TECHNISCHE UNIVERSITÄT DARMSTADT

Implementing and Composing MDSD-Typical DSLs

TR-Nr.: TUD-CS-2009-0156

Tom Dinkelaker, dinkelaker@st.informatik.tu-darmstadt.de
Technische Universität Darmstadt, Germany

Christian Wende, c.wende@tu-dresden.de
Technische Universität Dresden, Germany

Henrik Lochmann, HLochmann@gmx.net
Technische Universität Dresden, Germany

Abstract. In this document, we have studied two orthogonal approaches of building DSLs and their advantages and disadvantages with respect to MDSD. We show that embedded DSLs can be used to implement a MDSD-typical DSL rapidly. Further, we show that embedded DSLs and aspect-oriented programming can be used in concert. We also discuss how modular language engineering and language composition enables new reuse capabilities among modelling languages with a slightly higher initial development effort.

Acknowledgements. This work was partly supported by the feasiPLe project, Federal Ministry of Education and Research (BMBF), Germany.

Table of Contents

1. Introduction

The specification of software systems benefits from appropriate abstractions for specific system concerns. As usually software consists of multiple concerns, hence, a set of domain-specific languages (DSLs) is used in combination. The traditional approach of language development requires the developer to provide parsers, compilers, and development tools for every DSL, which is a cost-intensive task. When using multiple DSLs in model-driven software development (MDSD), and at the same time, and these DSLs even evolve, the effort to implement a DSL is increased massively. Hence, means to reduce development costs for DSLs are necessary.

One possibility is to make use of existing general-purpose languages for DSL development with the creation of so-called *embedded domain-specific languages* (EDSLs). This bottom-up approach allows rapidly providing new DSLs by shaping the syntax and semantics of an existing general-purpose host language to represent the concepts of the DSL as close as possible. Thus, EDSLs can partially reuse the parsers, compilers and development tools of their host languages. Implementing EDSLs is a well-known technique in many languages, like *Ruby* [Ruby], *Groovy* [Groovy], *Scala* [Scala], *Haskell* [Haskell], and *ML* [MTM+97]. For example, the popular *Ruby on Rails* Web framework [Rails] strongly utilizes the EDSL approach for a family of DSLs. Domain-specific literals and operators are introduced using language constructs provided by the host language and must be interpreted by a library that implements the domain logic. It follows that the domain-specific code needs to conform to the host language syntax. The embedding of DSL constructs into host language programs simultaneously addresses the need for semantic connection of expressions in different multiple DSLs.

Another possibility to improve the development of DSLs and their combination in particular scenarios is to extend traditional language development techniques with means for syntactic and semantic language composition. This top-down approach allows to implement single DSLs or even DSL parts in modular units and to integrate developed languages to form a platform for system specification. This approach tends to a higher initial effort for DSL implementation, but provides more flexibility regarding language syntax and feature reuse.

In this document we describe two exemplary representatives of both approaches and compare their advantages and disadvantages. Section 1.1 introduces a general classification scheme for DSL realisation and integration approaches. It distinguishes between invasive and non-invasive language composition techniques regarding syntax and semantics. Using this classification scheme we describe existing language composition approaches and discuss their appropriateness in different application scenarios. Section 1.3 introduces a flexible approach for embedding DSLs in dynamic object-oriented host languages (Groovy and Ruby) and discusses which language features are crucial for the host language to be applicable. A compositional approach for building DSLs independent of particular host languages is described in Section 1.8. In addition we present a language composition example involving several integrated DSLs. Section 1.13.3 compares the presented approaches for DSL engineering and our findings are concluded in Section 1.13.3.

2. A Classification Scheme for Language Composition Approaches

Composing languages is a difficult challenge that involves two major tasks [KL07]. One the one hand, the most difficult task is to derive a *composition specification* for merging the different language specifications to form a combined syntax and semantics. On the other hand, an implementation for the combined language must be provided, whereby a framework could ease this task through allowing *minimum performance overhead*, *maximum code reuse*, *auto-configuration*, and *manual override*.

We developed different approaches for the composition implementation of multiple DSLs, which composes their *syntax* and may also compose their *semantics*. We identified different composition approaches in related work that support for different flavours of composition, which we differentiate according to the influence the composed parts pose on each other. While *non-invasive* composition does not affect the implementation of the languages, *invasive* composition may change the language's interfaces and their implementation. We additionally distinguish between *centralized* and *decentralized* approaches for the specification of language composition. To round up the categorization of the language composition approaches, we provide a classification scheme that comprises the three above described dimensions, depicted in Figure 1.

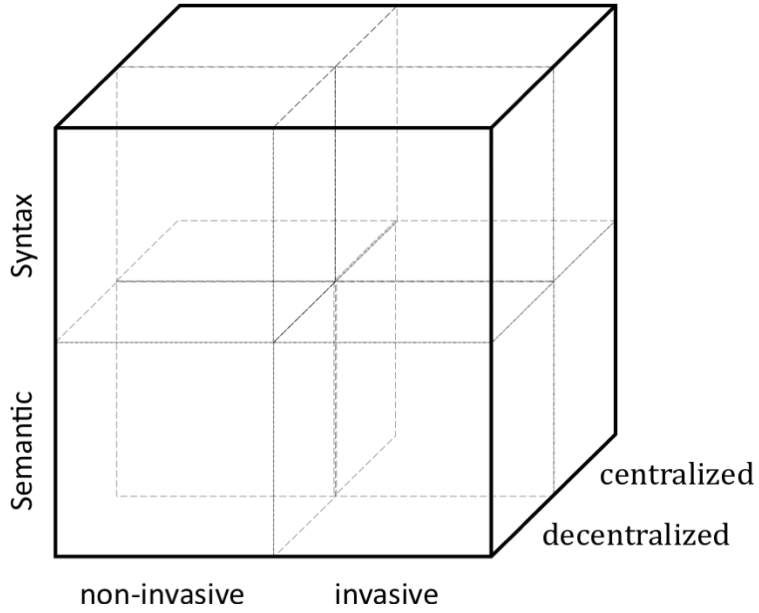


Figure 1 – Classification Scheme for Language Composition Approaches

1.1 Syntax / Semantic Composition

Concerning the syntax of two languages, their corresponding expressions (such as keywords, block statements, or other phrases) may remain untouched or not after composition. We call composition without adapting the syntax of the composed languages *non-invasive syntax composition*, while adapting the syntax of at least one of the languages is called *invasive-syntax composition*. Embedded DSLs are by definition syntax composition, whereby the DSL syntax does not introduce new concepts into its host language. Because an embedded DSL does not violate the syntax of the host

language, the host language's parser and compiler can be reused.

Good examples for non-invasive composition are *Java annotations* [Sun04] that allow adding domain-specific syntax to Java code. For instance, *Hibernate* [Hiber] uses annotations for specifying persistency requirements. Annotations are a generic extension mechanism of the language allows for DSL extensions to be added to the Java syntax inside annotation blocks, which does not affect the core Java syntax. Other examples are XML-based DSLs that are specified through a *XML Schema Definition* (XSD). Since each DSL conforms to the XML core syntax and multiple XML documents may use different XSD, an XML document may use primitives from multiple DSL definitions by using a *name space* prefix in front of the tag name. Note that used XSD Schemas must have been imported before. In effect, name space allow several DSLs to be composed without affecting DSL syntax of each other (or respectively the core XML syntax) since the XSD files are only imported.

In addition, the characteristics of such a composition could vary in the extent the syntaxes of the languages are mixed with each other. The least syntactical mixed form of composition would only allow different DSL syntax to be used in different type of modules. E.g., in the *Struts Web framework* [Struts], while for many purposes Java is used, it leverages DSL in several ways: views are created in JSP files which support domain-specific tag libraries, the page flows are specified in XML configuration files, and . There are examples in which the syntax is composed such that the programmer may use either the one syntax or the other to specify a part of a module. E.g., in *Pascal* [Pascal], one can open an assembler block to use assembler code instead of high-level code. In the strongest form of mixing the syntaxes, the syntaxes could be completely merged into one, such that keywords of different composed languages can be used in any module, scope, or context.

From the perspective of language semantics, on the one hand, *non-invasive language semantic composition* composes two embedded languages that do not interfere with each other. E.g., we can embed SQL into Java without changing the fundamental Java syntax and semantics. We call languages that do not interfere in their specification and implementation *independent*. Two independent languages can be integrated at a well-defined interface, i.e., by using a shared type, e.g., we can integrate Java and SQL using a *cursor type* that iterates over rows in a SQL result table. The composition of two independent languages can be using pre-processors and the composition specification is expected to be rather small. On the other hand, *invasive language semantic composition* is a difficult problem that must cope with syntactical and semantic interactions in language composition specification and composition implementation. E.g., the *Java Security Framework* [Oak01] of the Java VM composes Java code and a security policy. The policy is written in a DSL for describing permissions for un-trusted code that originates from various sources in particular Applets from the Internet. In case, general-purpose Java programs are execute under different semantics that enforces the defined policies. If a security policy is violated, the semantics of Java code is changed such that an exception is thrown.

Figure 2 shows the different dimensions/flavours of composition together with well-known examples:

Composition	Non-invasive	Invasive
Syntax	Java/JPA Annotations	Embedded SQL
Semantics	Embedded SQL	DSL for Java Security Policy

Figure 2 – Dimensions/flavours of composition and their representatives.

1.2 Non-Invasive / Invasive Composition

In case of non-invasive semantics composition the syntax of multiple languages is composed, while the semantics of each of the language stays unchanged. Concerning the composition specification, non-invasive semantics composition integrates multiple syntaxes without changing the semantics of the composed languages. E.g., *embedded SQL* is a good example of non-invasive composition in programming languages that makes the SQL syntax directly available in languages such a C or Java. Still, the base syntax of the language is not changed, i.e., the syntax and semantics of classes is not changed. The composed languages must be bridged so that one can transfer a value from one language's scope to another language's scope. E.g., use a Java value in an SQL query and iterate over the query result in Java. Such integration can be achieved by using variables and types shared between two or more domains. If no such shared type exists, the specifications of the languages can be appended with a new shared type.

DSL embeddings and compositions elaborated so far are black-box. EDSL semantics is defined on top of the semantic of the hosting language without changing the latter as well as the embeddings do not interfere with each other. However, such black-box embedding/composition is not always appropriate. E.g., we have identified situations in which DSL from different domains are composed in their execution. But such compositing of several DSLs that are embedded as libraries is complicated, because the specification and the implementations of the libraries become dependent on each other. In the first way, the composition specification is a hard problem, as the language designer must determine all point where the languages could interfere. For each interference, an appropriate way for resolution of potential conflicts must be provided. A complete discussion the specification problem is out of the scope. We refer to [CE00], which discusses this problem of *aspectual composition* of languages. Nonetheless, we elaborate a special case of invasive composition, namely *crosscutting composition*.

1.3 Decentralize / Centralized Composition

In our scheme, we additionally distinguish between centralized and decentralized approaches for the specification of language composition. Here, *centralized* characterizes approaches that implement language composition with a dedicated pivotal asset, such as a common interface or mapping paradigm. With *decentralized*, we indicate approaches that implement the composition of two or more languages case by case in a peer-to-peer manner, where each language pair is composed in isolation.

3. Implementing Embedded Domain-specific Languages

With the increasing complexity of applications, the use of domain-specific languages (DSLs) is becoming very important. Using domain-specific abstractions increases the abstraction level and decreases the representational gap between the way domain experts think and the way domains are modelled in programs, facilitating understanding and maintenance. However, these advantages of DSLs have their price. Traditionally, pre-processors are used for introducing domain-specific abstractions into general-purpose languages (GPLs). While it enables to implement domain-specific syntactic and semantic analyzes, this approach is labour intensive. A sophisticated language processing infrastructure has to be built on top of the infrastructure available for the hosting GPL. Furthermore, it is a well-known problem that DSLs implemented by means of this approach are hard to compose.

To address these problems, Hudak [Hud96] introduced the notion of a *domain-specific embedded language.* We will refer to such a language as an *embedded domain-specific language* (EDSL), while others refer to such a language an *internal domain-specific language*. Roughly speaking, these are DSLs that are implemented as libraries in a hosting language. Following this approach allows reusing the general-purpose features of the *host language*. The reuse of features implemented in the host language significantly reduces the development costs of language features of the embedded DSL [Fow05]. Further no parser and compiler has to be implemented, and tools for the host language can be used.

In this section, we investigate how a textual DSL that is often found in MDSD can be implemented as an embedded DSL in another programming language.

1.4 Language Features for Embedding Abstractions

Embedding textual domain-specific language has been used in many programming languages, such as the aforementioned Ruby, Groovy, Scala, Haskell, and ML. A question that remains to be answered is: what language features qualifies a language to be good for embedding. And, what is the best approach to embed a language. Giving a complete answer to these questions is out-of-scope of this document. Moreover, the questions may also be asked for non-textual languages, such as UML 2.0 that allows domain-specific syntax to be embedded using UML Profiles, that can be seen a light-weight extension mechanism or facility that supports embeddings. Further, one can ask with what features today languages should be extended with for better supporting embedding DSLs. A common denominator for all approaches for embedding DSLs is that they need some kind of *extension mechanism* in the host language that opens the host language for adding domain-specific abstractions.

In the Figure 3, an overview of languages, their support for embedding DSLs is given, and which extension mechanism can be used for embedding domain-specific abstractions.

Language	Techniques	Extension Mechanisms
Haskell	Monads [Hud96]	Algebraic types, functions
Java	Domain-specific libraries [Fowler05][FP06]	Class loading
Groovy	Design patterns, pretended method calls [Groovy]	Dynamic features / flexible syntax, closures, meta-object protocols
Ruby	Design patterns, pretended method calls [Ruby]	Dynamic features / flexible syntax, closures, meta-object protocols
Scala	Embedding into the type system [OSV07], Polymorphic embedding [HORM08]	Traits (mixin composition), imports
UML 2.0	Meta modelling	UML profiles, meta-object facility
XML	Extensible language platform	Namespaces

Figure 3 – Languages and feature for embedding

In this document, we discuss a selection of languages and approaches for embedding DSLs. In particular, we study dynamic object-oriented programming languages and how embedded DSL can be used in MDSD. We focus on Groovy and Ruby because of the openness in these languages.

1.4.1 Java

Although Java has no special language feature for embedding DSL, it has been used to embed languages in form of domain-specific libraries. This approach called a *fluent interface* is discussed by Fowler [Fow05b] and by Freeman and Pryce [FP06]. The knowledge and semantics of a problem domain are embedded, whereby domain objects are mapped to a set of classes and operations on them are mapped to methods. A major problem with embedding DSLs in Java is that there is a large *syntax noise* that requires DSL programs to use more or different characters in the EDSL syntax. Using additional or different characters is necessary in certain cases to make the concrete DSL syntax of the EDSL conforms to the host language syntax. This syntax noise can be measured relative to the *ideal syntax* of a standalone DSL implementation with its own syntax that can be designed without restriction only to serve the user's need. Possible metrics are the *Levenshtein distance* [Lev65].

1.4.2 Groovy

Groovy [Groovy] is a pure object-oriented scripting language that nicely integrates with Java [GJSB00]. The syntax is close to Java and one can call Groovy code from Java and vice versa without converting passed objects. Groovy provides special features that facilitate the embedding of DSLs: a *flexible syntax*, a *meta-object protocol*, and *closures* as first-class entities. Worth to mention, we later use Groovy to present one possible implementation of our EDSL architecture. This paragraph briefly introduces those features relevant for understanding the embedded DSL implementation. For a more comprehensive introduction to Groovy we refer to [Koe07].

Groovy's flexible syntax offers syntactic sugar for collection types and for passing parameters to methods. This flexibility results in EDSL syntax with a small *syntax noise*,

i.e., the amount of code one has to write in addition or which is written different to an idealized DSL syntax. Groovy provides a meta-object protocol (MOP), which enables *pretended method* calls (and *pretended properties*), that are methods (or respectively properties) that are not defined in the object's class and for those a method or respectively a field access can be forwarded to other methods or objects. When one invokes a method on such an object, respectively access a field, not defined in the object's class. Such "pretended" methods, respectively fields, are handled by the meta-object associated with the receiver, if any, called *delegate*. In particular, we later use the MOP to allow the dispatch of keywords in DSL programs that are treated as pretended method calls.

A Groovy closure is a first-class entity that can be referenced and that can be used to defer the evaluation of a piece of code. Closures are defined using curly brackets. E.g., `Closure cl = {x -> x*x}`, defines a closure that takes the parameter `x` and returns its square value, thus `cl.call(5)` will return `25`. A Groovy closure does not encapsulate a fixed evaluation context. A closure also may be assigned a *delegate* to whom any unbound symbols are dispatched. In particular, we later use closures to realize nested code structures and inject domain object as symbols in the evaluation context.

1.4.3 Ruby

Ruby [Ruby] is a dynamic object-oriented scripting language for which several implementations on different platforms are available. In comparison to Groovy, Ruby provides similar features that can be used to embed DSLs. In particular, Groovy was inspired by the Ruby language and its features. It is targeted to provide those features for Java. Note that we have used these features to repeat our EDSL implementation approach for Ruby, as we have used the corresponding features for Groovy. In contrast to Groovy, the Ruby language does provide more dynamicity with respect to its module system, e.g., Ruby support dynamic *mixins*.

There are minor differences with respect to the features that we use to embed DSLs. The syntax of Ruby has shown to be more flexible to design the concrete syntax of an EDSL closer to the abstract syntax [Fow2003]. Ruby supports closures in form of so-called *blocks*[1]. The Ruby MOP allows pretended methods as Groovy and moreover provides several abstractions on MOP operations.

From the perspective of model-driven software engineering, an advantage of Ruby is that the language implementation is not bound to a particular language platform, as there are Ruby implementations that are standalone, compiled, based on the Java and C# stack available. Thus, Ruby has a better prospect with respect to platform variability.

Despite we have implemented the architecture for EDSL implementation both in Groovy and in Ruby, for the sake of brevity, we present only the Groovy implementation.

---

[1] In contrast to Groovy closures, Ruby blocks do not support *delegates* that are necessary for the approach followed by the rest of the paper. However, we can use a different context for a block (with DSL code that uses DSL keywords) by evaluating the block in foreign context. This can be done by calling `instance_eval` method on a *delegate* object (that implements methods for DSL keywords). Whereby, passing the block as a parameter to the `instance_eval` method. This will dispatch calls inside the block to the evaluation context from which the block is called, thus to the instance in which the block call is evaluated – the delegate object.

1.5 An Architecture for EDSL Implementation

We have developed a new architecture and framework for implementing embedded DSLs, called POPART that is implemented in Groovy. We propose to use a host language for embedding DSLs that supports a set of language features including *closures*, a *meta-object protocol* (MOP), and *object-oriented programming*. We present an approach to implementing EDSLs that combines the power of these features to achieve modular, flexible, and composable embeddings of DSLs. On top of this, the OO mechanisms of interfaces and sub-classing enable pluggable interpreters and allow for defining new interpreters by reusing the infrastructure built for existing EDSLs, or other libraries. Object composition enables flexible black-box composition of EDSLs. Based on these features, the ongoing research investigates embedding aspect-oriented language constructs, which can be used to support what we call *crosscutting composition* of DSLs; as opposed to black-box composition, the interpretation of EDSLs to be composed is changed by the composition.

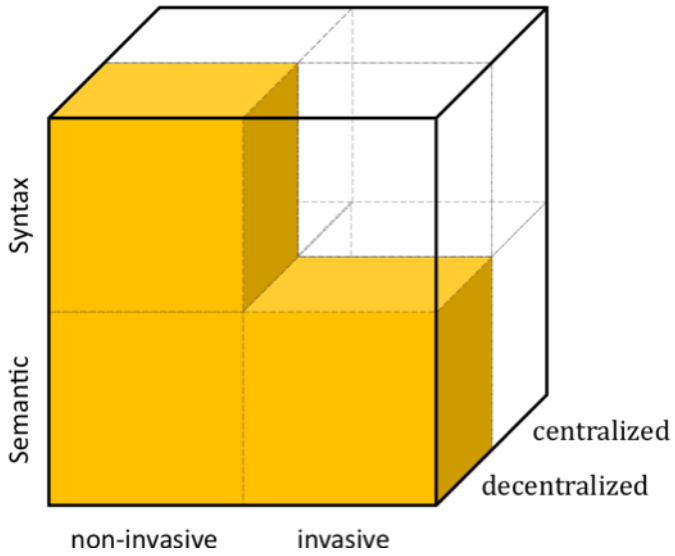


Figure 4 – Classification of the applicability of the POPART architecture

The POPART architecture we propose can be classified with respect to the classification schema introduced in Section 1.1. POPART supports a decentralized implementation of EDSLs and allows non-invasive syntax composition. Multiple domain-specific extensions that are implemented in POPART and on top of the same host language have the same concrete syntax of the host language. Because of this property, one can mix DSL keywords without writing a new parser. Further, POPART allows non-invasive semantic compositions of DSLs as modular and hierarchical DSL implementations and also provides support for *polymorphic embedding* [HORM2008]. Moreover, invasive semantic composition is supported by providing either using a generic semantic composition operator provided by POPART or by implementing a custom semantic composition comparator. In particular, invasive semantics composition is used for crosscutting composition of different DSLs, where the execution of one DSL program influences the execution (semantics) of another DSL program.

1.5.1 A Layered Architecture for Implementing EDSLs

We propose a four-layered architecture for embedding DSLs in Groovy (cf. right-hand side of Figure 5). At the first layer (P), there are DSL programs which use DSL elements, i.e., domain-specific primitives and domain-specific means of composition/abstraction [ASS96]. At the second layer (L), there are DSL interfaces, which declare operations for each DSL element. These operations are implemented by DSL interpreters in the third layer (I). DSL interpreters make use of classes from domain meta-models, each modelling domain abstractions, eventually by reusing existing types from the host language or from a library. Domain meta-models constitute the fourth layer (M) of our architecture. In the implementation of the architecture in Groovy, EDSL programs are enclosed in Groovy closures and the Groovy MOP automatically maps between DSL elements used in a program and the corresponding operations in a DSL interface. Together, the (L)-, (I)-, and (M)-layer form a run-time, that can be used to evaluate DSL programs, which we refer to in the following as a LIM run-time.

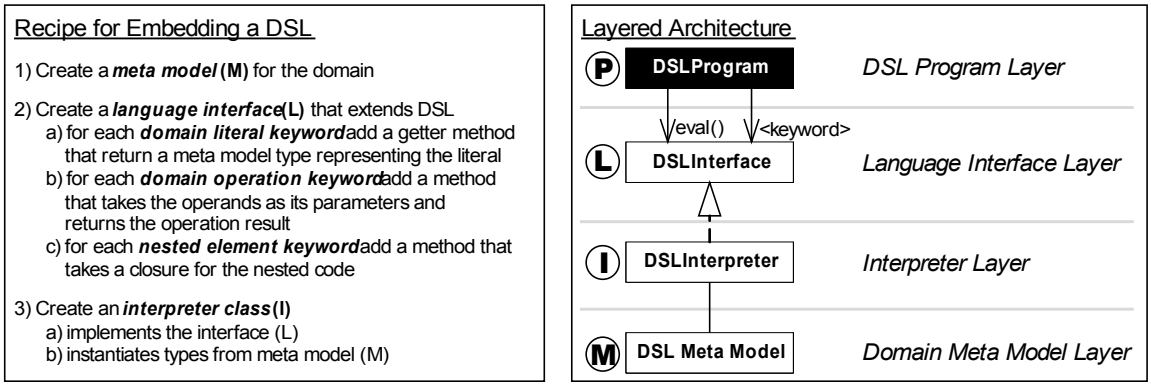


Figure 5 - Architecture and a Recipe for Embedding DSLs

A recipe of steps to follow for instantiating the proposed architecture is given on the left-hand side of Figure 5. The right-hand side of Figure 5 shows an overview of the instantiation of the DSL architecture for an embedding *DSL*.

1.5.2 Tailoring the DSL Syntax

```
machine Watch {
  start state reseted {
    entry: resetTimer;
    transitions {
      when start enter running;
      when switchOff enter off;
    }
  }

  state running {
    entry: startTimer;
    do: loopRunning;
    transitions {
      when split enter paused;
      when stop enter stopped;
    }
  }

  state paused {
    entry: pauseTimer;
    transitions {
      when unsplit enter running;
      when stop enter stopped;
    }
  }

  state stopped {
    entry: stopTimer;
    transitions {
      when reset enter stopped;
      when switchOff enter off;
    }
  }

  state off {
    exit: switchOff;
    transitions {
      when toEnd end;
    }
  }
}
```

(a) original DSL code

```
IFsmDSL dsl = new FsmDSL();
dsl.eval(name:"watch") {
  state(name:"reseted",type:"start") {
    entry "resetTimer";
    transitions {
      when (event:"start",enter:"running");
      when (event:"switchOff",enter:"off");
    }
  }

  state(name:"running") {
    entry "startTimer";
    perform "loopRunning";
    transitions {
      when(event:"split",enter:"paused");
      when(event:"stop",enter:"stopped");
    }
  }

  state(name:"paused") {
    entry "pauseTimer";
    transitions {
      when(event:"unsplit",enter:"running");
      when(event:"stop",enter:"stopped");
    }
  }

  state(name:"stopped") {
    entry "stopTimer";
    transitions {
      when(event:"reset",enter:"stopped");
      when(event:"switchOff",enter:"off");
    }
  }

  state(name:"off") {
    exit "switchOff";
    transitions {
      when(enter:end);
    }
  }
}
```

(b) DSL code in Groovy Syntax

Figure 6 - An Example FsmDSL Program modelling a Watch Clock

Figure 6 (a) shows an example program that defines a *state machine* to model a watch clock. The watch clock consists of five *states* (i.e., reseted, running, paused, stopped, and off). Each state may have a set of *action delegates* defined that have delegate type: a) *entry*, delegate type executes the action when entering the state, b) *do*, delegate type executes the action while remaining in the state, and c) *exit*, delegate type executes the action when leaving the state. Moreover, each state defines a set of legal *transitions* to other states. Each transition is defined that when an event (following the keyword *when*) occurs that the state machine should enter the next state (following the keyword *enter*).

Recall that an important requirement for implementing a DSL as an embedded DSL is that the DSL syntax must comply with the host language's syntax. Before one can implement a DSL as an embedded DSL in Groovy, the DSL syntax must be changed to comply with Groovy syntax. This restriction is an inherited drawback from the implementation approach of embedded DSL [Fow05]. The effect may vary from host language to host language and may even disqualify the embedded DSL approach in cases where it is not feasible to adapt the DSL syntax to the host languages syntax. The

embedded DSL approach assumes that the end user can posture with a more or less slightly changed DSL syntax that is close to the original DSL syntax, while savouring the advantages of the implementation efficiency of embedded DSLs.

As the DSL syntax in Figure 6 (a) is not compliant with Groovy syntax, the DSL code must be transformed to the code shown in Figure 6 (b). Note that the code snippets highlighted with a gray box show points at which the DSL syntax had to be heavily changed in order to comply with the Groovy host language syntax.

In general, one must construct legal host language code. Therefore, expressions in the original DSL syntax were transformed, so that they will be parsed as legal Groovy method calls, which is a necessary demand of our implementation architecture. In cases in which expressions consists of multiple tokens, these must start with a keyword that identifies the expression type followed by a list of named parameters, which are treated as parameters to that keyword.

In the example, the above transformation rule for deriving legal method calls had to be applied for the keywords: *state* and *when...enter*. Note that while we use brackets for the keywords *state* and *when*, the brackets in case of *entry*, *do*, and *exit* can be omitted. Further, the colon (`:`) has a special meaning in Groovy, therefore it cannot follow the keywords: *entry*, *do*, and *exit*.

In the state machine example, the DSL syntax violated the Groovy syntax in the following cases:

1. The top-level keyword *machine* of the original DSL is not available by default. To enter the syntactical environment to use DSL keywords a designated DSL interpreter instance must be created with **`new FsmDSL()`**. Next, one can evaluate DSL code passed to the **`eval`** method. (The instantiation of the interpreter poses additional syntax noise on the Groovy DSL syntax, which is presented this way for the sake of understanding. Later, we show how one can remove this syntax noise by using a so-called *bootstrap keyword*.)
2. The first state in the DSL program is defined as a *start state*. This violates Groovy syntax because the expression "*start state reseted"* cannot be resolved by the Groovy parser, which expects a well-formed method call with parameters at this place. Therefore, the position of the *start* keyword was moved to be an optional parameter (**`type`**) in the parameter list of the *state* keyword in the Groovy DSL.
3. In state "running", the original DSL keyword *do* conflicts with Groovy syntax, because the keyword **`do`** is already defined in Groovy. Therefore, the keyword has been renamed to **`perform`**.
4. In the last state "off", the keyword combination *when...end* has been transformed such that there is no event to take that transition and the state to enter is the *end state* that is referred to by using the keyword **`end`** passed as a keyword parameter to the *when* keyword.

1.5.3 From the Language Interface Layer to Implementation

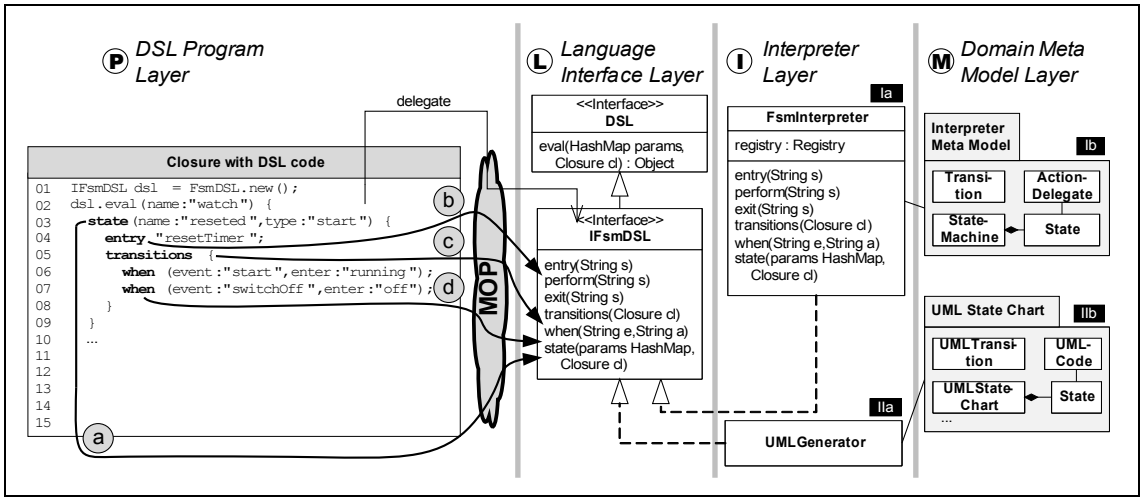


Figure 7 – The EDSL Architecture Instantiation for FsmDSL

In Figure 7, the instantiation of the EDSL architecture presented in Figure 5 is shown. To define states, a domain-specific composition element is introduced denoted by the keyword **`state`**. Each **`state`** in *StateMachineDSL* has a name and a closure (the code block in the curly brackets following the name declaration). The code inside a closure for defining a state (**`cl`**) may contain DSL abstractions, e.g., **`entry`** or **`state`**, as well as arbitrary Groovy code, e.g., Groovy's control structure **`each{}`** can be used in order to generate a number of states in the state machine using closures as templates.

At index (L) of Figure 7, the language interface of **`FsmDSL`** is declared in the **`IFsmDSL`** interface. The latter declares an operation for each DSL element: **`state`**, **`when`**. Mapping between the DSL elements used in the program at index (P) — a Groovy closure — and the operations in **`IFsmDSL`** is taken over by Groovy's MOP by assigning an object that implements **`IFsmDSL`** to the delegate field of the closure enclosing the DSL program.

Several implementations of **`IFsmDSL`** are possibly corresponding to different interpretations of **`FsmDSL`**. Two such interpretations are shown in Figure 7 ((Ia) and (IIa)). **`FsmInterpreter`** provides a custom interpretation of definitions of a state machine using the custom meta-model at index Ib, while **`UMLGenerator`** uses the meta-model of *abstract syntax tree* of UML (index (IIb)) to generate UML state charts from **`FsmDSL`** state machine definitions. The implementation of **`FsmInterpreter`** is elaborated in Figure 8. The method eval (cf. Figure 8, lines 6—11) (inherited from the marker interface **`DSL`**, which is extended by **`IFsmDSL`**) lays down what it means to evaluate a state machine definition. It creates an instance of the meta-class **`StateMachine`** for which subsequently defined states will be defined. Next, the executing DSL interpreter instance (`this`) is assigned to the delegate field of the **`machineDefinition`** closure (line 9). As a result, domain-specific elements with no meaning in the host language (e.g., state, or entry) encountered during the execution of the **`machineDefinition`** closure (line 10) are dispatched to the executing interpreter instance by the MOP (this mapping is schematically shown by the curved lines labelled a, b, c, and d in Figure 7).

To interpret DSL elements, following the recipe (in Figure 5), any DSL interpreter defines a) a property for each domain literal (e.g., **`end`** in lines 14—15 in Figure 8 in

FsmInterpreter), b) a method for each domain operation (e.g., **entry**, **perform**, **exit**, and **when** in lines 18—33 in **FsmInterpreter**), and c) a special method for each domain-specific abstraction/composition element (e.g., **state** in lines 36—43 and **transitions** in line 45 in **FsmInterpreter**).

```
01 class FsmInterpreter implementss IFsmDSL {
02   …
03   private StateMachine currentMachine;
04   private State currentState;
05   …
06   public eval(HashMap params, Closure machineDefinition) {
07     …
08     currentMachine = new StateMachine(params.name,…);
09     machineDefinition.delegate = this;
10     machineDefinition.call();
11   }
12
13   //domain literals
14   private State endState = new State("end",…);
15   public State getEnd() { return endState; }
16
17   //domain operation
18   public void entry(String name) {…}
19
20   public void exit(String name) {…}
21
22   public void perform(String name) {…}
23
24   public void when(HashMap params) {
25        State from = currentState;
26        State to = currentMachine.getState(params.enter);
27        if (to == null) {
28          to = new State(params.enter);
29          currentMachine.addState(to);
30        }
31        Transition t = new Transition(from,to,params.event);
32        from.addTransition(t);
33   }
34
35   //nested element
36   public void state(HashMap params,Closure stateDefinition) {
37     …
38     currentState = new State(params.name,…);
39     currentMachine.addState(currentState);
40     …
41     stateDefinition.delegate = this;
42     stateDefinition.call();
43   }
44
45   public void transitions(Closure transitionDefinitions) {...}
46 }
```

Figure 8 – The Interpreter for FsmDSL

1.6 Black-box Composition with POPART

For *black-box composition*, the syntax of multiple languages (EDSLs) is composed, while the semantics of each of the language stays unchanged. This allows using keywords from several DSLs in a program that is evaluated using a composed interpreter that reuses the modular EDSL implementations. We have classified black-box composition as a form of *non-invasive semantics composition*, whereby in contrast to *invasive semantics composition* (see Section 1.7), there is no interference between the semantics of composed languages.

Concerning the composition specification problem, black-box composition solves the technical integration between the multiple syntaxes, whereby the semantics of each composed language is considered as a black-box. For instance, it could be desirable to transfer a parameter value from one language's scope to another language's scope. Such integration can be achieved by using variables and types shared between two or more domains. If no such shared type exists, the specifications of the languages can be appended with a new shared type.

We have developed a new DSL framework for composing DSLs in the implementation space, called POPART. POPART supports this type of integration if each of the languages is developed as an embedded DSL. Interpreters are first-class entities that can be passed as values. This provides a great flexibility that can particularly be used for composing DSLs.

We can use different of these interpreters in order to evaluate code. Interpreter may reference other interpreters. Having interpreters as first class values facilitates very powerful, fine-grained and dynamic composition of DSLs using customizable composition. One can mix keywords within a program. Mixed DSL programs can be executed by calling the **`eval`** method on a interpreter instance that combines the DSLs to be composed. The code is passed in form of a first-class value, a closure. When a DSL keyword is encountered in the mixed DSL code, the keyword is delegated as a method call to their corresponding DSL that is composed.

Using factory objects, one can dynamically select interpreter implementations for a certain language interface. Calls to the interpreter's **`eval`** method can obviously be within conditional clauses or arbitrarily nested. POPART also provides support for declarative composition of interpreters with so called *interpreter combiners* - instances of the special interpreter class **`InterpreterCombiner`** or its subclasses. Figure 9 gives an overview of the composition infrastructure of POPART.

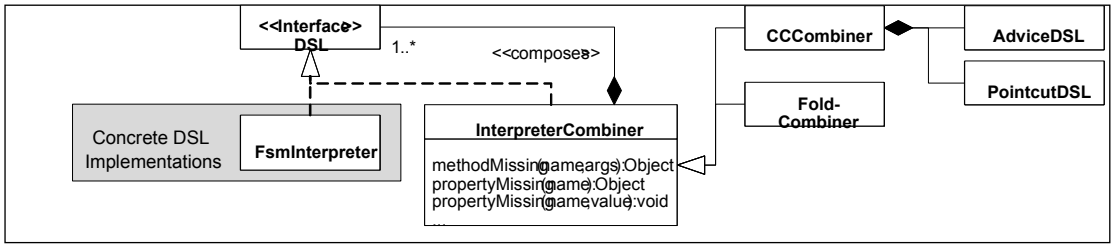


Figure 9 – Interpreter combiners for modular DSL Implementations.

Like any other POPART interpreter, an interpreter combiner implements the **`DSL`** interface, i.e., the method **`eval`** declared therein that takes a **`Closure`** as a parameter, containing DSL code that uses language elements from several DSLs. Unlike simple interpreters, interpreter combiners do not directly define any domain-specific semantics.

Instead, they hold references to other interpreters to which DSL elements are forwarded and implement two special methods of the Groovy MOP: **`methodMissing`** and **`propertyMissing`**. By convention, whenever a method is called on an object that is not defined in its class (a pretended method call), Groovy's MOP executes the method **`methodMissing`** of that class, passing the name of the method and the arguments of the pretended call as parameters. In a similar way, **`propertyMissing`** is invoked when accessing a (pretended) property. The methods **`methodMissing`** and **`propertyMissing`** implemented in **`InterpreterCombiner`** lay down DSL composition semantics. Programs that mix elements from several DSLs are defined in closures whose delegate object is an **`InterpreterCombiner`**. When a domain operation or a domain-specific nesting element is encountered, the **`InterpreterCombiner`** delegate receives a pretended method call and **`methodMissing`** is invoked by the MOP with the respective domain-specific keyword as the method name. In a similar way, the MOP dispatches literal keywords by invoking **`propertyMissing`** on the **`InterpreterCombiner`** delegate. Given their parameters and the inner interpreters referenced by their receiver, **`methodMissing`** and **`propertyMissing`** implement look-up semantics for the domain-specific abstractions. Several such semantics are conceivable and can be realized by a hierarchy of interpreter combiners in POPART. For instance, in case of the **`InterpreterCombiner`**, which holds a list of DSLs, **`methodMissing`** forwards a pretended method call to the first interpreter in the list that implements a keyword method with the same signature as the method of the pretended call. Another composition semantic is to forward to all interpreters that have an implementation of the pretended call. Special cases of the latter semantics are combiners that compose different interpreters of the same language. Other types of interpreter combiners are shown in Figure 9. The **`FoldCombiner`** implements more generic composition operators by taking a closure that entails the composition semantic as a parameter and a list of **`DSL`**s in its constructor. Specifically, when a keyword is received, it is forwarded to all interpreters in the list and the returned values are used as parameters to the closure that calculates the result

1.7 Using Aspect-Oriented Programming for invasive Semantic Composition

Consider the scenario when we would like to compose a DSL for describing workflows, called *ProcessDSL*, with another EDSL for enforcing secure communication with partners, called *SecurityDSL*.

The EDSL interface and interpreter of *ProcessDSL* provides keywords for describing workflows, such as **`task`** for defining the steps of a workflow, **`registry`** to retrieve a reference to the registry service that can be used to look up other services, and **`notify`** to send out an email to all stake holders of a process. We have implemented *ProcessDSL* as an EDSL in POPART in the class **`ProcessInterpreter`** that defines keyword methods for **`registry`**, **`notify`**, and **`task`**.

The EDSL interface and interpreter of *SecurityDSL* provides the necessary primitives for encrypting outgoing SOAP messages and decrypt incoming encrypted messages. *SecurityDSL* is implemented in a class **`SecurityDSL`** and provides two operations: **`encrypt(data,alg)`**, that encrypts the first argument using the encryption algorithm provided as the second argument, and **`decrypt(encData)`** for decrypting.

```
1 DSL process_dsl = new ProcessInterpreter();
2 DSL security_dsl = new SecurityDSL();
3 DSL combined_dsl = new InterpreterCombiner(process_dsl,security_dsl);
4
5 combined_dsl.eval(name:"EasyCreditProcess") {
6 def offers = [:];
7 task (name:"getOffers") {
8 def services = registry.find("Banking");
9 services.each { bank ->
10 def enc_request = encrypt(new RateRequest(),RSA);
12 def enc_response = bank.call("getRate",[enc_request]);
13 offers[bank] = decrypt(enc_response); ;
14 }
15 }
16
17 task (name:"selectOffer") {
18 def selectedBank = ... //get cheapest bank from offers
19 def enc_request = encrypt(new BorrowRequest(),RSA);
20 def enc_response = selectedBank.call("borrow",enc_request);
21 def response = decrypt(enc_response);
22 notify "Credit from $selectedBank.name"
23 }
24}
```

Figure 10 – A secured Workflow in a black-box composed DSL

Given *ProcessDSL* and *SecurityDSL*, we would like to write process definitions which encrypt/decrypt messages sent to external services. As illustrated in Figure 10, we could use black-box composition to compose *ProcessDSL* and *SecurityDSL* (lines 1-2) and then use the resulting combined interpreter (in line 3) to evaluate programs that use language elements from both domains (while *ProcessDSL* keywords are marked in blue, *SecurityDSL* keywords are red). The program, however, suffers from tangling and scattering - the code for security concerns is not localized (cf. lines 10-12 and 19-21). We would rather prefer to write the security code once using *SecurityDSL* elements and have a fine-grained composition mechanism that integrates the execution of security code into well-defined points during the interpretation of *ProcessDSL*. We call this style of composing DSLs crosscutting composition.

1.7.1 Combining Aspect-oriented Programming and EDSLs

To address the problem of code scattering and tangling in composed DSLs we would like to use aspect-oriented programming for semantic invasive composition embedded DSLs. However, all aspect-oriented programming languages that could be integrated with languages commonly used in MDSD did not provide the language features powerful enough to embed DSLs. There are two possibilities, either to extend an AO language with more powerful feature necessary for embedding DSLs, or to extend a language that provides powerful features with support for AOP. As providing features for embedding would demand the implementation of an AO language and the accompanying tools from scratch, we have chosen the latter option. Whereby, we follow the same approach for implementing an AO language that we followed for implementing DSLs. Based on the same features for embedding DSL syntax, we use these feature to embed aspect-oriented language constructs. The resulting AO language supports what we call crosscutting composition of DSLs; as opposed to black-box composition, the interpretation of EDSLs to be composed is changed by the composition. We have implemented the architecture as a framework in Groovy [Groovy], which we call POPART. Groovy has been chosen since it supports the features mentioned above and for its flexible syntax. Yet, any other language that supports the same set of features can be used as an implementation language.

1.7.2 Crosscutting Composition

To realize crosscutting composition of DSLs, we make use of aspect-oriented concepts. An overview of the approach is schematically given in Figure 11. In this approach, aspect

modules specify crosscutting composition semantics for programs written in different EDSLs (EDSL1-Program and EDSL2-Program), each with its own LIM runtime. To express composition semantics, an aspect makes use of pointcuts and advice, common language elements of aspect-oriented languages such as AspectJ [AspectJ] and composition operators such as **`before`**, **`after`**, **`around`**, and **`proceed`**. Roughly speaking, pointcuts define queries for selecting points in the interpretation of the programs of EDSL1, advice is a closure enclosing code in EDSL2, and composition operators determine the order of execution.

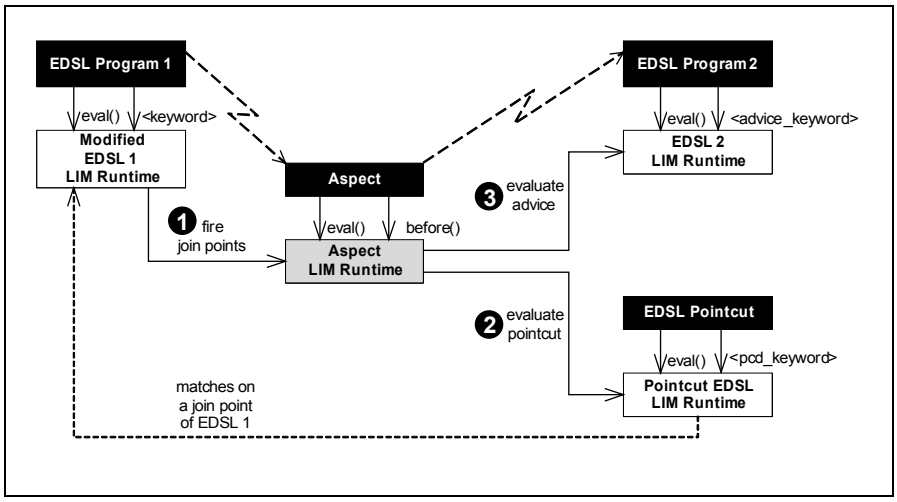


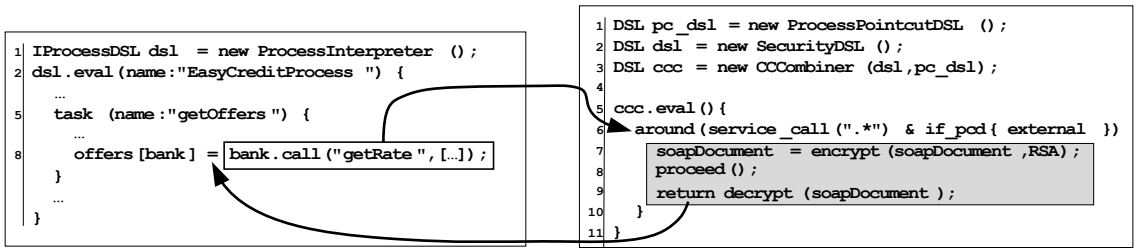
Figure 11 – Crosscutting composition of DSLs

The aspect language used for EDSL composition is itself implemented as an EDSL (cf. aspect LIM runtime in Figure 11). The aspect language interpreter, called **`CCCombiner`**, is a subclass of **`InterpreterCombiner`** (cf. Figure 9). It has a reference to the interpreter for one of the DSLs to be composed (EDSL2 in Figure 11) and to the interpreter of the pointcut language for querying points in the interpretation of the other DSL (EDSL1 in Figure 9). **`CCCombiner`** uses a meta-model for aspects and provides keywords before, around, after, and proceed. In a nutshell, the aspect meta-model consists of classes that model the core elements of AOP as first-class entities, such as **`Aspect`**, **`JoinPoint`**, and **`Pointcut`**. An aspect in the PlasmaJ meta-model maintains a list associating Pointcut objects to closures. **`JoinPoint`**s represent points in the interpretation of an EDSL exposed for composition by maintaining information about their type and the context exposed when these points are reached. The set of points in the interpretation of an EDSL that are identified as join points for composition is determined by a domain-specific join point model (DS-JPM). For illustration consider defining a DS-JPM for *ProcessDSL*, consisting of two kinds of join points. First, there are *service selection join points*: Points at which the registry is consulted to select services of a certain category; exposed properties include the pattern used to select services and the resulting set of selected services. Second, there are *service call join points*: Points at which a remote call to a Web service is done as part of interpreting a process level call; exposed properties include the name of the service, the SOAP document representing the call, whether the invocation is remote or local. A DS-JPM for an EDSL is defined by extending the core aspect meta-model with DS join point types and by creating a modified version of EDSL's LIM run-time that creates and fires join point objects during the execution of programs. Technically, the modified version of the original EDSL LIM runtime is automatically derived in POPART by using AspectJ [AspectJ] aspects.

Pointcuts are modelled in the meta-model as filter objects that select join point objects

based on their type and exposed context values. The filtering AOP and EDSLs queries are written in a *domain-specific pointcut language* (DS-PCL) which is also an EDSL with its own LIM run-time. The interface of a DS-PCL declares keywords for the so-called pointcut designators. For illustration, consider embedding a DS-PCL for *ProcessDSL* and the DS-JPM including service selection and service call join points. Such an embedding would include an EDSL interface, say *IProcessPointcutDSL*, declaring keyword methods **`service_call`** and **`service_selection`**. The latter would be implemented in a class, say **`ProcessPointcutDSL`** that uses the meta-model classes for modelling pointcuts. The pointcut expression "**`service_call("get.*") & if_pcd { external }`**" selects all service calls, where the operation name (one of the values in the context of service call join points) matches the regular expression "get.*" and where the call is remote, which is reflected by testing the Boolean variable **`external`** (also part of the context of a service call join point). The result of evaluating this pointcut expression would be a **`Pointcut`** object (class in aspect meta-model) that is composed of other **`Pointcut`** objects the sub-expressions in the pointcut.

Crosscutting composition takes place at run-time. When join points of the EDSL1 are fired at run-time to the aspect run-time (Figure 11 index 1), the latter calls match on the pointcut objects resulting from evaluating pointcut expressions defined in aspect modules and stored in Aspect instances, passing the fired join point that as a parameter. If a pointcut matches, the closure associated to it is evaluated by the corresponding run-time.



(a) *ProcessDSL* code (b) An example aspect

Figure 12 – Crosscutting composition example

For illustration, Figure 12 (b), lines 5-11 shows an aspect that composes code to secure outgoing and incoming soap messages into any external service call point. The aspect specifies that the code in the gray box should wrap service calls by means of the composition operator around/proceed (proceed can be thought of a place holder for the wrapped call). This aspect is evaluated by an instance of **`CCCombiner`** initialized with a **`ProcessPointcutDSL`** interpreter and a *SecurityDSL* interpreter (lines 1-3). The interpretation of keywords used in the pointcut, **`service_call`**, **`external`**, and **`if_pcd`** is dispatched by **`ccc`** to the instance of the **`ProcessPointcutDSL`** interpreter. When the process definition in Figure 12 (a) is evaluated and the interpretation of the bank service call at line 8 reaches the point where the corresponding web-service proxy is called, the control is passed to the aspect (Figure 12 (b), indicated by the arrow to the right), because the call matches the pointcut of the aspect. When executing the security code in the aspect, security operations are interpreted by the **`SecurityDSL`** interpreter. The context of the intercepted join point is also visible to the security code (e.g., **`SOAPDocument`**).

1.8 EDSLs in MDSD

To integrate EDSLs with the model-driven approach, we have implemented a prototype integration of the EDSL implementation and the metamodeling framework EMF. As an alternative to develop a generator for EMF metamodel, one can use the generated Java classes of an EMF metamodel. To realize an EDSL. In a nutshell, the Ecore metamodel and the corresponding EMF infrastructure is used to generate Java classes from the metamodel.

Next, the developer integrates the metamodel with the interpreter class in Groovy. The interpreter class is implemented using the same approach presented above, but it offers another method **`load`** which can be used to load a Reflective Ecore Model, which creates a metamodel instance that can be interpreted.

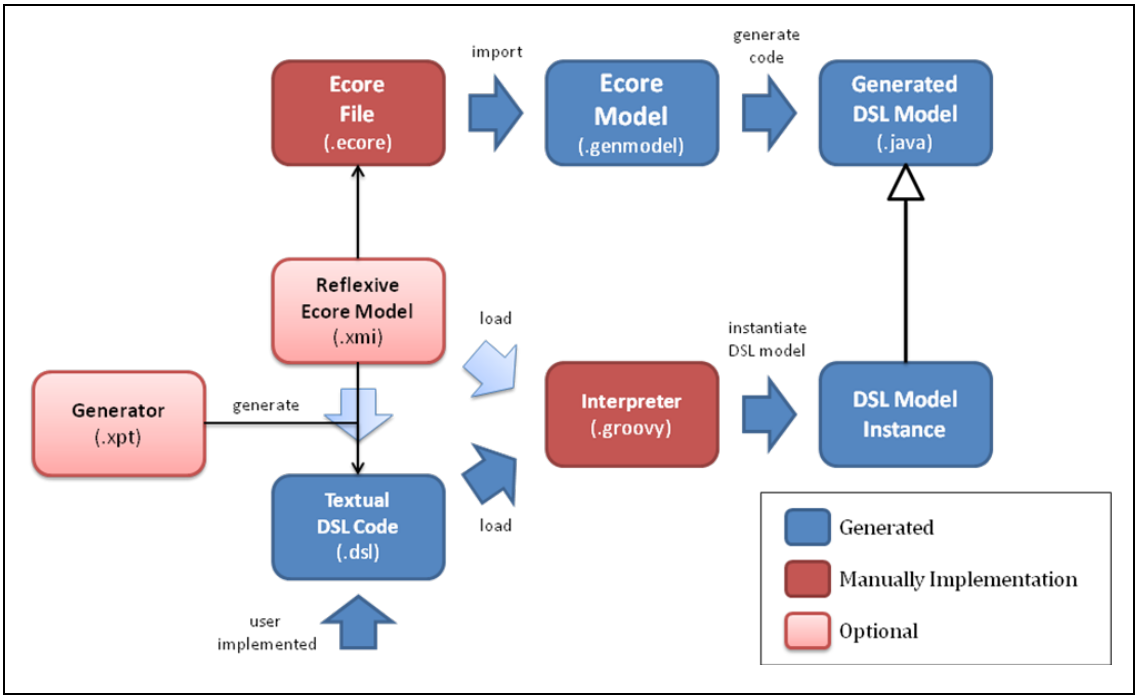


Figure 13 – The Interpreter for FsmDSL

4. Integration of MDSD-typical DSLs through Role-Based Language Composition

In the previous section we discussed techniques for the implementation and integration of DSLs using an EDSL approach. We have seen that it allows for a rapid implementation of new DSLs but also comes with some drawbacks. First, EDSLs are built on top of a general-purpose programming language which needs to provide special language features to enable proper embedding. Second, the syntax of the host language imposes restrictions on the DSL syntax. In this section we will discuss the integration of languages at the level of modelling languages where no general-purpose abstractions are available and concrete syntaxes are not limited to textual representations. In a model-driven software development process (MDSD) [Bet04][OMG03a] these languages are used during system design, but not for the final implementation of software. They can also be understood as DSLs, since the typically cover a specific domain of system design.

Due to the domain-specificity of modelling languages, they are often used in combination to describe several design aspects of a software system. These need to be integrated to realize a coherent executable system [BrLo07]. Therefore artefacts from different domains need to be matched and composed. In existing approaches this integration is typically realized during code generation [Fujaba][Jenerator]. Artefacts are matched by names and generated glue-code bridges the gap between abstractions of different domains. This has several disadvantageous implications:

- The integration happens during code-generation time. Hence, inconsistencies in related models cannot be detected during system design.
- The relationships between different languages are defined only implicitly in the code generators. We have no systematic understanding of the overall language infrastructure.
- Individual languages are hard to reuse and re-combine without a systematic consideration of language decomposition and composition.
- The integrated languages stay encapsulated syntactically and semantically.

These issues raise the need for a systematic approach to safe composition of modelling languages. We need to provide means to describe relationships between modelling languages at the level of language specifications. That is, we describe language composition for specifications of languages not specifications (or expressions) in languages.

Since modelling languages address specific aspects of a system, the concepts of their specifications intertwine in a crosscutting way. Therefore, we introduce a role-based approach for language composition. It employs the technique of role-modelling [Ree96] [And97] to address the aspectual character of language specifications and allow for their invasive superimposition. In the following we will discuss the foundations of this approach and its prototypical implementation in the language composition framework *LanGems Modeller*.

1.9 Foundation of Role-Based Language Composition

Considering the interaction of several modelling languages during the design of a software system, their combination can be again understood a language itself. In this

language the individual languages are used to realize a specific functional feature. W.r.t. the composed language, we call a modular sub-language realising a feature in the composed language *LanGem*. This notion is derived from the terms *morpheme* and *lexeme*: During the lexical analysis of a program, a lexeme describes the smallest unit in a parser's input stream. Morphemes denote the smallest entities with a defined semantics. Stretching this argumentation to the level of language specifications the term LanGem refers to a self-contained component that realize a particular language feature. To compose a language we choose from a collection of several language features and describe the connection of the LanGems realising them.

This section introduces the foundation of our role-based language composition system – LanGems Modeller.

1.9.1 Constituents of our Language Composition System

In this section we will present the conception and realisation of our role-based language composition system which contributes means to define aspectual language specifications and compose them to integrated languages.

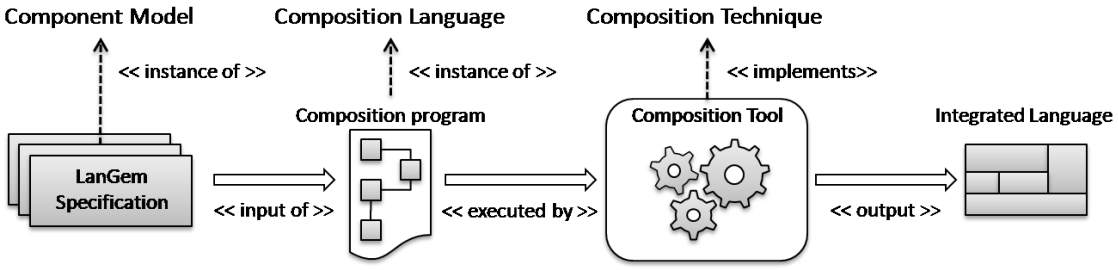


Figure 14 - Overview of the LanGems Composition System

Generally, a composition system is a triple consisting of a *component model*, a *composition language*, and a *composition technique* [Ass03]. The component model describes how components – in our case LanGems – look like and how they can be accessed. The composition language introduces the vocabulary used to describe concrete composition programs that specify the combination of several components to a system – in our case an integrated language. And finally, the composition technique defines the technological background that actually realizes the composition.

Before we introduce the key concepts of our language composition system *LanGems* according to these constituents, we will have a look at its coarse structure (cf. Figure 14). The LanGems approach contributes a dedicated composition system for languages. The specification of the individual LanGems is based on the *LanGems Module Specification Language* that institutes the concepts of the LanGems component model. How several LanGems are combined is specified in a composition program formulated in the *LanGems Composition Language*. This program is evaluated by a composition tool that implements our language composition technique and generates an integrated language from several LanGems.

Component Model

Every LanGem is built upon a concrete abstraction needed to describe the realisation of the LanGems language feature. This concrete abstraction constitutes the LanGems component model. The ongoing research in the area of *Model-Driven Software Development* (MDSD) [Bet04][OMG03a] introduced *metamodels* as an adequate methodology to describe a language's abstraction. In comparison to *abstract syntax trees* (ASTs) traditionally used for compiler construction the graph-like structure of *abstract*

syntax models (ASMs) allows representing references between language artefacts that are not related in terms of the natural containment hierarchy. The relevance of such references is indicated by examples found in nearly every language: consider the relationship of a procedure call and the procedure declaration in a procedural language, the references between states and transitions all contained in a state machine, or the definition of a classifier's property and navigation on this property in an OCL expression.

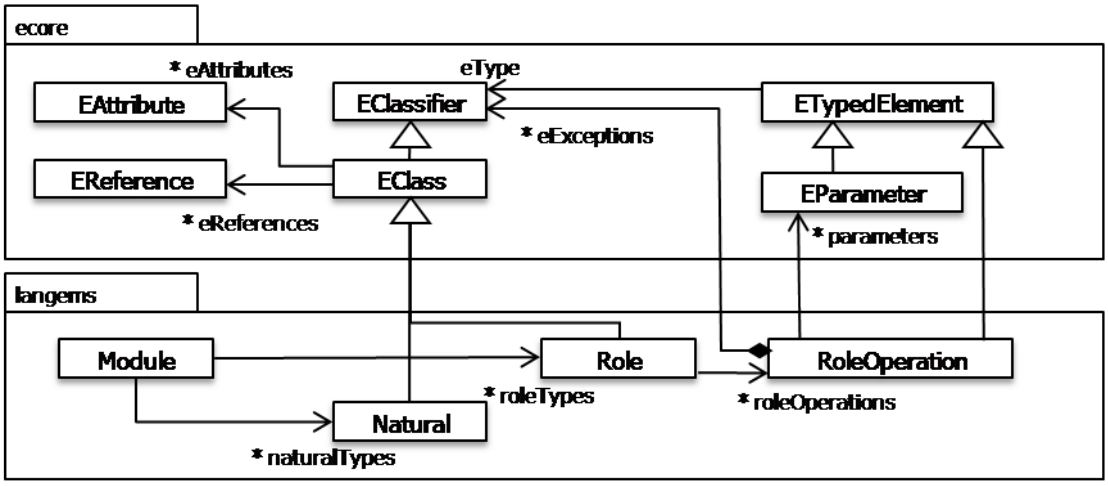


Figure 15 - Metamodel of the LanGems Module Specification Language

We want to keep the LanGems decoupled. But their combined use for the specification of software systems necessitates a tight integration of their conceptualisations. Every LanGem contributes a partial aspect of the system design and only their superimposition makes a coherent executable software.

Therefore, we extended the *Ecore* meta-modelling language (cf. *Eclipse Modeling Framework* (EMF) [BBM03]) with additional concepts (cf. Figure 15) which were inspired by the paradigm of role modelling [Ree96][And97]: The abstract conceptualisation of a LanGem is described by means of a module (**`Module`**) that contains a *collaboration* between *natural types* (**`Natural`**) and *role types* [Ste00] (**`Role`**). The types are distinguished by the fact that the identity and properties of naturals are found within the LanGem's application domain while roles describe generic variation points in the LanGem's collaboration. **`Natural`**s and **`Role`**s are special kinds of a metaclass (**`EClass`**). That means, their properties are described by attributes (**`EAttribute`**) and their interrelations by associations (**`EReference`**).

A number of considerations caused us to construct our component model upon the EMF Framework: The EMF type system provides advanced means to construct the abstraction a LanGem. EMF comes with a code generator to realize a Java implementation for EMF models. This code generation is extensible which allowed us to integrate our EMF extensions. The data and collection types shipped with the EMF standard library build the foundation for a uniform data exchange between several LanGems which is a fundamental premise to build compatible language modules.

It is important to understand that role types establish a type interface that is used in the LanGem's collaboration for the specification of its concrete syntax and semantics. The variability of a role type lies in the way this interface is implemented, because role types obtain their identity, and some structural and semantic properties from objects of other

types – their *role players* (cf. the next section on composition technique for details on role players). The role's semantic requirements are specified in terms of *role operations* (**`RoleOperation`**) that introduce an explicit composition interface between role types and role players. So role operations hide both the structural and the semantic adaptation the role player. In the other direction the role types hide the inner workings of a LanGem from the outside but provide an explicit composition interface to ensure semantic and structural safe compositions.

Composition Language

The composition of several LanGems and their adaptation for interoperation is externalized to a dedicated composition program. Thus, a maximal independence of solitary LanGems is achieved which allows their flexible combination and adaptation to allow for new combination of individual modelling languages.

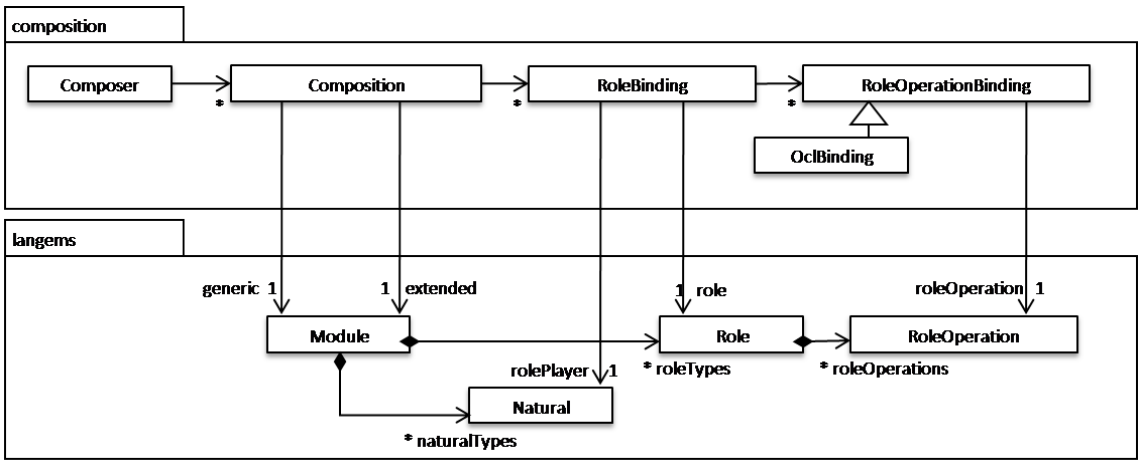


Figure 16 - Metamodel of the LanGems Composition Language

Figure 16 depicts the metamodel of the composition language used in LanGems. It is connected with the metamodel for the LanGems Module Specification Language to describe the combination of several LanGems using the concepts defined in their specification. Every composition program defines a **`Composer`** which consists of a number of **`Composition`**s each describing the integration of a **`generic`** LanGem with variation points (its role types) and an **`extended`** LanGem that binds these variation points. These **`Composition`**s comprise several **`RoleBinding`**s which impose a played-by relation between a natural type of the extended LanGem and a role type of the generic LanGem. The adaptation of the role player to the semantic and structural requirements of its role is described by means of **`RoleOperationBinding`**s for every **`RoleOperation`** defined in the **`Role`**. These bindings can be specified using OCL expressions which declare the implementation of the role operation in the context of the role player.

Composition Technique

Since language composition describes the connection of several LanGems, a role binding is typically established between a role type of one LanGem and a natural type of another LanGem. This binding between a role and the role player constitutes our central operator for language composition. It results in the superimposition of the LanGem's collaborations and specifies a combined language.

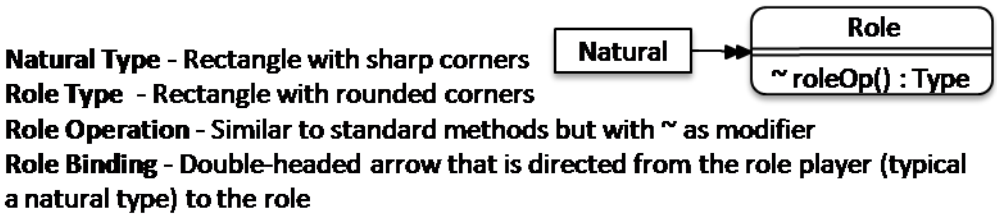


Figure 17 – Graphical notation for LanGems Specification extending the syntax of UML class diagrams

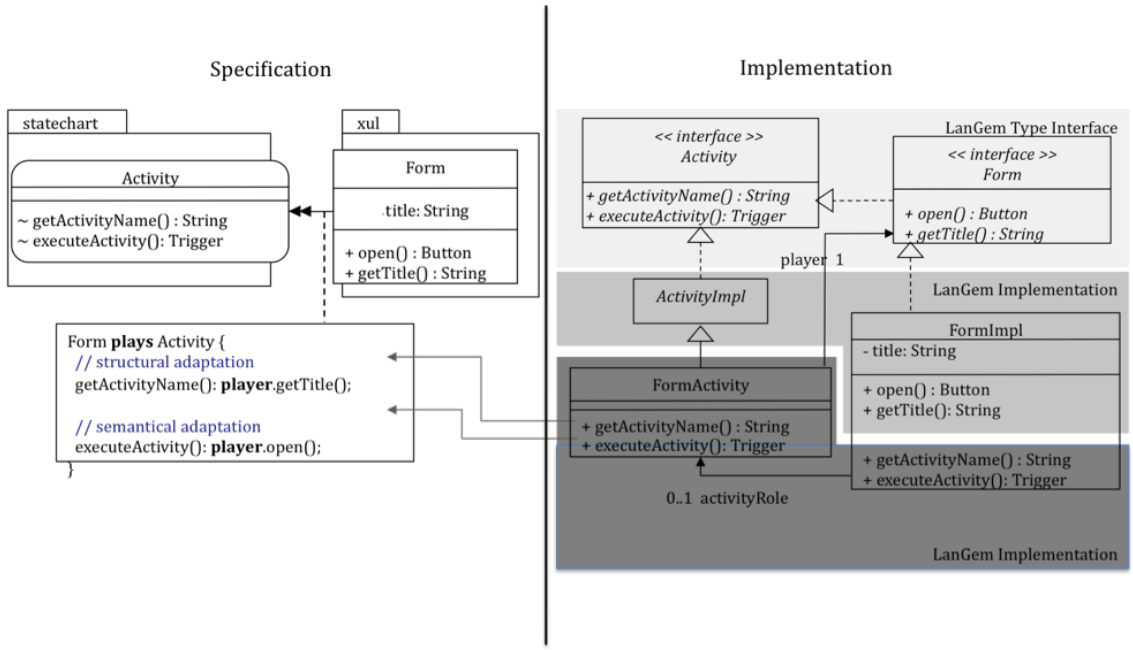


Figure 18 - Generative role-binding pattern

As stated above, we use Java and EMF as implementation technologies for the LanGems System. Java has no concept to represent roles and role bindings in the language. Thus, we used the generator pattern shown in Figure 18 to implement language composition. The left part of the Figure uses the notation introduced in Figure 17 to describe a role-binding between the `Form` natural from a user interface LanGem and the `Activity` role from a statechart DSL. The used pattern (shown right) implements the superimposition of the role models for the involved LanGems in accordance to the composition program and consists of three layers:

LanGem Type Interface Since the type interface specified in the LanGems component model is used during the specification of LanGem's syntax and semantics, it needs to be preserved during the composition. Therefore we generate the interfaces in correspondence to the role types and the natural types of the component model. In the generation step, role operations are simply mapped to normal operations. The role binding is mapped to an implements-relationship between the interface of the role player and the role interface.

LanGem Implementation The implementation of the type interfaces is encapsulated in the classes in a second layer of the implementation pattern. This layer implements the functionality relevant within the collaboration of the according LanGem, for instance, the persistence of an ASM (abstract syntax model) instance, the EMF-API for programmatic ASM manipulation, or LanGem semantics that are implemented operationally in Java. As

stated above, role types have no own identity, but obtain their identity from the role player. Hence, classes of role types are abstract and cannot be instantiated directly.

Composition Implementation The third layer encapsulates code used to actually implement the role bindings described in the composition program. Role binding affects the implementation of the role playing classes, which also needs to implement the role specific part of the role interface. This is done by delegating all calls to role-specific operations on the role player to a generated role adapter. This adapter extends the abstract implementation of the role type and therefore derives all properties of the role implementation. In addition, it implements the missing role operations according to the OCL expressions given in the composition program. The role playing object can be accessed from the adapter via the association `player`. Thus this layer also encapsulates the adaptations between the role players and their roles.

1.10 Constituents of a LanGem Specification

For our language composition approach we define a general structure of the constituents for the specification of a single LanGem. Clark et al. [CSW08] define a language as a combination of abstract syntax, concrete syntax, and semantics. Since we aim at composing LanGems of individual modelling languages, their specifications imitate this structure (cf. Figure 19). In the following we will describe the fundamental characteristics of these dimensions and their interrelation.

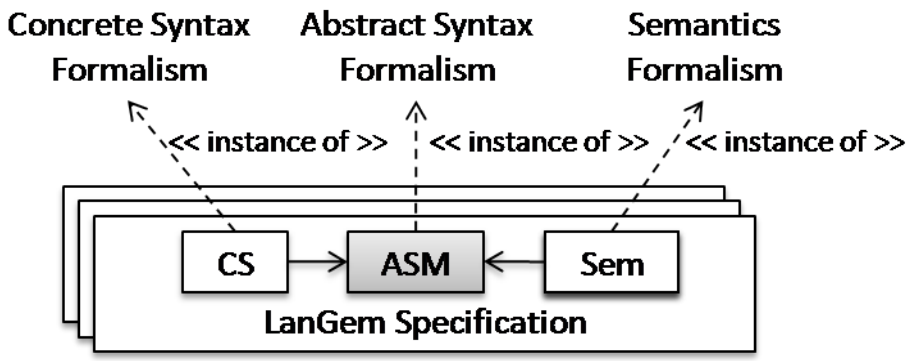


Figure 19 - Constituents of a LanGem Specification

Abstract Syntax (AS) The concepts of the LanGems's domain abstraction, their properties and relationships establish the foundation of every LanGem realisation. As discussed in Section 1.9.1, the abstract syntax of a LanGem constitutes the component model of our language composition system and the superimposition of the individual component models is the technique used for language composition. Hence, the abstract syntax specification is the central artefact of a LanGem specification. As depicted in Figure 19, every individual LanGem contributes its own abstract syntax and – in relation to this – its semantics and its concrete syntax.

Concrete Syntax (CS) The concrete syntax describes how language expressions are presented to the user. The possible representations are manifold: Traditionally, we think of a textual syntax, defined using a concrete syntax formalism like Extended Backus-Naur Form (EBNF) [ALSU06]. In the area of MDSD languages, diagrammatic syntaxes gained importance and even the tree-based model editors found in current modelling tools (e.g. EMF [BBM03]) provide a concrete syntax for language expression.

All concrete syntax formalisms have in common, that they are explicitly or implicitly related to the abstract syntax of a language. Parsers transform a textual syntax into an

ASM instantiating the language metamodel, diagram editors use specific graphical primitives to distinguish model entities regarding their abstract types, and tree-editors combine a graphical representation of the containment associations between model entities with a textual and form-based representation of entity attributes and references.

We integrated EMFText [ETE09] to realize composable concrete syntax specifications. EMFText provides an EBNF-like syntax to define parsing rules for each natural type of a LanGem's abstract syntax separately. Thus, the granularity of the concrete syntax specification matches the granularity of a LanGem. The parsing rules directly reference properties and associations of the types to relate features of the abstract syntax and their textual representation. The type information for properties and associations contained in the abstract syntax is used by EMFText to choose appropriate parsing rules for non-terminals in a rule's body: For attributes, regular expressions are used to parse their values w.r.t. the attributes primitive type, containment associations [BBM03] are parsed using the rules for the associations type or sub-types, and non-containment references are resolved in a second parser pass. Since the composition technique used in LanGems preserves the type interface of the composed LanGems, the concrete syntaxes defined for individual LanGems can be combined and used for the composed language. For the composition of the concrete syntaxes of several LanGems their rule sets are combined. The transition between the rule sets is directed by the role-playing relationships that are specified between the LanGems during language composition. That means the concrete syntax of role types results from the concrete syntax of the natural types which play the role. Hence, the concrete syntax specification of a LanGem inherits its variability from the LanGem's abstract syntax. The composed EMFText specification is used to generate an ANTLR-Parser [Paa07] that directly builds instances of the integrated ASM. Currently we only support textual concrete syntaxes. However, we argue, that the technique applied for composing textual syntaxes can be easily expanded to other syntax representations.

Semantics (Sem) A language semantics describes computations over language constructs. We divide *static semantics* and *execution semantics*. Typically, modelling languages have their execution semantics defined in the transformation to an implementation language. We will have a detailed discussion on the composition of translational semantics in Section 1.3.

Static semantics are applied for type checks in the ASM, to test the well-formedness of language expressions or in our special case to adapt domain abstractions from different LanGems. This needs to be done in a way the machine can interpret. Literature [Win93] [CSW08] distinguishes several formalisms often for this purpose (e.g., operational, denotational, translational, or extensional approaches).

Currently, the LanGems Modeller allows for two ways to work define language semantics. First language expressions and role operation bindings can be specified by an operational semantics defined in Java. Second, OCL expressions can be used.

1.11 Classification of Role-Based Language Composition

Figure 20 classifies our compositional approach for language integration in accordance to the facets introduced in Section 1.1. Language composition invasively merges the concrete syntaxes of the involved modelling languages in accordance to a given composition program. Non-invasive syntax integration is not feasible; because there is no central host language whose language constructs could be reused for language embedding.

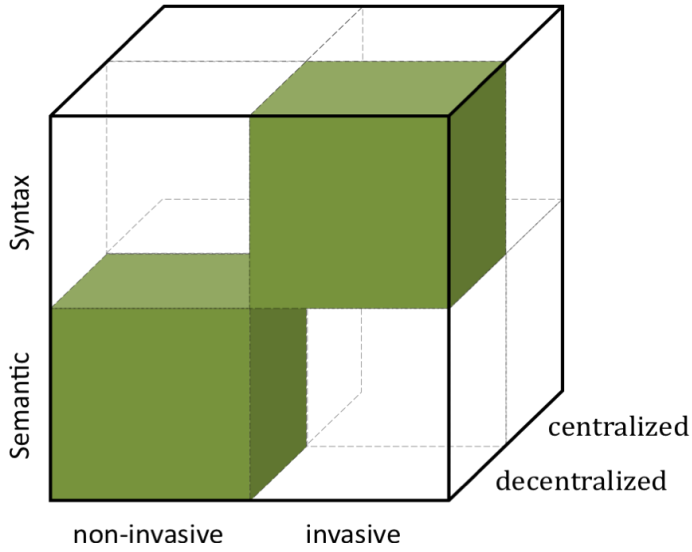


Figure 20 - Classification of Role-Based Language Composition

The semantic integration of the involved languages works non-invasively. It is expressed using predefined composition interfaces between language modules. That means, semantics of different languages interact only in anticipated ways. However, existing role modelling approaches used in software engineering provide means for aspectual and, thus, invasive semantic integration which will be adopted for our language composition approach in our future work.

The integration of multiple languages works decentralized. For language composition LanGems are connected directly by role-playing relationships. A central, universal integration mechanism is not provided.

1.12 Composition of language semantics using ontological foundations

Language composition case by case is pragmatic and often the way of choice to get quickly to desired results. However, to foster reuse of existing composition specifications not only for concrete syntax integration and to ensure a maximum of reutilisation also on final code level in model-driven development, a centralized approach to language mediation might be an adequate alternative. This section describes the HybridMDSD approach that provides the Unified Software Modelling Ontology (USMO) [BrLo08] – a universal ontological conceptualisation for modelling languages – acting as central semantic broker.

In MDSD one distinguishes between conceptual modelling and platform modelling. The Object Management Group (OMG) [OMG03a] furthermore distinguishes platform-

independent and platform-dependent models to abstract from concrete technical realisations. Platform or technical languages and instantiating system models form the basis for successive generation or interpretation steps that lead to executable programs. Therefore, platform languages and their models employ a limited set of semantic concepts and roles that can be described in a conceptualization for software systems. Here, languages and models reference themselves and comprise dependencies between each other, as illustrated in Figure 21.

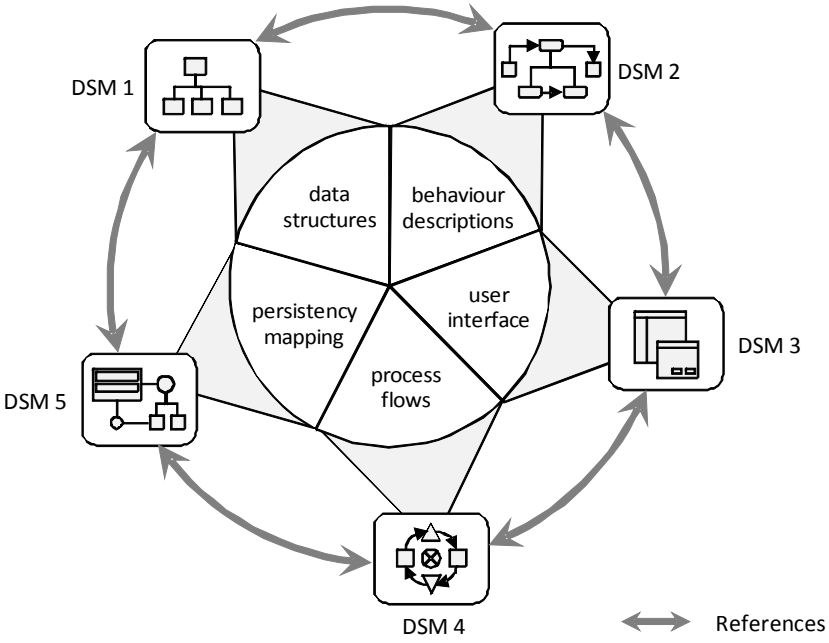


Figure 21 – Inter-Model dependencies in MDSD

In a centralized composition approach, the meaning of dedicated constructs of a modelling language may be mapped to concrete concepts and roles of a universal conceptualization. This way, each language and its instances obtains a semantics that was defined only once before and acts as a central interface for language composition.

Within the HybridMDSD project [COPL08], we followed this approach and defined the Unified Software Modelling Ontology (USMO) [BrLo08], which serves as central conceptualization to capture the semantics of modelling language constructs. To this end, HybridMDSD is to be classified as centralized approach for language composition, which is dedicated to language semantic only. An appropriate classification according to our scheme introduced in the section above, is illustrated in Figure 22.

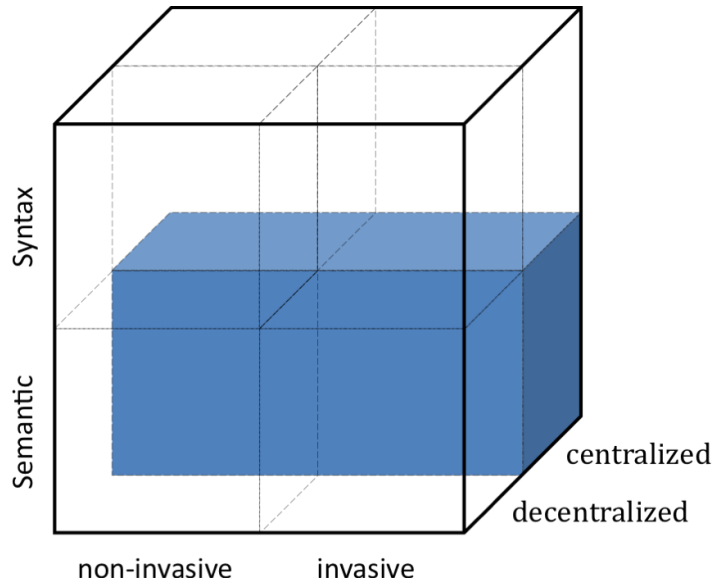


Figure 22 - Classification of the HybridMDSD approach

Our ontology contains concepts to describe structural and behavioural aspects of modelling languages semantics. This is comparable to above stated static and execution semantics. Besides the advantageous reuse possibilities through a central conceptualization, the semantics contained in our ontology can be used to derive certain composition patterns not only on modelling level but also on the level of actual program code. For instance, the knowledge about the sequential interaction of certain behavioural entities in different languages can be used to derive code patterns that are potentially useful for the integration of artefacts that are generated from each language and according models.

To give a summarizing example for language mediation/-composition based on a central semantic interface, we consider the composition of a structural language, a behavioural language that facilitates the modelling of dynamic semantics and a user interface language that represents various persistent structural entities. Figure 23 gives an informal overview about the example. The shown DSLs share concepts and relations with the ingredients of the example that discussed in more detail within the next section.

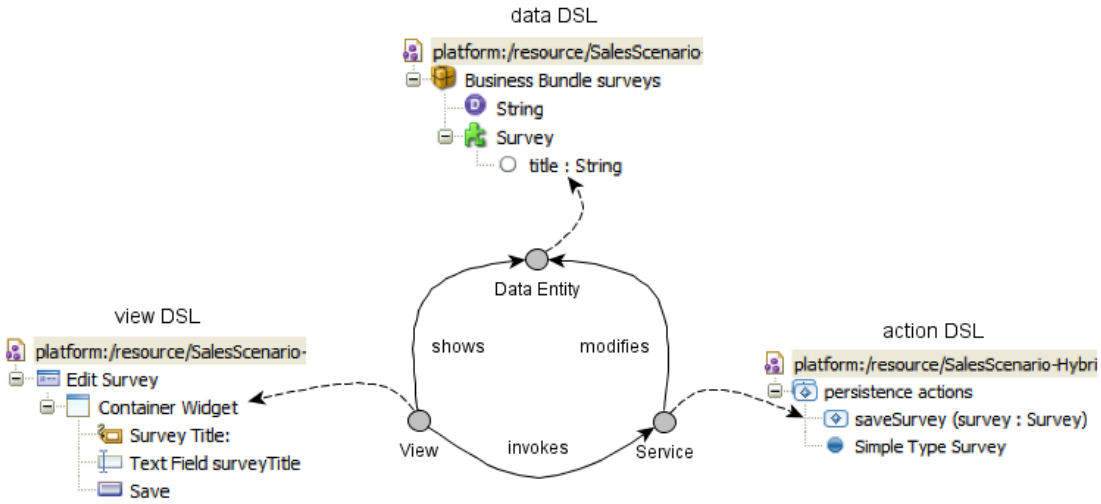


Figure 23 – Informal overview of the centralized composition of 3 DSLs

The illustration shows sample models of languages for 1) user interface dialogs (`view DSL`), 2) data entities (`data DSL`) and 3) persistence services (`action DSL`). They are interconnected using our central ontology by interpreting specific modelling constructs in terms of the semantic specification. As the figure shows, the example comprises a model in each of the 3 languages. The view model contains a graphical widget with a title, a text field and a `save` button. The data model contains a `Survey` business object with a `title` attribute. The action model contains a `saveSurvey` action which takes an argument of type `Survey` and persists that. The arrows in Figure 23 express the semantic connection between the different models: The widget from view model shows the `Survey` business object and invokes the `saveSurvey` action. The `saveSurvey` action furthermore modifies the `Survey` business object.

1.13 Composition of an Exemplary Language to Describe Graphical Wizard Dialogues

In this section we discuss the application of the LanGems Modeller for the integration of three individual DSLs to define graphical wizard dialogues.

1.13.1 Requirements for the Wizard Dialogue Language

For the implementation of a software system often behavioural, structural, and semantic properties need to be specified. A typical example is the description of the page flow in a graphical wizard dialogue. The specifications of the different dimensions typically involve very different conceptualisations. In the following we describe the design of an exemplary language to describe graphical wizard dialogues. This language is composed from three standard modelling languages.

Dialogue Execution Behaviour

A language for graphical wizards needs to provide means to specify the general dialogue execution. Wizards consist of several pages collecting a users input to achieve a standard, repetitive task. To structure this task in several logical sub-steps, wizards use pages that are passed in certain sequence. This sequence is not defined statically, but depends on decisions of the user of the wizard.

To describe this overall progress, we use state charts. State charts are a common technique [SCXML][OMG03b] to represent finite automata and several implementations [ASCXML][Sam08] can be found Pages of the wizards are represented by *States* that are connected through *Transitions* describing the possibility to change from one *State* to another. Every *State* can have several outgoing *Transitions* pointing to the *States* that can be reached next.

Besides these core concepts of state charts, we identify variable concepts that have a clear semantics within the state chart but are additionally used to leave the domain of finite automata. They describe variable points in the LanGems' conception where it can possibly be connected with other LanGems during language composition: During the execution of the wizard one of the outgoing *Transitions* is selected based on *Triggers* send from the runtime context of system the chart is applied in. *Activities* are executed as long as the chart is in a special State. Their behaviour depends on the concrete area the state chart is applied in. *Guards* describe additional conditions for passing a Transition. The concrete technique used to describe these conditions may also vary with the application.

Semantic Execution Constraints

In addition we need to define semantic constraints on the runtime state of the system to influence the dialogue behaviour. To define these constraints a declarative constraint language (e.g. OCL [OMG03c]) could be applied. It allows to efficiently describing conditions that need to be satisfied in the runtime context of the system to select a specific path in the wizard's page flow.

User Interface Structure

The wizard language also need to provide domain-specific means to describe the user interface (UI) displayed in the wizard pages. We know a manifold of declarative specification languages (XUL [XUL] and XAML [Mac08] which have been found beneficial [BV04] for the specification of graphical UIs. Their domain-specificity reduces the semantic gap between interface design and realisation, they abstract from a concrete implementation platform, and their structure respects the figuration of the actual interface. Concepts typically found in UI-languages are: *Form*, *Label*, *Text*, *Selection*, and *Button*.

1.13.2 Realisation of the Wizard Dialogue Language

For the realisation of the Wizard Dialogue Language we used the LanGems approach described in Section 1.8. First, we identified the LanGems according to the requirements defined above. They are depicted in Figure 24 which uses the notation depicted in Figure 17.

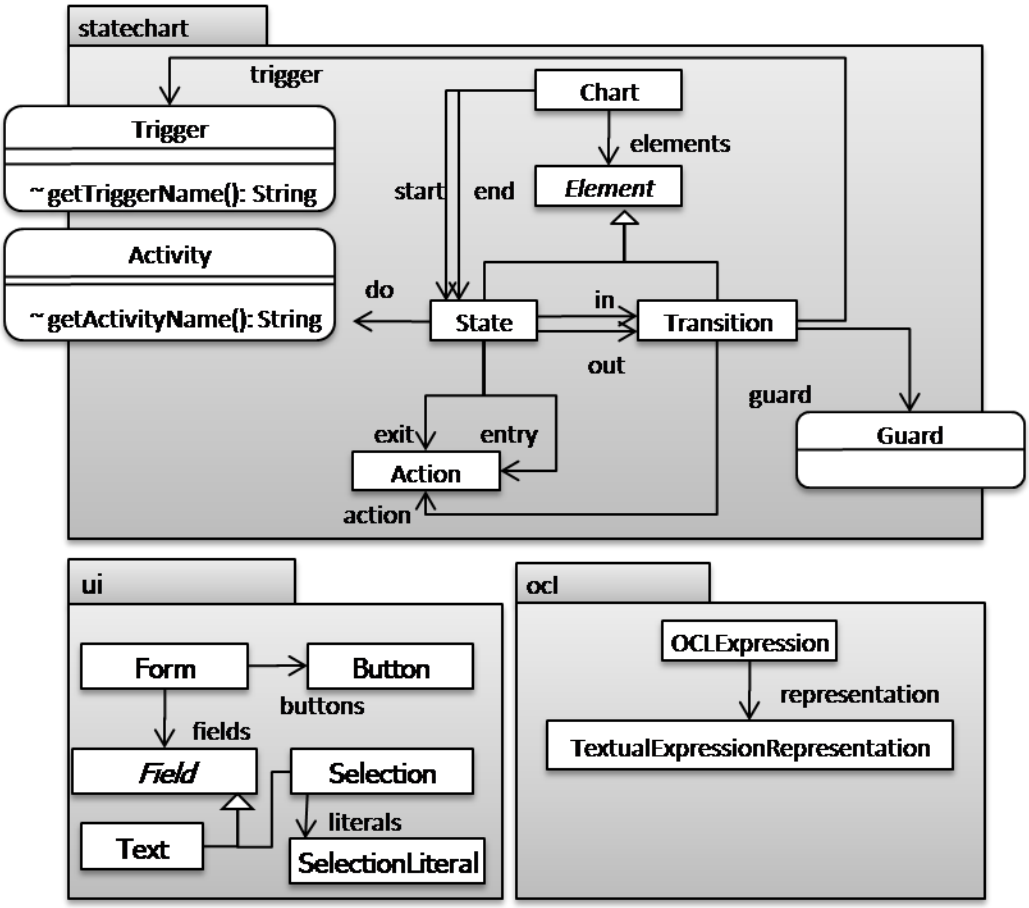


Figure 24 - LanGems of the exemplary wizard dialogue language

The `statechart` LanGem contributes the constructs of a state machine. A **`Chart`** consists of a number of **`State`**s and **`Transition`**s connected via the associations **`in`** and **`out`**. **`Transition`**s are associated with **`Trigger`**s and **`Guard`**s. **`Trigger`**s provoke state changes using **`Transition`**s they belong to, when all **`Guard`**s of the **`Transition`** hold. **`Trigger`**s, **`Guard`**s, and the **`Activity`**s performed in a **`State`** represent potential variation points in our **`statechart`** LanGem and are therefore modelled as role types with appropriate role operations.

The `ocl` LanGem allows the specification of **`OCLExpression`**s. For means of simplicity we restricted the representation of an expression to a purely textual format. This textual expression is fed into the Eclipse Model Development Tools (MDT) OCL Interpreter [MDT08] for evaluation. A more advanced realisation of the LanGem is in preparation and will provide means to represent OCL expressions using their ASM.

The `ui` LanGem introduces concepts to describe the structure of a **`Form`** dialogue. For this demonstration it is restricted to very basic constructs like **`Button`**s, **`Text`** and **`Selection`** fields. However, the LanGem can easily be extended to support more advanced user-interface elements.

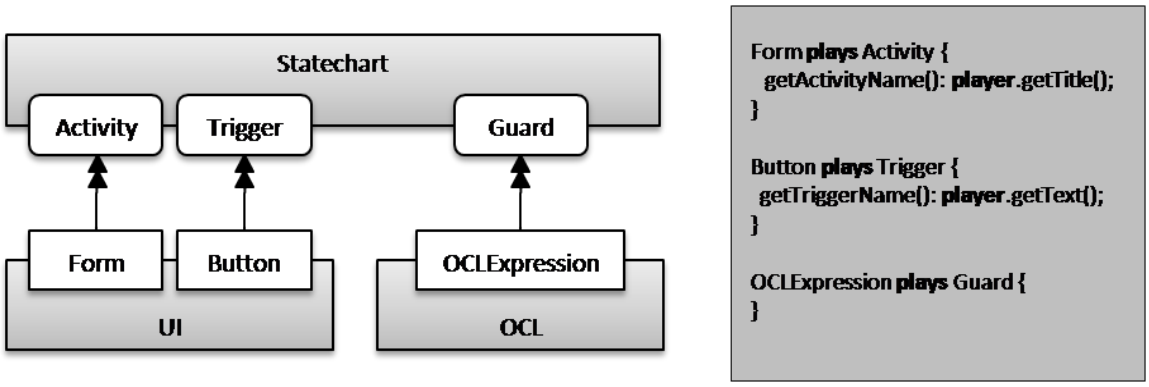


Figure 25 - Composition Program for the exemplary wizard dialogue language

Figure 25 depicts the program composing the three LanGems. It uses the notation introduced in Figure 17. The **`statechart`** LanGem realizes the integration of all sublanguages. Therefore, its role types are bound to natural concepts from the other LanGems. The **`ocl`** LanGem and the **`statechart`** LanGem are composed by using **`OCLExpression`**s for specifying **`Guard`**s in the state chart. Second, we compose the **`statechart`** LanGem and the **`ui`** LanGem: The **`Activity`** executed in a state of the wizard corresponds to opening the **`Form`** described with the UI-language and **`Button`**s pressed in the navigation area of a **`Form`** act as **`Triggers`** which provoke **`Transitions`**.

The differing domain abstractions are adapted by the role operation bindings depicted in the Listing in Figure 25.

Specification and Composition of LanGem Syntax and Semantics

Every LanGem introduces a special conceptualisation tailored to the purpose it is developed for. This eased the specification of its concrete syntax and the semantics. Figure 26 depicts the specification for the concrete syntax of the `statechart` LanGem. The rules use the EBNF-like syntax of EMFText: Rule heads refer to natural types they parse (e.g., `Chart`). Rule bodies use double-quoted strings to define terminal keyword tokens (e.g., **`"chart"`**) and non-terminals to refer to references (e.g., **`elements`**) and attributes (e.g., **`chartname`**) of the natural types. This connects concrete syntax elements and abstract syntax of a LanGem.

```
SYNTAXDEF chart
FOR     <http://de.tudresden/statechart>
START   Chart

RULES {
 Chart ::= "chart" chartname[] "Init: " init[]  "End: " end[]*
                                          "{" elements* "}";

 State ::= "state" stateName[]
                    ("entry:" "{" entry "}")?
                   "{" do "}"
                    ("exit:" "{" exit"}")?;


 Transition ::= "from" source[] "to" target[] "when" trigger[]
                    "[" guard[] "] "
                    "do " action[];

  Action ::= actionId "; "
}
```

Figure 26 - Specification of a concrete textual syntax for the statechart LanGem

Parsing rules for these non-terminals are derived from their types in the abstract syntax. For instance, to parse the non-terminal **`elements`** the rules for **`State`** and **`Transition`** are used alternatively, due to their inheritance relation with the type **`Element`** (cf. Figure 24).

Non-terminals which refer to role-types are handled likewise, with the difference that they are bound to the parsing rules of their role players during language composition. This integrates the concrete syntax of the combined LanGems. For instance, due to the role binding between **`Form`** and **`Activity`**, the non-terminal **`do`** in `State` is bound to the parsing rule for **`Form`** (cf. Figure 27. To derive a composed syntax specification for all LanGems, we combine their individual parsing rules.

```
SYNTAXDEF form
FOR     <http://de.tudresden/form>
START   Form

RULES {
 Form ::= "form"  heading['[', ']']
                  message['[', ']']
                  "{" (fields)* "}"
                  "buttons" (">" buttons)*;

 Text ::= fieldName[] ("=" defaultValue[])?;

 Selection ::= fieldName[]  cardinality['(', ')']
               "[" literals ("," literals)* "]"
               ("default" "[" defaultSelection[] ("," defaultSelection[])* "]")?;

 SelectionLiteral ::= literalValue[];

 Button ::= buttonName[];
}
```

Figure 27 - Specification of a concrete textual syntax for the form LanGem

1.13.3 Application of the Composed Language to Specify Wizard Dialogues

Figure 28 shows the application of the composed wizard dialogue language. It specifies an exemplary wizard to manage arbitrary items in a stock of inventory. As highlighted in the listing, constructs from the LanGems used form an integrated language which allows specifying different concerns of the software system using a well-suited conceptualisation. The composed parser translates the textual representation into an instance of the composed ASM.

Generating implementation code from this model would result in a multi-stepped wizard dialogue as depicted on the right of Figure 28. Single dialogue pages are represented by screenshots taken from the running application. Arrows between a form button and a page describe a path in dialogue flow triggered when the button is pressed. The OCL expressions annotated at these arrows specify the context conditions that must hold to walk the path.

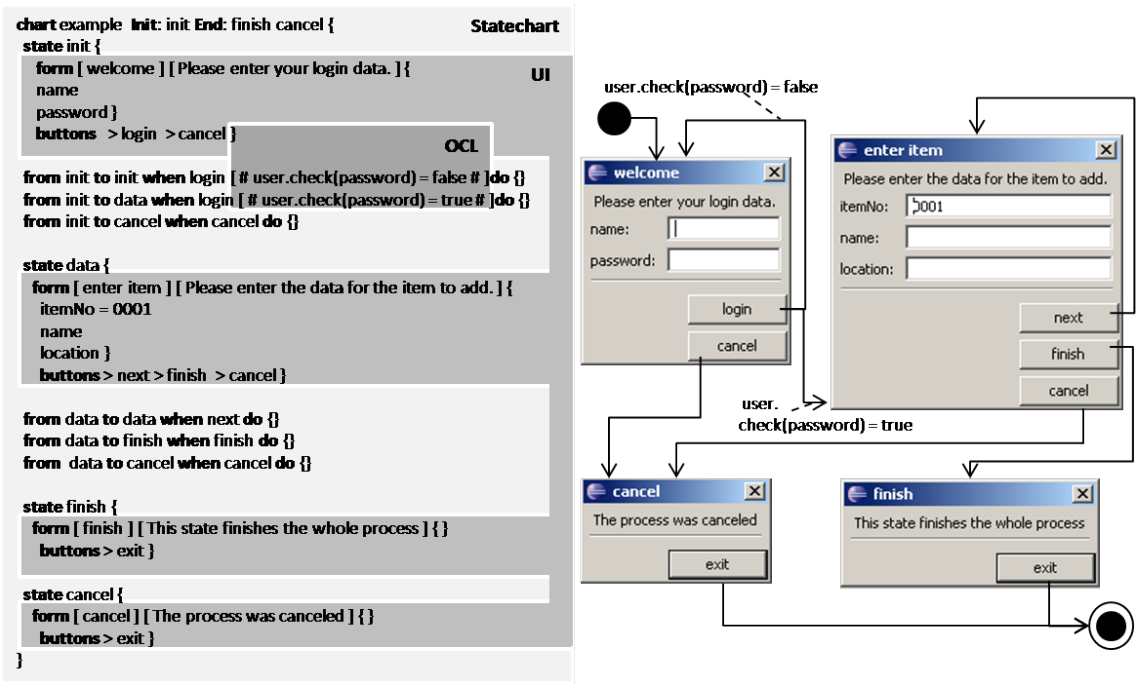


Figure 28 - Example for the Application of the composed Wizard Dialogue Language

5. Comparison of the different approaches

The presented approaches of building DSL and the different levels at which domain-specific abstractions are used have their advantages and disadvantages. There are differences in how close an implementation of a DSL is to the domain concepts and in the effort needed for DSL implementation. Here, we will discuss these differences.

1) **Abstraction Level:** The DSL engineering approaches takes place at different abstraction levels with differing premises for language composition. In Section 1.3, we have shown that at implementation level DSL embedding can be used for providing a specialized language infrastructure in general-purpose languages. In Section. 1.8, we have shown an approach for building DSL at the model level.

2) **Language Classification or Paradigm:** The languages used for DSLs implementation belong to different classes of languages or follow different paradigms. Programming languages have classifications, such as *functional programming*, *object-oriented programming*, *visual programming*, and so forth. Each of these languages and their paradigms have been studied for years and communities have revealed the advantages and disadvantages. In Section 1.3 we presented an approach to build DSLs by embedding them in host langauges. The selection of the right host language to base an embedded DSL implementation on should be driven by the requirements in language features and properties of the DSL under design. In Section 1.8 we have discussed how to build DSLs at modelling level where often no general-purpose abstraction is available. Here dedicated languages are used to specify language semantics and syntax.

3) **Language Features:** In Section 1.4, we discussed different languages that can be used for implementing EDSLs. What language features are needed to allow embedding of DSL is not the only important question for selection an adequate host language. Another important question is what (other) language features the host language provides that could be valuable in the target domain. For example, dynamic languages could be advantageous if the set of domain types is not fix and if we what to keep the set of domain types open. For example, strong typed languages allow automatically checking certain properties of DSL programs written for the EDSL. Another question is when an embedding DSL whether the host language features are inherited to the embedded DSL or not. We know that we cannot reuse all features in most languages and EDSL implementation approaches. E.g., the concept of modules cannot be reused, when embedding a DSL using our EDSL architecture in Groovy or Ruby. This is because an EDSL program currently is implemented in a method body in which the languages do not allow to define new modules. Thus DSL program must implement a new module concept without reusing the existing module feature. In Section. 1.8, discussed another way for reusing language features. We argued for modularising language specification in accordance to functional features and provided special techniques for composing these modules to integrated languages. This enables reuse of language features like name analysis and type checking among a family of DSLs.

4) **Syntax noise:** When embedding a language into a textual host language, the resulting EDSL syntax must be designed to conform to the host language syntax. In Section 1.5.2, we have discussed the concrete syntax noise in the POPART framework. Similar syntax noise exists for different host languages. The extend of

the syntax noise depends on how close the host language is to the DSL "ideal" syntax – the syntax one would define if the DSL would be implemented as an embedded DSL. Also the flexibility of the host language syntax is important, e.g., omissions of code fragments such as brackets can help to design the concrete DSL syntax to appear closer to the ideal syntax. In Section 1.8, we showed how language modules can be integrated syntactically by composing their language parsers. The composed textual syntax has no noise.

5) **Correctness and Safety:** The different host languages differ in the correctness and safety that is provided in the EDSL implementation. In Ruby, DSL programs are implemented as scripts in which domain objects do not have a special type that is checked before running the DSL program. This allows writing DSL programs that are incorrect and that evaluation results in errors at run-time [Ruby]. In contrast, EDSL can be implemented in Groovy such that types are optionally check or unchecked [Groovy]. Scala allows embedding a DSL in the type system, whereby correctness of DSL programs can be checked using the compiler [HORM08]. Also, in Omega [She04] other constraints not only types can be checked for embedded languages, this can be used to automatically guarantee domain-specific properties. The language composition approach presented in Section 1.8 uses a type-safe composition pattern that integrates languages at abstract syntax level. Thus, the domain-specific type system and type checking defined for individual languages is preserved during composition.

6) **Integrating Semantics:** Another important difference is how the semantics are provided. For instance, the EDSL implementation approach presented in Section 1.3 provides the semantics directly in the EDSL implementation. For integrating modelling languages semantically, Section 1.8 discusses a decentralized approach superimposing partial language semantics and a centralized approach for composing semantics using a universal conceptualisation.

7) **Reuse:** POPART allows modular implementation of EDSLs of which each is an extension of the host language syntax. EDSLs can be extended with new keywords and existing keywords can be overridden. Moreover, the EDSL approach in JRuby, Groovy, and Scala allows reusing standard Java libraries and their semantics built-in. Reusing libraries is particularly effective as a large set of libraries are available. The LanGems approach presented in Section 1.8 separates LanGem implementation and language composition and, thus, allows for reusing modularized language features and sublanguages across a family of modelling languages.

8) **Composability:** In Section 1.1, we have discussed what parts of modular language implementations can be composed. While certain approaches allow composing only the syntax, other allows composing semantics of the language implementation.

9) **DSL Integration:** An appropriate integration is needed to allow for embedding of EDSL code, transfer objects as parameters, and passing results back. Using Groovy for implementing EDSLs allows integration top of the Java platform. In contrast, several Ruby interpreters exist for different language platforms. This allows reusing, one EDSL implementation on different language platforms. Role-based language composition is a flexible technique to embed sublanguages into modelling languages, to enrich them with generic language features, and to integrate several modelling languages to a coherent system specification.

6. Conclusion

In this document, we have studied two orthogonal approaches of building DSLs and their advantages and disadvantages with respect to MDSD. We showed that embedded DSLs can be used to implement a MDSD-typical DSL rapidly. Further, we show that embedded DSLs and aspect-oriented programming can be used in concert. We also discussed how modular language engineering and language composition enables new reuse capabilities among modelling langauges with a slightly higher initial development effort.

In future work we will further investigate how embedded DSLs can be integrated with MDSD more tightly. We will elaborate how good support and integration should be designed. We will study known problem with the DSL implementation approaches. In particular, we strive for reducing the runtime overhead that is due to EDSL execution and for improving the reuse of the language infrastructure.

References

[ASS96] **Abelson H., Sussman, G., Sussman, J.:** *Structure and interpretation of computer programs*. MIT Press, Cambridge, Mass., 1996.

[ALSU06] **Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.**: *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley, 2006.

[And97] **Andersen, E.P.**: *Conceptual Modeling of Objects: A Role Modeling Approach.* Ph.D. Thesis. Oslo, Norway, University of Oslo 1997.

[ASCXML] **Apache Software Foundation**: *Apache Commons SCXML*. http://commons.apache.org/scxml/ (2008) Last accessed July 10 2008.

[Ass03] **Aßmann, U.***: Invasive Software Composition*. Springer-Verlag Inc., New-York 2003.

[Bet04] **Bettin, J**.: *Model-driven software development*. MDA Journal, 2004.

[BrLo07] **Bräuer M., Lochmann H.**: *Towards Semantic Integration of Multiple Domain-Specific Languages Using Ontological Foundations.* 4th International Workshop on (Software) Language Engineering (ATEM'07) at MoDELS, 2007.

[BrLo08] **Bräuer, M., Lochmann, H.**: *An Ontology for Software Models and its Practical Implications for Semantic Web Reasoning*. Proceedings of 5th European Semantic Web Conference, 2008.

[BV04] **Bravenboer, M., Visser, E.**: *Concrete Syntax for Objects: Domain-specific Language Embedding and Assimilation without Restrictions*. 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'04), 2004.

[CE00] **Czarnecki, K., Eisenecker, U**.: *Generative programming: methods, tools, and applications*. ACM Press, Addison-Wesley, New York, NY, USA, 2000.

[CSW08] **Clark, T., Sammut, P., Willans, J.**: *Applied Metamodelling a Foundation for Language Driven Development (2nd Edition)*. Ceteva, Available for download from http://www.ceteva.com/book.html, 2008.

[COPL08] **Lochmann, H.**: *The HybridMDSD Project*. Copenhagen Programming Language Seminar, Available for download from http://www.itu.dk/research/funtechs/coplas/2008-09-16.html, September, 2008.

[ETE09] **Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert and Christian Wende.:***Derivation and Refinement of Textual Syntax for Models.* Proc. of the 5th European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009).

[BBM03] **Budinsky, F., Brodsky, S.A., Merks, E.**: *Eclipse Modeling Framework*. Pearson Education, 2003.

[Fow05] **M. Fowler**: *Language Workbenches: The Killer-App for Domain Specific Languages?* http://martinfowler.com/articles/languageWorkbench.html. 2005.

[Fow05b] **M. Fowler:** *FluentInterface*. http://www.martinfowler.com/bliki/FluentInterface.html, 2005.

[Fujaba] **Fujaba Development Team**: *The Fujaba Toolsuite.* http://www.fujaba.de/. Last accessed July 10, 2008.

[GJSB00] **Gosling, J., Joy, B., Steele, G., Bracha G.:** *The Java Language Specication* (Second Edition). Addison-Wesley, Boston, Mass, 2000.

[Groovy] The Groovy Home Page. http://groovy.codehaus.org/.

[Haskell] The Haskell Programming Language. http://www.haskell.org/. Last accessed September 15, 2009.

[Hiber] Hibernate. http://www.hibernate.org/, Last accessed October 24, 2008.

[HORM08] **Hofer, C., Ostermann, K., Rendel, T., Morrs, A.**: *Polymorphic Embedding of DSLs*. In GPCE, 2008.

[Hud96] **Hudak, P.:** *Building domain-specific embedded languages.* ACM Computing
Surveys, 1996.

[Jenerator] **Völter M., Gärtner A.**: *Jenerator - Generative Programming for Java*. http://www.voelter.de/data/pub/jeneratorPaper.pdf, Last accessed July 10, 2008.

[Joe1997] **Joehanes R.:** Combining Pascal with Assembly. http://www.geocities.com/SiliconValley/Park/3230/pas/pasl2014.html. Last accessed September 16, 2009.

[KL07] **Kojarski, S., Lorenz, D.**: *Awesome: an aspect co-weaving system for composing multiple aspect-oriented extensions.* In OOPSLA, 2007.

[Koe07] **König, D., Glover, A.:** *Groovy in Action*. Manning, 2007.

[Lev65] **Levenštejn, V.:** *Levenshtein Distance*, 1965. http://en.wikipedia.org/wiki/Levenshtein_distance

[Mac08] **MacVittie, L.A.**: *XAML in a Nutshell*. O'Reilly Media, 2006.

[MDT08] **MDT Development Team**: *Model Development Tools (MDT) – OCL*. http://www.eclipse.org/modeling/mdt/?project=ocl, Last accessed July 10, 2008.

[MTM+97] **Milner, R. and Tofte, M. and Macqueen, D. and Harper, R.:** The definition of standard ML. The MIT Press, 1997.

[Oak01] Oaks, S.: *Java Security*. O'Reilly Media, 2001.

[OMG03a] **Object Management Group**: *MDA Guide Version 1.0.1*, 2003.

[OMG03b] **Object Management Group**: *Unified Modeling Language: Superstructure Version 2.0.* http://www.omg.org/cgi-bin/doc?ptc/03-08-02.pdf, OMG document number ptc/03-08-02, 2003.

[OMG03c] **Object Management Group***: UML 2.0 OCL Specification*. http://www.omg.org/cgibin/doc?ptc/03-10-14, OMG document number ptc/03-10-14, 2003.

[OSV07] **Odersky, M., Spoon, L., Venners, B.**: *Programming In Scala*. Artima Press, Mountain View, CA, USA, 2007.

[Paa07] **Parr, T.**: The Definitive ANTLR Reference: *Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[Rails] Ruby on Rails Homepage, http://www.rubyonrails.org/, Last accessed October 24, 2008.

[Ree96] **Reenskaug, T.**: *Working with objects. The OOram Software Engineering Method*. Manning Prentice Hall, 1996.

[Ruby] The Ruby Language. http://www.ruby-lang.org/. Last accessed October 27, 2008.

[Sam08] **Samek, M.***: Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems*. Butterworth Heinemann, 2008.

[Scala] The Scala Programming Language. http://www.scala-lang.org/. Last accessed September 15, 2009.

[SCXML] **W3C**: *State Chart XML (SCXML): State Machine Notation for Control Abstraction*. http://www.w3.org/TR/scxml/, Last accessed July 10, 2008.

[She04] **Sheard, T.**: *Languages of the future*. SIGPLAN Not., 39(12):119-132, 2004.

[Ste00] **Steimann, F.**: *On the Representation of Roles in Object-Oriented and Conceptual Modelling*. Data Knowledge Engineering 35(1), 2000.

[Struts] Apache: Struts Framework. http://struts.apache.org/, Last accessed October 24, 2008.

[Sun04] **Sun***: Java 1.5 - Annotations* http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html, Last accessed October 24, 2008.

[Win93] **Winskel, G.**: *Formal Semantics of Programming Languages*. The MIT Press, 1993.

[XUL] **Mozilla Foundation**: *XML User Interface Language (XUL) Project*. http://www.mozilla.org/projects/xul/, Last accessed July 10, 2008.