "a fragment is declared".

#include inserts text from an external file that may also contain preprocessor directives. ISC characteristic: $(slot, bind, \perp)$.

#define declares a macro with a name and optional macro parameters. From the ISC perspective, a macro directive declares a fragment in the composition environment that may contain slots and may be instantiated arbitrarily often by macro calls directives. ISC characteristic: $(rudiment, \perp, fragment)$.

#undef invalidates a previously declared macro so that it cannot be instantiated subsequently. ISC characteristic: $(rudiment, \perp, fragment)$.

#ifdef–#endif Depending on a certain macro being defined (visible) at that point, the directive is replaced by its inner content. in case the macro is not defined. ISC characteristic, if the macro is declared: $(slot, bind, \perp)$. Otherwise: $(rudiment, extract, \perp)$.

#ifndef–#endif tests for the absence of a macro definition. ISC characteristic, if the macro is not declared (visible): $(\mathit{slot}, \mathit{bind}, \perp)$. Otherwise: $(\mathit{rudiment}, \mathit{extract}, \perp)$.

#if–#else–#endif tests a given logical expression and, depending on the result, replaces the directive by the fragment contained in the `#if` part or the `#else` part. If an else is present or the expression is evaluated to `true`, ISC characteristic: $(\mathit{slot}, \mathit{bind}, \perp)$. Otherwise: $(\mathit{rudiment}, \mathit{extract}, \perp)$.

#error prints a message to standard I/O. ISC characteristic: $(\mathit{rudiment}, \mathit{extract}, \mathit{other})$

macro-call is expanded by creating a copy of the referenced fragment declaration and instantiating it by providing parameters as required by the macro. The syntactical representation of macro calls overlaps with that of C identifiers (*object-like* macros without parameters) and method calls (*function-like* macros with parameters). If no macro declaration with the same identifier or method signature is visible, the preprocessor ignores the call. ISC characteristic if a suitable declaration is visible: $(\mathit{slot}, \mathit{bind}, \perp)$. Otherwise: $(\perp, \perp, \perp)$.

For a complete and detailed overview of the directives supported by the CPP and exemplary macros it is kindly referred to standard literature on C programming (e.g., [Harbison III and Steele 2002]) or the C standardization document [ISO/IEC 1999].

Considering the generalized composers in Section 5.2.2, `#ifdef`, `#ifndef`, `#if` as well as `#include` are *primitive in-place composers* as they replace themselves with their owned fragment without any additional composition step. `#error` also is a primitive in-place composer as it removes itself from a fragment. Finally, macro calls are *local in-place composers* since they replace themselves with the fragment of a referenced macro definition and bind slots internally.

7.4.2. The Universal Preprocessor's Component Model

Similar to the VTpL, the UPP RAG specifications extend the SkAT/Minimal core. This is sketched in Figure 7.10. The UPP model covers seven specifications. The AST specifications `Template.ast` and `Expression.ast` provide the island nonterminals that are added to the minimal FCM for modeling the above described preprocessor directives and the expressions that can be used in `#if` directives. Concrete syntax is again specified as a PEG extension to the basic STpL slot-markup grammar. The ISC characteristics of the directives are specified in the `Points.jrag` and `Composer.jrag` attribute-grammar modules. An evaluator and type checker for the supported expressions is provided by the `Expression.jrag` module. Finally, a scope-based name analysis for associating macro-calls with macro declarations is contained in the `MacroLookUp.jrag` module. After generating the implementation with SkAT/Minimal, the UPP composition system consists of the basic JastAdd-generated AST classes and the composition-environment API and the UPP parser. The generated implementation classes are complemented by a `UPPProcessor` component that encapsulates the composition system and provides an entry point for clients to access and use the UPP macro-processing system. In the following, relevant parts of the UPP specifications will be discussed

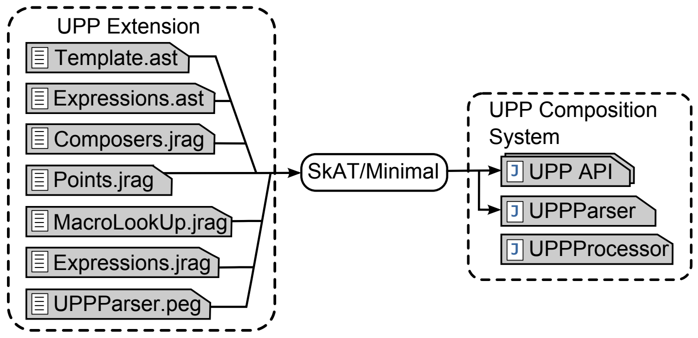


Figure 7.10.: Specification-setup and generation of the UPP composition system.

in more detail. This includes `Template.ast`, `Composer.jrag` and `Points.jrag`. The UPP PEG (`UPPParser.peg`), the macro name analysis (`MacroLookUp.jrag`) and the `Expression.ast` are included in the Appendix A.5.3, while the expression-related attributes are omitted as they are not relevant for understanding this chapter.

The UPP AST specification is shown in Listing 7.17. As to be expected from the preprocessor-directives overview of the previous section, each directive must be represented by a corresponding nonterminal. For convenience, the directive nonterminals are declared as subtypes of `Compositional` so that they are recognized as compositional island concepts by the SkAT system. The CPP does not have such an explicit AST structure because it is tightly coupled with the parser and directly works on the token stream when it is read. Hence, using ISC at least doubles the processing time (the AST has to be created and traversed by the UPP processor) and involves a larger memory footprint.

Concerning supported features, there are subtle differences to the CPP. The UPP supports macro declarations spanning multiple lines of source code (cf. `MacroDeclSL` and `MacroDeclML` in Lines 6 and 7) whereas the CPP only supports single-line macros. Multi-lines can only be mimicked by using the backslashes that escape the line-break symbol(s) from the token stream. Macro references from `MacroCall`s and `UnDefine`s to `MacroDecl` nodes are represented by the `MacroRef` nonterminal. Interested readers can inspect the JastAdd module specifying the name analysis in Appendix A.5.3, Listing A.22. `MacroCall`s can have an arbitrary number of comma-separated arguments which are meant to be bound to the parameters of the referenced `MacroDecl`. UPP has three node types for call arguments (cf. Lines 13–15). `PlainArg`s hold a single string-valued token that has been parsed by the UPP parser. `MultiArg`s can contain a collection of island and water elements being an UPP template themselves. In contrast to typical CPP implementations, `ExpArg`s provide the value of their `Expression` child as an argument. Hence, in UPP templates, users are enabled to write logical and arithmetic expressions over constants, macro declarations and system values. How this works will be exemplified later in this section. The same expressions used for `MacroCall`s are also used to specify the conditions of `#if` directives (cf. `IfCondition`, Line 26).

```
// #include
Include:Compositional;

// #define
abstract MacroDecl:Compositional ::= Parameters:MacroParam* Content:Element*;
MacroDeclSL:MacroDecl;
MacroDeclML:MacroDecl;
MacroParam ::= <Name:String>;

// macro-call with optional arguments
MacroCall:Compositional ::= Reference:MacroRef Args:MacroCallArg*;
abstract MacroCallArg;
PlainArg:MacroCallArg ::= <Value:String>;
MultiArg:MacroCallArg ::= Values:Element*;
ExpArg:MacroCallArg ::= Expression:Expression;

// macro references
MacroRef ::= <Name:String>;

// #undef
UnDefine:Compositional ::= Reference:MacroRef;

// #IfDef, #Ifndef, #if
IfDef:Compositional ::= Reference:MacroRef Content:Element*;
IfNotDef:IfDef;
IfCondition:Compositional ::= Condition:Expression Then:Element* Else:Element*;

// #error
Error:Compositional ::= <Message:String>;
```

Listing 7.17: UPP's JastAdd AST grammar `Template.ast`.

For generating the API, SkAT/Minimal weaves `Template.ast` with `Expression.ast` and the minimal FCM AST grammar—`Minimal.ast`, and uses JastAdd to generate the basic API classes of the composition system. The woven API class hierarchy is shown in Figure 7.11 as an UML class diagram using different colors to highlight parts of classes originating from the involved AST specifications. Derived from the woven AST grammar, all associations—except `macro`—are containment relations that characterize the spanning tree of instance-object graphs. The `macro` association is computed by the RAG via the name analysis attributes in the `MacroLookUp.jrag` specification. Similar to the previous case studies, the attributes that define the sets of point candidates are contained in the `Points.jrag` module, which is shown in Listing 7.18. According to the ISC characteristics of preprocessor directives developed in Section 7.4.1, the set of slot candidates is determined by the `isSlot` attribute in Lines 3–7. Hence, it contains the `#include`, `#ifdef`, `#ifndef` and `#if` directives, and macro calls. Furthermore, slots are supported by reusing the minimal FCM and the STpL base island-grammar. The set of rudiments (i.e., removable points) is specified between Lines 10 and 12. It contains the `#error`, `#define` and `#undef` directives. Additionally, it contains all slot candidates as these are—by default—rudiments which may be removed if they where not bound and the fragment would be valid after removal. Finally, the point names are specified in Line 15.

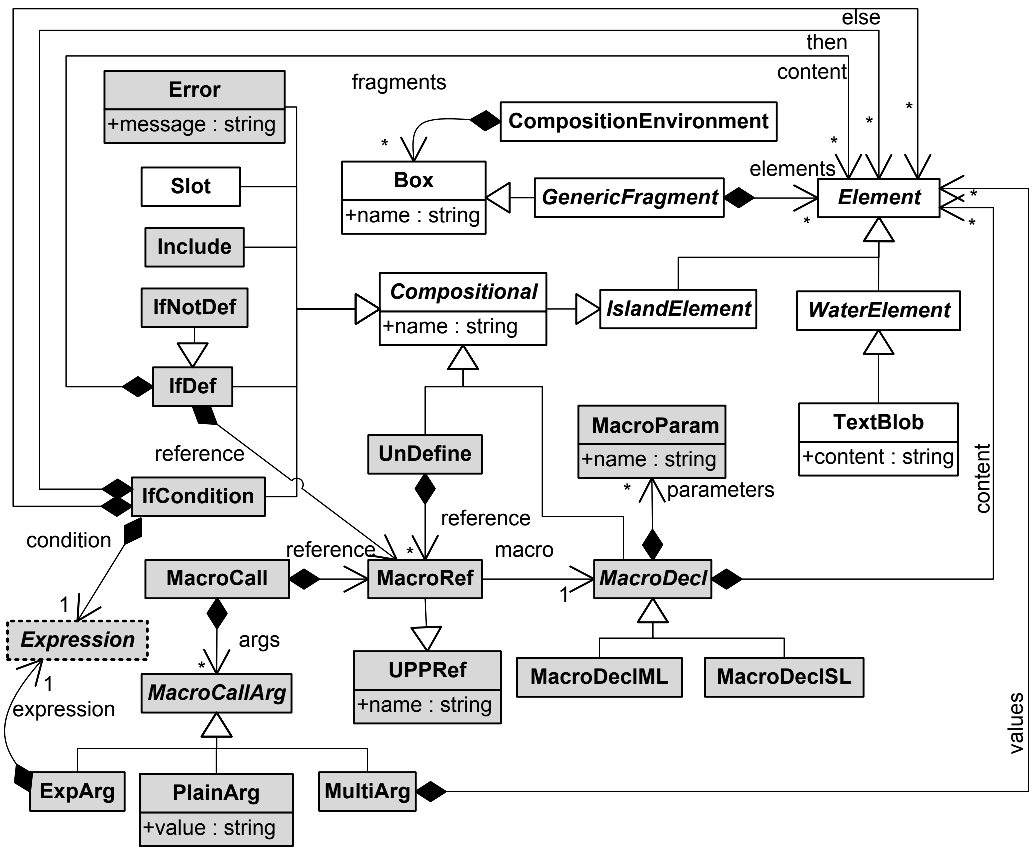


Figure 7.11.: Generated AST classes of the UPP in UML class-diagram notation. White concepts stem from the minimal FCM (from `Minimal.ast`), gray marks compositionals of the UPP (from `Template.ast`), dashed lines mark the common `Expression` class (from `Expression.ast`; expression nonterminals have been excluded).

7.4.3. Composers and Composition System

With the specifications discussed in the previous sections, it is possible to generate a SkAT-based composition API. Similar to the STpL and VTpL examples, programmers of client applications would be able to write composition recipes as Java programs against that API. However, in contrast to these examples, each UPP fragment is also a composition program itself—with its inherent semantics and control flow. The composition-program semantics of UPP is realized using the composer attributes of SkAT (cf. Section 5.2.1) and by implementing a simple interpreter on top of these attributes and the composition API. In the original implementation, the attributes have been put into the `Composers.jrag` specification. For convenience reasons, the specification has been split into several chunks that are documented in the remainder of this section.

```
aspect UPPPoints {
   // identifying slots
   eq Include.isSlot() = true;
   eq IfDef.isSlot() = true;
   eq IfNotDef.isSlot() = true;
   eq IfCondition.isSlot() = true;
   eq MacroCall.isSlot() = true;

   // identifying rudiments
   eq Error.isRudiment() = true;
   eq MacroDecl.isRudiment() = true;
   eq UnDefine.isRudiment() = true;

   // determining point names
   eq Compositional.pointName() = getName();
}
```

Listing 7.18: The RAG specification `Points.jrag` determining the compositional points of the UPP.

```
 1  aspect IncludeComposer {
 2     syn GenericFragment Include.associatedFragment();
 3
 4     eq Include.isBind() = true;
 5     eq Include.targetPoint() = this;
 6     eq Include.composer() = this;
 7     eq Include.associatedFragment() = (GenericFragment)findFragment(getName());
 8     eq Include.srcFragment() {
 9        GenericFragment fragment = associatedFragment();
10        if(fragment!=null){
11           return fragment.getElementsList().fullCopy();
12        }
13        return null;
14     }
15  }
```

Listing 7.19: Specification of the include composer.

The specification of the `#include` directive is shown in Listing 7.19. As known from the previous sections, the default values of the composer predicates `isHook`, `isBind` and `isExtract` are false. To determine `Include` as a primitive in-place composer (cf. Section 5.2.2) with binding semantics in any context, `isBind` always has to evaluate *true*. This is specified in Line 4 of Listing 7.19. An `Include`-composer's target point is the composer node itself (cf. Line 5) which is also a slot as specified in the previous section in the `Points.jrag` specification. Complementary, the `composer` attribute also refers to the `Include` node, which is specified in Line 6. The fragment component to be bound in place of the node is computed via the `srcFragment` attribute between Lines 8–14, which is also a default attribute of composer nonterminals. The attribute delegates the look-up of the fragment component in the composition environment to the custom `associatedFragment` attribute. Hence, if UPP encounters a

```
 1  aspect ErrorComposer {
 2     eq Error.isExtract() = true;
 3     eq Error.targetPoint() = this;
 4     eq Error.composer() = this;
 5
 6     eq Error.hasComposerFurtherSideEffects() = true;
 7     public boolean Error.composeSideEffects() {
 8        println("upp> " + this.owningBox().getName() + ": " + this.getMessage());
 9        return true;
10     }
11  }
```

Listing 7.20: The error message composer.

directive like `#include "name"` in a preprocessed source file, the `name` token is used to search for the fragment in the environment. After a successful look-up, the fragment is cloned and inserted at the `Include`-node's position by the composition engine.

The properties of the `#error` directive are defined via the attributes presented in Listing 7.20. The basic composer-identification attributes can be found between Lines 2–4. In contrast to the `Include` composer, `Error` has an *extract characteristic* only, which means that the `isExtract` attribute always evaluates *true* whereas the other composer-identification attributes evaluate *false*. Hence, if the UPP composition engine encounters a directive such as `#error "msg"`, it is removed from the composition environment. After removing, the composer is requested to produce a side effect by emitting its message to the system's I/O (cf. Line 8). The specification of the side-effect behavior can be found in Lines 6–10, where the presence of side effects causing behavior is indicated by the `hasComposerFurtherSideEffects` attribute and the operation that produces the side effect—`composeSideEffects()`.

All variability-related directives of UPP are specified in the `VariabilityComposers` RAG module shown in Listing 7.21. The composer-identification attributes of `#ifdef` and its negated counterpart `#ifndef` are to be found in Lines 3–7 and Lines 10–14, respectively. For both, the type of composer—bind or extract—depends on the presence or absence of a macro definition within the current scope, i.e., the already preprocessed parts of the fragment previous to the directive. The actual look-up of the macro declaration is delegated to the `macro` attribute of the owned `Reference` node that also holds the symbolic name of the referenced macro declaration. Depending on if a macro object can be resolved or not, the `IfDef` (`IfNotDef`) node under evaluation is treated as bind (extract) or extract (bind) composer. Similar to the directives described before, this behavior is specified by the corresponding attributes `isBind` and `isExtract`. In the case UPP recognizes the directive as a bind composer, its contents are retrieved via the `srcFragment` attribute and used for replacement.

Attributes related to the composer definition of `#if` are contained in Lines 17–27. Like `#ifdef` and `#ifndef`, the `#if` directive can be interpreted as a bind or extract composer, depending on a complex condition which is provided as a logical expression by the user according to the expression model of the UPP. Depending on the expression's value, the `Then` (*true*) or

```
aspect VariabilityComposers {
   // #ifdef related attributes
   eq IfDef.isBind() = getReference().macro()!=null;
   eq IfDef.isExtract() = getReference().macro()==null;
   eq IfDef.targetPoint() = this;
   eq IfDef.composer() = this;
   eq IfDef.srcFragment() = getContentList();

   // #ifndef related attributes
   eq IfNotDef.isBind() = getReference().macro()==null;
   eq IfNotDef.isExtract() = getReference().macro()!=null;
   eq IfNotDef.targetPoint() = this;
   eq IfNotDef.composer() = this;
   eq IfNotDef.srcFragment() = getContentList();

   // #if-#else related attributes
   syn boolean IfCondition.isValid();
   syn boolean IfCondition.hasElse();
   eq IfCondition.isBind() = getCondition().value()==Boolean.TRUE||hasElse();
   eq IfCondition.isExtract() = !isBind();
   eq IfCondition.targetPoint() = this;
   eq IfCondition.composer() = this;
   eq IfCondition.srcFragment() =
      getCondition().value()==Boolean.TRUE?getThenList():getElseList();
   eq IfCondition.isValid() =
      getCondition().type()==Type.BOOL && getThenList().getNumChild()>0;
   eq IfCondition.hasElse() = getElseList().getNumChild()>0;
}
```

Listing 7.21: The specification of UPP composers with variability semantics.

`Else` (false) parts of the directive are used for replacing it and the `isBind` attribute evaluates accordingly. In the case of the directive having no `Else` part specified and the provided expression turns out to be false, `isBind` turns `false` while `isExtract` evaluates to *true*. For checking purposes, `IfCondition` has a validating attribute. `isValid` uses the expression type checker if the condition evaluates to a Boolean value and if the `Then` part actually has child elements that could be used for binding.

Finally, it remains to specify the handling of the `#define` directive and the macro expansion via macro calls. The corresponding RAG implementation is presented in Listing 7.22. Lines 1–15 contain attributes that are related to `#define` and its parameters. The inherited `owningDecl` attribute provides `MacroParams` with a reference to their parent `MacroDecl` node. In the `associatedSlots` attribute, the parameter's parent is then used to collect all `Slots` in the `MacroDecl`'s body which have the name of the parameter (cf. Lines 7–15). Observe that `MacroDecls` never are composers. Semantics of macro expansion is associated with the `MacroCall` composer (cf. Lines 17–39). In C, macro-call syntax overlaps with the syntax of variable access and method calls. Macros without arguments are called *object-like macros* while macros with parameters are called *function-like macros* [Harbison III and Steele 2002]. Because of this overlap, the grammar is inherently ambiguous, since the type of nonterminal depends on

```
aspect MacroComposers {
   // #define and parameter-related attribute declarations and equations.
   inh MacroDecl MacroParam.owningDecl();
   syn lazy java.util.List<ASTNode> MacroParam.associatedSlots();

   eq MacroDecl.getParameters(int index).owningDecl() = this;
   eq MacroParam.associatedSlots() {
      final QRef ref = new QRef(getName());
      java.util.List<ASTNode> slots = owningDecl().collPoints(new Collector(){
         public boolean doEval(ASTNode node){
            return node.isSlot() && node.hasQName(ref);}
         public boolean cont(ASTNode node){return true;}
      });
      return slots;
   }

   // macroCall(...) related attribute equations.
   eq MacroCall.isBind() = getReference().macro()!=null &&
      getReference().macro().getParametersList().getNumChild() ==
         getArgsList().getNumChild();
   eq MacroCall.targetPoint() = this;
   eq MacroCall.composer() = this;
   eq MacroCall.srcFragment() {
      MacroDecl macroCopy = (MacroDecl)getReference().macro().fullCopy();
      List<MacroParam> params = macroCopy.getParametersList();
      List<MacroCallArg> args = getArgsList();
      for(int i = 0;i < params.getNumChild();i++){
            MacroParam param = params.getChild(i);
         for(ASTNode slot:param.associatedSlots()){
            StepResult result =
               ((Slot)slot).doBind((List<Element>)args.getChild(i).srcFragment());
            if(result!=StepResult.OK){
               env().status.add(StatusDescriptor.
                  ERROR("Binding failed during MacroCall. Reason: " + result));
            }
         }
      }
      return macroCopy.getContentList();
   }

   // argument-related attribute declarations and equations.
   syn ASTNode MacroCallArg.srcFragment();
   syn boolean MacroCallArg.isValid() = true;

   eq PlainArg.srcFragment() = List.newASTList(new TextBlob(getValue()));
   eq MultiArg.srcFragment() = getValuesList();
   eq ExpArg.srcFragment() =
      List.newASTList(new TextBlob(Type.asString(getExpression().value())));
   eq ExpArg.isValid() = getExpression().type()!=Type.BOT;
}
```

Listing 7.22: Specification of macro declarations and the macro-call composer.

if a corresponding macro declaration has been declared or not. In the UPP component model, this is reflected by the implementation of the `isBind` attribute. According to its specification, a `MacroCall` node is recognized as a bind composer if and only if a `MacroDecl` node with the same name is found in the left context of the call and the number of parameters is also the same (i.e., `MacroDecl` and `MacroCall` have an equivalent signature). Otherwise, the call is treated as a text node. The actual macro expansion works as follows. When the UPP processor encounters a `MacroCall` node and the node is successfully identified as a bind composer via the `isBind` attribute, the `srcFragment` attribute (cf. Lines 23–39) is evaluated to obtain the fragment that the call should be replaced with. The expansion of the fragment is done by (1) looking up and creating a copy of the referenced `MacroDecl` and (2) for each macro parameter the corresponding slots are looked up via the `associatedSlots` attribute and bound to the corresponding argument provided by the call. The evaluation of arguments is specified by the attributes and equations between Lines 41–49. Like for composers, `srcFragment` provides the actual value of the argument and is evaluated differently for the different argument types. For `PlainArg` this is simply a single `TextBlob` fragment. For `MultiArg` a list of arbitrary UPP nodes is provided for instantiation. Furthermore, arguments are allowed to be constant UPP expressions that can be evaluated at composition-time. Besides basic arithmetics and logics, calls to external arguments in the system environment are possible. Since SkAT emits Java code, system calls are checked and evaluated via Java's reflection API.

With the previous RAG modules, the semantics of each type of UPP composers has been defined. The UPP uses the API generated by SkAT and adds an interpreter and control flow. Algorithm 3 is an abstract representation of the simple algorithm used in the UPP implementation. It takes a runtime instance of the UPP composition environment as an input parameter *env* and produces a list of preprocessed fragments *fl* which—depending on the actual use case—can be serialized or compiled further by a language-specific compiler backend. In the order of occurrence, the algorithm iterates over all fragments in *env* and each child *elem*. Each *elem* is then checked to be a composer via the `isComposer` attribute, which is called via the generated JastAdd RAG API. If it is a composer, the composition is triggered via the composition API with potential problems being reported to the user.

7.4.4. Using the Preprocessor

Similar to the previous minimal composition systems STpL and VTpL, the UPP can be easily applied to a use-case scenario. Consider an application that has two variants rendering two-dimensional or three-dimensional geometric figures. Listing 7.23 contains the Java class `PointX` with UPP directives modeling 2D or 3D points. Depending on the presence or absence of the `THREED` parameter, UPP generates the respective versions. An example configuration is provided by the `Point.conf` file presented in Listing 7.24. It declares two macros. The `THREED` parameter is given as a single-line object-like macro (cf. Line 1) which would expand to an empty string if called by a macro call. Hence, it declares the presence of the parameter and has no further purpose. Additionally, `Point.conf` provides a second macro—`log(msg)`,

Algorithm 3: The `UPP` evaluation algorithm represented as pseudo code.

```
input  : $env$ – the composition environment
output : $fl$ – the preprocessed fragments

$fl \leftarrow$ initialize list
foreach fragment fgmt $\in$ env do
    $n \leftarrow$ count of direct child elements of fgmt
    $i \leftarrow 1$
    while $i \leq n$ do
        elem $\leftarrow$ reference to $i^{th}$ child of fgmt
        if isComposer(elem) then
            if elem is an IfCondition and not isValid(elem) then
                report problem (and stop processing)
            $\Delta n \leftarrow$ compose(elem)
            if $\Delta n < -1$ then
                report problems (and stop processing)
            else
                $n \leftarrow n + \Delta n$
        else
            $i \leftarrow i + 1$
    $fl \leftarrow fl +$ print(fgmt)
```

which is a multi-line function-like macro (cf. Lines 2–4). Macros that cover more than one line of code in UPP differ from those in the CPP as they require an extra `#end` directive instead of a backslash operator at the end of each line (except the last one) which concatenates them into one. The definition of `log(msg)` causes the `UPPProcessor` to replace occurrences of log statements with a direct call statement to Java's `System` interface. In the following, it is explained step by step how the `UPPProcessor` uses the UPP composition system to process the two fragments `PointX.jupp` and `Point.conf`. Correspondingly, the composition result `Point3D.java` can be inspected in Listing 7.25.

- Initially, the composition environment is set up with the two fragments: `PointX.jupp` and `Point.conf`.
- The system interprets `Point.conf`. As it only contains macro declarations and a slot (`#msg#`), which are passive constructs, nothing is changed.
- The system starts interpreting `PointX.jupp`. The single composition steps are described by the subsequent bullet points.

```
1  #include "Point.conf"
2  #if defined (THREED)
3  public class Point3D {
4  #else
5  public class Point2D {
6  #endif
7     private float x;
8     private float y;
9  #ifdef THREED
10    private float z;
11 #endif
12
13    public float getX() {
14       return x;
15    }
16    public void setX(float x) {
17       log("SettingX");
18       this.x = x;
19    }
20    public float getY() {
21       return y;
22    }
23    public void setY(float y) {
24       log("SettingY");
25       this.y = y;
26    }
27 #ifdef THREED
28    public float getZ() {
29       return z;
30    }
31    public void setZ(float z) {
32       log("SettingZ");
33       this.z = z;
34    }
35 #endif
36 }
```

Listing 7.23: `PointX.jupp` represents points in 2D or 3D space.

```
1  #define THREED
2  #define log(msg)
3  { System.out.println(#msg#); }
4  #end
```

Listing 7.24: `Point.conf` declares two macros that can be included.

```
1  public class Point3D {
2     private float x;
3     private float y;
4     private float z;
5
6     public float getX() {
7        return x;
8     }
9     public void setX(float x) {
10       { System.out.println("SettingX"); };
11       this.x = x;
12    }
13    public float getY() {
14       return y;
15    }
16    public void setY(float y) {
17       { System.out.println("SettingY"); };
18       this.y = y;
19    }
20    public float getZ() {
21       return z;
22    }
23    public void setZ(float z) {
24       { System.out.println("SettingZ"); };
25       this.z = z;
26    }
27 }
```

Listing 7.25: The resulting class `Point3D.java`, which is emitted by UPP for the inputs `PointX.jupp` and `Point.conf`.

- Line 1: the `#include` directive for `Point.conf` is encountered. The template processor recognizes the `Include` composer and triggers the composer evaluation in the composition system. The system looks up the `Point.conf` fragment and binds its contents to the `Include` slot (i.e., the directive is replaced with the contents of the fragment).

- Line 2 (Line 5 after composition): an `#if` directive is encountered and recognized as a composer by the template processor. The composition system evaluates the conditional expression, which checks if the `THREED` macro is defined. Thanks to the preceding `#include`, the look up of `THREED` is successful so that the expression yields *true*. Consequently, the whole `#if` directive up to Line 6 is replaced by the first branch.

- Line 9 (Line 8): an `#ifdef` directive is encountered. It is evaluated similarly to the above `#if` and thus replaced until Line 11 with enclosed field declaration for the third coordinate.

- Line 17 (Line 14) and Line 24 (Line 21): UPP consecutively recognizes two calls to the `log(msg)` macro. Triggered by the template processor, in both cases, the composition system again looks up the corresponding macro declaration, creates a copy of the declaration's contents and binds the #msg# slot with the provided call arguments, `"SettingX"` and `"SettingY"` respectively. Finally, the instantiated macro contents are bound to the macro calls.

- Line 27 (Line 24): a second `#ifdef` directive is encountered that also checks the presence of the `THREED` parameter and adds accessors for the z coordinate to the 3D point class.

- Line 32 (Line 28): after expanding the second `#ifdef`, a third macro call to `log(msg)` is encountered and bound to the corresponding contents by the composition system.

- After reaching the end of the fragment, the template processor optionally removes the remaining macro declarations previously included via rudiment extraction in the composition system. The final result after emitting can be inspected in Listing 7.25.

To summarize, the previous sections showed that RAG-based ISC is a suitable approach to realize composition systems or template languages with relatively low implementation efforts gaining a formal component model which enables access to clients through its meta API. However, this comes at a cost of no or little compositional guarantees, and potential runtime overheads in comparison to well-tuned optimized manual implementations. For example, it is obvious that a preprocessor implementation that builds upon internal AST data structure like the UPP introduces a runtime overhead for creating this data structure and also a memory overhead in comparison to hand-optimized preprocessors in industrial or established open-source compilers. On the other hand, generative approaches such as SkAT that are based on an expressive formalism like RAGs have a qualitative effect on the implementation. Specifications typically tend to be much more compact and explicit and are not tangled with technical or performance related aspects, which improves readability and maintainability of source code. These intangible properties are a cornerstone for component models that are extensible and can be enriched with additional concepts for composition. Moreover, depending on the expressiveness of the formalism, the strength of the coupling between component model, fragment and composition language can be *scaled* in such a way that a trade-off between the features supported by the composition and development effort becomes possible for developers.

Based on these observations, the following section proposes an agile development methodology for scalable fragment composition systems.

7.5. Agile Composition-System Development

Agile software development methods [Beck et al. 2001] like Extreme Programming [Beck and Andres 2004] are a flexible and customer-oriented way to develop software. Most notably, they foster a continuously developed and maintained software product that should be working from the beginning. Typically, the software is developed in multiple iterations, each adding new functionality or adjusting existing features of the software. Each iteration should be short, covering a small set of new functionality or changes conducted. In the heart of these small iterations stands the interaction between developers of the software and customers who ordered the software. The advantage of such a close collaboration is that changes requested by the customer and/or her users can be considered early and a continuous evaluation and steady improvement of the product is ensured. The values of agile software development are recorded in the *Manifesto for Agile Software Development* [Beck et al. 2001]:

- "Individuals and interactions over processes and tools",
- "Working software over comprehensive documentation",
- "Customer collaboration over contract negotiation" and
- "Responding to change over following a plan".

In the development of fragment composition systems as complex software systems, agility can also help to improve quality and process. For example, often it is difficult to identify the requirements of a composition system from the beginning, if domain knowledge needs to be integrated into the system to develop the *right* composition operators and composition language abstractions. Hence, it is required to develop the composition system in close collaboration with its future users (i.e., the *domain experts*). Monolithic software development processes such as the V-Model [Binder et al. 2006] do not encourage a steady communication, feedback and change so that important domain knowledge may not be considered accordingly in the final product causing a failure of the project. The *scalable ISC* approach presented in this section enables an agile development style for fragment composition systems. It covers a range of composition-system features starting from minimal component models with string-based slots, over island component models with partially syntax-safe slot composition, syntax-safe composition with slots and hooks and well-formed component models with context-sensitive composition contracts and domain-specific composition operators. These features have different demands in terms of development time and learning curves due to their inherent *implementation complexities*.

Section 7.5.1 discusses the implementation complexity of composition-system features of ISC systems and compares how they are supported by the existing ISC approaches such as the SkAT tool and the approaches evaluated in Chapter 4. Afterwards, in Section 7.5.2, *scalable ISC* is presented as a workflow diagram and it is discussed how SkAT supports this workflow.

7.5.1. Implementation Complexity of Fragment Composition Systems

The *implementation complexity* of composition-system features is a relative measure to estimate the effort needed to realize a composition system with a certain set of features. The following four basic feature sets are considered. (1) The system supports slots only (+ *slots*): fragment components may only contain declared variation points. (2) Slots and hooks, or hooks only are supported (+ *hooks*): fragments may contain variation points and may have implicit extension points supporting unforeseen extensions. (3) The system supports 2 and provides a composition language (+ *composition language*): composition recipes can be written in a language which is dedicated or embedded into the component language. (4) The system supports 3 and has advanced tooling (+ *composition tooling*): users are supported by typical IDE features such as editor services or fragment repositories.

Complementary, these compositional features can be supported at different levels of fragment-language support. The following four categories of fragment-language support are to be considered. (I) The system is completely language unaware (*string-based*): fragment components are not checked with their specification and the composition does not give any guarantees about the result of the composition. (II) The system is aware of the tokens that may appear in fragment components of the target language (*token-aware*): the system only accepts and composes fragments which consist of tokens that are terminals of the target language in order that at least some guarantees at the lexical level are possible. (III) The composition system is aware of the language's grammar or metamodel, which is the typical case for ISC systems (*syntax aware*): fragment components are typed w.r.t. the nonterminals of the grammar so that the composition result is guaranteed to be valid with respect to the context-free fragment language. (IV) The system is aware of parts of the language semantics and context-sensitive constraints (*semantics-aware*): the composition results are guaranteed to be correct w.r.t. the supported properties (e.g., fragment contracts in well-formed ISC).

Figure 7.12 shows the above features as fields of a 16-field "portfolio matrix", whose columns denote different degrees of language awareness while its rows represent supported composition-system features. To estimate the implementation complexity of a fragment composition system using this diagram, it is suggested to employ a simple block metric. If a global distance of 1 is assumed, the complexity measures range from 0 (field I_1) up to 9 (field IV_4). Observe that these numbers are only directly comparable if they are assumed for the same component and composition language. As an example, consider fragment components of a small DSL such as the person language of the BAF example and a general purpose language like C. A IV_4-composition system for the person DSL is possibly much simpler to be implemented than a III_1-composition system for C because of the huge number of language concepts and grammar constraints in C in comparison to the DSL. As string- and token-based composition systems are not aware of a fragment's tree structure, they cannot support context-dependent hooks, which require list nonterminals and nesting in the AST. Therefore, I_2 and II_2 are not counted in the proposed complexity measure.

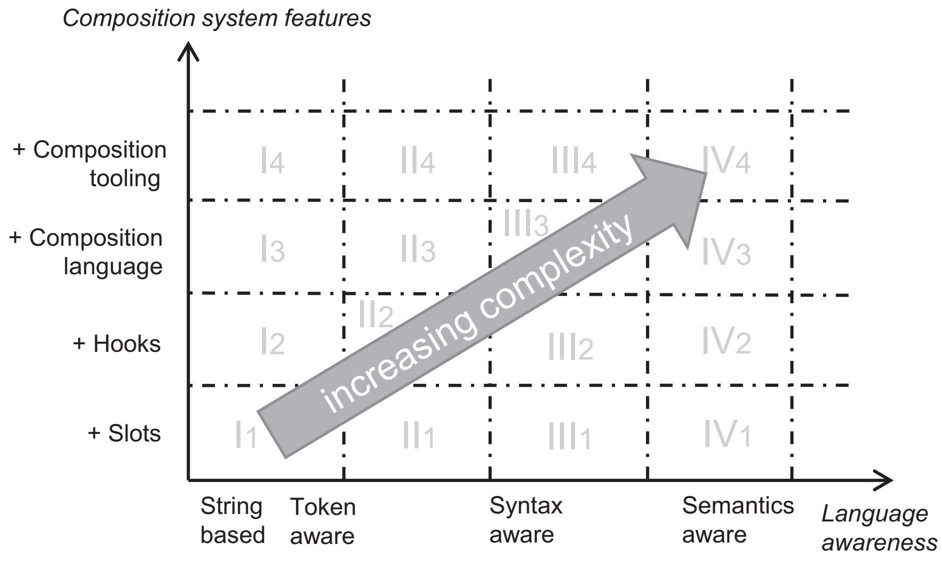


Figure 7.12.: Increasing specification and implementation complexity of fragment composition systems in dependency of the supported features.

In the following, the composition systems developed in this thesis and others are classified w.r.t. the matrix. The STpL is a I_1-composition system as it only supports slots and is completely unaware of the target language. Its complexity measure is 0. Its extensions VTpL and UPP are considered to be I_3 systems while UPP has a completely dedicated composition language and VTpL is a mixed approach with fragment markup and composition API. Also, simple template languages such as StringTemplate [Parr 2006] or xPand [Efftinge et al. 2008] operate on that level. The CPP is token-aware and thus can be considered as a II_3 composition system with a complexity measure of 2. The COMPOST system for Java uses Java as a host composition language and thus is a III_2 composition system. C++ templates and their instantiation are integrated with the programming language while type checks are the compiler's task. Consequently, C++ templates are a III_3 composition system for generic functions and classes. If its fragment contracts are also considered, the SkAT4J composition system for Java belongs to the IV_2 category. Without the fragment-contract support, the system is in the same category as COMPOST. As a command-line version, AspectJ [Kiczales et al. 2001] is a IV_3 composition system. It is tightly integrated with a Java compiler frontend. If the AJDT are also considered, IDE-like composition tooling is provided so that AJDT has an implementation complexity of 9.

To support an agile development style for invasive composition systems, the employed tools should not foil it or, even better, provide functionality supporting agility. Figure 7.13 shows how the ISC frameworks presented in Chapter 4 scale over composition features in comparison with the SkAT-based approaches developed in this thesis. Of course, the classic ISC systems

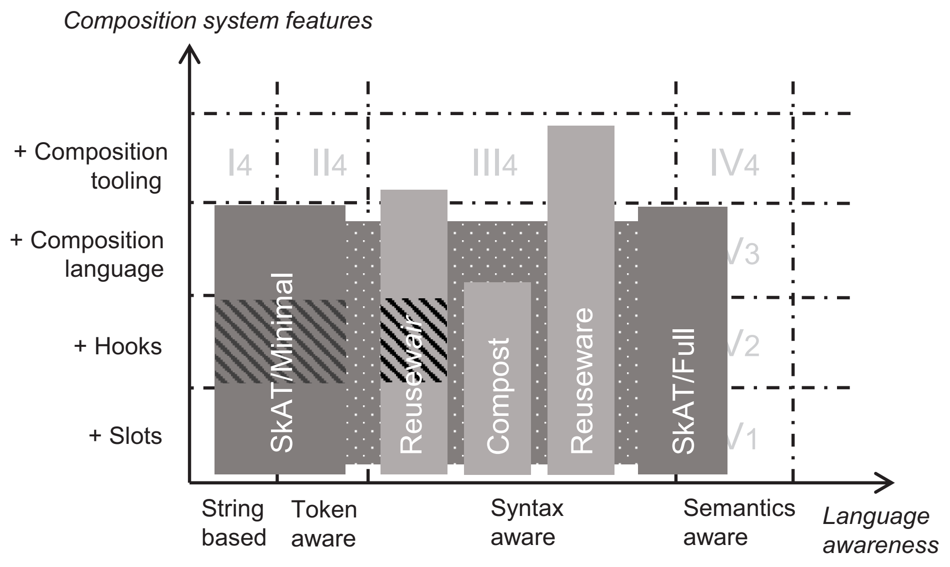


Figure 7.13.: A simple classification of the known ISC tools according to the complexity classes of Figure 7.12.

COMPOST and Reusew*air* support syntax awareness and thus can only scale in sectors of column III. Reusew*air* supports the embedding of composition operators and provides very basic tool support (e.g., textual editors based on EMFText). While the Reuseware approach has an extended ISC model, it still can be compared to the other approaches. The composition tooling provided by Reuseware (including a graphical composition language, a fragment repository and the instant update of composition) is very complex and well-integrated into the Eclipse IDE. Thus, it supports III_4-composition systems.

For agile software development, a restriction to syntax-aware composition is counterproductive: for creating a working product that fully supports a fragment language, a full AST grammar or metamodel has to be provided and compositional points have to be specified w.r.t. that grammar. While this works well in software development processes where requirements have been analyzed exhaustively and the component and language models have been designed in the respective process phases, it is not well-suited for agile processes, where requirements and designs are evolved in collaboration with customers and domain experts over the whole project lifespan. Hence, it would be worthwhile to be able to create "small", productive composition systems from the beginning. These versions can practically be evaluated by their users reporting feedback for improvements. Based on this feedback, systems can be *refined*, and new features can be integrated or others be improved.

As demonstrated in the previous sections of this chapter, small productive fragment composition systems are supported by SkAT/Minimal, which supports string-based and token-aware FCMs (cf. Figure 7.13). For example, the VTpL (cf. Section 7.3.2) is such a "small" productive composition system. It has been employed to investigate possibilities in restructuring the JastAdd code generator, making some of its implementation patterns explicit as composition language constructs or composers, respectively. In an agile development style, VTpL would be handed to developers who can check practicability of the solution and suggest further changes by applying it to recreate other parts of the JastAdd generator. After the conceptual evaluation using the productive minimal systems, and depending on the discovered customer (user) requirements, a syntax or even semantics-aware composition system can be designed. As shown in Figure 7.13, SkAT/Full supports these levels of language awareness. Hence, the concepts investigated with the early products are to be converted into syntax-aware concepts—combining compositional constructs with the actual fragment-language model.

De facto, it still remains a huge complexity gap between the simple composition systems of the two leftmost quadrants and the ones located in the two rightmost quadrants. If it is required to also bridge this gap in an agile development style, it is suggested to use island grammars and island component models to allow developers to step by step refine the composition system with additional syntax and semantics support for parts of the fragment language. In Figure 7.13, this is implied by horizontal bar connecting SkAT/Minimal and SkAT/Full. Besides supporting diverse degrees of language awareness, SkAT directly inherits the aspect-oriented modularity features provided by the underlying RAG system. This also has positive effects on agility. Extensions to FCMs can be modeled as extra JastAdd RAG modules that are woven into the system while other aspects remain untouched.

The following section deepens the discussion on scalable ISC and proposes a general workflow for agile development of ISC systems.

7.5.2. Agile Composition-System Development

The refining activities involved in agile composition-system development, and their inter-dependencies can be captured as a *workflow diagram*. Workflow diagrams help to document process knowledge in descriptive or even executable ways. The Business Process Model and Notation (BPMN) [Object Management Group (OMG) 2011a] is a common standardized modeling language for workflows providing “a notation that is readily understandable by all business users, from the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes, and finally, to the business people who will manage and monitor those processes” [Object Management Group (OMG) 2011a, p. 1]. One of the ancestors of BPMN are UML Activity Diagrams [Object Management Group (OMG) 2011d], which makes it a suitable notation for software engineers (especially composition-system developers) as well as software architects and other stakeholders in software development projects.

A scalable ISC workflow definition should support the creation of any kind of fragment composition system, i.e., string-based and token-aware systems, syntax-aware systems and semantics-aware systems. Also, it should consider different starting points and requirements, since compatible grammars or metamodels—including a semantics specification—might be

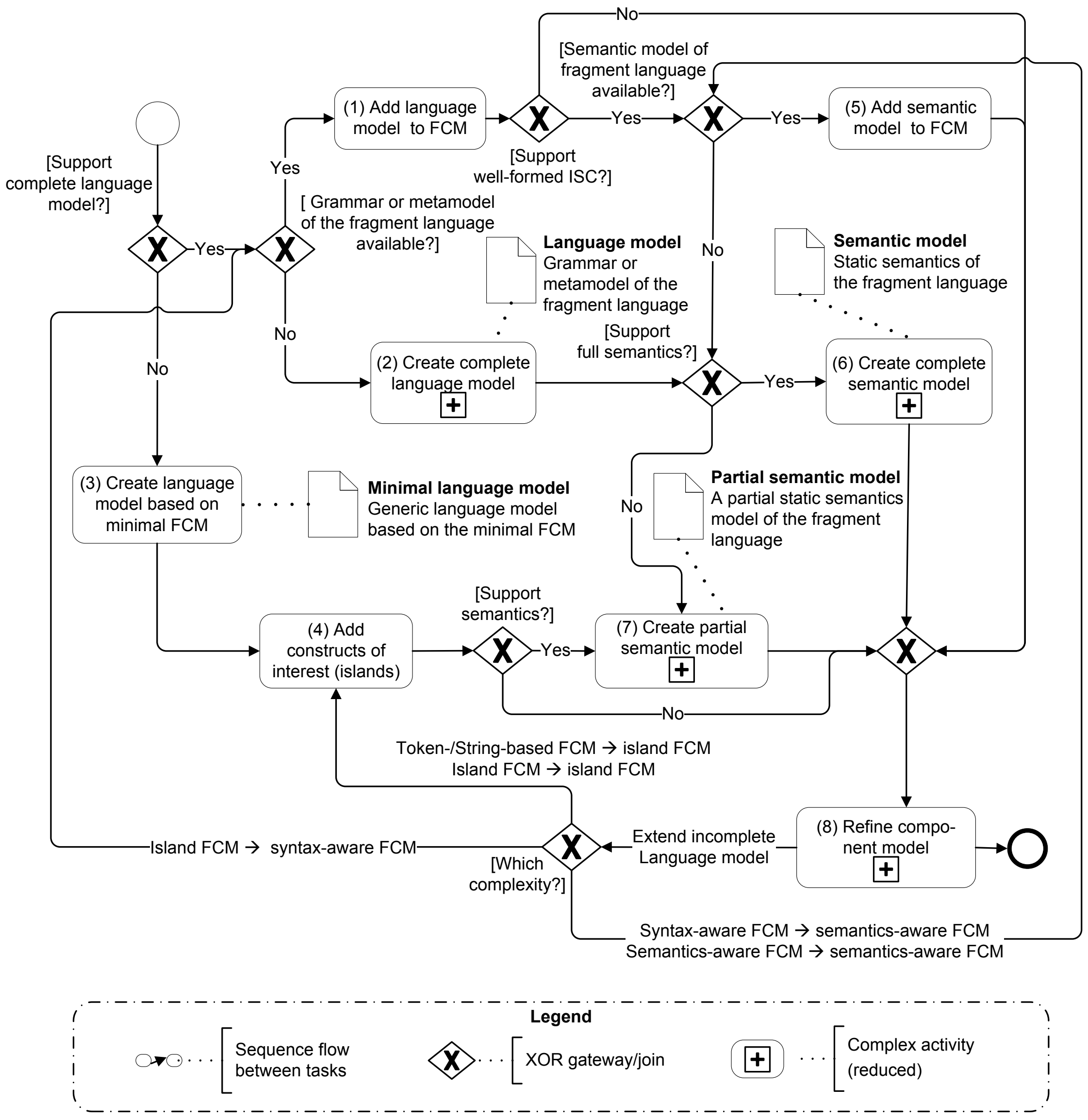

[Support complete language model?]
Yes
No
[Grammar or metamodel of the fragment language available?]
(1) Add language model to FCM
[Support well-formed ISC?]
No
Yes
[Semantic model of fragment language available?]
(5) Add semantic model to FCM
Language model
Grammar or metamodel of the fragment language
Semantic model
Static semantics of the fragment language
(2) Create complete language model
[Support full semantics?]
(6) Create complete semantic model
(3) Create language model based on minimal FCM
Minimal language model
Generic language model based on the minimal FCM
Partial semantic model
A partial static semantics model of the fragment language
(4) Add constructs of interest (islands)
[Support semantics?]
(7) Create partial semantic model
Token-/String-based FCM → island FCM
Island FCM → island FCM
Island FCM → syntax-aware FCM
[Which complexity?]
Extend incomplete Language model
(8) Refine component model
Syntax-aware FCM → semantics-aware FCM
Semantics-aware FCM → semantics-aware FCM
Legend
Sequence flow between tasks
XOR gateway/join
Complex activity (reduced)

Figure 7.14.: The top-level workflow of scalable ISC.

present or absent. Figure 7.14 shows a top-level BPMN workflow which corresponds to these requirements. It consists of eight activity nodes and the control flow (also called *sequence flow* in BPMN) between them, as well as several *gateway* nodes that typically are decision points. Activities associated with artifacts produce fresh specifications (e.g., language metamodels or AST grammars, or semantic specifications) or extend existing specifications with new content which is not part of a refactoring activity (this will be explained later in this section). Furthermore, the diagram distinguishes normal and *complex* activities. In BPMN, complex activities are typically used to fold complex sub-processes into one node in such a way that the diagram can be decomposed to improve readability. In the process of Figure 7.14, complex activities emphasize activities that are to be expected to cause high modeling efforts, but are not necessarily backed by a sub-process diagram in this thesis. Only the essential "Refine component model" task will be discussed in detail later in this section. In the following, the activities and outcomes of the top-level process are described.[5]

Top-Level Activities

(1) Add language model to FCM. If a compatible model of the fragment language is available, it can be added to the set of specifications that contribute to the FCM. For SkAT, this would be a JastAdd AST grammar. In metamodel-based approaches such as Reuseware a language metamodel would be added to the FCM.

(2) Create complete language model. If no compatible model is available or if an island model has been developed during preceding iterations of the current process, a complete AST specification or metamodel can be developed and added to the FCM (if an island model existed before it is replaced). The creation of the language model is considered a complex activity and thus may involve a long-running subprocess itself.

(3) Create language model based on minimal FCM. In case the composition-system developers have decided to create a string-based or token-aware composition system, the minimal FCM introduced in Section 7.1.4 is used as a basic language model. In SkAT, this corresponds to the inclusion of the SkAT/Minimal models.

(4) Add constructs of interest (islands). As a direct successor of activity (3), this activity adds new island constructs or concrete syntax like slot markup to the FCM. The STpL, developed in Section 7.3.1, is a typical initial island FCM while VTpL and UPP, developed in Section 7.3.2 and Section 7.4, are typical island FCMs after several refinement iterations.

(5) Add semantic model to FCM. If a well-formed composition system should be developed and a compatible semantics specification is available, it can be added to the FCM. In SkAT, the semantic model can be provided as a set of JastAdd `.jrag` modules.

(6) Create complete semantic model. If a well-formed composition system should be developed and *no* compatible semantics specification is available, is has to be developed and added to the FCM. Depending on concept coverage, implementing static language semantics is typically a very complex and error-prone task. Therefore, it is modeled as a complex activity.

[5]It is silently assumed that an extension of the language grammar (or metamodel) includes concrete syntax.

(7) Create partial semantic model. Developers can also decide to not support the full language semantic in the composition system. It seems reasonable to implement only those aspects which are indeed required to realize the envisaged fragment contracts, e.g., name-analysis-dependent contracts. The task is modeled as a complex activity, because depending on the complexity of the fragment language model, implementing partial semantics can be complex and error-prone.

(8) Refine component model. This complex activity deals with the actual development of the component model and the composer model, if the latter should be considered. Its internals will subsequently be unfolded and explained by a separate BPMN diagram.

The outgoing sequence flows of the refinement activity should be described in context of the top-level diagram. Depending on the refinement steps and evolving goals emerging from the agile development style, it might be required to extend or modify the underlying fragment-language model. This is modeled by the flow-edge labeled "Extend incomplete language model" that leads to an exclusive gateway that—depending on the situation—decides which activity of the top-level workflow can be executed next. Three different situations are distinguished. First, a token-aware, string-based or island FCM can be extended to support additional island constructs. This is modeled by a flow edge connecting to activity (4). Second, an island FCM can be transformed into a syntax-aware FCM. This is realized as a flow edge leading to the gateway which models the decision on using an existing language model or creating a fresh one. Third, a syntax-aware FCM shall be extended with a semantics model or a partial semantics model should be extended. This is modeled by the flow edge leading to the decision about adding an existing full semantics specification or creating a fresh or additional partial model. The refinement workflow is shown in Figure 7.15 while its activities are described subsequently.

Refinement Activities

(1) Specify extension grammar/metamodel. In this activity, grammar or metamodel extensions that represent composers or points can be added to the language model. In SkAT, this can be an extra AST specification whereas in Reuseware this would normally be added to the language metamodel.

(2a) Specify slot model. If developers decide to add variation points to the FCM, a slot model has to be added to the composition system. Conceptually, this model represents the slot-identification function introduced in the common FCM Definition 4.1. In SkAT, this is realized by adding or modifying an according JastAdd RAG aspect. In Reuseware, this is realized by the Reuse EXtension language for Component Model Configuration (REX_{CM}).

(2b) Specify hook model. This activity models the introduction of new extension points to the FCM. A hook model will be created and added to the composition system. Conceptually, this model represents the hook-identification function according to Definition 4.1. Like for

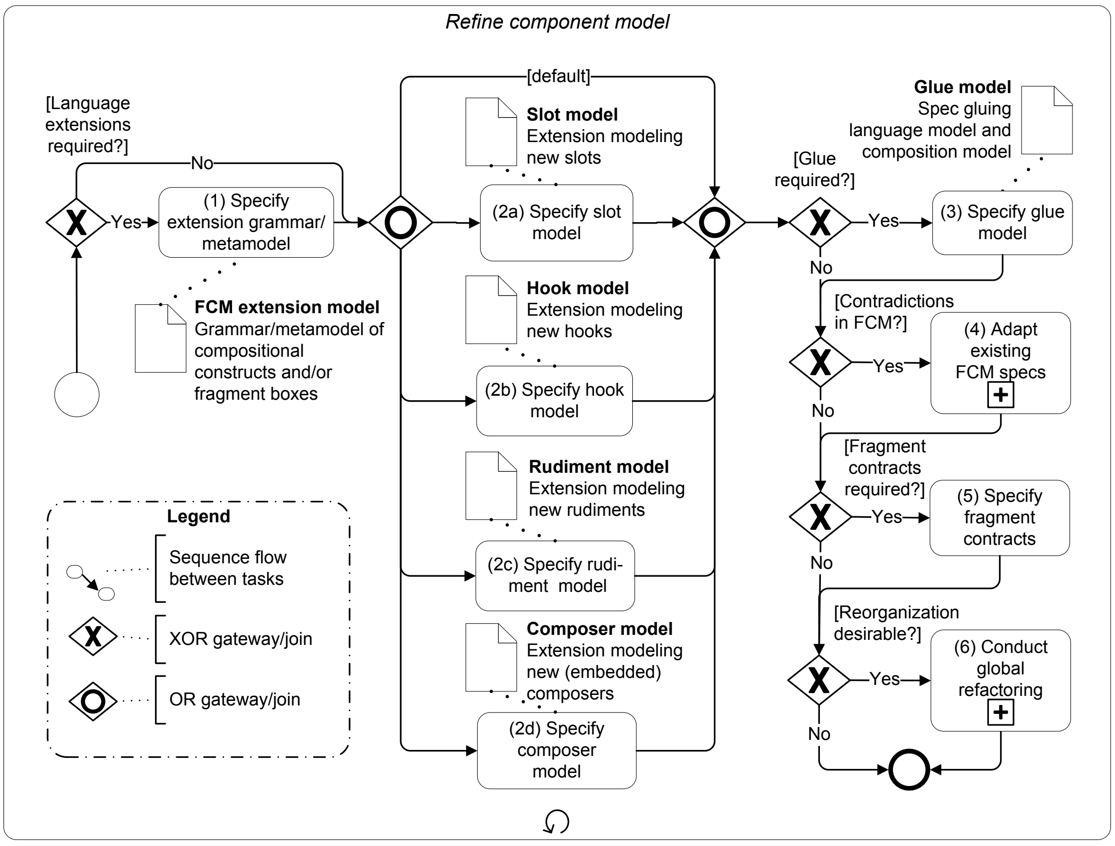

Refine component model
[default]
[Language extensions required?]
No
Yes
(1) Specify extension grammar/ metamodel
FCM extension model
Grammar/metamodel of compositional constructs and/or fragment boxes
Slot model
Extension modeling new slots
(2a) Specify slot model
Hook model
Extension modeling new hooks
(2b) Specify hook model
Rudiment model
Extension modeling new rudiments
(2c) Specify rudi- ment model
Composer model
Extension modeling new (embedded) composers
(2d) Specify composer model
Glue model
Spec gluing language model and composition model
[Glue required?]
(3) Specify glue model
[Contradictions in FCM?]
(4) Adapt existing FCM specs
[Fragment contracts required?]
(5) Specify fragment contracts
[Reorganization desirable?]
(6) Conduct global refactoring
Legend
Sequence flow between tasks
XOR gateway/join
OR gateway/join

Figure 7.15.: Component-model refinement workflow in BPMN.

slots, in SkAT this can be realized by adding or modifying a corresponding JastAdd aspect while in Reuseware it is specified in the REX_{CM} specification.

(2c) Specify rudiment model. In this step, nonterminals or concepts whose instances can be removed via the extract composer can be defined as rudiments in this activity. Similar to slots and hooks, rudiments are supported via attributes in SkAT.

(2d) Specify composer model. In this activity, specific composition operators like embedded import and weaving constructs can be specified. Again, SkAT supports composers via special attributes dedicated to that purpose that can be added to a fresh or existing module.

(3) Specify glue model. After compositional constructs or fragment-box types have been added to the FCM, it might be necessary to define a glue model that closes the gap between context-dependent semantic functions. For example, name and type analysis typically depend on a scoped context. In RAGs, context-dependent attributes are typically inherited

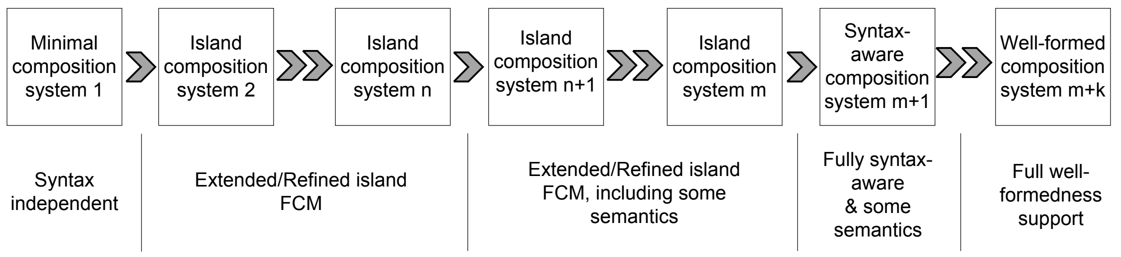

Minimal composition system 1
Island composition system 2
Island composition system n
Island composition system n+1
Island composition system m
Syntax-aware composition system m+1
Well-formed composition system m+k
Syntax independent
Extended/Refined island FCM
Extended/Refined island FCM, including some semantics
Fully syntax-aware & some semantics
Full well-formedness support

Figure 7.16.: A potential execution sequence of agile iterations.

attributes or attributes which indirectly depend on an inherited attribute. Hence, contexts of inherited attributes not considered in the integrated FCM have to be closed in form of additional equations for these contexts. Moreover, due to broadcasting of inherited attributes, which silently passes inherited values down via implicit replicating attributes, unexpected contexts causing erroneous computations may occur. For more information on related issues, it is kindly referred back to Section 5.1.5.

(4) Adapt existing FCM specifications. Adding new points or composers to the FCM may cause inconsistencies with the established concepts in the FCM, e.g., making them invalid or contradicting. In this activity, such inconsistencies shall be resolved by unifying, removing or adapting the contradicting formerly established parts of the FCM.

(5) Specify fragment contracts. During this activity, fragment contracts supporting well-formed composition can be specified using portions of the specified fragment language semantics to check if the fragment remains well-formed after composition. In SkAT, this is supported by special contract attributes which are automatically checked during composition (cf. Section 5.4 and Section 6.3.5).

(6) Conduct global refactoring. After one or more complex refinements and additions to the FCM, it may be advisable to conduct a refactoring of the specifications and models involved to reorganize, group or integrate new modules. This improves the quality of the specification models and ensures extensibility of the FCM in later iterations.

The sequence flows between the root workflow diagram in Figure 7.14 and the refinement diagram in Figure 7.15 enable the proposed agile development style. Figure 7.16 exemplifies a potential of iterations based on the composed workflow. The development starts with a simple minimal composition system based on the minimal FCM, e.g., supporting slots and some variability concepts. In a second iteration, some constructs of interest are added as islands to the FCM to provide a small degree of syntax awareness, e.g., adhering the language's block structure. Several iterations (until composition system n) add more constructs of the fragment language as island to the FCM, e.g., expressions. During iterations $n + 1$ to m, some semantics support is added, e.g., the look-up of method names or other context-sensitive properties. Finally, the last iterations replace the island FCM with a full grammar or metamodel of the fragment language.

7.6. Summary and Conclusions

This chapter introduced scalable ISC as a novel agile approach to composition-system creation. With the approach it becomes possible to develop fragment composition systems in small iterations considering new requirements or changes. The main advantage over the classic ISC approaches and well-formed ISC is that composition abstractions can be tested and evaluated earlier, and the effort of developing a full-fledged ISC system can be avoided initially.

Scalable ISC is supported by three other inventions of this thesis. The RAG-based specification method of fragment component models, developed in Chapter 5 and implemented in Chapter 6, enables a decomposition of FCM specifications along user-specific concerns. This property is essential for extensions to the FCM in iterations of the scalable ISC workflow and for decomposing the model as it is adequate for the process. Moreover, island FCMs were introduced. Based on island grammars, island FCMs allow specifying fragment component models w.r.t. an incomplete language specification consisting of islands (the specified parts) and water (the unspecified parts). Hence, with island FCMs it is not required to provide a complete grammar of a CnL, instead a partial one with CnL-specific islands suffices. As a corner case, a minimal fragment component model has been introduced. The minimal FCM is an island FCM which only has slots as islands while everything else is water. Consequently, the minimal FCM is an island FCM w.r.t. any CnL with slots (modulo CnL-specific hedge delimiters) and thus an adequate starting point in composition system development according to suggested workflow of scalable ISC.

Supporting the argumentation in this chapter, three case studies of island-based composition systems were conducted. To implement these systems, SkAT provides SkAT/Minimal as an extension of SkAT/Core and an implementation of the minimal FCM, which can be included and extended by specific systems. The Slot Template Language (STpL) is an instance of SkAT/Minimal that adds a concrete slot-delimiter syntax and provides a SkAT-based composition API. Extending STpL, the Variant Template Language (VTpL) adds compositional constructs to declare variants in fragment components that can be instantiated and parametrized via an API. As a side effect, this case study also demonstrates how SkAT supports embedded composer signatures and how string-based code generators can use such a language to untangle control flow from fragment representation. The Universal Extensible Preprocessor (UPP) is an extension to the STpL that adds an extensible macro language and an ISC-based macro expander. In comparison with the VTpL, UPP demonstrates how SkAT supports embedded composers with an active expansion semantics. In comparison with “classic” preprocessors like the CPP, UPP provides extensibility in two dimensions: new directives can be added as composers and language awareness is scalable towards an improved support of CnL-specific constructs, e.g., brackets, declarations, statements.

Since the discussions of the main contributions of this thesis have now been completed, it remains to comment on additional related work in the next chapter.

8 Related Work

This chapter discusses work related to the research presented in this thesis that has not yet been discussed sufficiently in any of the other chapters. As the thesis topic has relations with a number of research areas in theoretical and applied computer science, only a small selection of related work can be covered here. This selection has been chosen carefully and to the best of the author's knowledge, but still may miss important results from other sources as the body of acquired knowledge steadily grows exponentially.

8.1. Safe Template Languages

Template languages are important tools for generating source code from a generic specification which is parametrized with data from application-specific models. While many template tools operate on character or token-stream level (e.g., StringTemplate [Parr 2006]), only a few consider the grammars of object languages (i.e., the fragment language or CnL in ISC terminology) and even fewer consider their semantics. This section focuses on template languages and tools of the latter two categories.

Repleo

[Arnoldus 2011] presents a declarative approach for the specification of syntax-safe template engines for arbitrary context-free object languages. Therefore, a reusable metalanguage and template-evaluation engine is defined and implemented as a Java-based prototype called *Repleo*. Following the spirit of StringTemplate [Parr 2006], the metalanguage is designed toward being "strong enough to express the unparser" and minimal w.r.t. "the possibilities for expressing calculations in the template" [Arnoldus 2011, p. 61]. Thus, the metalanguage is kept small, supporting concepts such as placeholder replacement (i.e., slot binding), control flow and iterations over input-model elements. Under the hood, the approach is based on the *SDF* grammar language and its corresponding SGLR implementation [Visser 1997; Brand et al. 2002]. The advantage of SDF over conventional parser generators is that it provides a module system and can handle ambiguous grammars. Thus, the metalanguage of Repleo is provided as SDF modules and can be combined with an object-language grammar by specifying a *combination module*. The combination module defines where placeholders may occur by relating them with the nonterminals of the object language. Hence, spoken in ISC terminology, combination modules realize slot-identification functions. Given the SDF modules of metalanguage, object language and combination, SDF generates an integrated template-language parser which is used by Repleo to create parse trees from syntactically correct template files. Based on the parse tree, the object-language grammar and an input model, the template engine performs a tree traversal and interprets the metalanguage constructs by performing rewrites on the parse tree. Interestingly, [Arnoldus 2011] also investigates static semantic checks for Repleo templates based on JastAdd. Therefore, a prototypical JastAdd-based realization of the metalanguage is discussed as well as a name-based mapping of object-language grammars in SDF to JastAdd. In parts of the experiment, it was possible to specify semantic checks of metalanguage concepts as well as reusing checks of a small toy language. However, in a larger experiment an integration of Repleo and JastAddJ failed because of mismatches in the SDF Java grammar and the JastAddJ grammar. This is due to the fact that JastAddJ has both an AST grammar *and* a parser grammar while Repleo only supports declarative SDF grammars. During the parse, JastAddJ creates AST nodes which already are abstractions from the original parse tree and cannot be covered by simple mappings in all cases.

In comparison with the research conducted in this thesis, the work of [Arnoldus 2011] is much more focused on a small class of template abstractions, assuming that everything else already is in ideal shape in the input model so that additional computations or non-linear transformations are not required. In contrast, the scope of this thesis is broader, covering abstractions for templates but also cross-cutting concerns and composition interfaces. Considering a typical code generator, different concerns are tangled and scattered throughout the generator's implementation. Using a simple template language, it is hard to express and organize these concerns as they have to be organized linearly along templates—in the worst case on a per file basis. A fragment composition system does not force a code generator to be organized along template files. Instead, it can be much better organized along the generator's logic.

Model-aware templates

[Heidenreich et al. 2009c] describe an approach on extending model-based textual languages with template-metaprogramming constructs. Similar to Repleo, the approach features a predefined set of metalanguage constructs which is provided as an EMOF metamodel (cf. Section 3.2.2) including constructs for iteration, conditionals and placeholders. Object languages are assumed to be provided as a metamodel for abstract syntax and an EMFText grammar defining the metamodel's textual representation. To combine the object language with template concepts, the language's metamodel is transformed automatically in such a way that *every* reference declaration in the metamodel refers to an abstract class whereas the original classes become their subclasses (cf. the slotification operator of Reuse*wair* discussed in Section 4.3). The so prepared metamodel and the model of the metalanguage are then imported into a third metamodel which combines both via subclassing. Besides abstract syntax, the newly derived template language needs to be provided with a concrete syntax. This is achieved by supplying another EMFText grammar that imports the object language syntax and adding template specific symbols (e.g., hedge symbols). Observe that although EMFText does not support scannerless or generalized parsing, grammar conflicts can be avoided by carefully choosing keywords and markup syntax. Concerning well-formedness, the authors of [Heidenreich et al. 2009c] do not follow a clear approach. At least, it is suggested to use the reference-resolver stubs of EMFText which are generated for each non-containment reference declared in the metamodel. This has the disadvantage that reference resolvers in EMFText can only be used practically for name analysis and the algorithms cannot be modularized nicely along their semantic concerns.

In comparison with the approach of this thesis, the statements concerning Repleo also hold for the approach of [Heidenreich et al. 2009c]. Moreover, an automatic extension of the object-language metamodel seems inadequate as it tends to pollute the specification and easily breaks other tooling.

SafeGen

SafeGen [Huang et al. 2011] is a template-metaprogramming language for safely generating Java code. SafeGen combines axioms in first-order logic with basic code-analysis algorithms to check a template's correctness using an automatic theorem prover on the axioms. If a template is recognized as correct, it is guaranteed to produce well-formed Java code for any valid input parameters. Observe that "correctness" here means correct w.r.t. the axioms. According to [Huang et al. 2011], this includes checking the presence of a non-final super class id declared, valid argument types of a method and a correct typing of a returned value. As the generic approaches above, SafeGen's expressiveness is restricted to control-flow and iteration statements. Moreover, the parameters of a template can only be obtained by reflective queries over the existing class base (e.g., querying a class or method name), which means that no custom input models are supported.

In comparison to the approach of RAG-based fragment contracts developed in this thesis, axioms in SafeGen play a role similar to attribute-based assertions in fragment contracts. They formulate well-formedness conditions that are checked before or during composition or template expansion. RAGs are much more expressive than first-order axioms as they can easily implement typical static analysis algorithms for complex real-world languages. At a downside, RAGs cannot be checked by a theorem prover. However, it could be a viable approach to combine first-order axioms as fragment assertions mapping them to characteristic attributes as "built-in predicates" provided by the underlying RAG.

C++ Templates and Concepts

The C++ programming language provides a standardized set of class and function templates provided with the C++ standard library [ISO/IEC 2011]. Templates in C++ are different from the template languages described above as they are part of the language by definition and allow parametrizing class and function declarations with type names, constants as well as function and member pointers. Hence, their main purpose is providing generic-programming abstractions for C++ developers (e.g., providing type-independent data structures) not code generation per se. A template is instantiated when it is used in a program by binding type names to template parameters (i.e., type slots). As they are integrated with the object language, the parser checks if templates are syntactically correct. However, the adequateness of types is mostly checked after the instantiation of a template by the compiler, which tries to compile classes or functions generated from templates. Thus, the error messages delivered to the user in case of compilation problems are not related to the template but to the generated artifacts, which makes it difficult to find the problems' actual causes. To improve there, *concepts* have been suggested as a declarative extension to the C++ standard [Sutton and Stroustrup 2012]. Using *concepts*, requirements on type parameters can be specified w.r.t. template parameters by constraints and axioms. Constraints define functions and operators a type argument must provide, while axioms are semantic requirements that must hold at execution time. Consequently, type-related problems during template instantiation can be detected earlier and provide better understandable error messages.

With ISC and SkAT it is easy to implement template languages such as C++ templates. Moreover, using well-formed ISC and fragment contracts, static problems can be detected earlier and provide better error messages (e.g., by checking type arguments). However, also RAGs can only reason about information which is available to the system. For example, type information may not be completely derivable so that checks are still incomplete. An approach such as *concepts* can provide the missing information and complete the analysis. As *concepts* are declared by template developers, they can be considered as user-defined fragment contracts.

8.2. Macros and Preprocessors

As the term "macro" is frequently used in a very broad sense (e.g., [Tatsubori et al. 2000]), it is first approximately defined what is considered as a macro in this section. Macros are static metaprograms nested in a host language which are specified by a *macro definition* consisting of a pattern (e.g., a name and a list of parameters) and a replacement rule. A *macro call* consists of a declared macro's name and a list of arguments matching the requirements of the declaration's pattern. Knowing all macro definitions, a macro processor locates all occurrences of macro calls in a program and replaces them according to the respective replacement rules. Typically, macro syntax is transparently integrated with the host language in such a way that using macros "feels" similar to using a non-macro language feature. Replacements are usually specified using templates whereas the template may respect the host language's syntax (*syntactic macro languages*) or not (*lexical macro languages*) [Brabrand and Schwartzbach 2002]. Well-known representatives of both classes are pattern-based macros in Scheme [Kelsey et al. 1998] and macros supported by the C preprocessor (CPP) [Harbison III and Steele 2002].

The relation between *embedded* ISC, and definition and expansion of syntactic macros is investigated in [Henriksson 2009, p. 204ff]. The ideas presented there are closely related to [Brabrand and Schwartzbach 2002], who suggest a macro-definition approach with custom syntax *metamorphing* macro calls into host-language syntax. In terms of embedded ISC, a macro call is represented by a composer node with custom syntax in the AST and the macro definition is given as a composition program. Although not studied extensively, the RAG-based specification method for FCMs presented in this thesis is capable of defining such macro abstractions by employing generalized composers (cf. Section 5.2.2). In comparison, it is more flexible than the embedded ISC approach since it does not dictate a specific language-extension pattern and allows reusing attributes from the CnL's RAG (e.g., to check static semantic properties).

Moreover, based on minimal ISC and island FCMs, the SkAT framework can be used to implement lexical preprocessors and preprocessors with partial language awareness. The UPP discussed in Section 7.4 demonstrates this. A tool like the UPP is of practical interest since its syntax and semantics are derived from the CPP which is widely used in any kinds of C/C++ programs. Based on its extensible component model, RAGs and island grammars, it can be used to provide a safer preprocessing or to detect preprocessor-related errors in existing programs. Research in that direction has received some attention during the last years. [Kästner and Apel 2009] analyze the (bad) effects of `#ifdef` directives on code quality and suggest several tool-supported improvements for better handling these effects, e.g., view-based coloring of source-code variants or disciplined annotations. The UPP could be a basis of such tooling. In a later work, [Kästner et al. 2011a] suggest a partial preprocessor which expands macro calls and `#include` directives only leaving `#ifdefs` intact to be able to perform a variability-aware analysis of the source code in context of a larger tool chain called *TypeChef* [Kästner et al. 2011b]. The integrated tool chain can be used to detect preprocessor-related syntax errors [Medeiros et al. 2013]. Recently, the authors of [Heumüller et al. 2014] reported on an approach to fully parse a C code base including preprocessor statements. The process involves two preparsing transformations to be able to fully parse the very most of the code. The first transformation is normalizing `#ifdefs` in such a way that only complete C constructs are contained in the `#ifdefs` while the second transformation distinguishes macro calls from normal identifiers or type declarations by crawling the code base for finding corresponding macro or type declarations. Interestingly, the authors also mention island grammars as an alternative approach for solving the parsing problem, but discard it because island grammars may not consider contextual information sufficiently. This is certainly true in a sense that only an exact parser and analysis tool can provide all contextual information. However, an island-grammar-based approach such as the UPP can be useful to perform the two described preparsing transformations as islands are essentially syntactic patterns to be detected in a stream of characters or tokens.

8.3. Aspect Languages

As the basic ideas of aspect-oriented programming (AOP) have already been introduced in Chapter 1, it is kindly referred back to that discussion in case it is required to refresh one's knowledge in this respect. Moreover, in AOP the following common terminology is used: *Aspects* are compilation units encapsulating crosscutting concerns. They mainly consist of *advice* and *pointcut* definitions. An advice declaration defines a piece of code (i.e., a fragment) that should be *woven* (i.e., composed) with a program and information on how it should be woven (e.g., by extension or replacement). Pointcuts denote sets of *joinpoints*, where joinpoints are points in the static structure or execution of a program that can be extended by aspects (i.e., hooks). Pointcuts are defined using *pointcut expressions*, which are essentially patterns matching joinpoints.

As already discussed by [Aßmann 2003] and [Henriksson 2009], AOP abstractions can be implemented using ISC. This section therefore focuses on some specific AOP implementations which are specifically related to the contents of this thesis.

Static Aspects in JastAdd

First of all, it should be recalled that the SkAT implementation is based on the aspect-based module system of JastAdd. This is a natural approach because static semantic analysis algorithms often are crosscutting in nature. Since JastAdd aspects only support attributes, inter-type declarations, and static joinpoints and advices in form of refining equations, it would immediately be possible to "bootstrap" the JastAdd module system with SkAT. As a positive side effect, slots could be introduced into the RAG specification language in such a way that generic attribute declarations and equations become possible. Thinking further, these may be used to provide composition-system developers with templates of SkAT FCM equations, which currently have to be handcrafted. Moreover, common patterns of name- and type-analysis algorithms could be provided as reusable templates.

AspectBench Compiler

The AspectBench Compiler (ABC) presented in [Avgustinov et al. 2006] is an extensible compiler of the AspectJ AOP implementation for Java [Kiczales et al. 2001]. It provides an extensible component model for aspects, an extensible parser and a Java-bytecode weaver. The main purpose of ABC is to facilitate the development and testing of additions to the original set of AOP abstractions in AspectJ. Conceptually on a broader scale, the SkAT framework provides a similar infrastructure for ISC systems. SkAT/Core provides an extensible FCM infrastructure and composition semantics. The SkAT/Full and SkAT/Minimal projects are then extensions that are used by concrete composition systems such as SkAT4J, which themselves are also extensible. Extensions in ABC are organized in packages for syntax extensions, AST classes, compiler passes, behavioral changes, type-system changes and weaving. SkAT systems are typically organized in groups of RAG modules, AST grammar files and parser-grammar modules. Thanks to the

RAG modules and JastAdd's own aspect-based abstractions, composition-system specifications can be organized more efficiently and compact than in the ABC system, which is mainly based on delegation and the visitor pattern [Gamma et al. 1995]. The paper [Avgustinov et al. 2008] supports this claim by studying a replacement of parts of the ABC frontend with a JastAdd-based counterpart. In comparison with the original implementation, the JastAdd-based implementation has a smaller specification, provides better extensibility and is faster.

Generic Aspects

The authors of [Lohmann et al. 2004] propose an extension of AspectC++ [Spinczyk et al. 2002]—an AOP implementation for C++ which adds support for generic types to the aspect language. They discuss generic extensions to AspectC++ in two dimensions. The first dimension is concerned with improvements of the pointcut-expression language and joinpoint model enabling the weaving of C++ templates and instances. The second dimension is concerned with *generic advices*, which may statically access information on parametrized types (e.g., from parametrized argument or return types) available at specific joinpoints. Using this information, advices can parametrize templates themselves, which may be part of and aspect or defined anywhere else in the program.

In ISC, genericity and extensibility are naturally supported via slots and hooks. However, implementing context-dependent, generic advices in a dedicated composition language (such as AspectC++) is cumbersome with the previous CFG-based approaches to ISC since it requires program-analysis algorithms to provide context-dependent information. Fortunately, the RAG-based approach presented in this thesis provides a better support in this respect since it is based on RAGs that can also host implementations of such algorithms.

8.4. General Approaches to Software Composition

There are of course other general approaches to software composition with fragments. Two of them will be discussed below.

AHEAD

The *Algebraic Hierarchical Equations for Application Design (AHEAD)* model and tool collection [Batory et al. 2004] is a software-composition approach based on *step-wise refinements* [Wirth 1971]—a general development methodology for step by step refining programs with high-level abstractions to full-fledged programs. Using AHEAD, the refinement concept has been applied to model multi-dimensional separation of concerns (MDSoC) [Batory et al. 2003] and AOP [Apel et al. 2007] abstractions. In the AHEAD model, a refining composition is realized as "a function that transforms arbitrarily nested containment hierarchies down to the most primitive artifacts. A refinement may alter a containment hierarchy by adding new nodes [. . .] or it may refine existing nodes" [Batory et al. 2004, p. 357]. Refinements are expressed using a form of mathematical equation system whose equations consist of *constants*, *functions* and *collectives*. Constants denote the most basic artifacts in the containment hierarchy, e.g., a class declaration. Functions denote binary refinement operations applied to constants or collectives. Collectives define a composite set-based structure, as they may contain constants, functions or collectives themselves. The refinement operation is polymorphic, i.e., it can be applied to nested collectives. However, at the level of constants, a *refinement composition operator* has to be provided as a black box for each kind of artifact used in the equation system (e.g., if Java class declarations shall be refined, a corresponding composition-operator implementation is required). Given an equation system, basic composition operators and a set of artifacts, the AHEAD tool suite generates an application. Observe that different equation systems produce different variants of a program.

The AHEAD approach and ISC have some commonalities. Refinement as a composition operation is related to the extend composition operator of ISC, which can be considered as a fine-grained refinement operation applied to fragment hooks. Moreover, the authors of [Batory et al. 2004] suggest that refinement is a generalization of mixin-based inheritance. As shown in Section 6.4.2, mixins can easily be implemented using ISC and may be integrated in larger composition scenarios. Hence, ISC provides an adequate model and implementation technique to provide composition operators on artifacts in AHEAD while its equation language is a high-level composition language.

Considering their differences, the model of ISC is more detailed and more complete than the refinement model of AHEAD, which merely abstracts from composer implementation at a certain degree. Moreover, the equation language of AHEAD seems rather simple, which gives rise to questions concerning its expressiveness (e.g., how can recursion and constrained compositions be supported). On the other hand, AHEAD supports composition of fragments in different languages and may also support bytecode composition. In this thesis, this has not been investigated for ISC. However, multiple languages could be supported by a multi-language environment aggregating multiple fragment languages or multiple composition systems.

Choice Calculus

The *choice calculus* [Erwig and Walkingshaw 2011] is a general approach to formally represent software variation (e.g., as it occurs with `#ifdefs` in the CPP). The main objective of the calculus is to have a common theoretical model to represent software variation. The calculus works on and transforms an artifact's (or a fragment's) tree representation. To encode variability, the tree contains specific expressions which are recognized and transformed by the calculus' replacement rules (e.g., to derive a specific variant or to reduce the number of possibilities). *Choices* are the most basic constructs supported by the calculus as they model equally ranked variants of artifact-language expressions as well as calculus expressions. Moreover, *variability dimensions* provide a flexible handling of different variability concerns (e.g., variable names, operator implementation, concurrency) by decoupling the mapping of variant names and variants declared within choices. Remaining concepts are *references* and *bindings*. References group equally named points that can be varied by variant bindings.

In comparison with ISC, the choice calculus is a strict formal model to describe software variation on paper with the purpose of promoting the discourse on theoretical software-variation research while ISC is rather an implementation technique for fragment composition systems and the related research.[1] While the model of ISC also supports extensibility, grammars and component interfaces, the choice calculus only supports the concepts discussed above. Nevertheless, it could be used (or extended) to describe a subset of ISC.

In [Chen et al. 2014], the *variational lambda calculus* is introduced as an advancement of the choice calculus. It is completely based on the lambda calculus and considers the type system of the underlying language for the inference of variational types. Hence, the calculus could provide a strict formal basis for variability languages based on well-formed ISC.

8.5. Formal Models of ISC

In this thesis, RAGs are used to specify a model of ISC as well as to check and validate fragment components via fragment contracts. Moreover, a formal definition of the core concepts of ISC concepts—ISC_{core}—was given in Section 4.1.3. Both approaches do not support *automatic verification* directly: RAGs are rather an expressive implementation technique than a framework for proving things while ISC_{core} could be used to prove certain properties of FCMs or composition scenarios on paper (e.g., termination of a composition program), but not automatically. For such purposes, logics-based or tool-supported formalizations might be a better choice.

F-Logic

In [Azurat 2007], the author suggests to use *F-Logic* [Kifer et al. 1995] to describe fragment-composition scenarios and apply automatic reasoning to conclude certain properties of the composition and to specify constraints. F-Logic integrates object-oriented features required by ISC such as encapsulation, polymorphism and containment-relations with resolution-based reasoning. The paper provides a small case study that realizes fragment components, implicit hooks and composers using F-Logic concepts. Fragment classes are realized as object facts, specific fragment components are fragments with specific instance rules (called *f-molecules* in F-Logic) and composition operations are expressed using normal f-logic rules. Although the example is incomplete and does not provide a full FCM (only fragment identification is considered), it at least sketches how composition-constraint checking could principally work in F-Logic.

Theorem Proving

The authors of [Kezadri et al. 2012] present a formalization of basic ISC concepts using set theory and metamodel-based multigraphs with the purpose of providing a framework for semi-automatic composition verification using the *Coq* theorem prover [Bertot and Castéran 2004].

[1] In the broadest sense, there is of course something like a tree-based calculus behind ISC, whose basic operations are the binding of slots and the extension of hooks.

Therefore, the notions of fragment-box interface, compositional points as well as the bind and extend operations are formally defined. A fragment-box metamodel is constructed by extending the original fragment metamodel (i.e., the CnL definition) with explicit metaclasses for hooks, prototypes (cf. Section 4.4) and fragment nodes. This modification allows *any* node (object) in a fragment component to be an instance of a hook or a prototype. The composition operators bind and extend are then defined on the basis of the extended, explicit fragment-box interface by graph-node and -edge replacement. Structural properties of the composition are then proved using a semi-automatic proof assistant. For example, it is verified that the resulting structure also is a multigraph, that the fragment-box interface is respected by the composition and that the resulting fragment always is an instance of the original CnL metamodel. In comparison, the formalizations of ISC used in this thesis are based on the assumption that fragments are basically trees with additional reference edges (i.e., multigraphs with spanning trees), while above general typed multigraphs are used. The checks proposed by the authors of the above paper are quite reasonable and could also be applicable for ISC_{core}. However, a more explicit consideration of component-model specifications at the meta level would make a verification more valuable. Moreover, proving properties such as confluence of a specific composition scenario with a set of compositions would be a reasonable add-on.

Recently, in [Kezadri Hamiaz et al. 2014] the authors of [Kezadri et al. 2012] extended their work towards consistent model composition. This means that syntactic constraints of the CnL's metamodel such as feature cardinalities, bidirectional references and containment are verified. Moreover, the authors suggest to use pre- and postconditions to avoid compositions that violate well-formedness constraints (e.g., duplicate attributes in a metaclass). Hence, the approach in [Kezadri Hamiaz et al. 2014] is related to the notion of fragment contracts developed in this thesis (and originally in the paper [Karol et al. 2012]), while the focus of both works is complementary: the first is verification while the latter is specification and implementation.

8.6. Metaprogramming and Rewriting

In this thesis, JastAdd was used for implementing composition systems based on JastAdd's API and AST rewrites. There are other metaprogramming systems that support language extensions by AST manipulation via rewriting.

Stratego/XT and TXL

Stratego/XT [Bravenboer et al. 2008] is a metaprogramming system that supports a declarative specification of program transformations. It splits up into a rewrite specification language and execution engine (Stratego), and an accompanying infrastructure and tool set (XT), which amongst others includes the SDF syntax specification language and SGLR parser generator [Visser 1997; Brand et al. 2002]. The foundation of Stratego is term rewriting [Baader and Nipkow 1999] of annotated terms using the ATerm format [Brand and Klint 2007] as a term representation.

Thus, Stratego is adequate for transforming arbitrary (abstract) syntax trees represented as terms. Transformations in Stratego are specified by a set of declarative rewrite rules consisting of a matching pattern and a replacement pattern, and optionally a rewrite condition. A rewrite rule is applicable to a term if it is matched by the matching pattern and its condition succeeds whereas the replacement is given by the unification of the match and the replacement pattern. As term rewriting is context-free, normal rewrite rules cannot use context-sensitive information. However, Stratego offers so-called *dynamic rewrite rules* to circumvent this problem. Dynamic rewrite rules generate new rewrite rules at transformation time so that a local match may imply rewrites at distant nodes in a syntax tree. One of the main problems of declarative rewriting in general is that a rewrite system given by a set of rewrite rules and an input data structure (e.g., a term or graph) may yield different results for different orders of rewrite applications (i.e., the system is not confluent [Baader and Nipkow 1999]) or may not terminate. As a tool for handling confluence and termination problems, Stratego offers programmable rewrite strategies. These strategies can be employed by transformation developers to define traversal strategies (e.g., bottom-up or top-down traversal) and rule application strategies (e.g., rule interdependency).

TXL [Cordy 2006] also is a program-transformation language and engine with roots in term rewriting. Like Stratego/XT, TXL comprises a rich tool set for solving typical tasks in source-to-source transformations. This includes a parser generator and specification language based on agile parsing, and a language for pretty printing. Rules are specified by matching patterns and replacements, whose semantics can be compared to that of Stratego rules. While TXL does not support strategies as first class concepts, it allows developers to “program” traversals, guards and scopes using first-order functional programming.

Stratego/XT and TXL are suitable tools for implementing translational semantics of language extensions and for transforming trees in general. Although they are not a direct counterpart, the composition strategies developed in this thesis were inspired by Stratego. Nevertheless, Stratego/XT and TXL have not been chosen as an implementation framework in this thesis for the following reasons. Expressing fragment component models and composition environments seems difficult as it “feels” not natural to express them as ATerms (Stratego) and rewrite rules since they introduce many context-sensitive constraints that are better expressed using RAGs. Moreover, RAG tools like JastAdd allow falling back to imperative code if it is necessary and do not force users to rely on a specific set of tools, which may not be required to solve the actual problem. Regarding fragment contracts, plain term rewriting seems to not be an adequate implementation technique since a language’s name and type analysis algorithms are to be reused and extended. In this regard, term rewriting offers no advantages [Klint et al. 2005a].

DMS

The *Design Maintenance System® (DMS)* [Baxter et al. 2004] is a commercial program transformation and analysis toolkit that has been applied in a row of large-scaled industrial projects. Similar to TXL and Stratego/XT, it features a rule-based transformation engine based on term rewriting, and tools such as parser generators and pretty printers. Besides that, it has support for a wide variety of additional analysis algorithms concerned with static semantics and symbolic execution, e.g., control-flow analysis, name analysis, call graphs. Moreover, it comprises a parallel attribute evaluation engine and equation language to implement custom analyses. Out of the box, the DMS provides implementations of frontends of more than 30 languages (cf. [Semantic Designs, Inc. 1995–2014]), including an extensible implementation of the current standard of C++ [ISO/IEC 2011] and dialects as well as current versions of Java. Besides, DMS also has cross-language support, i.e., applications may mix several languages in the same workspace.

It seems likely that SkAT could have been implemented using DMS as Java and AGs are supported. However, it is not clear from the publications how the toolkit is organized internally and what kind of attribute grammar is supported so that it is not clear if the component's interplay would be adequate. Moreover, since DMS is commercial, it has not been an option.

RACR

The *RACR* tool for reference attribute grammar controlled rewriting [Bürger 2012] is a demand-driven RAG library for Scheme [Kelsey et al. 1998] that includes a rewrite engine for program transformation. Since RACR orientates itself to JastAdd, its rewrite rules are not based on pattern matching, but are specified w.r.t. node types and expressions over attribute values (cf. Section 5.3.4). In comparison with the metaprogramming tools described above, RACR enables a strong integration of semantic analysis algorithms on reference-attributed ASTs and term rewrites with alternating phases of attribute evaluation and rewriting, which is also different from JastAdd whose rewrite and attribute evaluation phases are interleaved. Moreover, like Stratego, RACR supports basic rewrite strategies. Internally, RACR caches all attribute values and maintains a fine-grained dynamic attribute dependency graph. After a rewrite phase, these dependencies are used to compute a nearly optimal revaluation of only those attribute values that directly or indirectly depend on information that has been changed by a rewrite (e.g., an inserted node or a changed terminal value). Such automatic optimizations are hardly possible with loosely coupled tools, which would need extra manual tweaking effort to achieve the same results or must strictly separate analyses from rewriting as independent passes.

The RACR system is a good candidate for implementing well-formed ISC systems such as SkAT. In comparison to recent versions of JastAdd, it has a more fine-grained dynamic dependency graph which is nicely integrated with the rewriting API and it supports strategic rewrites. Thus, it can be expected that RACR metaprograms are likely more efficient out of the box than their JastAdd-based counterparts if many rewrites are used. Recent experiments comparing naïve implementations of hook composition in RACR and JastAdd showed that RACR indeed scales significantly better in presence of constant well-formedness checking after each composition step [Тасић 2014]. Nevertheless, still JastAdd has been chosen as an implementation framework because RACR was still in its early development stage at that time while JastAdd already was available and actively maintained frontend implementations for full programming languages such as JastAddJ.

9 Epilogue

As this is the final chapter of this thesis, it is now time to lean back and see what has been achieved. A general observation from the previous chapters is that RAGs are well-suited as a specification and implementation framework for fragment composition systems based on the ISC model. This is due to features that are essential for advanced ISC systems: typed AST structures, overlay graphs induced by reference attributes and modular specifications. Moreover, the implementation style of modern RAG evaluators is advantageous over previous implementation techniques of ISC. Dynamic and demand-driven attribute evaluation strategies are combinable with AST manipulation operations such as *bind* and *extend*. The advantages of RAGs pay-off in the SkAT system, which, on the one hand, is an *extensible* implementation framework for classic ISC systems and, on the other hand, supports the novel extensions to the classic ISC model—*well-formed* and *scalable ISC*.

The remainder of this chapter is structured as follows: Section 9.1 recapitulates the main contributions. Section 9.2 discusses open issues and gives an outlook on future work.

9.1. Achievements in Detail

In Chapter 2, the research objectives and claimed contributions (C1, ..., C4) of this thesis were discussed. In the following, realization and quality of these contributions are evaluated and their provided progress beyond the state of the art in software-composition research is justified.

C1: Well-Formed ISC

The ingredients of well-formed ISC were presented in Chapter 5. This includes a RAG-based model for fragment *composition environments*. The specified attributes handle the identification of compositional points such as slots, hooks and rudiments, which were introduced as a new category of deletable points. The basic composition operators bind, extend and extract are supported via explicit composer declarations contained in fragment environments, where attributes are used to provide the basic infrastructure of composer and point look-up as well as for checking the adequateness of syntactic types (cf. *[C1a]* in Section 2.2).

For composer interpretation, three parametrizable composition strategies were suggested which give users a better control over problems related to composition interactions (cf. *[C1b]* in Section 2.2). *Operator-determined composition* resembles term rewriting where sets of rewrite rules are matched and applied to a given fragment tree. *Point-determined composition* resembles weaving algorithms where fragments are inserted when a point is reached in program execution. *Attribute-determined composition* integrates composition execution with attribute evaluation.

Finally, *fragment contracts* were introduced as a novel concept to incorporate static semantic analyses in fragment composition. Fragment contracts allow specifying composition-related pre- and postconditions as well as invariants based on attributes and equations. Using fragment contracts, a composition system can provide better cause-related error messages in case of problems that are typically only detected in later compilation phases (cf. *[C1c]* in Section 2.2).

Progress. The combination of RAGs and ISC is beneficial since the implementation of many tasks of ISC systems can be simplified using RAGs. Moreover, strategies for composition are a novel concept to ISC and provide a precise configurable semantics to composer interpretation, which has not yet been addressed by other fragment-composition approaches so far. Fragment contracts are a step toward a better consideration of type systems and other static semantic properties of fragment languages. While this is completely new to ISC implementations, there are only a few non-language-specific approaches to software composition addressing related problems in any way (cf. Chapter 8).

C2: Scalable ISC

Scalable ISC and its ingredients have been presented in Chapter 7. Based on island grammars [Moonen 2001], *island fragment component models* were introduced as a means to support component models for only a subset of the concepts of a fragment language (or CnL) while still being able to process the whole language. Island FCMs are an instance of the basic RAG model of ISC. The model provides predefined concepts of islands and water, where islands denote constructs of the CnL and compositional constructs such as slots or composer signatures, while water covers parts that are not considered by the system (cf. *[C2b]* in Section 2.2).

Minimal ISC covers a special class of islands FCMs which do not depend on any syntactic construct of any language, only operating on strings or tokens respectively. Thus, minimal ISC

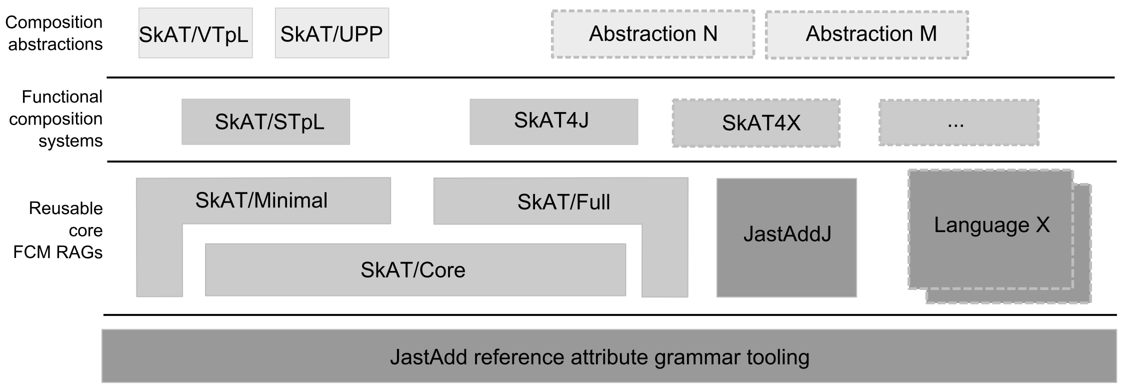


Figure 9.1.: A complete overview of SkAT.

can be used to realize string-based code generation in the style of lexical macro or template languages. As a special case, minimal FCMs only support slots as islands that may occur at arbitrary positions in a fragment (cf. *[C2a]* in Section 2.2).

Agile composition-system development was described as a generic workflow specified with BPMN. The workflow consists of a composite diagram with several complex activities concerned with syntax and semantic specification as well as FCM refinement activities which typically occur in the development of ISC systems. Since this thesis is concerned with composition, component-model refinement was detailed using an extra workflow diagram. The refinement workflow covers “small” incremental changes to FCMs including syntax and semantics based on islands, as well as “large” increments based on the full well-formed ISC approach (cf. *[C2c]* in Section 2.2).

Progress. Scalable ISC bridges the gaps between lexical, syntactic and well-formed fragment composition—a problem that has not yet been addressed by any preceding approach to ISC. Moreover, the author of this thesis is not aware of any other general approach to fragment composition covering such a broad class of fragment languages. Leveraging modular FCM specifications based on RAGs and island component models, the scalable ISC workflow suggests a novel style of agile composition-system development, enabling test and evaluation of composition-system features at early development stages without requiring a full grammar and parser of the CnL. An evaluation of the workflow in a larger scenario over a longer period remains as future work.

C3: The SkAT Framework

The SkAT framework has been introduced in Chapter 6 (SkAT/Full and applications) and Chapter 7 (SkAT/Minimal and applications). The now completed layered architecture of SkAT is replicated in Figure 9.1.

SkAT/Core provides the basic infrastructure of all derived SkAT tools and extensions based on the JastAdd RAG tool. This comprises basic RAG modules with attributes and equations for point identification and look-up, as well as the fragment environment. Augmenting that infrastructure, SkAT/Full adds composer declarations for bind, extend and extract composition operators, as well as support of composition strategies. Moreover, it supports fragment-contract specifications as JastAdd attributes and incorporates these checks into composer execution (cf. *[C3a]* in Section 2.2).

Extending SkAT/Full, SkAT4J implements a full-fledged fragment composition system for Java 1.5. It has a COMPOST-like composition API for programming composition programs in Java which is also its composition language. SkAT4J provides some fragment contracts based on pre- and postcondition that check well-formedness of expression types and avoid duplicate declarations (cf. *[C3b]* in Section 2.2).

SkAT4J was used to implement a case study of a fragment library for parallel algorithmic skeletons [Cole 1989]. The fragment library incorporates single and multi-threaded implementations of map, reduce and mapreduce skeletons which were used to generate single and multi-threaded variants of a keyword analysis on Wikipedia documents (cf. *[C3c]* in Section 2.2).

SkAT/Minimal augments SkAT/Core with support for scalable ISC. It adds island FCMs and a minimal FCM whose slot syntax can be parametrized with custom hedge symbols (cf. *[C3d]* in Section 2.2).

Extending SkAT/Minimal, the Slot Template Language (STpL), the Variant Template Language (VTpL) and the Universal Extensible Preprocessor (UPP) are three extensible island composition systems where UPP additionally is a composition language. STpL is a minimal slot language that adds a concrete syntax for slots. VTpL extends STpL with markup for variants in arbitrary fragment languages while UPP extends STpL with C-like macro syntax and preprocessor directives (cf. *[C3e]* in Section 2.2).

Progress. SkAT is the first ISC framework that supports fragment contracts and well-formed ISC enabling a cause-related error detection and reporting w.r.t. context-sensitive fragment-language constraints. Further, it is the first ISC system that can be parametrized with different composition strategies which enable users to choose the most adequate interpretation w.r.t. to their respective problems. Moreover, by now, SkAT4J is the most reliable and complete ISC system for code generation in Java as demonstrated by the business application framework (BAF) case study. Additionally, SkAT is the first ISC framework that supports scalable ISC by island FCMs and extensible composition-system construction. This is achieved by leveraging island grammars and RAG modules. Scalable ISC tools like the UPP are a testbed for combining existing composition abstractions such as fragment-language-unaware macros with language-specific syntactic constructs.

C4: Review of ISC Systems

Existing ISC systems have been discussed in Chapter 4 while SkAT/Full was introduced and compared in Chapter 6.

ISC$_{\text{core}}$ was introduced as a formal model of essential ISC constructs based on CFGs. The model abstractly defines concepts such as compositional points, fragments, and how they are related to productions and nonterminals of a given CFG by point and fragment-identification functions (cf. *[C4a]* in Section 2.2).

Besides SkAT4J, COMPOST, Reuse*wair* and Reuseware have been chosen to implement the BAF example. Their respective FCM specification and implementation approaches where related by describing how they realize the corresponding part defined by ISC$_{\text{core}}$. While with SkAT4J it was possible to fully implement all requirements of the BAF generator as proposed in Chapter 2, from the other approaches only COMPOST could implement the BAF code generator nearly as it was required (cf. *[C4b]* in Section 2.2).

A detailed comparison of SkAT/Full and the above-mentioned existing ISC approaches was provided by Table 6.5 in Chapter 6 as well as in the corresponding discussion (cf. *[C4c]* in Section 2.2).

Progress. The analysis of ISC systems has shown that not all systems provide sufficient means to fully support the concepts defined in ISC$_{\text{core}}$ (e.g., Reuse*wair*). Moreover, it was discovered that the approach of Reuseware has severe issues w.r.t. expressiveness of its composition language and specification DSLs. In contrast, RAGs are a good compromise between a completely declarative specification approach as in Reuseware and implementation-centric approaches as in COMPOST.

9.2. Outlook

This section discusses open issues that have not been addressed in this thesis and suggests potential starting points for future work.

Support of Model-Based Languages

As it is completely based on JastAdd, the current version of SkAT does not support modeling languages out of the box so that composition systems for textual or graphical modeling languages cannot be realized with SkAT. However, an integration with model-based languages has turned out as an advantage in the Reuseware/Reuse*wair* frameworks since those languages share a standardized metamodel. Hence, in future versions of SkAT, it would be beneficial if model-based languages would be supported. As a basis for this, JastEMF [Software Technology Group 2013] or a similar tool can be used as a starting point. Using JastEMF would have the positive effect that JastAdd RAGs can be used transparently with EMF-based modeling languages (cf. Section 3.2.2 and [Bürger et al. 2011]). While this would enable fragment composition systems for textual modeling languages immediately, for graphical languages more work has to be done.

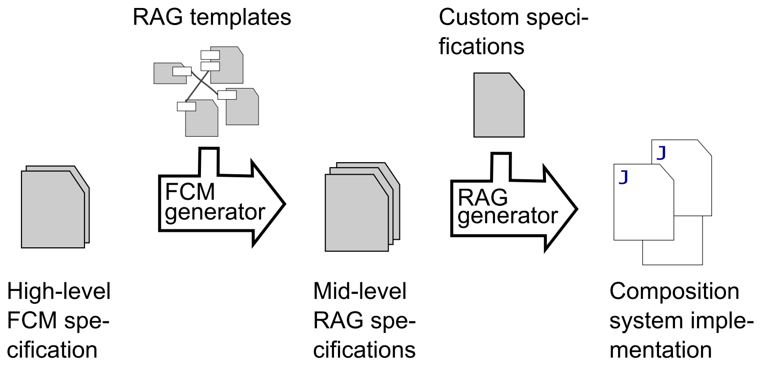

RAG templates
Custom speci-fications
FCM generator
RAG generator
High-level FCM spe-cification
Mid-level RAG spe-cifications
Composition system imple-mentation

Figure 9.2.: Generating RAG-based FCM specifications from a high-level DSL-based specification.

This includes an adequate co-composition of layout information [Johannes 2011] and a seamless integration of RAGs with graphical editors and views which are highly interactive and may interfere with the attribute evaluation [Bürger et al. 2011].

Component-Model Specification Support

While RAGs are an adequate and expressive approach for specifying FCMs, it requires developers to invest a significant effort in learning their concepts and how they are used efficiently. Moreover, composition-system developers need to know the basic attribution patterns of ISC and their interplay to be able to use them. Future RAG-based ISC frameworks may support their users in these issues by providing specification tools. For example, templates of the required attributes and equations can be supplied and parametrized by using a fragment composition system. Moreover, DSLs for FCM specification as employed in Reuseware/Reusew*air* could be used to provide high-level specifications and generate attribute grammars from this. Figure 9.2 sketches this: given a high-level specification, the FCM generator instantiates a set of internal templates to generate a RAG-based FCM specification from it. A RAG tool like JastAdd takes these specifications to generate a corresponding evaluator implementation. Additionally, if something was not expressible using the high-level language, developers may add custom RAG modules providing the required functionality. However, finding the right abstractions for such high-level DSLs turned out to be difficult in the past. Future research and development activities should consider this.

Macro Languages Based on Scalable ISC

In Chapter 7, scalable ISC has been used to implement UPP, a macro language and processor in the style of C's preprocessor. Such a tool has several potential applications. It can be extended with fragment-language-specific constructs to support partially syntax-aware macro definitions and directives. For example, it may enforce a matching number of opening and closing curly braces in a fragment in such a way that developers are informed about related issues when writing preprocessor constructs. Another opportunity could be experimenting with different textual representations of macro constructs, e.g., using Java comment syntax or XML syntax for preprocessor usage in Java or XML, as language-specific representations may be better received by developers who use such languages. Moreover, the fragment AST structure and RAGs provide static analysis capabilities that can be used to analyze dependencies between macros or to impose additional semantic constraints on preprocessor constructs.

Besides classic preprocessors, preprocessors bound to programming languages could mimic macros as specific language constructs and expand them as required, e.g., for providing an advanced module system or backend-specific variants of a parallel loop. Currently, a SkAT-based *Fortran Extensible Preprocessor (FPP)* for the Fortran programming language is developed to investigate research opportunities in that direction.

Heterogeneous Skeleton Libraries

The map and reduce skeleton library based on SkAT4J that was introduced in Section A.4.3 showed that ISC is a good choice for implementing parallel algorithmic skeletons to support parallel programming. However, the current version of the SkAT-based skeleton tooling only supports a small set of skeletons and built-in operations as single- or multi-threaded variants programmed in Java. In future work, the library should be extended with more skeletons and operations, as well as support for non-shared memory scenarios, e.g., using Java's remote method invocation capabilities. Moreover, different dimensions of heterogeneity should be considered. Other languages typically used in the development of highly parallel applications should be supported with skeleton components, e.g., C/C++ and Fortran. Including such languages is important because they are frequently used in scientific computing and are well-integrated with established standards for parallel programming such as OpenMP [OpenMP Architecture Review Board 2013]. As a second dimension of heterogeneity, different target architectures should be supported by the skeleton tooling. This can be achieved by providing variants optimized for different computing platforms. For example, a variant that runs well enough on a desktop PC may not perform adequately on embedded devices or high-performance clusters, because of their very different architectures. Hence, a heterogeneous skeleton library should support those variants by an extensible design and a model for architecture metadata.

Performance Optimizations

In favor of a good readability and extensibility, the current implementation of the SkAT framework has not been realized towards a performance-optimal evaluation of attributes. That is, almost all attributes in the composition environment are not cached and only some performance-critical attributes were supplied with performance-optimal equations. To improve here, two options are viable. The first is introducing attribute caching as a manually maintained side effect in attribute equations and composition execution. Second is using a RAG framework like RACR [Bürger

2012] with automatic caching and cache management during rewrite execution. Both approaches have their advantages: automatic cache management is transparent and does not introduce extra implementation efforts. In contrast, manual caching introduces additional implementation efforts but one may choose custom data structures and custom dependency management reducing the memory footprint of the caches. As discussed in Section 8.6, essentially in combination with heavy usages of fragment contracts caching will significantly improve performance.

Conceptual Improvements

Considering the RAG-based model of ISC developed in Chapter 5, several opportunities of improvement remain as sketched below.

The current approach to fragment contracts can be improved in several ways. Besides contracts in FCM specifications, also user-defined contracts could be supported by a specific contract language and a set of characteristic attributes. Based on such contracts, users of a composition system can specify their own constraints on specific fragments or points that must hold as pre- or postconditions before or after composition. A typical application of a user-defined contract is annotating information about the expected type of an expression or declaration to slots. Such annotations can be necessary to enable a more precise type analysis in cases of fragments with many slots [Sutton and Stroustrup 2012]. Another issue is a better consideration of cyclic dependencies between fragments induced by contracts, which are not supported by the current composition strategies. As an example of such a dependency consider two fragments: a declaration of a global variable v and a declaration of a method m. Further, assume that the variable declaration has an initialization expression that calls m and m using v in its declaration's body. If there is an active contract that checks that all uses of variables and methods in a fragment need to have a corresponding declaration, v and m will not be inserted at composition time as v requires m (which is not present) while m requires v (which is not present). As a simple solution to this kind of problem, the composition environment could provide composite composers to group interdependent fragments and contracts based on postconditions or invariants.

Another potential way of improving the model is a support of more advanced point-matching capabilities such as patterns or a point-matching language since the current approach relies on regular path names only. Moreover, users could benefit from a static analysis of composer declarations w.r.t. potential composition conflicts rising from accidental matches of the same compositional points (cf. [Karol et al. 2011]).

A Appendix

A.1. Supplements of Chapter 2

A.1.1. Full Code Examples

Listing A.1: Resulting Java class for `Employee`.

```java
public class Employee extends Person {

   private int workload;

   public int getWorkload() {
      return workload;
   }

   public void setWorkload(int workload) {
      this.workload = workload;
   }

   private Date employed;

   public Date getEmployed() {
      return employed;
   }

   public void setEmployed(Date employed) {
      this.employed = employed;
   }

   public String asString(){
      String v = "Employee";
      v+= "\\n id:" + getID();
      v+= "\\n name:" + getName();
      v+= "\\n workload:" + getWorkload();
      v+= "\\n employed:" + getEmployed();
      return v;
   }
}
```

Listing A.2: Resulting Java class for `EmployeeCustomer`.

```
public class EmployeeCustomer extends Employee
   implements IEmployee,ICustomer {

   public EmployeeCustomer(){
      discount = 20;
   }

   private int discount;

   public int getDiscount() {
      return discount;
   }

   public void setDiscount(int discount) {
      this.discount = discount; }

   public String asString(){
      String v = "EmployeeCustomer";
      v+= "\n id:" + getID();
      v+= "\n name:" + getName();
      v+= "\n workload:" + getWorkload();
      v+= "\n employed:" + getEmployed();
      v+= "\n discount:" + getDiscount();
      return v;
   }
}
```

A.1.2. Business DSL Specifications

Concrete Syntax

Listing A.3: Syntax of the business DSL in EMFText.

```
SYNTAXDEF bm
FOR <http://www.emftext.org/language/businessmodel>
START BusinessModel

RULES {
   BusinessModel ::= "application" "roles" "{" roleDefinitions* "}";
   RoleDefinition ::=
    "object" name[] ("is_a" superRoles[] ("," superRoles[])* )? (":" properties+)?;
   PropertyDefinition ::= name[] ":" type;
   Date ::= "Date" ("[" default[] "]")?;
   Hours ::= "Hours" ("[" default[] "]")?;
   Percentage ::= "Percentage" ("[" default[] "]")?;
}
```

Abstract Syntax

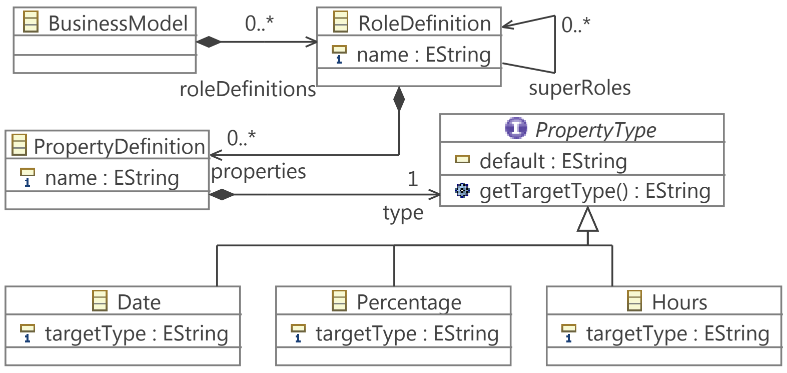


Figure A.1.: EMF Metamodel of the business DSL.

A.2. Supplements of Chapter 4

Listing A.4: Resulting `EmployeeCustomer.java` of COMPOST and RECODER using the composition program in Listing 4.5.

```
public class EmployeeCustomer extends Employee {

   public String asString() {
      String v = "EmployeeCustomer";
      v += "\n" + " id:" + getID();
      v += "\n" + " name:" + getName();
      v += "\n" + " discount:" + getDiscount();
      v += "\n" + " workload:" + getWorkload();
      v += "\n" + " employed:" + getEmployed();
      return v;
   }
   public int getDiscount() {
      return discount;
   }
   public void setDiscount(int discount) {
      this.discount = discount;
   }
   private int discount;
   public EmployeeCustomer() {
      super();
      System.out.println("setting defaults ...");
      setDiscount(20);
   }
}
```

Listing A.5: Resulting `EmployeeCustomer.java` of Reusew*air* and Java^{-}using the composition program of Listing 4.10.

```java
public class EmployeeCustomer extends Employee {
   private int discount=20;

   public void setDiscount(int discount) {
      this.discount=discount;
   }

   public int getDiscount() {
      return discount;
   }
   public String asString(){
      String v ="EmployeeCustomer";
      v+="\n" + " id:" + getID();
      v+="\n" + " name:" + getName();
      v+="\n"+" employed:"+getEmployed();
      v+="\n"+" workload:"+getWorkload();
      v+="\n"+" discount:"+getDiscount();
      return v;
   }
}
```

Listing A.6: Resulting `EmployeeCustomer.java` of Reuseware and JaMoPP using the composition program of Figure 4.11.

```java
public class EmployeeCustomer extends Person {
 public String asString(){
   String v = "EmployeeCustomer";
   v+= "\\nid:" + getID();
   v+= "\\nname:" + getName();
   return v;
 }

   private Mixin.int discount;

   public Mixin.int getDiscount() {
      return discount;
   }

   public void setDiscount( Mixin.int discount) {
      this.discount = discount;
   }
}
```

A.3. Supplements of Chapter 5

The following definition extends Definition 4.1 on fragment component models with a notion of removable points.

Definition A.1 (fragment component model with rudiments):
A fragment component model FCM with rudiments is a 7-tuple $(G,\mathcal{S},\mathcal{H},\mathcal{R},\mathcal{F},\int,\mathcal{L}_{\int})$ with

- $\mathcal{R} \subseteq N$ the finite set of *rudiment candidates*,
- $\int = \int_{\mathcal{R}} \cup \int'$, where $\int_{\mathcal{R}}$ assigns identifiers to rudiments and the compartments of $\int' = \int_{\mathcal{S}} \cup \int_{\mathcal{H}} \cup \int_{\mathcal{F}}$ are given Definition 4.1,
- $(G,\mathcal{S},\mathcal{H},\mathcal{F},\int',\mathcal{L}_{\int})$ is a fragment component model according to Definition 4.1.

If $n \in \mathcal{R}$, then there must exist $l \to \alpha\, n\, \beta \in P$, where P is the set of productions of G and $\alpha, \beta \in (\Sigma \cup N)^*$, so that l is a list nonterminal or an optional nonterminal, i.e., there also exists $l \to \alpha\, \beta \in P$. Subsequently, let $T = (V, E, \mathrm{Lab}, \mathcal{L}_\Sigma) \in \mathcal{T}(\mathcal{L}_\Sigma)$ be a syntax tree w.r.t. $G_{|n} = (N_{|n}, \Sigma_{|n}, P_{|n}, n)$, so that $n \Rrightarrow^*_G T$, and $name \in \mathcal{L}_{\int}$ is an identifier. The rudiment identification function $\int_{\mathcal{R}}$ is defined as follows:

$$\int_{\mathcal{R}}(T, v) : \mathcal{T}(\mathcal{L}_\Sigma) \times V \rightsquigarrow \mathcal{L}_{\int} \text{ so that}$$
$$\int_{\mathcal{R}}(T, v) = \begin{cases} name & \text{if } \mathrm{Lab}(v) \in \mathcal{R} \text{ and } v \in V \text{ shall be a \textit{rudiment} in } T, \\ \perp & otherwise. \end{cases}$$

◇

Complementing the above extended FCM definition, Definition A.2 specifies the *extract* (de-) composer, which can be used to remove fragments at rudiments.

Definition A.2 (extract composer):
Let $FCM = (G, \mathcal{S}, \mathcal{H}, \mathcal{R}, \mathcal{F}, \int, \mathcal{L}_{\int})$ be a fragment component model with rudiments, where $G = (N, \Sigma, P, S)$ is a CFG. The *extract* composer $\bar{\pi}_{FCM}(T, m) : \mathcal{T}(\mathcal{L}_\Sigma) \times (\mathcal{T}(\mathcal{L}_\Sigma) \to 2^V) \rightsquigarrow \mathcal{T}(\mathcal{L}_\Sigma)$ is a partial function removing subtrees from a given FCM fragment component $T = (V, E, \mathrm{Lab}, \mathcal{L}_\Sigma)$ with $T \in \mathcal{T}(\mathcal{L}_\Sigma)$ at points $\mathrm{m}(T) \subset V$. The result of applying $\bar{\pi}_{FCM}$ is a valid syntax tree w.r.t. G or undefined ($\perp$), if the transformation cannot yield a valid tree.

In the following, let $f = \mathrm{Lab}(R)$ where $f \Rrightarrow^*_G T$, $\mathcal{Q} = \mathrm{m}(T)$ and R is the root of T. $\bar{\pi}_{FCM}$ is given as follows:

$$\bar{\pi}_{FCM}(T, \mathrm{m}) = \begin{cases} T[\mathcal{Q}/\emptyset] = T' & \text{if } \mathcal{Q} \neq \emptyset \\ & \quad \text{and } f \in \mathcal{F} \\ & \quad \text{and } \int_{\mathcal{F}}(T) \neq \perp \\ & \quad \text{and } \forall v \in \mathcal{Q} : \int_{\mathcal{R}}(T, v) \neq \perp \wedge v \neq R \\ & \quad \text{and } f \Rrightarrow^*_G T', \\ T & \text{if } \mathcal{Q} = \emptyset, \\ \perp & otherwise. \end{cases}$$

$T_1[\mathcal{Q}/\emptyset]$ removes all $k = |\mathcal{Q}|$ subtrees (rudiments) $t_i = l_i[\ldots] = (V_i, E_i, \mathrm{Lab}_i, \mathcal{L}_\Sigma)$ in $T = f[\ldots, T_1, \ldots, t_i, \ldots, t_k, \ldots]$ where $R_i \in \mathcal{Q}$ is the root of T and $1 \leq i \leq k$. Hence, $T' = (V', E', \mathrm{Lab}', \mathcal{L}_\Sigma)$ with $V' = V \setminus \bigcup_{i=1}^k V_i$, $E' = E \setminus \bigcup_{i=1}^k E_i$, $\mathrm{Lab}' = \mathrm{Lab} \setminus \bigcup_{i=1}^k \mathrm{Lab}_i$ and root $R' = R$. ◇

A.4. Supplements of Chapter 6

A.4.1. Specifications of SkAT/Core

Listing A.7: Fragment API from the `FragmentSystem.jadd` module of SkAT/Core.

```
public abstract class FragmentSystem extends java.lang.Object {
   // Returns the (absolute) base/output/input URI.
   public String getBaseURI() { ... }
   public String getOutURI() { ... }
   public String getInURI() { ... }

   // Obtain the underlying composition environment.
   public CompositionEnvironment getEnv() { ... }

   // Adds a fragment to the environment.
   public void addFragment(Box fragment) { ... }

   // Adds a list of fragments to the environment.
   public void addFragment(Collection<Box> fragments) { ... }

   // Uses the environment's look-up functionality to obtain the fragment with
   // the provided 'name'.
   public Box findFragment(String name){ ... }

   // Create a copy of the fragment with 'oldName', set its name to 'newName',
   // add it to the environment and return the new box.
   public Box copyBox(String oldName, String newName){ ... }

   // Persist all fragments using their input names or their output names if specified.
   public void persistFragments() throws IOException { ... }

   // Persist all fragments whose names match the given pattern.
   public void persistFragments(String regex) throws IOException{ ... }

   // Composition systems implement this in a subclass to adapt their pretty printer.
   public abstract void doPersist(Box fragment) throws IOException;

   // Composition systems need to implement this in a subclass to adapt their
   // initialization needs such as parsing.
   protected abstract CompositionEnvironment createEnvironment() throws IOException;

   // Composition systems need to implement this to define when the underlying
   // environment is in consistent state, e.g., if validating attributes say so.
   public abstract boolean isConsistent();

   // (De-)activate recovery mode. In recover mode the system remains healthy unless
   // a failure status is pushed to the status stack. With recovery mode enabled,
   // the system may execute compositions even if an error is on the status stack.
   public void setRecoverMode(boolean mode){ ... }

   // If the environment is healthy, changes/compositions are allowed to be processed.
   public boolean isEnvHealthy(){ ... }

   // Prints status information on the composition system to console.
   public void printStatus(){ ... }
}
```

Listing A.8: Excerpt from the `Rudiments.jrag` specification of SkAT/Core.

```
aspect Rudiments{
   // Default attributes, determining rudiments. By default, slots which are direct
   // children of an Opt or List are rudiments by default.
   syn boolean ASTNode.isRudiment() =
         isSlot()?(hasParent()&&(getParent().isList()||getParent().isOpt())):false;
   syn String ASTNode.rudimentName() = isSlot()?slotName():"";
}

aspect RudimentCollection{
   // Attribute decls for collecting rudiments.
   syn java.util.List<ASTNode> ASTNode.collRudiments();
   syn ASTNode ASTNode.findRudiment(Pattern qName);
   syn java.util.List<ASTNode> ASTNode.findRudiments(Pattern qName);
   ...
}
```

Listing A.9: Excerpt from the `Hooks.jrag` specification of SkAT/Core.

```
aspect Hooks {
   // The hook determining attributes. Since lists are transparent in JastAdd, they
   // cannot have inherited attributes. Thus, these are simulated using synthesized
   // ones which delegate decisions to the parent.
   syn boolean ASTNode.isHook() = false;
   syn String ASTNode.hookName() = "";
   syn String[] ASTNode.hookAliases() = new String[0];
   syn int List.hookIndex(String hookName);

   // Mirrored attributes defined at the parent of a hook node taking the hook-list
   // node as an argument.
   syn boolean ASTNode.isHook(List hook) = false;
   syn String ASTNode.hookName(List hook) = "";
   syn String[] ASTNode.hookAliases(List hook) = new String[0];
   syn int ASTNode.hookIndex(List hook, String hookName) = 0;
   syn Class[] ASTNode.compatibleFragmentTypes(List hook)
                        = new Class[]{BottomFragmentType.class};

   // The gluing default equations.
   eq List.isHook()
      = isInEnvironment()&&(owningBox()!=null)?getParent().isHook(this):false;
   eq List.hookName() = getParent().hookName(this);
   eq List.hookAliases() = getParent().hookAliases(this);
   eq List.hookIndex(String hookName) = getParent().hookIndex(this,hookName);
   eq List.compatibleFragmentTypes() = getParent().compatibleFragmentTypes(this);
}

aspect HookCollection{
   // Attribute declarations for collecting hooks
   syn java.util.List<List> ASTNode.collHooks();
   syn List ASTNode.findHook(QRef qName);
   syn java.util.List<List> ASTNode.findHooks(QRef qName);
   ...
}
```

A.4.2. Detailed SkAT4J Specifications

Figure A.2 contains a graphical representation of SkAT4J's box model.

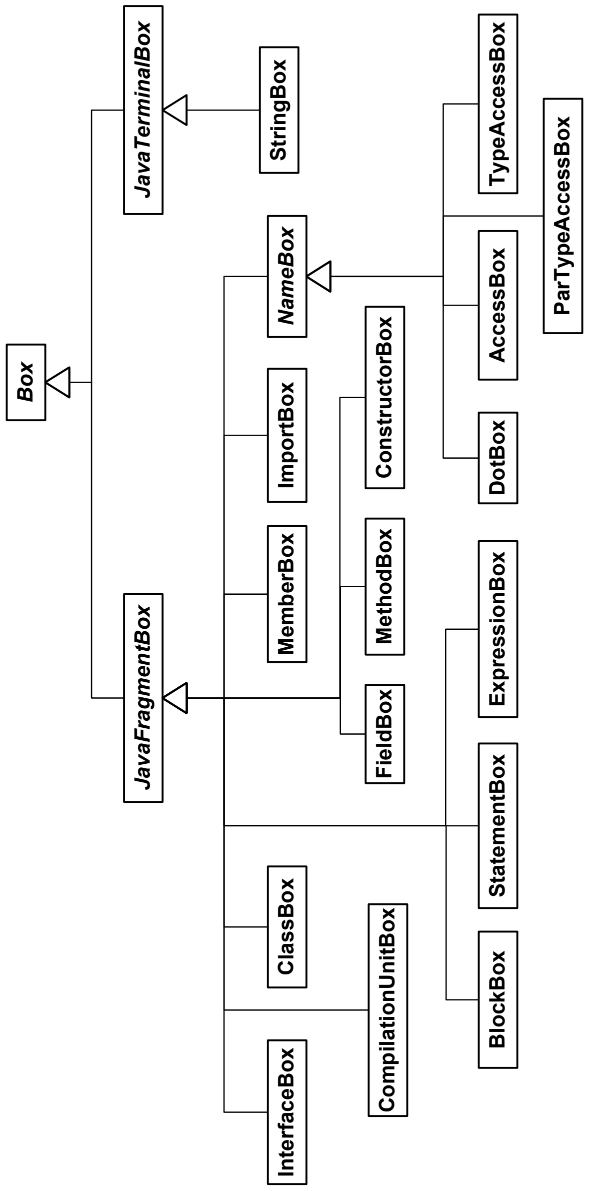


Figure A.2.: SkAT4J's *Boxology* in UML class-diagrammatic representation.

Listing A.10: `Slots.jrag` specification of SkAT4J.

```
aspect Slots {
   eq ExprSlot.isSlot() = true;
   eq ExprSlot.slotName() = isSlot()?extract(getSlotName(),"[[","]]"):"";
   eq ExprSlot.compatibleFragmentTypes() = new Class[]{Expr.class};
   eq ExprSlot.checkContractPre(Object fragment) = true;
   eq ExprSlot.checkContractPost(Object fragment) {
      if(fragment instanceof Expr){
         Object result = assertCompatibleType((Expr)fragment);
         if(result!=Boolean.TRUE)
            return result;
      }
      return true;
   }

   eq StmtSlot.isSlot() = true;
   eq StmtSlot.slotName() = isSlot()?extract(getSlotName(),"[[","]]"):"";
   eq StmtSlot.compatibleFragmentTypes() = new Class[]{Stmt.class};

   eq MemberDeclSlot.isSlot() = true;
   eq MemberDeclSlot.slotName() = isSlot()?extract(getSlotName(),"[[","]]"):"";
   eq MemberDeclSlot.compatibleFragmentTypes() = new Class[]{MemberDecl.class};

   eq TypeAccessSlot.isSlot() = true;
   eq TypeAccessSlot.slotName() = isSlot()?extract(getSlotName(),"[[","]]"):"";

   eq TypeAccessSlot.compatibleFragmentTypes()
     = new Class[]{TypeAccess.class,ParseName.class /* 3 more */};
   eq GenericTypeAccessSlot.compatibleFragmentTypes()
     = new Class[]{ParTypeAccess.class,AmbiguousAccess.class,ParseName.class};
   eq ArrayTypeAccessSlot.compatibleFragmentTypes()
     = new Class[]{ArrayTypeAccess.class,AmbiguousAccess.class,ParseName.class};

   eq TypeVariableSlot.isSlot() = true;
   eq TypeVariableSlot.slotName() = isSlot()?extract(getID(),"[[","]]"):"";
   eq TypeVariableSlot.compatibleFragmentTypes()
     = new Class[]{TypeVariable.class,ParseName.class};

   eq VarAccessSlot.isSlot() = true;
   eq VarAccessSlot.slotName() = isSlot()?extract(getSlotName(),"[[","]]"):"";
   eq VarAccessSlot.compatibleFragmentTypes()
    = new Class[]{VarAccess.class,ParseName.class,AmbiguousAccess.class};

   eq MethodDecl.isSlot() = name().endsWith("Slot");
   eq MethodDecl.getChild(int i).isInSlot() = isSlot();
   eq MethodDecl.slotName() = isSlot()?name().substring(0,name().length()-4):"";

   // Terminal slots:
   inh TypeDecl TypeDeclNameSlot.SlotOwner();
   eq SlotableClassDecl.getTypeDeclNameSlot().SlotOwner() = this;
   eq SlotableInterfaceDecl.TypeDeclNameSlot().SlotOwner() = this;

   eq TypeDeclNameSlot.isSlot() = isHedged(SlotOwner().getID(),"[[","]]"); // eq. (1)
   eq TypeDeclNameSlot.slotName()
    = isTerminalSlot()?extract(SlotOwner().getID(),"[[","]]"):""; // eq. (2)
   eq TypeDeclNameSlot.compatibleFragmentTypes()
    = new Class[]{StringValue.class,ParseName.class,AmbiguousAccess.class}; // (3)
```

```

  // eq. (4), using syn attribute instead of inh, because usage is not supported.
  syn MethodDecl MethodDeclNameSlot.SlotOwner() = (MethodDecl)getParent().getParent();
  eq MethodDeclNameSlot.isSlot() = ... // like eq. (1)
  eq MethodDeclNameSlot.slotName() = ... // like eq. (2)
  eq MethodDeclNameSlot.compatibleFragmentTypes() = ... // like eq. (3)

  syn ConstructorDecl ConstructorDeclNameSlot.SlotOwner() = ... // like eq. (4)
  eq ConstructorDeclNameSlot.isSlot() = ... // like eq. (1)
  eq ConstructorDeclNameSlot.slotName() = ... // like eq. (2)
  eq ConstructorDeclNameSlot.compatibleFragmentTypes() = ... // like eq. (3)

  syn MethodAccess MethodAccessNameSlot.SlotOwner() = ... // like eq. (4)
  eq MethodAccessNameSlot.isSlot() = ... // like eq. (1)
  eq MethodAccessNameSlot.slotName() = ... // like eq. (2)
  eq MethodAccessNameSlot.compatibleFragmentTypes() = ... // like eq. (3)

  syn StringLiteral StringValueSlot.SlotOwner() = ... // like eq. (4)
  eq StringValueSlot.isSlot() = slotName()!=null;
  eq StringValueSlot.slotName(){
   String literal = SlotOwner().getLITERAL();
   int open = literal.indexOf("[[");
   if(open>=0){
      int close = literal.indexOf("]]");
      if(close>open){
         return literal.substring(open+2,close);
      }
   }
   return null;
  }
  eq StringValueSlot.compatibleFragmentTypes()
   = new Class[]{StringValue.class,Literal.class,Access.class};

  syn FieldDeclaration FieldDeclNameSlot.SlotOwner() = ... // like eq. (4)
  eq FieldDeclNameSlot.isSlot() = ... // like eq. (1)
  eq FieldDeclNameSlot.slotName() = ... // like eq. (2)
  eq FieldDeclNameSlot.compatibleFragmentTypes() = ... // like eq. (3)

  syn VariableDecl VariableDeclNameSlot.SlotOwner() = ... // like eq. (4)
  eq VariableDeclNameSlot.isSlot() = ... // like eq. (1)
  eq VariableDeclNameSlot.slotName() = ... // like eq. (2)
  eq VariableDeclNameSlot.compatibleFragmentTypes() = ... // like eq. (3)

  syn VariableDeclaration VariableDeclarationNameSlot.SlotOwner() =... // l. eq. (4)
  eq VariableDeclarationNameSlot.isSlot() = ... // like eq. (1)
  eq VariableDeclarationNameSlot.slotName() = ... // like eq. (2)
  eq VariableDeclarationNameSlot.compatibleFragmentTypes() = ... // like eq. (3)

  syn ParameterDeclaration ParameterDeclNameSlot.SlotOwner() = ... // like eq. (4)
  eq ParameterDeclNameSlot.isSlot() = ... // like eq. (1)
  eq ParameterDeclNameSlot.slotName = ... // like eq. (2)
  eq ParameterDeclNameSlot.compatibleFragmentTypes() = ... // like eq. (3)
}
```

Listing A.11: `Hooks.jrag` specification of SkAT4J.

```
aspect Hooks {
   // "statements" hook.
   syn boolean Block.isMethodRootBlock() =
      getParent().getParent() instanceof MethodDecl;
   eq Block.isHook(List hook) = hook == getStmtList() && !isInSlot() ;
   eq Block.hookName(List hook) = "statements";
   eq Block.hookAliases(List hook) {
      if(isMethodRootBlock()){
         return new String[] {"methodEntry","methodExit","statementsEnd"};
      }
      else if(!isMethodRootBlock() && endsWithReturn()){
         return new String[] {"methodExit" + numReturns(),"statementsEnd"};
      }
      else /* All other situations, e.g., constructors, static blocks. */
         return new String[] {"statementsEnd"};
   }
   eq Block.hookIndex(List hook, String hookName) {
      if("methodEntry".equals(hookName) || "statements".equals(hookName)){
         return 0;
      }
      else if (hookName.startsWith("methodExit") && endsWithReturn()){
         return hook.numChildren()-1;
      }
      else /* All other situations: length of the list. */
         return hook.numChildren();
   }
   eq Block.compatibleFragmentTypes(List hook) = new Class[]{Stmt.class};

   // "imports" hook.
   syn boolean CompilationUnit.isImportDeclHook(List hook) =
         getImportDeclList()==hook && !isSlot() && !isInSlot();

   eq CompilationUnit.isHook(List hook) = isImportDeclHook(hook);
   eq CompilationUnit.hookName(List hook)
         = isImportDeclHook(hook)?"imports":super.hookName(hook);
   eq CompilationUnit.hookIndex(List hook, String hookName) = hook.numChildren();
   eq CompilationUnit.compatibleFragmentTypes(List hook) = isImportDeclHook(hook)?
            new Class[]{ImportDecl.class}:new Class[]{BottomFragmentType.class};

   // "members" hook.
   syn boolean TypeDecl.isMembersHook(List hook) =
         getBodyDeclList()==hook && !isSlot() && !isInSlot();

   eq TypeDecl.isHook(List hook) = isMembersHook(hook);
   eq TypeDecl.hookName(List hook) =
         isMembersHook(hook)?"members":super.hookName(hook);
   eq TypeDecl.hookAliases(List hook) = isMembersHook(hook)?
         new String[]{"membersEntry","membersExit"}:super.hookAliases(hook);
   eq TypeDecl.hookIndex(List hook, String hookName)
      = (isMembersHook(hook)&&!"membersEntry".equals(hookName))?hook.numChildren():0;
   eq TypeDecl.compatibleFragmentTypes(List hook) = isMembersHook(hook)?
         new Class[]{BodyDecl.class}:new Class[]{BottomFragmentType.class};

   eq TypeDecl.checkContractPre(List hook, Object fragment){
      Object contractValue = Boolean.TRUE;
      if(isMembersHook(hook)){
         if(fragment instanceof MethodDecl)
            contractValue = assertNotDeclared((MethodDecl)fragment);
         if(contractValue == Boolean.TRUE && fragment instanceof FieldDeclaration)
            contractValue = assertNotDeclared((FieldDeclaration)fragment);
         if(contractValue == Boolean.TRUE)
            contractValue = assertVariablesProvided((ASTNode)fragment);
      }
      return contractValue;
   }
   eq TypeDecl.checkContractPost(List hook, Object fragment) = true;

   // "implements" hook.
   syn boolean ClassDecl.isImplementsHook(List hook) =
      getImplementsListNoTransform()==hook && !isSlot() && !isInSlot();

   eq ClassDecl.isHook(List hook) = isImplementsHook(hook)||super.isHook(hook);
   eq ClassDecl.hookName(List hook) =
      isImplementsHook(hook)?"implements":super.hookName(hook);
   eq ClassDecl.hookIndex(List hook, String hookName)
      = isImplementsHook(hook)? hook.numChildren(): super.hookIndex(hook,hookName);
   eq ClassDecl.compatibleFragmentTypes(List hook) = isImplementsHook(hook)?
     new Class[]{TypeAccess.class,ParseName.class}:super.compatibleFragmentTypes(hook);

   // "paramters" hook.
   syn boolean MethodDecl.isParametersHook(List hook) =
      getParameterList()==hook && !isSlot() && !isInSlot();

   eq MethodDecl.isHook(List hook) = isParametersHook(hook);
   eq MethodDecl.hookName(List hook) =
      isParametersHook(hook)?"parameters":super.hookName(hook);
   eq MethodDecl.hookIndex(List hook, String hookName)
       = isParametersHook(hook)?hook.numChildren():0;
   eq MethodDecl.compatibleFragmentTypes(List hook) = isParametersHook(hook)?
         new Class[]{ParameterDeclaration.class}:super.compatibleFragmentTypes(hook);
}
```

Listing A.12: `Assertions.jrag` specification of SkAT4J.

```
aspect FragmentAssertions{
   // Assert that method is not declared in this TypeDecl.
   syn Object TypeDecl.assertNotDeclared(MethodDecl decl){
      String signature = decl.signature();
      if(localMethodsSignatureMap().containsKey(signature))
         return "Sig. '" + signature + "' already declared in type '" + getID() + "'.";
      return true;}
   // Assert that method is not declared in this TypeDecl or super type.
   syn Object TypeDecl.assertNotDeclaredGlobal(MethodDecl decl){
      String signature = decl.signature();
      if(methodsSignatureMap().containsKey(signature))
        return "Sig. '" + signature + "' already declared in type '" + getID()
          + "' or super type.";
      return true;}

   // Assert that field is not declared in this TypeDecl.
   syn Object TypeDecl.assertNotDeclared(FieldDeclaration decl) {
      String name = decl.getID();
      if(localFieldsMap().containsKey(name))
        return decl.assertionID() + " already declared in type '" + getID() + "'.";
      return true;}
   // Assert that field is not declared in this TypeDecl or super type.
   syn Object TypeDecl.assertNotDeclaredGlobal(FieldDeclaration decl) {
      String name = decl.getID();
      if(memberFieldsMap().containsKey(name))
         return decl.assertionID() + " already declared in type '" + getID()
            + "' or super type.";
      return true;
   }
   // Assert that all variables used in fgmt are provided by this TypeDecl.
   syn Object TypeDecl.assertVariablesProvided(ASTNode fgmt){
      Collection<String> requires = fgmt.danglingVars(null);
      String missing = "";
      for(String var:requires){
         if(memberFields(var).isEmpty()&&lookupVariable(var).isEmpty())
            missing += " '" + var + "'";
      }
      if(!missing.equals(""))
         return fgmt.assertionID() + " requires variable(s)" + missing + ".";
      return true;
   }
   // Assert that all variables called in fgmt are provided by this TypeDecl.
   syn Object TypeDecl.assertMethodsProvided(ASTNode fgmt){
      Collection<String> requires = fgmt.danglingCalls(null);
      String missing = "";
      for(String cal:requires){
         if(memberMethods(cal).isEmpty()&&lookupVariable(cal).isEmpty())
            missing += " '" + cal + "'";
      }
      if(!missing.equals(""))
         return fgmt.assertionID() + " requires method(s)" + missing + ".";
      return true;
   }
   // Assert an expression's type compatibility comparing its both sides' types.
   syn Object ExprSlot.assertCompatibleType(Expr fgmt){
      Expr expr = getParent() != null?this:fgmt; // Pre- or postcondition situation?
      if(expr.getParent() instanceof AssignExpr){
         AssignExpr parent = (AssignExpr)expr.getParent();
         if(parent.getSource() == expr){
            TypeDecl sourceType = fgmt.type();
            TypeDecl destType = parent.getDest().type();
            if(sourceType == expr.unknownType() || destType==expr.unknownType()
                || !sourceType.wideningConversionTo(destType))
                 return "Type of Expr fragment '" + sourceType.getID()
                       + "' does not fit left-hand type '" + destType.getID() + "'.";
         }
         else if(parent.getDest() == expr){
            TypeDecl sourceType = parent.getSource().type();
            TypeDecl destType = fgmt.type();
            if(sourceType == expr.unknownType() || destType==expr.unknownType()
                || !sourceType.wideningConversionTo(destType))
               return "Type of Expr fragment '" + sourceType.getID()
                     + "' does not fit right-hand type '" + destType.getID() + "'.";
      } }
      return true;}
}
```

Listing A.13: Excerpt from the `fragments.parser` Beaver grammar of SkAT4J (AST construction has been removed).

```
ASTNode goal = java_fragment_box | java_fragment_list;

// Boxes:
JavaFragmentBox java_fragment_box = method_declaration.decl |
  constructor_declaration.decl | field_declaration.decl | block.block |
  if_then_statement.stmt | if_then_else_statement.stmt |
  while_statement.stmt | for_statement.stmt |
  java_fragment_box_prefixed | java_fragment_box_prefixed_type ;

JavaFragmentBox java_fragment_box_prefixed = FRAGMENT block_statement.stmt |
  FRAGMENT literal.expr | FRAGMENT type.type | FRAGMENT import_declaration.decl ;

JavaFragmentBox java_fragment_box_prefixed_type= FGMT_STATEMENT block_statement.stmt |
  FGMT_LITERAL literal.expr | FGMT_EXPRESSION expression.expr |
  FGMT_IMPORT import_declaration.decl | FGMT_FIELD field_declaration.decl |
  FGMT_VARIABLE local_variable_declaration_statement.stmt | FGMT_BLOCK block.block |
  FGMT_METHOD method_declaration.decl | FGMT_NAME name.name | FGMT_TYPE type.type ;

List java_fragment_list = java_fragment_box.box1 java_fragment_box.box2 |
  java_fragment_list.l java_fragment_box.box ;

// Slots:
Stmt statement = SLOTIDENTIFIER SEMICOLON ; //StmtSlot
ClassDecl class_declaration
  = modifiers.m? CLASS SLOTIDENTIFIER super.s? interfaces.i? class_body.b ;
InterfaceDecl interface_declaration
  = modifiers.m? INTERFACE SLOTIDENTIFIER extends_interfaces.i? interface_body.b ;
MethodDecl method_header = modifiers.m? type.t SLOTIDENTIFIER LPAREN
                     formal_parameter_list.l? RPAREN dims.d? throws.tl?
 | modifiers.m? VOID SLOTIDENTIFIER LPAREN formal_parameter_list.l? RPAREN throws.tl?;
ParameterDeclaration formal_parameter = modifiers.m? type.t SLOTIDENTIFIER dims.d? ;
VariableDecl variable_declarator_id = SLOTIDENTIFIER dims.d? ;
ConstructorDecl constructor_declaration
  = modifiers.m? SLOTIDENTIFIER LPAREN formal_parameter_list.pl? RPAREN throws.tl?
      LBRACE explicit_constructor_invocation.c? block_statements.l? RBRACE;
BodyDecl class_member_declaration = SLOTIDENTIFIER SEMICOLON ;

Expr class_instance_creation_expression
  = NEW normal_type_slot.t LPAREN argument_list.l? RPAREN ;
Expr array_creation_uninit
  = NEW normal_type_slot.t dim_exprs.d | NEW normal_type_slot.t dim_exprs.d dims.e ;
Expr assignment_expression = SLOTIDENTIFIER ;
Expr primary = LPAREN SLOTIDENTIFIER RPAREN ;
Expr assignment = SLOTIDENTIFIER EQ assignment_expression.source ;

Access class_type = SLOTIDENTIFIER ;
Access interface_type = SLOTIDENTIFIER ;
Access method_invocation = SLOTIDENTIFIER LPAREN argument_list.l? RPAREN ;
Access field_access = primary.p DOT SLOTIDENTIFIER ;
Access type = normal_type_slot | array_type_slot ;
Access normal_type_slot = SLOTIDENTIFIER ;
Access array_type_slot = SLOTIDENTIFIER dims.d ;

TypeVariable type_parameter_1 = SLOTIDENTIFIER GT ;
Access reference_type_1
```

```
  = SLOTIDENTIFIER GT | SLOTIDENTIFIER LT type_argument_list_2.l;
Access reference_type_2
  = SLOTIDENTIFIER RSHIFT | SLOTIDENTIFIER LT type_argument_list_3.l;
Access reference_type_3 = SLOTIDENTIFIER URSHIFT ;
```

A.4.3. Map and Reduce Skeletons

Listing A.14: Contents of `simple_map.jbx`.

```
{
   // Initialize a list to maintain output results.
   List<[[OUT_TYPE]]> results = new LinkedList<[[OUT_TYPE]]>();
   // Request the first input data chunk.
   [[IN_TYPE]] inValue = [[IN_PORT_INIT]];

   while(inValue!=null){
      // Call map operation with input data.
      [[OUT_TYPE]] result = [[MAP_OP]](inValue);
      results.add(result);
      // Request next chunk of data.
      inValue = [[IN_PORT_CONT]];
   }
   // Pass results to the output port.
   [[OUT_PORT]] = results;
}
```

Listing A.15: Contents of `simple_reduce.jbx`.

```
{
   // Initialize result sink.
   [[OUT_TYPE]] result = [[OUT_INIT]];
   // Request the first input data chunk.
   [[IN_TYPE]] inValue = [[IN_PORT_INIT]];

   while(inValue!=null){
      // Call reduce operation with input data.
      [[REDUCE_OP]](inValue,result);
      // Request next chunk of data.
      inValue = [[IN_PORT_CONT]];
   }
   // Pass reduced value to the output port.
   [[OUT_PORT]] = result;
}
```

Listing A.16: Contents of `concurrent_map.jbx`.

```
{
   // Allocate worker threads:
   ExecutorService executor = Executors.newFixedThreadPool([[WORKERS]]);
   // Futures contain the thread output.
   List<Future<[[OUT_TYPE]]>> futures = new LinkedList<Future<[[OUT_TYPE]]>>();
   // Initialize a list to maintain output results.
   final List<[[OUT_TYPE]]> results = new LinkedList<[[OUT_TYPE]]>();
   // Request the first input data chunk.
   [[IN_TYPE]] inValue = [[IN_PORT_INIT]];

```

```
    // As long as input data is available:
    while(inValue!=null){
        final [[IN_TYPE]] inArg = inValue;
        futures.add(executor.submit(
                new Callable<[[OUT_TYPE]]>(){
                        public [[OUT_TYPE]] call() throws Exception {
                            // Call map operation with input data.
                            [[OUT_TYPE]] result = [[MAP_OP]](inArg);
                            synchronized (results) {
                                results.add(result);
                            }
                            return result;
                        }}
                    ));
        // Request next chunk of data.
        inValue = [[IN_PORT_CONT]];
    }

    executor.shutdown();
    while(!executor.isTerminated()){
        // As long as there's work in the thread pool: wait.
    }
    // Pass reduced value to the output port.
    [[OUT_PORT]] = results;
}
```

Listing A.17: Contents of `concurrent_mapreduce.jbx`.

```
{
    // Allocate worker threads:
    ExecutorService executor = Executors.newFixedThreadPool([[WORKERS]]);
    // Futures contain the thread output and the map results.
    List<Future<[[OUT_TYPE]]>> futures = new LinkedList<Future<[[OUT_TYPE]]>>();
    // Initialize result sink.
    [[OUT_TYPE]] reduceResult = [[OUT_INIT]];
    // Request the first input data chunk.
    [[IN_TYPE]] inValue = [[IN_PORT_INIT]];

    // Map related tasks.
    // As long as input data is available:
    while(inValue!=null){
        final [[IN_TYPE]] inArg = inValue;
        // Submit work to workers.
        futures.add(executor.submit(
                new Callable<[[OUT_TYPE]]>(){
                        public [[OUT_TYPE]] call() throws Exception {
                        // Call map operation with input data.
                        [[OUT_TYPE]] mapResult = [[MAP_OP]](inArg);
                        return mapResult;
                    }}
                ));
        // Request next chunk of data.
        inValue = [[IN_PORT_CONT]];
    }

    // Reduce-related tasks.
    // As long as not all threads were reduced:
    while(futures.size()>1){
      Future<[[OUT_TYPE]]> current1 = futures.remove(0);
      if(current1.isDone()){
         // Request processed data from thread (1).
         final [[OUT_TYPE]] arg1 = current1.get();
         while(futures.size()>0){
            // Request processed data from thread (2).
            Future<[[OUT_TYPE]]> current2 = futures.remove(0);
            if(current2.isDone()){
               final [[OUT_TYPE]] arg2 = current2.get();
               // Submit work to workers.
               futures.add(executor
                     .submit(new Callable<[[OUT_TYPE]]>() {
                        public [[OUT_TYPE]] call() throws Exception {
                           // Call map operation on input data.
                           [[REDUCE_OP]](arg2,arg1);
                           return arg1;
                        }
                     }));
              break;
            }
            else{
               futures.add(current2);
            }
         }
      }
      else{
         futures.add(current1);
      }
   }

   executor.shutdown();
   Future<[[OUT_TYPE]]> lastResult = futures.get(0);
   while (!lastResult.isDone()) {
   }
   // Request final result.
   reduceResult = lastResult.get();

   // Pass reduced value to the output port.
   // Use a list output type to provide a compatible type to map.
   [[OUT_PORT]] = java.util.Collections.singletonList(reduceResult);
}
```

Listing A.18: Single-threaded MapReduce variant `WikiCrawler1.java`.

```
public class WikiCrawler1 {
  public Map<String, Integer> compute(Collection<String> in) throws Exception {
    Iterator<String> it = in.iterator();
    Collection<Map<String, Integer>> mapResult = null;
    {
      List<Map<String, Integer>> results = new LinkedList<Map<String, Integer>>();
      String inValue = it.hasNext() ? it.next() : null;
      while(inValue != null){
        Map<String, Integer> result = map(inValue);
        results.add(result);
        inValue = it.hasNext() ? it.next() : null;
      }
      mapResult = results;
    }
    Iterator<Map<String, Integer>> mIt = mapResult.iterator();
    Map<String, Integer> reduceResult = null;
    {
      Map<String, Integer> result = new HashMap<String, Integer>();
      Map<String, Integer> inValue = mIt.hasNext() ? mIt.next() : null;
      while(inValue != null){
        reduce(inValue, result);
        inValue = mIt.hasNext() ? mIt.next() : null;
      }
      reduceResult = result;
    }
    return reduceResult;
  }
  public static Map<String, Integer> map(String text) {
   ... 1:1 from map_op.jbx ...
  }
  public static boolean isWS(char c) {
   ... 1:1 from map_op.jbx
  }
  public static void reduce(Map<String, Integer> in1, Map<String, Integer> inout2) {
   ... 1:1 from reduce_op.jbx
  }
}
```

Listing A.19: Dual-threaded MapReduce variant `WikiCrawler2.java`.

```
public class WikiCrawler2 {
  public Map<String, Integer> compute(Collection<String> in) throws Exception {
    Iterator<String> it = in.iterator();
    Collection<Map<String, Integer>> mapResult = null;
    {
      ExecutorService executor = Executors.newFixedThreadPool((2));
      List<Future<Map<String, Integer>>> futures
                    = new LinkedList<Future<Map<String, Integer>>>();
      Map<String, Integer> reduceResult = new HashMap<String, Integer>();
      String inValue = it.hasNext() ? it.next() : null;
      while(inValue != null){
        final String inArg = inValue;
        futures.add(executor.submit(new Callable<Map<String, Integer>>() {
           public Map<String, Integer> call() throws Exception {
             Map<String, Integer> mapResult = map(inArg);
             return mapResult;
           }
        }));
        inValue = it.hasNext() ? it.next() : null;
      }
      while(futures.size() > 1){
        Future<Map<String, Integer>> current1 = futures.remove(0);
        if(current1.isDone()) {
          final Map<String, Integer> arg1 = current1.get();
          while(futures.size() > 0){
            Future<Map<String, Integer>> current2 = futures.remove(0);
            if(current2.isDone()) {
              final Map<String, Integer> arg2 = current2.get();
              futures.add(executor.submit(new Callable<Map<String, Integer>>() {
                 public Map<String, Integer> call() throws Exception {
                   reduce(arg2, arg1);
                   return arg1;
```

```
         }
       }));
       break ;
      }
      else {
       futures.add(current2);
      }
     }
    }
    else {
     futures.add(current1);
    }
   }
   executor.shutdown();
   Future<Map<String, Integer>> lastResult = futures.get(0);
   while(!lastResult.isDone()){
   }
   reduceResult = lastResult.get();
   mapResult = java.util.Collections.singletonList(reduceResult);
  }
  Iterator<Map<String, Integer>> mIt = mapResult.iterator();
  Map<String, Integer> reduceResult = null;
  reduceResult = mIt.next();
  return reduceResult;
 }
 public static Map<String, Integer> map(String text) {... 1:1 from map_op.jbx ...}
 public static boolean isWS(char c) {... 1:1 from map_op.jbx }
 public static void reduce(Map<String, Integer> in1, Map<String, Integer> inout2) {
  ... 1:1 from reduce_op.jbx
 }
}
```

A.5. Supplements of Chapter 7

A.5.1. Minimal ISC Specifications

Listing A.20: JastAdd AST grammar of the minimal FCM and STpL.

```
CompositionEnvironment ::= fragments:Fragment*;

abstract Fragment;
GenericFragment:Fragment ::= <name:String> elements:Element*;

abstract Element;
abstract WaterElement:Element;
abstract IslandElement:Element;
TextBlob:WaterElement ::= <content:String>;
abstract Compositional:IslandElement ::= <name:String>;
Slot:Compositional;
```

Listing A.21: JastAdd AST grammar of the VTpL extension.

```
VariantList:Compositional ::= variants:Element* <ActiveName:String>;
Prototype:Compositional ::= content:Element*;
Variant:Compositional ::= content:Element*;
```

VTpL grammar in SimpAG notation:

```
CompositionEnvironment ::= fragments:Box⋆
GenericFragment▷Fragment ::= elements:Element⋆
@Box ::= name:<string>
@Element ::= ϵ
@WaterElement▷Element ::= ϵ
TextBlob▷WaterElement ::= content:<string>
@IslandElement▷Element ::= ϵ
@Compositional▷IslandElement ::= name:<string>
Slot▷Compositional ::= ϵ
VariantList▷Compositional ::=
            variants:Element⋆ activeName:<string>?
Variant▷Compositional ::= content:Element⋆
Prototype▷Compositional ::= content:Element⋆
```

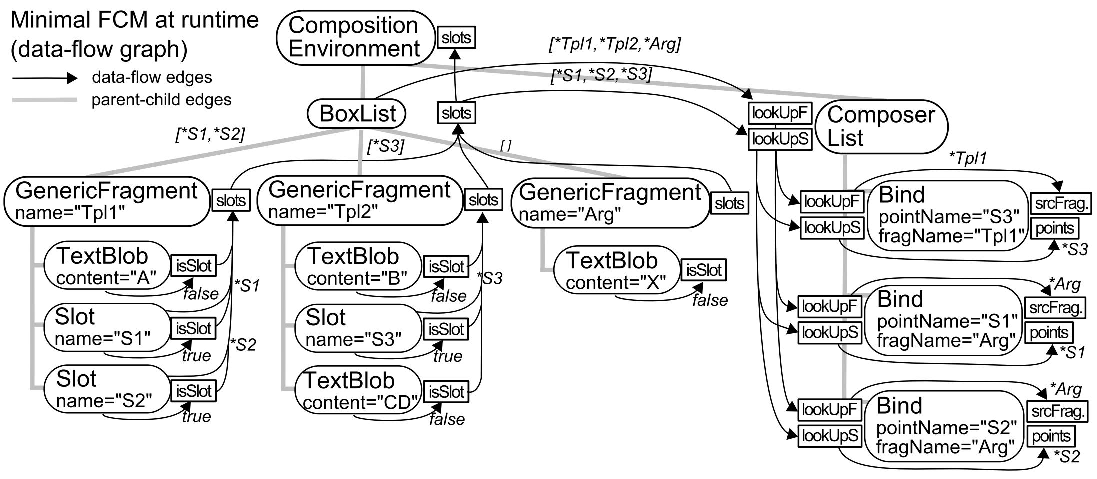
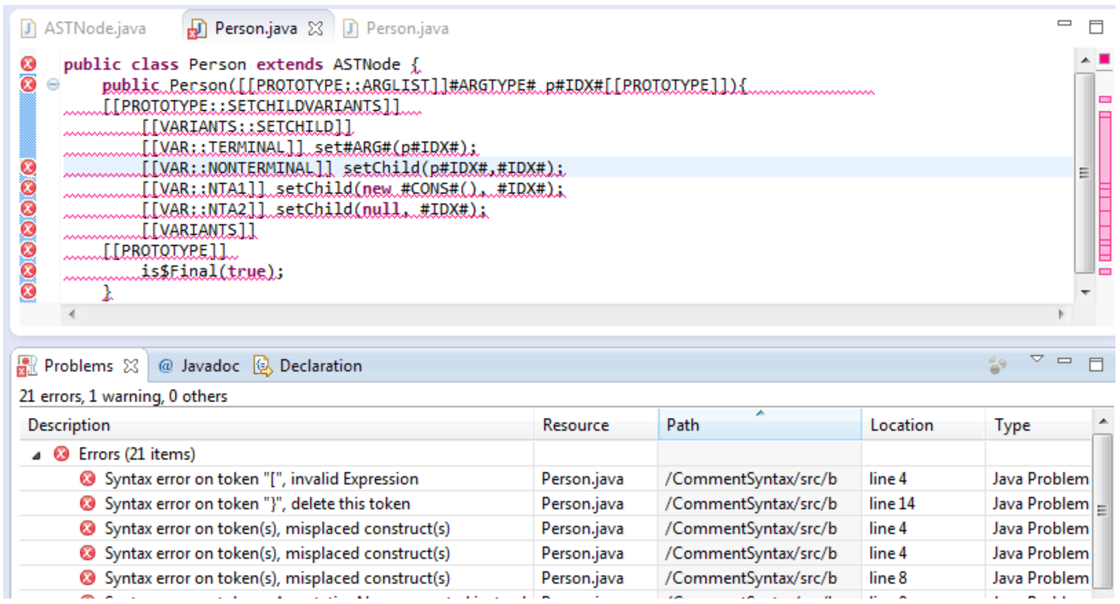


Figure A.3.: Data-flow representation of Figure 7.5.

A.5.2. A Comment-Based Syntax for VTpL

The VTpL is language independent and was not designed to support a specific target language. Hence, if a file containing VTpL code (variants and prototypes) is opened in a language-specific editor, it will most likely complain about lexical and syntactical errors. Figure A.4 shows a screenshot of a Java file with embedded VTpL in the Eclipse Java IDE. In the error view the editor complains about 21 errors of which 20 are lexical or syntactical and fails to give semantic feedback. However, if the delimiters of composers and slots are chosen carefully


ASTNode.java | Person.java | Person.java

```
public class Person extends ASTNode {
    public Person([[PROTOTYPE::ARGLIST]]#ARGTYPE# p#IDX#[[PROTOTYPE]]){
    [[PROTOTYPE::SETCHILDVARIANTS]]
        [[VARIANTS::SETCHILD]]
        [[VAR::TERMINAL]] set#ARG#(p#IDX#);
        [[VAR::NONTERMINAL]] setChild(p#IDX#,#IDX#);
        [[VAR::NTA1]] setChild(new #CONS#(), #IDX#);
        [[VAR::NTA2]] setChild(null, #IDX#);
        [[VARIANTS]]
    [[PROTOTYPE]]
        is$Final(true);
    }
```

Problems | Javadoc | Declaration

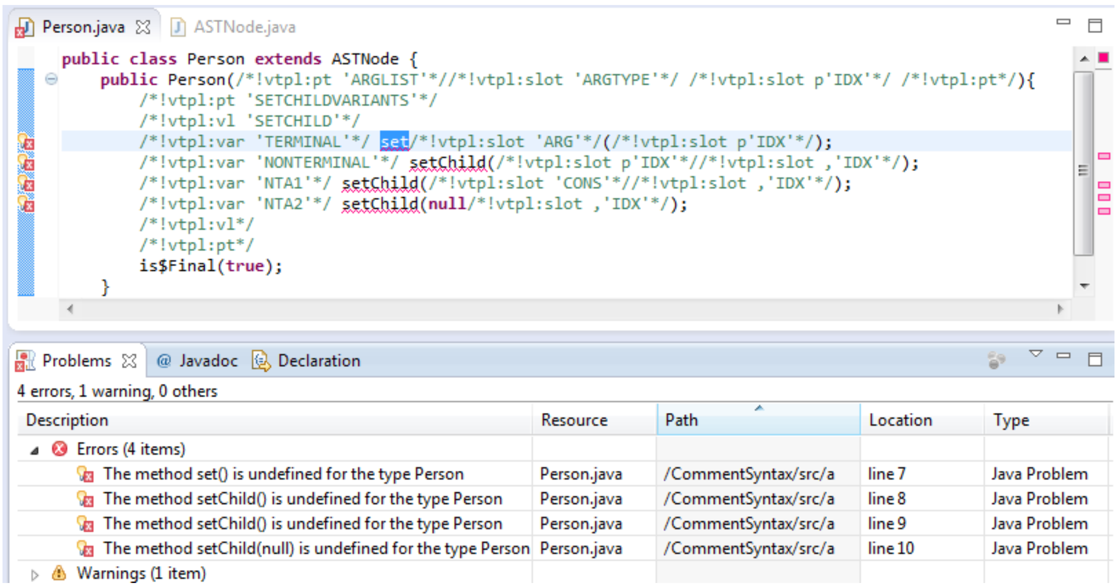
21 errors, 1 warning, 0 others

Description	Resource	Path	Location	Type
Errors (21 items)				
Syntax error on token "[", invalid Expression	Person.java	/CommentSyntax/src/b	line 4	Java Problem
Syntax error on token "}", delete this token	Person.java	/CommentSyntax/src/b	line 14	Java Problem
Syntax error on token(s), misplaced construct(s)	Person.java	/CommentSyntax/src/b	line 4	Java Problem
Syntax error on token(s), misplaced construct(s)	Person.java	/CommentSyntax/src/b	line 4	Java Problem
Syntax error on token(s), misplaced construct(s)	Person.java	/CommentSyntax/src/b	line 8	Java Problem

Figure A.4.: VTpL template opened in the JDT Java editor of Eclipse 4.2. The code corresponds to Listing 7.12.

(i.e., more language specific), the editor tool's massive stumbling on lexical errors can be avoided in most cases such that the editor can give much more usable feedback. As an example, consider Figure A.5. Here the normal VTpL markup is mimicked in Java comment syntax. `/*!vtpl:DEFINITION*/` denotes the common prefix of all compositional constructs of the VTpL. Slots are declared by `/*!vtpl:slot PFX'SLOTNAME'SFX */` where `PFX` and `SFX` are optional patterns that are concatenated to the bound value. The other concepts are straight-forward, `pt` stands for prototype, `vl` for variant list, and `var` stands for actual variants. According to Figure A.5, the feedback given by the Java editor is more valuable. The number of errors decreased to 4 and, since the file is correctly parsed as a Java file, all of them stem from the analysis of the semantic analyzer. Three messages report a missing `setChild()` method, which is indeed not declared in the example used for taking the screenshot and thus not related to the template. In contrast, the fourth error is related to the template since the suffix of `set` has to be provided by binding the `ARG` slot.

Hence, using *nondestructive* syntax—like comments in free-form languages like Java or C—for compositional constructs in minimal ISC systems enables composition-unaware editor tools (or compiler frontends) to provide better syntactic semantic analysis results than with destructive syntax. However, it does not give compositional guarantees about the composition result like full/well-formed ISC.



```
public class Person extends ASTNode {
    public Person(/*!vtpl:pt 'ARGLIST'*//*!vtpl:slot 'ARGTYPE'*/ /*!vtpl:slot p'IDX'*/ /*!vtpl:pt*/){
        /*!vtpl:pt 'SETCHILDVARIANTS'*/
        /*!vtpl:vl 'SETCHILD'*/
        /*!vtpl:var 'TERMINAL'*/ set/*!vtpl:slot 'ARG'*/(/*!vtpl:slot p'IDX'*/);
        /*!vtpl:var 'NONTERMINAL'*/ setChild(/*!vtpl:slot p'IDX'*//*!vtpl:slot ,'IDX'*/);
        /*!vtpl:var 'NTA1'*/ setChild(/*!vtpl:slot 'CONS'*//*!vtpl:slot ,'IDX'*/);
        /*!vtpl:var 'NTA2'*/ setChild(null/*!vtpl:slot ,'IDX'*/);
        /*!vtpl:vl*/
        /*!vtpl:pt*/
        is$Final(true);
    }
```

4 errors, 1 warning, 0 others

Description	Resource	Path	Location	Type
Errors (4 items)				
The method set() is undefined for the type Person	Person.java	/CommentSyntax/src/a	line 7	Java Problem
The method setChild() is undefined for the type Person	Person.java	/CommentSyntax/src/a	line 8	Java Problem
The method setChild() is undefined for the type Person	Person.java	/CommentSyntax/src/a	line 9	Java Problem
The method setChild(null) is undefined for the type Person	Person.java	/CommentSyntax/src/a	line 10	Java Problem
Warnings (1 item)				

Figure A.5.: The same template as in Figure A.4, but in alternative comment syntax.

A.5.3. Detailed UPP Specifications

Listing A.22: Reference attributes to perform the look-up of macro declarations in `MacroLookUp.jrag`.

```
aspect MacroLookUp {
   inh MacroDecl ASTNode.lookUpMacro(String macroName);
   syn MacroDecl Element.lookUpMacroLocal(String macroName) = null;
   syn MacroDecl MacroRef.macro() = lookUpMacro(getName());

   syn MacroDecl MacroDecl.lookUpMacroLocal(String macroName)
            = macroName.equals(getName())?this:null;

   eq GenericFragment.getElements(int index).lookUpMacro(String macroName) {
      // counting backwards looking for a #define or #undefine with the
      // corresponding macro name
      for(int i=index-1;i>=0;i--){
         MacroDecl decl = getElements(i).lookUpMacroLocal(macroName);
         if(decl!=null)
            return decl;
         if(getElements(i) instanceof UnDefine){
            if(((UnDefine)getElements(i)).getName().equals(macroName)){
               return null;
            }
         }
      }
      return null;
   }
}
```

Listing A.23: UPP JastAdd AST grammar for expressions in `Expression.ast`.

```
abstract Expression;
abstract UnaryExpression:Expression ::= Arg:Expression;
abstract BinaryExpression:Expression ::= LeftArg:Expression RightArg:Expression;
abstract AdditiveExpression:BinaryExpression;
abstract MultiplicativeExpression:BinaryExpression;
abstract BooleanExpression:BinaryExpression;

Constant:Expression ::= <StringValue:String> <Type:Type>;
Call:Expression ::= <OpName:String> Args:Expression*;
Defined:Expression ::= Reference:MacroRef;
Nested:Expression ::= Expression;

Or:BooleanExpression;
And:BooleanExpression;
LessThen:BooleanExpression;
GreaterThen:BooleanExpression;
Equals:BooleanExpression;

Plus:AdditiveExpression;
Minus:AdditiveExpression;

Mult:MultiplicativeExpression;
Div:MultiplicativeExpression;

Not:UnaryExpression;
```

Listing A.24: The *Rats!* UPP island grammar modulo AST construction actions.

```
module parser.UPPParser(ISCBase);
modify ISCBase;

// spacing
transient void Space = ' ' / '\t' ;
transient void LS = Space*;
transient void LB = '\r' '\n' / '\r' / '\n' ;
transient String LBS = '\r' '\n' / '\r' / '\n' ;
transient void WS = ( Space / LB )*;

// constant types
String Str = (![\"] _ )* ;
String Literal = "\"" (![\"] _ )* "\"" ;
transient String IntNum = [0-9]+;
transient String Bool = "true" / "false";

// delimiters
transient String Pfx = PfxDelimiter LS;
transient String PfxDelimiter = "#";
transient String SlotDelimiter = PfxDelimiter;
transient String Delimiters = PfxDelimiter;
transient String EscapeDelimiter = "\\" ;

// tethering directives as islan elements
IslandElement Island := yyValue:Include
                  / yyValue:MacroDecl
                  / yyValue:IfCondition
                  / yyValue:UnDefine
                  / yyValue:Error
                  / yyValue:IfDef
                  / yyValue:IfNotDef
                  / yyValue:SlotDecl
                  / yyValue:MacroCall {};

// #include declarations
Include Include = Pfx "include" LS '"' name:Str '"' LB {};

// #define: two alternatives - single-line and multi-line
MacroDecl MacroDecl = MacroDeclML / MacroDeclSL {};

// #define: multi-line ending with #end (different from CPP)
MacroDeclML MacroDeclML = Pfx "define" LS name:Identifier LS
   params:("(" yyValue:MacroParams ")" LS )?
   content:(yyValue:SlotDecl/ yyValue:MacroCall/ yyValue:TextElement)* Pfx "end"
   LB {};

// #define: single-line (like in the C standard)
MacroDeclSL MacroDeclSL =
   Pfx "define" LS name:Identifier LS params:("(" yyValue:MacroParams ")" LS )?
   content:(yyValue:SlotDecl / yyValue:MacroCall / yyValue:MacroContentSL)* LB {};

TextBlob MacroContentSL = value:MacroWater {};
transient String MacroWater = ( (EscapeDelimiter LBS ) / !( LBS / Island) _ )+ ;

// parameter list of a macro definition
List<MacroParam> MacroParams =
   name:Identifier {} param:("," LS name:Identifier {} )* {};

// #undef to invalidate a defined macro via its name
UnDefine UnDefine = Pfx "undef" LS macro:MacroRef LB {};

// #ifdef
IfDef IfDef = Pfx "ifdef" LS macro:MacroRef LB content:TplElements Pfx "endif" LB {};

// #ifndef like #ifdef, but checks if a macro is undefined
IfNotDef IfNotDef = Pfx "ifndef" LS macro:MacroRef LB
            content:TplElements Pfx "endif" LB {};

// #if-then-#else clause
IfCondition IfCondition = Pfx "if" LS yyValue:IfBody {};

// the tail and body of an #if-then-#else-#endif clause
IfCondition IfBody =
   condition:Or LB then:TplElements elz:( Pfx "elif" LB body:IfBody {}
  / Pfx "else" LB yyValue:TplElements)? Pfx "endif" LB {};

// #error to issue error messages and fail the transformation process
Error Error = Pfx "error" LS message:Literal LB {};

// call a macro in a function like style
MacroCall MacroCall = ref:MacroRef LS "(" args:MacroCallArgs ")"{} / ref:MacroRef {};
List<MacroCallArg> MacroCallArgs =
   head:MacroCallArg {} tail:("," yyValue:MacroCallArg)* {};

// call args can be expressions or elements
MacroCallArg MacroCallArg = "$" LS value:Or {} / value:MArgWater {};

```

```
transient String MArgWater = Literal / SimpleMArg / NestedMArg;

// normal case for call args: no parentheses and no comma, but everything else
transient String SimpleMArg = ( !( "(" / ")" / "," ) _ )+ ;

// special case for call args: parentheses allowed in balance and also nested commas
transient String NestedMArg = "(" ( MArgWater / ( !( "(" / ")" ) _ )+ ) ")";

// the leafs of the expression tree
Expression Atomic = Constant / NestedExpression / CallExpression / Defined;
Expression NestedExpression = "(" LS nested:Or LS ")" {};
Constant Constant = value:Literal {} / value:IntNum {} / value:Bool {};
Defined Defined = "defined" LS "(" macro:MacroRef ")" {};
Call CallExpression = ident:Identifier LS "(" LS ")" {} /
                 ident:Identifier LS "(" LS args:CallArgs LS ")" {};
List<Expression> CallArgs = head:Or {} tail:("," LS yyValue:Or )* {};

MacroRef MacroRef = ident:Identifier {};

Expression Or = leftArg:And LS "||" LS rightArg:Or {} / yyValue:And;

Expression And = leftArg:Comparatives LS "&&" LS rightArg:And {}
             / yyValue:Comparatives;

Expression Comparatives = leftArg:Additives LS "<" LS rightArg:Comparatives {}
                  / leftArg:Additives LS ">" LS rightArg:Comparatives {}
                  / leftArg:Additives LS "==" LS rightArg:Comparatives {}
                  / yyValue:Additives;

Expression Additives = leftArg:Multiplicatives LS "+" LS rightArg:Additives {}
                 / leftArg:Multiplicatives LS "-" LS rightArg:Additives {}
                 / yyValue:Multiplicatives;

Expression Multiplicatives = leftArg:Unary LS "*" LS rightArg:Multiplicatives {}
                 / leftArg:Unary LS "/" LS rightArg:Multiplicatives {}
                 / yyValue:Unary;

Expression Unary = "!" arg:Atomic {} / yyValue:Atomic ;
```

List of Figures

List of Tables

List of Listings

List of Algorithms

List of Abbreviations

ABC AspectBench Compiler

AG attribute grammar

AHEAD Algebraic Hierarchical Equations for Application Design

AJDT AspectJ Development Tools

ANCAG absolutely non-circular attribute grammar

AOP aspect-oriented programming

API application programming interface

ASG abstract syntax graph

AST abstract syntax tree

BAF business application framework

BPMN Business Process Model and Notation

CFG context-free grammar

CM component model

CMOF Complete Meta Object Facility

$\mathbf{C_m}$SL component-model specification language

CnL component language

CORBA Common Object Request Broker Architecture

CPP C preprocessor

CsL composition language

CST concrete syntax tree

CT	composition technique
DFA	deterministic finite automaton
DMS	Design Maintenance System®
DOM	Domain Object Model
DSL	domain-specific language
E-ISC	embedded invasive software composition
EBNF	Extended Backus Naur Form
EMF	Eclipse Modeling Framework
EMOF	Essential Meta Object Facility
FCM	fragment component model
FPP	Fortran Extensible Preprocessor
GLR	generalized LR parsing
GPU	graphics processing unit
HiDec	hierarchical decomposition
HOAG	higher-order attribute grammar
IDE	integrated development environment
IDL	Interface Description Language
ISC	invasive software composition
JaMoPP	Java Model Parser and Printer
JAXB	Java Architecture for XML Binding
JDT	Java Development Tools
LogProg	Logic “Programming” Language
MDA	Model-Driven Architecture
MDE	model-driven engineering
MDSD	model-driven software development

MDSoC	multi-dimensional separation of concerns
MOF	Meta Object Facility
NTA	nonterminal attribute
OAG	ordered attribute grammar
OCL	Object Constraint Language
OMG	Object Management Group
OpenMP	Open specifications for Multi Processing
PDA	push-down automaton
PEG	parsing expression grammar
RACR	reference attribute grammar controlled rewriting
RAG	reference attribute grammar
ReRAG	rewritable reference attribute grammar
$\textbf{REX}_{\textbf{CM}}$	Reuse EXtension language for Component Model Configuration
$\textbf{REX}_{\textbf{CL}}$	Reuse EXtension language for Composition Language Integration
SDF	Syntax Definition Formalism
SGLR	scannerless generalized LR parsing
SimpAG	Simple Attribute Grammar Specification Language
SkAT	Skeletons and Application Templates
SkAT/Core	SkAT core implementation
SkAT/Full	SkAT for full-fledged ISC
SkAT/Minimal	SkAT for minimal ISC
SkAT4J	SkAT for Java
SoC	separation of concerns
SoCC	separation of cross-cutting concerns
STpL	Slot Template Language

TMP	template metaprogramming
U-ISC	universal invasive software composition
U-ISC/Graph	universal invasive software composition for typed graphs
UCL	Universal Composition Language
UML	Unified Modeling Language
UPP	Universal Extensible Preprocessor
VTpL	Variant Template Language
XMI	XML Metadata Interchange
XML	eXtensible Markup Language
XSD	XML Schema Definition Language

References

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullmann (1986). *Compilers: Principles, Techniques and Tools*. 1st ed. Reading, MA, USA: Addison-Wesley. ISBN: 0-201-10088-6 (cit. on pp. 19, 34).

Alanen, Marcus and Ivan Porres (2003). *A Relation Between Context-Free Grammars and Meta Object Facility Metamodels*. Technical report. Turku, Finland: TUCS Turku Center for Computer Science, Åbo Akademi University (cit. on p. 41).

Alblas, Henk (1991). "Introduction to Attribute Grammars." In: *Proceedings of the International Summer School on Attribute Grammars, Applications and Systems*. Vol. 545. Lecture Notes in Computer Science. Berlin / Heidelberg, Germany: Springer, pp. 1–15. ISBN: 978-3-540-54572-9. DOI: `10.1007/3-540-54572-7_1` (cit. on p. 156).

Apel, Sven, Christian Kästner, Thomas Leich, and Gunter Saake (2007). "Aspect Refinement – Unifying AOP and Stepwise Refinement." In: *Journal of Object Technology* 6.9: *TOOLS EUROPE 2007*, pp. 13–33. ISSN: 1660-1769. DOI: `10.5381/jot.2007.6.9.a1` (cit. on p. 277).

Apel, Sven, Christian Lengauer, Bernhard Möller, and Christian Kästner (2008). "An Algebra for Features and Feature Composition." In: *Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology (AMAST 2008)*. Vol. 5140. Lecture Notes in Computer Science. Berlin / Heidelberg, Germany: Springer, pp. 36–50. ISBN: 3-540-79979-6. DOI: `10.1007/978-3-540-79980-1_4` (cit. on p. 245).

Aracic, Ivica, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann (2006). "An Overview of CaesarJ." In: *Transactions on Aspect-Oriented Software Development I*. Vol. 3880. Lecture Notes in Computer Science. Berlin / Heidelberg, Germany: Springer, pp. 135–173. ISBN: 978-3-540-32972-5. DOI: `10.1007/11687061_5` (cit. on p. 63).

Arnoldus, Bastiaan J. (2011). "An Illumination of the Template Enigma: Software Code Generation with Templates." PhD thesis. Eindhoven, The Netherlands: Technische Universiteit Eindhoven. ISBN: 978-90-386-2418-1 (cit. on pp. 11, 271, 272).

Aßmann, Uwe (2000). "Graph Rewrite Systems for Program Optimization." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22.4, pp. 583–637. ISSN: 0164-0925. DOI: `10.1145/363911.363914` (cit. on p. 139).

– (2003). *Invasive Software Composition*. 1st ed. Berlin / Heidelberg, Germany: Springer. ISBN: 3-540-44385-1 (cit. on pp. 1, 3, 4, 7, 12, 13, 61, 62, 64, 65, 72–74, 81, 194, 276).

Aßmann, Uwe, Sacha Berger, François Bry, Tim Furche, Jakob Henriksson, and Jendrik Johannes (2007). "Modular Web Queries—From Rules to Stores." In: *On the Move to Meaningful*

Internet Systems 2007: OTM 2007 Workshops. Vol. 4806. Lecture Notes in Computer Science. Berlin / Heidelberg, Germany: Springer, pp. 1165–1175. ISBN: 978-3-540-76890-6. DOI: 10.1007/978-3-540-76890-6_44 (cit. on p. 81).

Aßmann, Uwe, Andreas Bartho, Christoff Bürger, Sebastian Cech, Birgit Demuth, Florian Heidenreich, Jendrik Johannes, Sven Karol, Jan Polowinski, Jan Reimann, Julia Schroeter, Mirko Seifert, Michael Thiele, Christian Wende, and Claas Wilke (2012). "DropsBox: the Dresden Open Software Toolbox." In: *Software & Systems Modeling* 13.1, pp. 133–169. ISSN: 1619-1366. DOI: 10.1007/s10270-012-0284-6 (cit. on pp. ix, 232).

Avgustinov, Pavel, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble (2006). "abc: An Extensible AspectJ Compiler." In: *Transactions on Aspect-Oriented Software Development I*. Vol. 3880. Lecture Notes in Computer Science. Berlin / Heidelberg, Germany: Springer, pp. 293–334. ISBN: 978-3-540-32972-5. DOI: 10.1007/11687061_9 (cit. on p. 276).

Avgustinov, Pavel, Torbjörn Ekman, and Julian Tibble (2008). "Modularity First: A Case for Mixing AOP and Attribute Grammars." In: *Proceedings of the 7th International Conference on Aspect-Oriented Software Development (AOSD '08)*. New York, NY, USA: ACM, pp. 25–35. ISBN: 978-1-60558-044-9. DOI: 10.1145/1353482.1353486 (cit. on p. 277).

Azurat, Ade (2007). "Mechanization of Invasive software Composition in F-Logic." In: *Proceedings of the 2007 annual Conference on International Conference on Computer Engineering and Applications (CEA '07)*. Stevens Point, Wisconsin, USA: World Scientific, Engineering Academy, and Society (WSEAS), pp. 89–94. ISBN: 978-960-8457-58-4 (cit. on p. 279).

Baader, Franz and Tobias Nipkow (1999). *Term Rewriting and All That*. 1st ed. Nottingham, U.K.: Cambridge University Press. ISBN: 0-521-77920-0 (cit. on pp. 136, 280, 281).

Batory, Don, Jia Liu, and Jacob N. Sarvela (2003). "Refinements and Multi-Dimensional Separation of Concerns." In: *Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-11)*. New York, NY, USA: ACM, pp. 48–57. ISBN: 1-58113-743-5. DOI: 0.1145/940071.940079 (cit. on p. 277).

Batory, Don, Jacob N. Sarvela, and Axel Rauschmayer (2004). "Scaling Step-Wise Refinement." In: *IEEE Transactions on Software Engineering* 30.6, pp. 355–371. ISSN: 0098-5589. DOI: 10.1109/TSE.2004.23 (cit. on pp. 3, 277, 278).

Baxter, Ira D., Christopher Pidgeon, and Michael Mehlich (2004). "DMS®: Program Transformations for Practical Scalable Software Evolution." In: *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. ICSE '04. Washington, DC, USA: IEEE Computer Society, pp. 625–634. ISBN: 0-7695-2163-0. DOI: 10.1109/ICSE.2004.1317484 (cit. on p. 281).

Beck, Kent and Cynthia Andres (2004). *Extreme Programming Explained: Embrace Change*. 2nd ed. Reading, MA, USA: Addison-Wesley. ISBN: 978-0321278654 (cit. on p. 260).

Beck, Kent, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas (2001). *Manifesto for Agile Software Development*. URL: `http://agilemanifesto.org/` (visited on 01/20/2014) (cit. on p. 260).

Benkner, Siegfried, Sabri Pllana, Jesper Larsson Traff, Philippas Tsigas, Uwe Dolinsky, Cédric Augonnet, Beverly Bachmayer, Christoph Kessler, David Moloney, and Vitaly Osipov (2011). "PEPPHER: Efficient and Productive Usage of Hybrid Computing Systems." In: *Micro* 31.5, pp. 28–41. ISSN: 0272-1732. DOI: `10.1109/MM.2011.67` (cit. on p. 201).

Bergmans, Lodewijk and Mehmet Aksit (2001). "Composing Crosscutting Concerns Using Composition Filters." In: *Communications of the ACM* 44.10, pp. 51–57. ISSN: 0001-0782. DOI: `10.1145/383845.383857` (cit. on p. 2).

Berstel, Jean and Luc Boasson (1990). "Context-Free Languages." In: *Handbook of Theoretical Computer Science*. Ed. by Jan van Leeuwen. Vol. B. Cambridge, MA, USA: The MIT Press, pp. 59–102. ISBN: 978-0-262-72015-9 (cit. on pp. 19, 20).

– (2002). "Formal Properties of XML Grammars and Languages." In: *Acta Informatica* 38.9, pp. 649–671. ISSN: 0001-5903. DOI: `10.1007/s00236-002-0085-4` (cit. on p. 37).

Bertot, Yves and Pierre Castéran (2004). *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. 1st ed. Berlin / Heidelberg, Germany: Springer. ISBN: 978-3-540-20854-9 (cit. on p. 279).

Binder, Thomas, Dietmar Winkler, and Stefan Biffl (2006). *Quo Vadis V-Modell*. German. Technical report IFS-QSE-06/03. Vienna, Austria: Vienna University of Technology (cit. on p. 260).

Boyland, John T. (2005). "Remote Attribute Grammars." In: *Journal of the ACM* 52.4, pp. 627–687. ISSN: 0004-5411. DOI: `10.1145/1082036.1082042` (cit. on p. 49).

Brabrand, Claus and Michael I. Schwartzbach (2002). "Growing Languages with Metamorphic Syntax Macros." In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '02)*. New York, NY, USA: ACM, pp. 31–40. ISBN: 1-58113-455-X. DOI: `10.1145/503032.503035` (cit. on pp. 3, 274, 275).

Bracha, Gilad and William Cook (1990). "Mixin-Based Inheritance." In: *Proceedings of the European Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA/ECOOP '90)*. New York, NY, USA: ACM, pp. 303–311. ISBN: 0-89791-411-2. DOI: `10.1145/503032.503035` (cit. on pp. 12, 192, 194, 195).

Brand, Mark G. J. van den, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser (2002). "Disambiguation Filters for Scannerless Generalized LR Parsers." In: *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*. Vol. 2304. Lecture Notes in Computer Science. London, UK: Springer, pp. 143–158. ISBN: 978-3-540-43369-9. DOI: `10.1007/3-540-45937-5_12` (cit. on pp. 35, 272, 280).

Brand, Mark G. J. van den and Paul Klint (2007). "ATerms for Manipulation and Exchange of Structured Data: It's All About Sharing." In: *Information and Software Technology* 49.1: *Most Cited Journal Articles in Software Engineering - 2000*, pp. 55–64. ISSN: 0950-5849. DOI: `10.1016/j.infsof.2006.08.009` (cit. on p. 280).

Bravenboer, Martin, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser (2008). "Stratego/XT 0.17. A Language and Toolset for Program Transformation." In: *Science of Computer Pro-*

gramming 72.1-2: *Second Issue of Experimental Software and Toolkits (EST)*, pp. 52–70. ISSN: 0167-6423. DOI: `10.1016/j.scico.2007.11.003` (cit. on p. 280).

Brüggemann-Klein, Anne and Derick Wood (2004). "Balanced Context-Free Grammars, Hedge Grammars and Pushdown Caterpillar Automata." In: *Proceedings of Extreme Markup Languages® 2004*. Alexandria, VA, USA: IDEAlliance (cit. on p. 37).

Bürger, Christoff (2012). *RACR: A Scheme Library for Reference Attribute Grammar Controlled Rewriting*. Technical report TUD-FI12-09. Dresden, Germany: Technische Universität Dresden (cit. on pp. 50, 107, 127, 282, 289).

Bürger, Christoff and Sven Karol (2007). "Entwurf und Implementierung des Übersetzergenerators Babylon." German. Bachelor thesis. Dresden, Germany: Technische Universität Dresden (cit. on p. 232).

– (2010). *Towards Attribute Grammars for Metamodel Semantics*. Tech. rep. TUD-FI10-03. Dresden, Germany: Technische Universität Dresden (cit. on pp. 50, 232, 233).

Bürger, Christoff, Sven Karol, and Christian Wende (2010). "Applying Attribute Grammars for Metamodel Semantics." In: *Proceedings of the International Workshop on Formalization of Modeling Languages (FML '10)*. New York, NY, USA: ACM, 1:1–1:5. ISBN: 978-1-4503-0532-7. DOI: `10.1145/1943397.1943398` (cit. on p. x).

Bürger, Christoff, Sven Karol, Christian Wende, and Uwe Aßmann (2011). "Reference Attribute Grammars for Metamodel Semantics." In: *Proceedings of the Third International Conference on Software Language Engineering (SLE 2010)*. Vol. 6563. Lecture Notes in Computer Science. Berlin / Heidelberg, Germany: Springer, pp. 22–41. ISBN: 978-3-642-19439-9. DOI: `10.1007/978-3-642-19440-5_3` (cit. on pp. ix, 50, 107, 233, 287, 288).

Chalabine, Mikhail and Christoph Kessler (2006). "Crosscutting Concerns in Parallelization by Invasive Software Composition and Aspect Weaving." In: *Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS '06)*. Vol. 9. Los Alamitos, CA, USA: IEEE Computer Society, pp. 214–226. ISBN: 0-7695-2507-5. DOI: `10.1109/HICSS.2006.106` (cit. on p. 201).

Chen, Sheng, Martin Erwig, and Eric Walkingshaw (2014). "Extending Type Inference to Variational Programs." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 36.1, 1:1–1:54. ISSN: 0164-0925. DOI: `10.1145/2518190` (cit. on p. 279).

Ciechanowicz, Philipp, Michael Poldner, and Herbert Kuchen (2009). "The Münster Skeleton Library Muesli - A Comprehensive Overview." In: *Working Papers, ERCIS - European Research Center for Information Systems* 7. ISSN: 1614-7448 (cit. on p. 201).

Cole, Murray (1989). *Algorithmic Skeletons: Structured Management of Parallel Computation*. Cambridge, MA, USA: MIT Press. ISBN: 0-262-53086-4 (cit. on pp. 5, 17, 200, 286).

– (2004). "Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming." In: *Parallel Computing* 30.3, pp. 389–406. ISSN: 0167-8191. DOI: `10.1016/j.parco.2003.12.002` (cit. on p. 161).

Cordy, James R. (2006). "The TXL Source Transformation Language." In: *Science of Computer Programming* 61.3: *Special Issue on The Fourth Workshop on Language Descriptions, Tools,* *and Applications (LDTA '04)*, pp. 190–210. ISSN: 0167-6423. DOI: 10.1016/j.scico.2006.04.002 (cit. on p. 281).

Czarnecki, Krysztof and Ulrich W. Eisenecker (2000). *Generative Programming: Methods, Tools and Applications*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co. ISBN: 0-201-30977-7 (cit. on p. 3).

Czarnecki, Krzysztof (1999). "Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models." PhD thesis. Ilmenau, Germany: Technical University of Ilmenau (cit. on p. 8).

Dean, Jeffrey and Sanjay Ghemawat (2008). "MapReduce: Simplified Data Processing on Large Clusters." In: *Communications of the ACM* 51.1, pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492 (cit. on pp. 192, 201, 202).

Demenchuk, Alexander (2012). *Beaver—a LALR Parser Generator*. URL: http://beaver.sourceforge.net/ (visited on 09/28/2014) (cit. on pp. 166, 177).

Dijkstra, Edsger W. (1982). "On the Role of Scientific Thought." In: *Selected Writings on Computing: A Personal Perspective*. New York, NY, USA: Springer, pp. 60–66. ISBN: 0-387-90652-5 (cit. on p. 1).

Douence, Rémi, Pascal Fradet, and Mario Südholt (2002). "A Framework for the Detection and Resolution of Aspect Interactions." In: *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002)*. Vol. 2487. Lecture Notes in Computer Science. London, UK: Springer, pp. 173–188. ISBN: 3-540-44284-7. DOI: 10.1007/3-540-45821-2_11 (cit. on pp. 14, 135).

Dueck, Gerald D. P. and Gordon V. Cormack (1990). "Modular Attribute Grammars." In: *The Computer Journal* 33.2, pp. 164–172. ISSN: 0010-4620. DOI: 10.1093/comjnl/33.2.164 (cit. on pp. 50, 55).

Durr, Pascal, Lodewijk Bergmans, and Mehmet Aksit (2007). "Static and Dynamic Detection of Behavioral Conflicts Between Aspects." In: *Proceedings of the 7th International Workshop on Runtime Verification (RV '07)*. Vol. 4839. Lecture Notes in Computer Science. Berlin / Heidelberg, Germany: Springer, pp. 38–50. ISBN: 978-3-540-77394-8. DOI: 10.1007/978-3-540-77395-5_4 (cit. on p. 14).

Eclipse Foundation (2013a). *AspectJ Development Tools*. URL: http://www.eclipse.org/ajdt/ (visited on 06/06/2013) (cit. on pp. 4, 14).

– (2013b). *Eclipse Modeling Framework Project (EMF)*. URL: http://www.eclipse.org/modeling/emf/ (visited on 02/20/2013) (cit. on pp. 42, 89).

– (2013c). *Eclipse Project*. URL: http://www.eclipse.org/ (visited on 02/06/2013) (cit. on p. 13).

Efftinge, Sven, Peter Friese, Arno Haase, Dennis Hübner, Clemens Kadura, Bernd Kolb, Jan Köhnlein, Dieter Moroff, Karsten Thoms, Marcus Völter, Patrick Schönbach, Moritz Eysholdt, and Steven Reinisch (2008). *openArchitectureWare User Guide v.4.3.1*. URL: http://www.openarchitectureware.org/pub/documentation/4.3.1/ (visited on 03/21/2014) (cit. on pp. 232, 262).

Ehrig, Hartmut, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer (2010). *Fundamentals of Algebraic Graph Transformation*. 1st ed. Berlin / Heidelberg, Germany: Springer. ISBN: 978-3-540-31187-4 (cit. on p. 136).

Ekman, Torbjörn (2006). “Extensible Compiler Construction.” PhD thesis. Lund, Sweden: University of Lund. ISBN: 91-628-6839-X (cit. on pp. 50, 161).

Ekman, Torbjörn and Görel Hedin (2004). “Rewritable Reference Attributed Grammars.” In: *Proceedings of the European Conference on Object-Oriented Programming Systems, Languages, and Applications (ECOOP ’04)*. Vol. 3086. Lecture Notes in Computer Science. Berlin / Heidelberg, Germany: Springer, pp. 147–171. ISBN: 978-3-540-22159-3. DOI: 10.1007/978-3-540-24851-4_7 (cit. on pp. 50, 107, 127, 145–147, 149, 162).

– (2007a). “The JastAdd System—Modular Extensible Compiler Construction.” In: *Science of Computer Programming* 69.1-3: *Special Issue on Experimental Software and Toolkits*, pp. 14–26. ISSN: 0167-6423. DOI: 10.1016/j.scico.2007.02.003 (cit. on pp. 17, 49, 161).

– (2007b). “The Jastadd Extensible Java Compiler.” In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (OOPSLA ’07)*. New York, NY, USA: ACM, pp. 1–18. ISBN: 978-1-59593-786-5. DOI: 10.1145/1297027.1297029 (cit. on pp. 49, 165).

Erwig, Martin and Eric Walkingshaw (2011). “The Choice Calculus: A Representation for Software Variation.” In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 21.1, 6:1–6:27. ISSN: 1049-331X. DOI: 10.1145/2063239.2063245 (cit. on pp. 3, 278).

Farrow, Rodney (1986). “Automatic Generation of Fixed-Point-Finding Evaluators for Circular, but Well-defined, Attribute Grammars.” In: *Proceedings of the 1986 SIGPLAN symposium on Compiler Construction (SIGPLAN ’86)*. New York, NY, USA: ACM, pp. 85–98. ISBN: 0-89791-197-0. DOI: 10.1145/12276.13320 (cit. on p. 49).

Ford, Bryan (2002). “Packrat Parsing:: Simple, Powerful, Lazy, Linear Time, Functional Pearl.” In: *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP ’02)*. New York, NY, USA: ACM, pp. 36–47. ISBN: 1-58113-487-8. DOI: 10.1145/581478.581483 (cit. on p. 231).

– (2004). “Parsing Expression Grammars: A Recognition-Based Syntactic Foundation.” In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’04)*. New York, NY, USA: ACM, pp. 111–122. ISBN: 1-58113-729-X. DOI: 10.1145/964001.964011 (cit. on pp. 35, 36, 221).

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0-201-63361-2 (cit. on pp. 81, 138, 175, 277).

Goos, Gerhard (1997). *Vorlesungen über Informatik*. German. Vol. 3: Berechenbarkeit, formale Sprachen, Spezifikationen. Berlin / Heidelberg, Germany: Springer. ISBN: 978-3-540-60655-6 (cit. on p. 19).

Goswami, Dhrubajyoti, Ajit Singh, and Bruno R. Preiss (2002). “From Design Patterns to Parallel Architectural Skeletons.” In: *Journal of Parallel and Distributed Computing* 62.4, pp. 669–695. ISSN: 0743-7315. DOI: 10.1006/jpdc.2001.1809 (cit. on p. 161).

Grimm, Robert (2006). “Better Extensibility through Modular Syntax.” In: *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’06)*. New York, NY, USA: ACM, pp. 38–51. ISBN: 1-59593-320-4. DOI: `10.1145/1133981.1133987` (cit. on pp. 221, 230).

Grosch, Jan (1990). *Object-Oriented Attribute Grammars*. Tech. rep. Aachen, Germany: CoCoLab Datenverarbeitung (cit. on p. 49).

Gruska, Jozef (1976). “Descriptional Complexity (of Languages) a Short Survey.” In: *Proceedings of the 5th Symposium on Mathematical Foundations of Computer Science 1976*. Vol. 45. Lecture Notes in Computer Science. Berlin / Heidelberg, Germany: Springer, pp. 65–80. ISBN: 978-3-540-07854-8. DOI: `10.1007/3-540-07854-1_162` (cit. on p. 219).

Harbison III, Samuel P. and Guy L. Steele Jr. (2002). *C: A Reference Manual*. 5th ed. Upper Saddle River, New Jersey, USA: Prentice-Hall, Inc. ISBN: 978-0-130-89592-9 (cit. on pp. 246, 248, 254, 275).

Hedin, Görel (1989). “An Object-Oriented Notation for Attribute Grammars.” In: *Proceedings of the 3rd European Conference on Object-Oriented Programming (ECOOP’89)*. BCS Workshop Series. Nottingham, U.K.: Cambridge University Press, pp. 329–345. ISBN: 0-521-38232-7 (cit. on p. 49).

– (2000). “Reference Attributed Grammars.” In: *Informatica: An International Journal of Computing and Informatics* 24.3, pp. 301–317. ISSN: 0350-5596 (cit. on pp. 4, 49, 162).

– (2011). “An Introductory Tutorial on JastAdd Attribute Grammars.” In: *Generative and Transformational Techniques in Software Engineering III*. Vol. 6491. Lecture Notes in Computer Science. Berlin / Heidelberg, Germany: Springer, pp. 166–200. ISBN: 978-3-642-18022-4. DOI: `10.1007/978-3-642-18023-1_4` (cit. on p. 5).

Heidenreich, Florian, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende (2009a). “Derivation and Refinement of Textual Syntax for Models.” In: *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2009)*. Vol. 5562. Lecture Notes in Computer Science. Berlin / Heidelberg, Germany: Springer, pp. 114–129. ISBN: 978-3-642-02673-7. DOI: `10.1007/978-3-642-02674-4_9` (cit. on pp. x, 10, 41, 82, 232).

– (2009b). “EMFText and JaMoPP - Tool Presentation.” In: *5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2009): Proceedings of the Tools and Consultancy Track*. Vol. WP09-12. CTIT Proceedings Series. Enschede, The Netherlands: University of Twente, pp. 73–77 (cit. on p. 232).

Heidenreich, Florian, Jendrik Johannes, Mirko Seifert, Christian Wende, and Marcel Böhme (2009c). “Generating Safe Template Languages.” In: *Proceedings of the Eighth International Conference on Generative Programming and Component Engineering (GPCE 2009)*. New York, NY, USA: ACM, pp. 99–108. ISBN: 978-1-60558-494-2. DOI: `10.1145/1621607.1621624` (cit. on pp. 272, 273).

Heidenreich, Florian, Jakob Henriksson, Jendrik Johannes, and Steffen Zschaler (2009d). “On Language-Independent Model Modularisation.” In: *Transactions on Aspect-Oriented Software Development VI: Special Issue on Aspects and Model-Driven Engineering*. Vol. 5560. Lecture

Notes in Computer Science. Springer Berlin / Heidelberg, pp. 39–82. ISBN: 978-3-642-03763-4. DOI: `10.1007/978-3-642-03764-1_2` (cit. on p. 64).

Heidenreich, Florian, Jendrik Johannes, Sven Karol, Mirko Seifert, Michael Thiele, Christian Wende, and Claas Wilke (2011). "Integrating OCL and Textual Modelling Languages." In: *Workshops and Symposia at MODELS 2010, Reports and Revised Selected Papers*. Vol. 6627. Lecture Notes in Computer Science. Berlin / Heidelberg, Germany: Springer, pp. 349–363. ISBN: 978-3-642-21209-3. DOI: `10.1007/978-3-642-21210-9_34` (cit. on p. x).

Heidenreich, Florian, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende (2013). "Model-Based Language Engineering with EMFText." In: *Generative and Transformational Techniques in Software Engineering IV*. Vol. 7680. Lecture Notes in Computer Science. Berlin / Heidelberg, Germany: Springer, pp. 322–345. ISBN: 978-3-642-35991-0. DOI: `10.1007/978-3-642-35992-7_9` (cit. on pp. ix, 49, 232, 233).

Henadeera, Chinthaka Henadeera (2014). "A Fragment Component Library for Parallel Architectural Skeletons in the Context of GPGPU Programming." Master thesis. Dresden, Germany: Technische Universität Dresden (cit. on p. 212).

Henriksson, Jakob (2009). "A Lightweight Framework for Universal Fragment Composition—-with an Application in the Semantic Web." PhD thesis. Dresden, Germany: Technische Universität Dresden. URL: `http://nbn-resolving.de/urn:nbn:de:bsz:14-ds-1231251831567-11763` (cit. on pp. 3–5, 7, 64, 65, 81, 82, 88, 131, 275, 276).

Henriksson, Jakob, Florian Heidenreich, Jendrik Johannes, Steffen Zschaler, and Uwe Aßmann (2008). "Extending Grammars and Metamodels for Reuse: The Reuseware Approach." In: *IET Software* 2.3, pp. 165–184. ISSN: 1751-8806. DOI: `10.1049/iet-sen:20070060` (cit. on p. 112).

Heumüller, Robert, Jochen Quante, and Andreas Thums (2014). "Parsing Variant C Code: An Evaluation on Automotive Software." In: *Softwaretechnik-Trends* 34.2: *Berichte und Beiträge vom 16. Workshop "Software-Reengineering und -Evolution" und 6. Workshop "Design for Future"*. ISSN: 0720-8928. (Visited on 10/07/2014) (cit. on p. 275).

Heuzeroth, Dirk, Uwe Aßmann, Mircea Trifu, and Volker Kuttruff (2006). "The COMPOST, COMPASS, Inject/J and RECODER Tool Suite for Invasive Software Composition: Invasive Composition with COMPASS Aspect-Oriented Connectors." In: *Generative and Transformational Techniques in Software Engineering*. Vol. 4143. Lecture Notes in Computer Science. Berlin / Heidelberg, Germany: Springer, pp. 357–377. ISBN: 978-3-540-45778-7. DOI: `10.1007/11877028_14` (cit. on pp. 3, 102).

Heuzeroth, Dirk, Mircea Trifu, and Tobias Gutzmann (2013). *The RECODER Refactoring Engine*. URL: `http://recoder.sourceforge.net` (visited on 02/14/2014) (cit. on p. 73).

Hogeman, Erik (2014). "Extending JastAddJ to Java 8." Master Thesis. Lund, Sweden: University of Lund (cit. on p. 165).

Huang, Shan Shan, David Zook, and Yannis Smaragdakis (2011). "Statically Safe Program Generation with SafeGen." In: *Science of Computer Programming* 76.5: *Special Issue on Generative Programming and Component Engineering (Selected Papers from GPCE 2004/2005)*, pp. 376–391. ISSN: 0167-6423. DOI: `10.1016/j.scico.2008.09.007` (cit. on p. 273).

ISO/IEC (1991). *Fortran 90 - International Standard - ISO/IEC 1539 : 1991*. Published: online available specification. URL: http://www.iso.org/iso/catalogue_detail.htm?csnumber=6128 (visited on 11/12/2013) (cit. on p. 246).

– (1999). *C - International Standard - ISO/IEC 9899 : 1999*. Published: online available specification. URL: http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=29237 (visited on 03/21/2014) (cit. on pp. 246, 248).

– (2011). *C++ - International Standard - ISO/IEC 14882 : 2011*. Published: online available specification. URL: http://www.iso.org/iso/catalogue_detail.htm?csnumber=50372 (visited on 07/10/2014) (cit. on pp. 274, 282).

Johannes, Jendrik (2011). "Component-Based Model-Driven Software Development." PhD thesis. Dresden, Germany: Technische Universität Dresden. URL: http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-63986 (cit. on pp. 3, 5, 7, 64, 89, 112, 288).

Johnson, Stephen C. (1975). *YACC - Yet Another Compiler Compiler*. Tech. rep. CS TR 32. Murray Hill, NY, USA: Bell Labs (cit. on p. 34).

Jonge, Maartje de and Eelco Visser (2012). "An Algorithm for Layout Preservation in Refactoring Transformations." In: *Proceedings of the Fourth International Conference on Software Language Engineering (SLE 2011)*. Vol. 6940. Lecture Notes in Computer Science. Berlin / Heidelberg, Germany: Springer, pp. 40–59. ISBN: 978-3-642-28829-6. DOI: 10.1007/978-3-642-28830-2_3 (cit. on p. 45).

Jonge, Merijn (2000). "A Pretty-Printer for Every Occasion." In: *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000)*. Wollongong, Australia: University of Wollongong, pp. 68–77. ISBN: 978-0-864-18725-3 (cit. on p. 45).

Kang, Kyo C., Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson (1990). *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. Pittsburgh, PA, USA: Carnegie Mellon University (cit. on p. 245).

Karol, Sven (2008). "Komposition Syntaktischer Sprachdefinitionen." German. Master thesis. Dresden, Germany: Technische Universität Dresden (cit. on p. 134).

Karol, Sven and Steffen Zschaler (2010). *Providing Mainstream Parser Generators with Modular Language Definition Support*. Tech. rep. TUD-FI10-01. Dresden, Germany: Technische Universität Dresden (cit. on p. 35).

Karol, Sven, Martin Heinzerling, Florian Heidenreich, and Uwe Aßmann (2010). "Using Feature Models for Creating Families of Documents." In: *Proceedings of the 10th ACM Symposium on Document Engineering (DocEng '10)*. New York, NY, USA: ACM, pp. 259–262. ISBN: 978-1-4503-0231-9. DOI: 10.1145/1860559.1860618 (cit. on pp. x, 232).

Karol, Sven, Matthias Niederhausen, Daniel Kadner, Uwe Aßmann, and Klaus Meißner (2011). "Detecting and Resolving Conflicts Between Adaptation Aspects in Multi-staged XML Transformations." In: *Proceedings of the 11th ACM Symposium on Document Engineering (DocEng '11)*. New York, NY, USA: ACM, pp. 229–238. ISBN: 978-1-4503-0863-2. DOI: 10.1145/2034691.2034738 (cit. on pp. ix, 138, 139, 290).

Karol, Sven, Christoff Bürger, and Uwe Aßmann (2012). “Towards Well-Formed Fragment Composition with Reference Attribute Grammars.” In: *Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE ’12)*. New York, NY, USA: ACM, pp. 109–114. ISBN: 978-1-4503-1345-2. DOI: `10.1145/2304736.2304755` (cit. on pp. ix, 109, 280).

Kastens, Uwe (1980). “Ordered Attributed Grammars.” In: *Acta Informatica* 13.3, pp. 229–256. ISSN: 0001-5903. DOI: `10.1007/BF00288644` (cit. on p. 48).

Kelsey, Richar, William Klinger, and Jonathan Rees (eds.) (1998). “5th Revised Report on the Algorithmic Language Scheme.” In: *Higher-Order and Symbolic Computation* 11.1, pp. 7–105. ISSN: 1388-3690. DOI: `10.1023/A:1010051815785` (cit. on pp. 3, 275, 282).

Kennedy, Ken and Scott K. Warren (1976). “Automatic Generation of Efficient Evaluators for Attribute Grammars.” In: *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages (POPL ’76)*. New York, NY, USA: ACM, pp. 32–49. DOI: `10.1145/800168.811538` (cit. on pp. 48, 151).

Kent, Stuart (2002). “Model Driven Engineering.” In: *Proceedings of the 3rd International Conference on Integrated Formal Methods (IFM ’02)*. Vol. 2335. Lecture Notes in Computer Science. Berlin / Heidelberg: Springer, pp. 286–298. ISBN: 978-3-540-43703-1. DOI: `10.1007/3-540-47884-1_16` (cit. on p. 4).

Kessler, Christoph and Welf Löwe (2012). “Optimized Composition of Performance-Aware Parallel Components.” In: *Concurrency and Computation: Practice and Experience* 24.5, pp. 481–498. ISSN: 1532-0634. DOI: `10.1002/cpe.1844` (cit. on p. 201).

Kezadri Hamiaz, Mounira, Marc Pantel, Benoît Combemale, and Xavier Thirioux (2014). “Correct-by-Construction Model Composition: Application to the Invasive Software Composition Method.” In: *Electronic Proceedings in Theoretical Computer Science* 147: *Proceedings of the 11th International Workshop on Formal Engineering Approaches to Software Components and Architectures (FESCA 2014)*, pp. 108–122. ISSN: 2075-2180. DOI: `10.4204/EPTCS.147.8` (cit. on p. 280).

Kezadri, Mounira, Benoît Combemale, Marc Pantel, and Xavier Thirioux (2012). “A Proof Assistant Based Formalization of MDE Components.” In: *Proceedings of the 8th International Symposium Formal Aspects of Component Software (FACS 2011)*. Vol. 7253. Lecture Notes in Computer Science. Berlin / Heidelberg, Germany: Springer, pp. 223–240. ISBN: 978-3-642-35742-8. DOI: `10.1007/978-3-642-35743-5_14` (cit. on pp. 279, 280).

Kiczales, Gregor, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin (1997). “Aspect-Oriented Programming.” In: *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP ’97)*. Vol. 1241. Lecture Notes in Computer Science. Berlin / Heidelberg: Springer, pp. 220–242. ISBN: 978-3-540-63089-0. DOI: `10.1007/BFb0053381` (cit. on pp. 2, 63).

Kiczales, Gregor, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold (2001). “An Overview of AspectJ.” In: *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP ’01)*. Vol. 2072. Lecture Notes in Computer

Science. Berlin / Heidelberg: Springer, pp. 327–353. ISBN: 3-540-42206-4. DOI: `10.1007/3-540-45337-7_18` (cit. on pp. 2, 4, 13, 14, 63, 119, 262, 276).

Kifer, Michael, Georg Lausen, and James Wu (1995). "Logical Foundations of Object-oriented and Frame-based Languages." In: *Journal of the ACM* 42.4, pp. 741–843. ISSN: 0004-5411. DOI: `10.1145/210332.210335` (cit. on p. 279).

Klint, Paul, Tijs van der Storm, and Jurgen Vinju (2005a). "Term Rewriting Meets Aspect-Oriented Programming." In: *Processes, Terms and Cycles: Steps on the Road to Infinity*. Vol. 3838. Lecture Notes in Computer Science. Berlin / Heidelberg, Germany: Springer, pp. 88–105. ISBN: 978-3-540-30911-6. DOI: `10.1007/11601548_8` (cit. on p. 281).

Klint, Paul, Ralf Lämmel, and Chris Verhoef (2005b). "Toward an Engineering Discipline for Grammarware." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 14.3, pp. 331–380. ISSN: 1049-331X. DOI: `10.1145/1072997.1073000` (cit. on pp. 37, 81).

Kniesel, Günter (2009). "Detection and Resolution of Weaving Interactions." In: *Transactions on Aspect-Oriented Software Development V*. Vol. 5490. Lecture Notes in Computer Science. Berlin / Heidelberg: Springer, pp. 135–186. ISBN: 978-3-642-02058-2. DOI: `10.1007/978-3-642-02059-9_5` (cit. on p. 135).

Kniesel, Günter and Uwe Bardey (2006). "An Analysis of the Correctness and Completeness of Aspect Weaving." In: *Proceedings of the Working Conference on Reverse Engineering 2006 (WCRE 2006)*. Los Alamitos, CA, USA: IEEE, pp. 324–333. ISBN: 0-7695-2719-1. DOI: `10.1109/WCRE.2006.10` (cit. on pp. 4, 14, 135).

Knuth, Donald E. (1968). "Semantics of Context-Free Languages." In: *Mathematical Systems Theory* 2.2, pp. 127–145. ISSN: 0025-5661. DOI: `10.1007/BF01692511` (cit. on pp. 4, 46, 48, 49).

– (1971). "Semantics of Context-Free Languages: Correction." In: *Mathematical Systems Theory* 5.2, pp. 95–96. ISSN: 0025-5661. DOI: `10.1007/BF01702865` (cit. on pp. 4, 46, 48).

Kristensen, Bent B., Ole L. Madsen, and Birger Møller-Pedersen (2007). "The When, Why and Why Not of the BETA Programming Language." In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (HOPL III)*. New York, NY, USA: ACM, pp. 10–1–10–57. ISBN: 978-1-59593-766-7. DOI: `10.1145/1238844.1238854` (cit. on p. 64).

Kunert, Andreas (2008). "Semi-Automatic Generation of Metamodels and Models From Grammars and Programs." In: *Electronic Notes in Theoretical Computer Science (ENTCS)* 211: *Proceedings of the Fifth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006)*, pp. 111–119. ISSN: 1571-0661. DOI: `10.1016/j.entcs.2008.04.034` (cit. on p. 41).

Kästner, Christian and Sven Apel (2009). "Virtual Separation of Concerns - A Second Chance for Preprocessors." In: *Journal of Object Technology* 8.6, pp. 59–78. ISSN: 1660-1769. DOI: `10.5381/jot.2009.8.6.c5` (cit. on p. 275).

Kästner, Christian, Paolo G. Giarrusso, and Klaus Ostermann (2011a). "Partial Preprocessing C Code for Variability Analysis." In: *Proceedings of the 5th Workshop on Variability Modeling* *of Software-Intensive Systems (VaMoS '11)*. New York, NY, USA: ACM, pp. 127–136. ISBN: 978-1-4503-0570-9. DOI: `10.1145/1944892.1944908` (cit. on p. 275).

Kästner, Christian, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger (2011b). "Variability-aware Parsing in the Presence of Lexical Macros and Conditional Compilation." In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*. New York, NY, USA: ACM, pp. 805–824. ISBN: 978-1-4503-0940-0. DOI: `10.1145/2048066.2048128` (cit. on p. 275).

Kühnemann, Armin and Heiko Vogler (1997). *Attributgrammatiken – Eine grundlegende Einführung*. German. Braunschweig / Wiesbaden, Germany: Vieweg. ISBN: 3-528-05582-0 (cit. on p. 46).

Leeuwen, Jan van, ed. (1994). *Handbook of Theoretical Computer Science*. Vol. B: Formal Models and Semantics. Cambridge, MA, USA: The MIT Press. ISBN: 978-0-26272-015-1 (cit. on p. 19).

Lohmann, Daniel, Georg Blaschke, and Olaf Spinczyk (2004). "Generic Advice: On the Combination of AOP with Generative Programming in AspectC++." In: *Proceedings of the Third International Conference on Generative Programming and Component Engineering (GPCE 2004)*. Vol. 3286. Lecture Notes in Computer Science. Berlin / Heidelberg, Germany: Springer, pp. 55–74. ISBN: 978-3-540-23580-4. DOI: `10.1007/978-3-540-30175-2_4` (cit. on p. 277).

Ludwig, Andreas (2002). "Automatische Transformation großer Softwaresysteme." German. PhD thesis. Karlsruhe, Germany: Universität Karlsruhe. ISBN: 978-3-83221-106-6 (cit. on p. 73).

Maddox III, William H. (1997). "Incremental Static Semantic Analysis." UCB//CSD-97-948. PhD thesis. Berkeley, CA, USA: University of California (cit. on p. 50).

Madsen, Ole L., Birger Møller-Pedersen, and Kristen Nygaard (1993). *Object-Oriented Programming in the BETA Programming Language*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co. ISBN: 0-201-62430-3 (cit. on pp. 7, 65).

Magnusson, Eva (2007). "Object-Oriented Declarative Program Analysis." PhD thesis. Lund, Sweden: University of Lund. ISBN: 978-91-628-7306-6 (cit. on pp. 49, 234).

Mcilroy, Doug (1969). "'Mass Produced' Software Components." In: *Proceedings of Software Engineering Concepts and Techniques*. Brussels, Belgium: NATO Scientific Affairs Division, pp. 138–155 (cit. on p. 1).

Medeiros, Flávio, Márcio Ribeiro, and Rohit Gheyi (2013). "Investigating Preprocessor-Based Syntax Errors." In: *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences (GPCE 2013)*. New York, NY, USA: ACM, pp. 75–84. ISBN: 978-1-4503-2373-4. DOI: `10.1145/2517208.2517221` (cit. on p. 275).

Meyer, Bertrand (1992). "Applying 'Design by Contract'." In: *Computer* 25.10, pp. 40–51. ISSN: 0018-9162. DOI: `10.1109/2.161279` (cit. on pp. 1, 154, 155).

Mezini, Mira and Klaus Ostermann (2003). "Conquering Aspects with Caesar." In: *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD '03)*. New

York, NY, USA: ACM, pp. 90–99. ISBN: 1-58113-660-9. DOI: 10.1145/643603.643613 (cit. on p. 2).

Moonen, Leon (2001). "Generating Robust Parsers Using Island Grammars." In: *Proceedings of the Eighth Working Conference on Reverse Engineering 2001*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 13–22. ISBN: 0-7695-1303-4. DOI: 10.1109/WCRE.2001.957806 (cit. on pp. 6, 219, 221, 284).

Murata, Makoto, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi (2005). "Taxonomy of XML Schema Languages Using Formal Language Theory." In: *ACM Transactions on Internet Technology (TOIT)* 5.4, pp. 660–704. ISSN: 1533-5399. DOI: 10.1145/1111627.1111631 (cit. on p. 37).

Niederhausen, Matthias, Sven Karol, Uwe Aßmann, and Klaus Meißner (2009). "HyperAdapt: Enabling Aspects for XML." In: *Proceedings of the 9th International Conference on Web Engineering (ICWE '09)*. Vol. 5648. Lecture Notes in Computer Science. Berlin / Heidelberg: Springer, pp. 461–464. ISBN: 978-3-642-02817-5. DOI: 10.1007/978-3-642-02818-2_38 (cit. on pp. x, 232).

Nierstrasz, Oscar and Theo Dirk Meijler (1995). "Research Directions in Software Composition." In: *ACM Computing Surveys (CSUR)* 27.2, pp. 262–264. ISSN: 0360-0300. DOI: 10.1145/210376.210389 (cit. on p. 62).

Nierstrasz, Oscar and Dennis Tsichritzis, eds. (1995). *Object-Oriented Software Composition*. Hertfordshire, UK: Prentice Hall International Ltd. ISBN: 0-13-220674-9 (cit. on p. 1).

Nilsson-Nyman, Emma, Görel Hedin, Eva Magnusson, and Torbjörn Ekman (2009). "Declarative Intraprocedural Flow Analysis of Java Source Code." In: *Electronic Notes in Theoretical Computer Science (ENTCS)* 238.5: *Proceedings of the 8th Workshop on Language Descriptions, Tools and Applications (LDTA 2008)*, pp. 155–171. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2009.09.046 (cit. on p. 49).

Object Management Group (OMG) (2003–2013). *Modeling and Metadata Specifications*. Published: online available specifications. URL: http://www.omg.org/spec/\#M&M (visited on 05/08/2013) (cit. on p. 38).

– (2003). *MDA Guide, Version 1.0.1*. Ed. by Joaquin Miller and Jishnu Mukerji. URL: http://www.omg.org/cgi-bin/doc?omg/03-06-01 (visited on 03/21/2014) (cit. on p. 38).

– (2006). *OCL 2.0 Specification*. Published: online available specification. URL: http://www.omg.org/spec/OCL/2.0/ (visited on 03/21/2014) (cit. on p. 94).

– (2011a). *Business Process Model and Notation (BPNM) Specification, Version 2.0*. Published: online available specification. URL: http://www.omg.org/spec/BPMN/2.0/ (visited on 03/21/2014) (cit. on p. 264).

– (2011b). *MOF-to-XMI Mapping Specification, Version 2.4.1*. Published: online available specification. URL: http://www.omg.org/spec/XMI/2.4.1/ (visited on 04/17/2013) (cit. on p. 41).

Object Management Group (OMG) (2011c). *Meta Object Facility (MOF) Core Specification, Version 2.4.1*. Published: online available specification. URL: `http://www.omg.org/spec/MOF/2.4.1/PDF` (visited on 04/17/2013) (cit. on pp. 37–40).

– (2011d). *Unified Modeling Language (UML), Version 2.4.1*. Published: online available specification. URL: `http://www.omg.org/spec/UML/2.4.1/` (visited on 04/17/2013) (cit. on pp. 38, 96, 264).

– (2012a). *CORBA Component Model, Version 3.3*. Published: online available specification. URL: `http://www.omg.org/spec/CORBA/3.3/Components/PDF` (visited on 02/27/2013) (cit. on p. 63).

– (2012b). *Common Object Request Broker Architecture (CORBA), Version 3.3*. Published: online available specification. URL: `http://www.omg.org/spec/CORBA/3.3/` (visited on 02/27/2013) (cit. on p. 63).

Odersky, Martin, Lex Spoon, and Bill Venners (2008). *Programming in Scala: A Comprehensive Step-by-step Guide*. 1st ed. Walnut Creek, CA, USA: Artima Incorporation. ISBN: 978-0-98153-160-1 (cit. on p. 13).

OpenMP Architecture Review Board (2013). *OpenMP Application Program Interface Version 4.0*. Published: online available specification. URL: `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf` (visited on 11/12/2013) (cit. on pp. 246, 289).

Öqvist, Jesper and Görel Hedin (2013). "Extending the JastAdd Extensible Java Compiler to Java 7." In: *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '13)*. New York, NY, USA: ACM, pp. 147–152. ISBN: 978-1-4503-2111-2. DOI: `10.1145/2500828.2500843` (cit. on p. 165).

Ossher, Harold and Peri Tarr (2001). "Using Multidimensional Separation of Concerns to (Re)shape Evolving Software." In: *Communications of the ACM* 44.10, pp. 43–50. ISSN: 0001-0782. DOI: `10.1145/383845.383856` (cit. on p. 2).

Paakki, Jukka (1995). "Attribute Grammar Paradigms—a High-Level Methodology in Language Implementation." In: *ACM Computing Surveys (CSUR)* 27.2, pp. 196–255. ISSN: 0360-0300. DOI: `10.1145/210376.197409` (cit. on p. 49).

Parr, Terence (2006). *A Functional Language For Generating Structured Text*. Tech. rep. San Fransisco, CA, USA: University of San Francisco (cit. on pp. 11, 232, 262, 271).

Parr, Terence and Russell Quong (1995). "ANTLR: A Predicated-LL(k) Parser Generator." In: *Software—Practice & Experience* 25.7, pp. 789–810. ISSN: 0038-0644. DOI: `10.1002/spe.4380250705` (cit. on p. 34).

Pop, Adrian, Ilie Savga, Uwe Aßmann, and Peter Fritzson (2005). "Composition of XML Dialects: A ModelicaXML Case Study." In: *Electronic Notes in Theoretical Computer Science (ENTCS)* 114: *Proceedings of the Software Composition Workshop (SC 2004)*, pp. 137–152. ISSN: 1571-0661. DOI: `10.1016/j.entcs.2004.02.071` (cit. on pp. 64, 80).

Potts, Colin (1993). "Software-Engineering Research Revisited." In: *IEEE Software* 10.5, pp. 19–28. ISSN: 0740-7459. DOI: `10.1109/52.232392` (cit. on p. 61).

Pratt-Szeliga, Philip C., James W. Fawcett, and Roy D. Welch (2012). "Rootbeer: Seamlessly Using GPUs from Java." In: *Proceedings of the 14th International Conference on High Performance Computing and Communication (HPCC'12)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 375–380. ISBN: 978-1-4673-2164-8. DOI: `10.1109/HPCC.2012.57` (cit. on p. 212).

Reps, Thomas, Tim Teitelbaum, and Alan Demers (1983). "Incremental Context-Dependent Analysis for Language-Based Editors." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 5.3, pp. 449–477. ISSN: 0164-0925. DOI: `10.1145/2166.357218` (cit. on p. 50).

Schöning, Uwe (2001). *Theoretische Informatik – kurzgefasst*. German. 4th ed. Heidelberg / Berlin: Spektrum. ISBN: 3-8274-1099-1 (cit. on pp. 19, 23, 33, 34).

Semantic Designs, Inc. (1995–2014). *DMS®Language/Compiler Front Ends*. URL: `http://www.semdesigns.com/Products/FrontEnds/` (visited on 10/13/2014) (cit. on p. 282).

Simonyi, Charles (1976). "Meta-Programming: A Software Production Method." XEROX PARC: CSL-76-7. PhD thesis. Stanford, CA, USA: Stanford University (cit. on p. 74).

Sloane, Anthony M. (2011). "Lightweight Language Processing in Kiama." In: *Generative and Transformational Techniques in Software Engineering III*. Vol. 6491. Lecture Notes in Computer Science. Berlin / Heidelberg: Springer, pp. 408–425. ISBN: 978-3-642-18022-4. DOI: `10.1007/978-3-642-18023-1_12` (cit. on p. 49).

Sloane, Anthony M., Lennart C. L. Kats, and Eelco Visser (2010). "A Pure Object-Oriented Embedding of Attribute Grammars." In: *Electronic Notes in Theoretical Computer Science (ENTCS)* 253.7: *Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009)*, pp. 205–219. ISSN: 1571-0661. DOI: `10.1016/j.entcs.2010.08.043` (cit. on p. 49).

Software Technology Group (2013). *JastEMF: JastAdd Attribute Grammars for EMF Metamodels, Version 0.1.7*. URL: `http://www.jastemf.org/` (visited on 03/28/2014) (cit. on pp. 50, 287).

Software Technology Group and DevBoost GmbH (2013a). *Java Model Parser and Printer, Version 1.4.0*. URL: `http://www.jamopp.org/` (visited on 02/13/2013) (cit. on p. 96).

– (2013b). *Reuseware, Version 1.0.1*. URL: `http://www.reuseware.org/` (visited on 02/13/2013) (cit. on pp. 82, 92).

Spinczyk, Olaf, Andreas Gal, and Wolfgang Schröder-Preikschat (2002). "AspectC++: An Aspect-oriented Extension to the C++ Programming Language." In: *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications (CRPIT '02)*. Darlinghurst, Australia: Australian Computer Society, pp. 53–60. ISBN: 0-909925-88-7 (cit. on p. 277).

Steele Jr., Guy L. (1990). *Common LISP: The Language*. 2nd ed. Newton, MA, USA: Digital Press. ISBN: 1-55558-041-6 (cit. on p. 3).

Steinberg, David, Frank Budinsky, Marcelo Paternostro, and Ed Merks (2009). *Eclipse Modeling Framework*. 2nd ed. Boston, MA, USA: Pearson Education, Inc. ISBN: 978-0-321-33188-5 (cit. on p. 42).

Sun Microsystems, Inc. (2009). *JSR 222: Java Architecture for XML Binding (JAXB), Version 2.2*. Published: online available specification. URL: `https://jcp.org/en/jsr/detail?id=222` (visited on 04/08/2014) (cit. on p. 38).

Sutton, Andrew and Bjarne Stroustrup (2012). "Design of Concept Libraries for C++." In: *Proceedings of the Fourth International Conference on Software Language Engineering (SLE 2011)*. Vol. 6940. Lecture Notes in Computer Science. Berlin / Heidelberg, Germany: Springer, pp. 97–118. ISBN: 978-3-642-28829-6. DOI: `10.1007/978-3-642-28830-2_6` (cit. on pp. 274, 290).

Synytskyy, Nikita, James R. Cordy, and Thomas R. Dean (2003). "Robust Multilingual Parsing Using Island Grammars." In: *Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '03)*. Indianapolis, IN, USA: IBM Press, pp. 266–278 (cit. on p. 219).

Szyperski, Clemens (2002). *Component Software: Beyond Object-Oriented Programming*. 2nd ed. Boston, MA, USA: Addison-Wesley. ISBN: 978-0-201-74572-6 (cit. on pp. 1, 62).

Söderberg, Emma (2012). "Contributions to the Construction of Extensible Semantic Editors." PhD thesis. Lund, Sweden: University of Lund. ISBN: 978-91-976939-8-1 (cit. on pp. 50, 234).

Tarr, Peri, Harold Ossher, William Harrison, and Stanley M. Sutton Jr (1999). "N Degrees of Separation: Multi-Dimensional Separation of Concerns." In: *Proceedings of the 21st International Conference on Software Engineering (ISCE'99)*. New York, NY, USA: ACM, pp. 107–119. ISBN: 1-58113-074-0. DOI: `10.1145/302405.302457` (cit. on p. 2).

Tarski, Alfred (1955). "A Lattice-Theoretical Fixpoint Theorem and its Applications." In: *Pacific Journal of Mathematics* 5.2, pp. 285–309. ISSN: 0030-8730. DOI: `10.2140/pjm.1955.5-2` (cit. on p. 49).

Тасић, Марта (2014). *Benchmarking Incremental Dynamic Attribute Evaluation - RACR/JastAdd Performance Case Study in the Domain of Invasive Software Composition*. Published: online available slide set. Dresden, Germany: Technische Universität Dresden. URL: `http://tinyurl.com/pr8p8r2` (visited on 10/14/2014) (cit. on p. 282).

Tatsubori, Michiaki, Shigeru Chiba, Marc-Olivier Killijian, and Kozo Itano (2000). "OpenJava: A Class-Based Macro System for Java." In: *Reflection and Software Engineering*. Vol. 1826. Berlin / Heidelberg, Germany: Springer, pp. 117–133. ISBN: 978-3-540-67761-1. DOI: `10.1007/3-540-45046-7_7` (cit. on p. 274).

Teitelman, Warren (1966). "PILOT: A Step Toward Man-Computer Symbiosis." AITR-221. PhD thesis. Cambridge, MA, USA: Massachusetts Institute Of Technology (cit. on p. 63).

The Apache Software Foundation (2014). *The Apache ANT Project*. URL: `http://ant.apache.org/` (visited on 10/02/2014) (cit. on p. 175).

The Unicode Consortium (2014). *Unicode Standard, Version 7.0.0*. Published: online available specification. URL: `http://www.unicode.org/versions/Unicode7.0.0/` (visited on 09/24/2014) (cit. on p. 54).

Thiele, Michael (2011). “Attribute Grammars for Weaving EMF-based Metamodels.” Master thesis. Dresden, Germany: Technische Universität Dresden (cit. on pp. 151, 152, 160).

Tomita, Masaru (1991). *Generalized L.R. Parsing*. Norwell, MA, USA: Kluwer Academic Publishers. ISBN: 978-0-792-39201-9 (cit. on p. 35).

Vinju, Jurgen (2005). *A Type Driven Approach to Concrete Meta Programming*. Tech. rep. E 0507. Amsterdam, The Netherlands: Centrum Wiskunde & Informatica, pp. 1–20 (cit. on p. 11).

Visser, Eelco (1997). “Syntax Definition for Language Prototyping.” PhD thesis. Amsterdam, The Netherlands: University of Amsterdam (cit. on pp. 221, 272, 280).

– (2004). “Program Transformation with Stratego/XT.” In: *Proceedings of the International Seminar on Domain-Specific Program Generation*. Vol. 3016. Lecture Notes in Computer Science. Springer, pp. 216–238. ISBN: 978-3-540-22119-7. DOI: `10.1007/978-3-540-25935-0_13` (cit. on p. 4).

Vogt, Harald H., Doaitse Swierstra, and Matthijs F. Kuiper (1989). “Higher Order Attribute Grammars.” In: *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation (PLDI ’89)*. New York, NY, USA: ACM, pp. 131–145. ISBN: 0-89791-306-X. DOI: `10.1145/73141.74830` (cit. on pp. 49, 127).

Weise, Daniel and Roger Crew (1993). “Programmable Syntax Macros.” In: *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI ’91)*. New York, NY, USA: ACM, pp. 156–165. ISBN: 0-89791-598-4. DOI: `10.1145/155090.155105` (cit. on p. 3).

Wikimedia Foundation, Inc (2014). *MediaWiki API Documentation*. URL: `http://www.mediawiki.org/wiki/API:Main_page` (visited on 09/01/2014) (cit. on p. 208).

Wikipedia contributors (2014a). *Algorithmic Skeleton*. URL: `http://en.wikipedia.org/wiki/Algorithmic_skeleton` (visited on 08/22/2014) (cit. on p. 200).

– (2014b). *Wiki Markup*. URL: `http://en.wikipedia.org/wiki/Help:Wiki_markup` (visited on 09/01/2014) (cit. on p. 206).

Wilhelm, Reinhard and Dieter Maurer (1997). *Übersetzerbau – Theorie, Konstruktion, Generierung*. German. 2nd ed. Berlin / Heidelberg: Springer. ISBN: 3-540-61692-6 (cit. on pp. 34, 35, 46, 145).

Wirth, Niklaus (1971). “Program Development by Stepwise Refinement.” In: *Communications of the ACM* 14.4, pp. 221–227. ISSN: 0001-0782. DOI: `10.1145/362575.362577` (cit. on p. 277).

World Wide Web Consortium (W3C) (1998–2009). *Extensible Markup Language (XML)*. Published: online available specification. URL: `http://www.w3.org/XML` (visited on 09/23/2014) (cit. on p. 37).

– (1998–2012). *Document Object Model (DOM), Technical Reports*. Published: online available specification. URL: `http://www.w3.org/DOM/DOMTR` (visited on 04/18/2013) (cit. on p. 37).

– (2012). *XML Schema, Version 1.1*. Published: online available specification. URL: `http://www.w3.org/XML/Schema` (visited on 03/21/2013) (cit. on p. 37).

Wyk, Eric Van, Lijesh Krishnan, Derek Bodin, and August Schwerdfeger (2007). “Attribute Grammar-Based Language Extensions for Java.” In: *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP ’07)*. Vol. 4609. Lecture Notes in Computer Science. Berlin / Heidelberg: Springer, pp. 575–599. ISBN: 978-3-540-73588-5. DOI: `10.1007/978-3-540-73589-2_27` (cit. on p. 50).

Wyk, Eric Van, Derek Bodin, Jimin Gao, and Lijesh Krishnan (2008). “Silver: an Extensible Attribute Grammar System.” In: *Electronic Notes in Theoretical Computer Science (ENTCS)* 203.2: *Special Issue on ETAPS 2006 and 2007 Workshops on Language Descriptions, Tools, and Applications (LDTA ’06 and ’07)*, pp. 103–116. ISSN: 1571-0661. DOI: `10.1016/j.scico.2009.07.004` (cit. on p. 49).