Supporting Variability with Late Semantic Adaptations of Domain-Specific Modeling Languages

Tom Dinkelaker, Martin Monperrus, Mira Mezini

Supporting Variability with Late Semantic Adaptations of Domain-Specific Modeling Languages

Tom Dinkelaker Martin Monperrus Mira Mezini

Technische Universität Darmstadt, Germany
{dinkelaker,monperrus,mezini}@cs.tu-darmstadt.de

ABSTRACT

Meta-object protocols are used to open up the implementations of object-oriented general-purpose languages to support semantic variability. They enable performing application-level semantic adaptations to the language even at runtime. However, such meta-object protocols are not available for domain specific-modeling languages. Also, existing approaches to implementing domain-specific modeling languages do not support semantic adaptations, where the application basically redefines specific parts of the language semantics. We propose a new approach for the implementation of domain-specific modeling languages that uses meta-objects and meta-object protocols to open up the implementation of domain-specific abstractions. This approach enables runtime semantic variability of the form of application-specific late semantic adaptations of domain-specific modeling languages that depend on the runtime application context.

Categories and Subject Descriptors

D.3.3 [**Software Engineering**]: Language Constructs and Features—*Classes and Objects, Frameworks*; D.2.11 [**Software Architectures**]: Languages

General Terms

Design, Languages

Keywords

Domain-Specific Modeling Languages, Variability, Semantic Adaptation, Meta-Object Protocols

1. INTRODUCTION

Domain-specific modeling languages (DSMLs) facilitate the development of software in a certain application domain by providing direct means to express domain-specific abstractions and operations. DSMLs are supported by domain-specific interpreters or compilers, which implement DSML syntax and semantics [22].

Previous work showed that most of the current methods for implementing DSMLs are closed with respect to changes in their semantics [30, 22]. For instance, van Deursen pointed that extensible DSL compilers and interpreters [30] have been little explored and Mernik stated [22] that *"building DSLs [...] in an extensible way"* is an open problem. To support the need for extensible modeling languages, note that even UML2 [25] defines an extension mechanism of its semantics called *semantic variation point*. This is where this makes its contribution: we propose a new approach for implementing DSMLs which supports semantic variability.

This approach allows DSML users to define the DSML semantics that exactly fits their needs, in the spirit of semantic variation points of UML2 [25]. For illustration, consider a model of a travel package booking Web service defined in a DSML for composing Web services. Let us assume that the initial DSML semantics only supports synchronous events consumption, thus DSML programs can only handle synchronous Web service partners. What happens if the default Web service for booking flights fails and that the only other partner available works asynchronously? The DSML application has to be rewritten using another modeling language. If the user could change the DSML semantics in an application-specific manner, she could implement an adaptation of the DSML semantics in order to enable asynchronous event consumption to also support asynchronous partners, while still reusing the initial DSML application and most of the default DSML semantics. If the DSML application has to support at runtime both synchronous and asynchronous partners, i.e. be self-adaptive to recover dynamically if a partner fails [2], the semantic adaptation has to depend on the execution context.

In [31], van Gurp coined the term *late variability* in the context of product lines, where it means changing a product after its delivery. In this paper, we explore the use of *late variability* in the context of DSML, which means being able to change the DSML semantics after the default interpreter or compiler has been delivered. To do so, we define the concept of *late semantic adaptation* as a replacement of one or more parts of the default semantics of the DSML within a DSML program; *late* meaning that the adaptation occurs after the delivery of the DSML and even as late as during the execution of a DSML program.

Let us now list and define what could be adapted in a DSML: A *domain type* is a type of the metamodel of the DSML, i.e. a type representing a domain abstraction. Adapt-

First International Workshop on Composition and Variability colocated with AOSD'2010 Rennes, St. Malo, France

ing a domain type means that every instance created after the adaptation will have the new semantics. Domain types contain *domain operations* that may change the state of domain objects. A *domain object* is an instance of a domain type. Adapting a domain object means that the semantics (the implementation) of its domain operations (and only the operations of this particular object) are changed.

Existing approaches to implementing DSMLs (e.g. DSML compilation [1], domain virtual machine [23], and polymorphic embedding [14]) do not support such late semantic adaptations, where the application basically redefines specific parts of the language semantics. In existing approaches, changing the semantics at runtime is only possible if the semantic adaptations had already been anticipated at design time. However, it is not possible to envision every possible semantic adaptation a priori at design-time. Even if it would be possible to embed into the DSML variation points for the known adaptations, the resulting implementation of the DSML semantics would be bloated with additional attributes and conditional logic. Such a *one-size-fit-alls* solution hampers the design of the default semantics which is used by most of the DSML programs. Last but not least, the DSML semantics could not be causally connected to the application state (i.e., dependent on the application state).

Our contribution is a method to implement DSMLs which are able to support runtime semantic adaptations. Meta-object protocols (MOPs) are interfaces to change the semantics of object-oriented programming languages [17]. MOPs define meta-objects that for instance handle the dispatch of method calls. Our key insights are that: 1) domain objects can be linked to a meta-object and 2) by implementing a DSML in a specific manner, an existing general-purpose MOP enables to change the semantics of the DSML itself. The method supports unanticipated semantic adaptations after the default DSML implementation has been delivered to a particular domain, as late as during the execution of a DSML program.

To evaluate our approach, we instantiate the method by building a DSML for state machines. This DSML supports semantic adaptations discussed previously in literature [25, 3].

The remainder of this paper is structured as follows. Section 2 presents different dimensions along which semantic adaptations may be defined. The proposed DSML method is presented in Section 3. Section 4 evaluates the support for semantic adaptations of a DSML implemented following our method. Related work is discussed in Section 5. Section 6 concludes the paper and discusses future research directions.

2. DIMENSIONS OF LATE SEMANTIC ADAPTATIONS

It's not straightforward to adapt the DSML semantics at the application level. DSML programmers require analysis means to design their adaptations. Hence, we have identified the following dimensions of semantic adaptations.

2.1 Scope of Variability

The first dimension along which we classify DSML semantic adaptations is the scope of variability. This dimension is discrete and has two values: (a) *domain type semantic adaptation* and (b) *domain object semantic adaptation.* A domain type semantic adaptation affects the semantics of all domain objects of a given domain type. On the contrary, a domain object semantic adaptation affects only one particular domain object.

(D3.a) Pre-Execution
Semantic Adaptation

```
--- DSL Loading
+++ Adaptation
+++ Adaptation
%%% DSL Execution
%%%
%%%
%%%
%%% End of Execution
```

(D3.b) Execution Context Dependent
Semantic Adaptation

```
--- DSL Loading
%%% DSL Execution
%%%
+++ Adaptation
%%%
+++ Adaptation
%%%
%%% End of Execution
```

Figure 1: Semantic Adaptations and DSML Program Execution

2.2 Granularity of Changes

The second dimension of semantic adaptations is the granularity of adaptations that are made. The size of these adaptations ranges from one single domain operation to multiple parts of the DSML semantics. Indeed changing one part of the semantics often requires also changing another part of the semantics, and multiple "elementary" semantic adaptations have to be packed into a unit of change.

2.3 Relation to DSML Execution

The third dimension characterizes the relation between the point in time in which the semantic adaptations happen and the point in time when DSML programs are executed. In most cases, the right semantics for a program execution can be determined beforehand and stays fixed for a complete program run. Sometimes, the selection of the right semantics depends on the execution state of a DSML program, i.e., a change in the DSML program context triggers a semantic adaptation.

Let us consider the following two abstract examples of execution traces that illustrate the two possible points in time when adaptations may take place. Figure 1 shows the difference in the execution traces of a pre-execution adaptation and a context-dependent adaptation. In both traces, the first step is to load the DSML program of which the corresponding trace is prefixed by "−−−". Then, two kinds of execution steps can occur: semantic adaptation ("+++") and normal DSML execution ("%%%").

The left-hand side of figure 1 schematically depicts a pre-execution semantic adaptation: the semantics of the DSML changes before any domain object is created, or any call to a domain operation has occurred. Note that multiple adaptations can be applied as indicated. Then, the DSML program is evaluated until completion. In this case, the adaptation is independent of the DSML execution. The right-hand side of figure 1 depicts an execution context-dependent semantic adaptation, as used in the running example. Unlike the previous trace, the adaptation happens during DSML execution, depending on the concrete values of domain objects. This is symbolized by the interlacing of several regular domain operation execution and semantic adaptation steps. Such context-dependent adaptations enable semantic self-adaptation of DSML programs.

3. A NEW METHOD FOR IMPLEMENTING DSMLS

This section presents a method for implementing DSMLs. DSMLs interpreters implemented with this method have the particularity to allow late semantic adaptations (as described in 2), i.e. semantic adaptations of the DSML inside DSML programs. We use the Groovy programming language to demonstrate the feasibility of the approach, as well as to fully instantiate the approach later in section 4.

3.1 Using Groovy to Implement DSMLs

Groovy [5, 18] is an object-oriented scripting language that nicely integrates with Java [12]. We have selected Groovy as the implementation language of our method for the following reasons:

1. Groovy provides a runtime MOPs in which meta-objects are first-class entities that can be directly accessed and modified by users[1].

2. Groovy has a *flexible syntax* that enables the definition of embedded DSMLs with a small syntax overhead. While for other host languages, such as Haskell, a large syntax overhead has been measured [19], Groovy supports *named parameters* and *command expressions* that allow the DSML implementer to design the syntax of the embedded DSML more openly.

3. Groovy is accessible to a broad community of developers since it has a syntax that is close to the Java syntax. Groovy is seamlessly integrated into Java and Groovy code can be called from Java code and vice versa. Hence, DSML programs can be called from Java and DSML programs can call existing Java libraries. All these argument allow an easy dissemination of our method.

While our method for implementing DSMLs could be implemented using other programming languages that come with a meta-object protocol (Smalltalk [10], CLOS [17], Ruby [27]), none of these languages satisfy all the aforementioned requirements.

Let us now give a quick overview of the features of Groovy that our method uses for implementing DSMLs[2]. Every Groovy object is bound to a meta-object [17]. This meta-object has several responsibilities: 1) it contains the logic related to introspection (e.g. the method `getMethods`) and 2) it handles every method call to this object. It is possible to change or replace this meta-object at runtime.

Also, there is a registry that links a class name to its default meta-object. Every new instance of a class, say `x`, is bound to the meta-object for its class in the registry. Hence, when the registry is updated, already existing objects keep the old meta-object and the new generation of objects is bound to the updated meta-object.

Groovy supports first-class closures. A closure can be created dynamically, passed as parameters to methods and functions, and executed. Listing 1 illustrates these points

[1]Note that users do not have to understand and use the MOP as long as they use the default semantics of a DSML and do not need to adapt it.

[2]Note that our approach is not bound to Groovy specifically, but to dynamic languages with a MOP. For instance, our approach is completely applicable in the context of Ruby.

```
// creating a closure
aClosure = {x->
print ”hello ”+x }

def m(Closure c) {
  c(”world”) // executing the closure
}

// passing the closure as parameter
m(aClosure)
```

Listing 1: Closures in Groovy

```
// creating a closure
aClosure = {-> bar() }

// two different contexts
class Context1 {
  def bar() { println ”bar” } }
class Context2 {
  def bar() { println ”bar2” } }

// executing the closure with Context1
aClosure.delegate=new Context1()
aClosure() // output ”bar”
// executing the closure with Context2
aClosure.delegate=new Context2()
aClosure() // output ”bar2”
```

Listing 2: Delegates in Groovy

Also, an important feature of Groovy closures is that their execution can be parameterized by a delegate context. By default, a closure has access to the *lexical context* in which it has been created. The lexical context contains all local variables of the closure. If it has been created within a method body, all variables available in the method are also available for the closure. In particular, all instance attributes and methods of the object that has created the closure are available when the closure is executed. This creating object is called the *owner* of the closure. In addition to the lexical context, the available context can be extended. When using a *delegate* for the closure, by changing its `delegate` attribute to refer to the *delegate*, the lexical context of the delegate becomes accessible in the closure. This way, the instance attributes and methods of another object than its owner can be used. If a function is not found in the closure's lexical context, a method with same signature is looked up in the delegate context, as shown in listing 2. Note that depending on the current binding of the delegate, the execution of the same closure can produce different results.

Technically, extending the available context for a closure is possible because Groovy uses a special meta-object for every closure. This meta-object first tries to lookup attribute accesses and method calls in the lexical context of a closure, i.e., in the local variables and in the owner. If no attribute or method with a corresponding name or signature is available in the lexical context and if the closure's delegate attribute is set, then the meta-object tries to lookup the attribute or method in the class of the delegate object. Only if the attribute or method can be found neither in the lexical context nor in the delegate, Groovy throws a runtime error.

3.2 The Embedding of DSMLs

Our method is based on Hudak's method to implement DSLs [15], i.e., no parser and compiler has to be written.

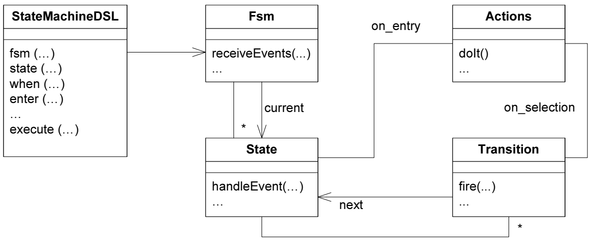

Figure 2: Architecture of the Default Interpreter

Implementing a DSML relies on two main steps. First, a metamodel specifies domain types, domain operations, and associated semantics in terms of a set of interrelated Groovy classes. Second, a syntactic language interface – a Groovy class – maps DSML syntax to DSML semantics by mapping DSML keywords to domain objects. There is a method in the syntactic language interface for each keyword in the DSML. DSML programs are enclosed in Groovy closures and the latter are assigned an instance of the language interface class as their delegate. The delegation mechanism of Groovy closures then maps DSML keywords to the corresponding method calls to a closure's delegate.

For instance, let us consider the implementation of the default semantics for a DSML for state machines. Figure 2 depicts the design of this DSML. The metamodel consists of classes `Fsm`, `State`, `Transition`, `Actions`. The `Fsm` class maintains a set of states, refers to a current state and implements some default semantics of state machines, e.g., the method `receiveEvents(...)` defines the dispatch mechanism of events received by a state machine. `State` instances maintain a set of outgoing transitions and may have an `on_entry` action and implement the state semantics. For instance, the method `handleEvent(...)` defines the state event handling mechanism. A `Transition` points to the `next` state. The `fire(...)` method is called whenever a transition is selected. The class `Action` encapsulates a set of domain-specific actions; the semantics of an action execution is encoded in the `doIt` method. Finally, the syntactic language interface is implemented in class `StateMachineDSML`. There is a method in `StateMachineDSML` for each keyword in the DSML. When called, these methods instantiate domain objects.

Listing 3 shows an excerpt of an embedded DSML program for state machines. The DSML program is contained in the closure `dslPackage` (cf. line 3) which is configured in line 22 to have an instance of `StateMachineDSML` as its delegate and whose evaluation starts at line 24. The evaluation is performed in two steps.

The first step transforms the textual DSML program (from line 5 to 9), embedded into the host language syntax, to a representation as a network of interrelated domain objects (instances of the domain classes from the metamodel, e.g., the instance `MyFsm` of class `Fsm`). During the execution of the closure, keywords, e.g. `fsm`, `state`, and `when`, are encountered in the DSML program. These keywords are turned into method calls due to the flexible syntax of Groovy. When using curly brackets at the end of a keyword method call, Groovy creates a closure and passes the closure to the method call as the last parameter. For instance, the program segment `fsm 'MyFsm', {...}` is turned

```
1  // this closure contains
2  // the DSML program + adaptations
3  def dslPackage = {
4    // the DSML program
5    fsm 'MyFsm', {
6      state 'S1', { ... }
7      ...
8      state 'SX', { ... }
9    }
10
11   // will execute the DSML program
12   // when the closure dslPackage is called
13   // with the event list passed as a parameter
14   MyFsm.execute({'ok','error',...})
15 }
16
17 // in Groovy a delegate is
18 // the interpretation context for closures
19 // we set the interpretation context for dslPackage
20 //  to be the default interpreter for state machines
21 dslPackage.delegate =
22   new de.tud.statemachine.StateMachineDSML()
23
24 dslPackage(); // evaluates the closure
```

Listing 3: State Machine Embedded in Groovy

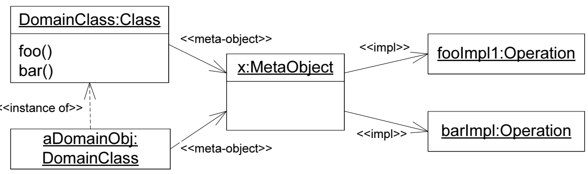

Figure 3: A Meta-Level for Domain Objects

into a method call of the form `fsm('MyFsm', closure-in-brackets)` with the `dslPackage` closure as the receiver. These calls are dispatched to closure's delegate, in this case a `StateMachineDSML`, of which the method with the corresponding keyword name and signature is called. These methods serve mostly as factories of domain objects. The second step, in line 14, is the execution of the DSML program as a method call to a domain object (resp. `MyFsm` and `execute`), given a specific execution context (`{'ok','error',...}`). This triggers a cascade of method calls on domain objects created during the first step.

3.3 How to Support Late Semantic Adaptations in DSMLs

In the following, we explain how to use meta-objects to enable late semantic adaptations. The meta-level introduced by our method is schematically depicted in Figure 3. Every domain type is mapped to a domain class A domain class, e.g., `DomainClass` in Figure 3, defines the domain operations of its instances – the domain objects, e.g., `aDomainObj`. The semantics of domain objects is reified in *meta-objects*, which are responsible for handling the execution of domain operations. Every domain class is associated with a meta-object, which is the default meta-object of any new instance of the domain class. Meta-objects, e.g., `x` in Figure 3, dispatch method calls received by domain objects to concrete implementations of domain operations. The links `meta-object` and/or `impl` can be changed at runtime, which is the key to allow dynamic semantic adaptations.

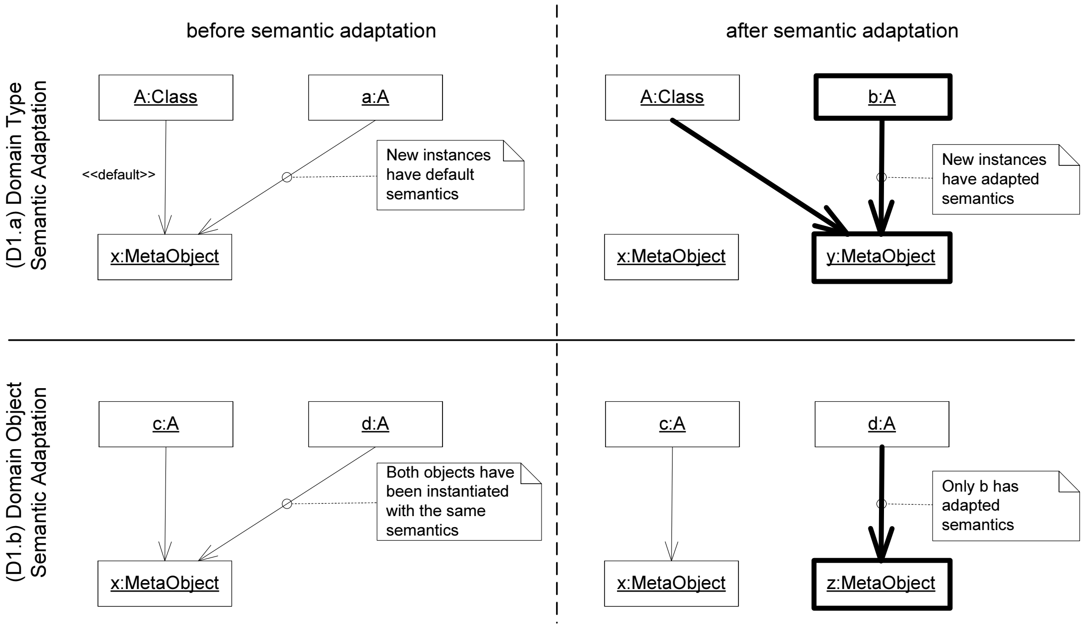

Figure 4: Dimension 1 – Scope of Variability

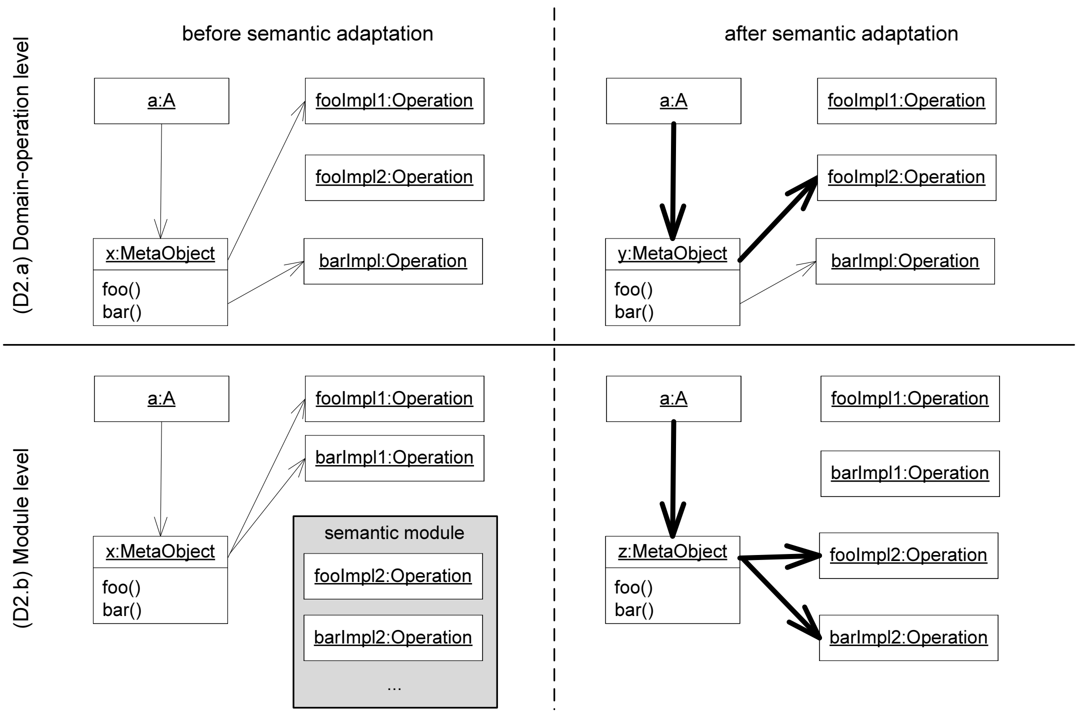

Figure 5: Dimension 2 – Granularity of Changes

Our base embedding method in Groovy presented above supports this meta-level: 1) all domain classes are Groovy classes whose semantics can be modified at runtime; 2) all domain objects are Groovy objects, and the corresponding meta-object can be changed for a single instance only.

3.3.1 *Scope of Variability*

Figure 4 shows how the two kinds of variability with regard to the scope dimension – domain type versus domain object – are supported in the proposed meta-level.

The upper part shows the effects of semantic adaptations whose scope is an entire domain type; the lower part corresponds to an adaptation that is specifically scoped for a particular domain object, thus, only domain objects are shown there. Both parts show the relation between domain types and domain objects to their corresponding semantics (encapsulated in a meta-object) before and after the adaptation.

In the upper left quadrant, the domain type `A` is bound to the default semantics represented by the meta-object `x`. Every new domain object that is created, e.g., `a`, runs under the default semantics. The semantic adaptation defines new semantics for domain type `A`. In the upper right quadrant, a new meta-object `y` is defined to represent the new semantics and `A` is associated with it. Any domain object created subsequently runs under the new semantics: the domain object `b` is created after the adaptation, hence, it is linked to the new meta-object `y`. Objects that were created before the semantic adaptation continue to run under the previous semantics, e.g., `a` is still linked to the meta-object `x`.

In the lower left quadrant, the domain objects `c` and `d` are created with the same semantics. The object-level semantic adaptation depicted here modifies the semantics of `d` only. The lower right quadrant shows the domain objects, meta-objects, and their relations after the semantic adaptation has taken place. While `c` keeps the former semantics, `d` uses the new semantics represented by meta-object `z`.

3.3.2 *Granularity of Changes*

Figure 5 depicts how adaptations at different levels of granularity are also supported by the proposed meta-level. The upper part shows the most fine-grained semantic adaptation at the level of an atomic domain operation. Unlike figure 4, the meta-objects are represented along with the domain operations. Doing so, we can highlight that the adaptation can separately impact a particular operation. In the upper left quadrant, the object `a` is attached to meta-object `x`; in the upper right quadrant, the same object is attached to a new meta-object `y`, which is the result of cloning `x` and binding `foo` to a new implementation, called `fooImpl2`. This way, the whole default semantics gets reused except the re-bound domain operation(s).

In general, it is likely that a semantic adaptation affects several places in the default implementation of the semantics. Obviously, it is preferable to apply the changes together as a unit of semantic adaptation. In contrast to the atomic adaptation at the level of a single domain operation, in this case the changes have to be packed into a bigger variability unit. This abstract unit is depicted as the gray rectangle in the lower left part of figure 5. When an adaptation happens, all changes of this adaptation unit are performed in concert. The impacted domain objects are then bound to a new meta-object, which is the result of mixing the previous meta-object and the semantic adaptation unit. In figure

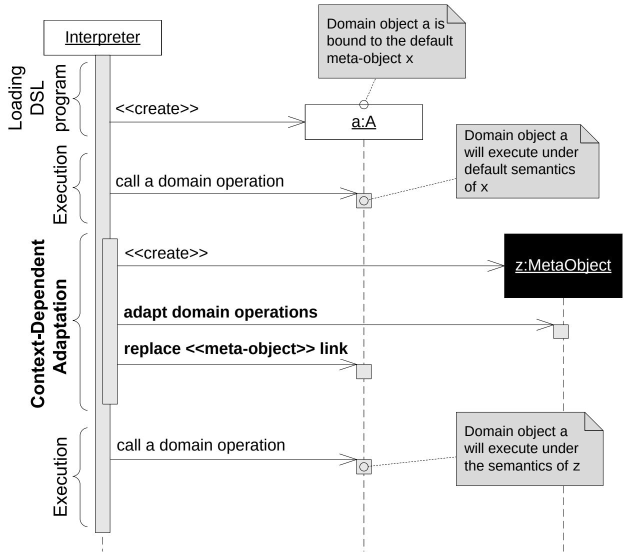

Figure 6: Context-Dependent Semantic Adaptation

5, after the change, the domain object `a` is attached to the meta-object `z`, which binds both `foo` and `bar` to new implementations.

3.3.3 *Relation to DSML Execution*

Semantic adaptations may occur at loading time or at runtime of DSML programs. Figure 6 depicts the sequence diagram of context-dependent semantic adaptation. Normal execution and semantic adaptation are interlaced. After *loading DSML program* the first *execution* phase starts. In this phase, while executing the DSML program and when entering a context that requires a semantic adaptation, a special phase is started that consists of applying semantic adaptations onto domain objects (respectively domain types). At the end of the adaptation phase, control is passed back to DSML execution. The subsequent *execution* phase will run under the new tailored semantics. It is worth mentioning that several adaptation phases can be executed, e.g., the adaptation can be reverted or other semantics can be installed.

4. APPLYING THE METHOD

This section discusses an implementation of a DSML for finite state machines (FSM DSML) using the method described in section 3. We show why the DSML interpreter supports semantic adaptations and how to implement them in at the level of DSML programs.

4.1 Possible Semantic Adaptations for the FSM DSML

The UML specification [25] discusses several semantic adaptations for state machines. We consider here two of them.

4.1.1 *Synchronous vs. asynchronous event handling.*

Listing 4 shows two possible implementations of `State`'s domain operation `handleEvent`. The first implementation is the default one and encodes synchronous event handling; the second implementation supports asynchronous event handling.

```
// default: synchronous event handling
def handleEvent(Event e) {
  this.fsm.currentState =
    this.transitionSelection(e).fire()
}

// alternative: asynchronous event handling
def handleEvent(Event e) {
  if (this.queue.isEmpty) {
    this.fsm.currentState =
      this.transitionSelection(e).fire()
  } else {
    this.fsm.queue.add(e)
  }
}
```

Listing 4: Two Implementations of handleEvent

```
// default semantics: deterministic transition selection
def transitionSelection(Event e) {
  return this.transitions.findAll(event).first
}

// alternative semantics: random transition selection
def transitionSelection(Event e) {
  return this.transitions.findAll(event).getRandom()
}
```

Listing 5: Two Implementations of transitionSelection

4.1.2 Deterministic vs. random transition selections.

A given state of a state machine can have several transitions matching a given event. In this case, a state machine implementation has to provide a transition selection policy. Following our method, the semantics of transition selection is encoded in the `transitionSelection` method of the domain class `State`. Listing 5 shows two possible implementations of `transitionSelection`. The default implementation of the transition selection policy is a deterministic one. It returns the first element of the collection of matched transitions. The alternative implementation selects a random item in the collection of matched transitions for a fairer load-balancing.

The following sub-sections will discuss scenarios of using our method to apply alternative semantics for event handling and transition selection thereby varying the kind of adaptation along the three dimensions discussed previously. Section 4.3, specifically, will discuss how to combine several variation points in one semantic module; for instance, how to use the tailored version of both `handleEvent` and `transitionSelection` in a concise and elegant manner.

4.2 Scope of Adaptations

Listing 6 shows the implementation of the first dimension presented in section 3.3.1.

Listing 6 illustrates domain type semantic adaptation. The module `dslPackage` (lines 5-9) is a Groovy closure, which consists of three parts: (a) the declaration of a DSML program (lines 5 to 9), (b) a piece of meta-program that tailors the semantics of the DSML (lines 17 to 19), and (c) a piece of code that starts the execution of the DSML program (line 26). Parts (a) and (c) have been explained in 3.2. The adaptation consists in changing the `transitionSelection` method of the default state meta-object in line 17. As explained earlier, all instances of class `State` are affected by

```
1  // this closure contains the DSML package
2  // (DSML program + adaptations)
3  def dslPackage = {
4    // the DSML program
5    fsm 'MyFsm', {
6      state 'S1', { ... }
7      ...
8      state 'SX', { ... }
9    }
10
11   // adaptation of the default semantics
12   // of the domain class State
13   // by replacing the implementation of
14   // the transitionSelection method
15   // of the default meta-object associated
16   // with the State class
17   State.metaClass.transitionSelection = { event ->
18     return this.transitions.findAll(event).last
19   }
20
21   // executing the DSML program
22   MyFsm.execute({'ok','error',...})
23 }
24 dslPackage.delegate =
25   new de.tud.statemachine.StateMachineDSML()
26 dslPackage();
```

Listing 6: Domain Type Semantic Adaptation

this kind of adaptation and will execute with the tailored transition selection semantics.

For sake of space, we cannot elaborate on a complete DSML program that performs a semantic adaptation at the level of a single domain object. The code is similar to that in listing 6, except that it is not the `metaClass` of class `State` (line 17) that is changed but the `metaClass` of a domain object. For example, we can change the meta-object of `S1` using `MyFsm.S1.metaClass.transitionSelection = {...}` .

4.3 Granularity of Adaptations

This section illustrates the second semantic dimension, presented in section 3.3.2. On the one extreme in this dimension, a semantic adaptation affects a single domain method; on the other extreme, a semantic adaptation may imply the construction of a completely new meta-object.

Listing 7 shows DSML code that is embedded similarly to listing 6. It focuses on the adaptation in lines 7 to 10. The only element that is changed is a domain method of a domain class.

On the contrary, listing 8 shows the creation of a semantic module and its use for tailoring the semantics of a domain class. Similarly to using classes to represent domain types, we use a new subclass for modularizing alternative semantics for a domain type. In the example, lines 2–9 define such a subclass called `TailoredState`. Subclassing a domain type to create a new meta-object allows leveraging two key Groovy features used in listing 8:

1. The possibility to attach new semantics to an existing domain class, using a registry mechanism (line 13).

2. The automatic creation of a meta-object for each new class (line 14).

A meta-object for the new subclass is automatically created and stored in the class variable `TailoredState.metaClass`. In listing 8 lines 13–14, we register this meta-object

```
fsm 'MyFsm', {
  ...
}

// we tailor only transition selection
// part of the semantics of States
State.metaClass.transitionSelection = {
  event ->
  return this.transitions.findAll(event).last
}
// executing the DSML program
MyFsm.execute({'ok','error',...})
```

Listing 7: Method-Level Adaptation

```
// we use classes for modularizing the semantics
class TailoredState extends State {
  def transitionSelection(event) {
    /* cf. variation point transitionSelection */
  }
  def handleEvent(event) {
    /* cf. variation point handleEvent */
  }
}

// we tailor the semantics of State in one unit
// for both event handling and transition selection
InvokerHelper.metaRegistry.setMetaClass(State,
  TailoredState.metaClass)
// executing the DSML program
MyFsm.execute({'ok','error',...})
```

Listing 8: Semantic Module Adaptation

also for the State class. As a consequence, the domain operation implementations of TailoredState will be used for State objects.

4.4 Moment of Adaptations

As shown figure 1, a pre-execution adaptation is simply a piece of code preceding the DSML program that changes the semantics of the language itself. While our method enables such adaptations, for sake of space, we cannot elaborate on them.

We now present a complete example of a semantic adaptation that occurs during the execution of a DSML program, i.e. a context-dependent adaptation. Consider now the self-adaptive DSML program in listing 9. This DSML program is a state machine representing a Web service composition for a travel booking process. The machine consists of two states, first booking a flight and second booking a hotel. In both states, a call to a Web service is made in the on_entry blocks. States transitions are triggered by the reception of Web service responses.

This program is self-adaptive since it is able to recover from failing synchronous partners, thanks to our FSM DSML which supports semantic adaptations: the TravelPackage program is able:

1. to handle the error event in the BookingFlight state (line 12);
2. to replace the failing partner whith an asynchronous one (line 14)
3. to adapt itself to the new webservice by changing the event reception semantics of the DSML only for state BookingFlight in order to listen to asynchronous events (lines 12–21).

```
1  def dslPackage = {
2    // declaration of the DSML program
3    fsm 'TravelPackage', {
4      state 'BookingFlight', {
5        on_entry {
6          /* synchronous call to flight_webservice */ }
7
8        // nominal mode
9        when 'done', { enter 'BookingHotel'}
10
11       // error recovery mode
12       when 'error', {
13         // change to an asynchronous webservice
14         targetService = "another_flight_webservice"
15         // and we have to change the semantics too
16         this.metaClass.handleEvent = { event ->
17           /* new asynchronous implementation */
18         }
19         // re-enter current state
20         enter 'BookingFlight'
21       } // end when
22     } // end state
23
24     state 'BookingHotel', {
25       on_entry {
26         /* synchronous call to hotel_webservice */ }
27       }
28     }
29   }
30
31   // executing the DSML program
32   TravelPackage.execute({'ok','error','done',...})
33 }
34 dslPackage.delegate =
35   new de.tud.statemachine.StateMachineDSML()
36 dslPackage();
```

Listing 9: Context-Dependent Adaptation

In such cases, the semantic adaptation code becomes part of the code of the DSML program and the adaptation logic is executed only when DSML execution reaches the lines 12–21.

In this section, we have presented an instantiation of our method as a proof of concept that has shown its real implementability. From the viewpoint of end-users of our method, i.e. DSML designers, this section is fully complementary to the conceptual presentation of our method in sections 2 and 3 and enables them to implement on their own a DSML that supports late semantic adaptations.

5. RELATED WORK

Domain-Specific Languages.

The implementation of DSMLs using "traditional, closed" compilers (e.g. [1]) does not allow semantic adaptations. In contrast, extensible compilers, such as Polyglot [24] or JastAdd [13, 8], target semantic adaptations of the form of extensions to Java language. Akesson et al. [33] address the implementation of extensible DSMLs. However, extensible compilers do not support all the semantic adaptation dimensions discussed in this paper. Only class-level adaptations are supported, in the sense that the adaptation granularity is the class as well as in the sense that all domain objects are executed under the same semantics. Furthermore, dynamic semantic adaptation that depends on application execution state is not supported. Last but not least, the application adaptation are very different as opposed to our approach, where – due to using the language embedding technology –

the same language is used for implementing an application and the semantics of the DSML.

Our approach follows the *domain virtual machine* pattern [9], i.e., it is a DSML interpreter realized by a set of domain classes implementing the domain semantics in their methods. Kermeta [23] is a language to implement DSML interpreters following the domain virtual machine pattern. However, our domain classes are embedded into a host language which allows to seamlessly integrate DSML programs and programmatic semantic adaptations. More importantly, our approach supports non-invasive, application-specific, and even execution context specific semantic adaptations.

Steele [28] proposes to build interpreters out of a set of building blocks called *pseudomonads*, in reference to Haskell monads [32]. Achieving a semantic adaptation can be done by composing interpreters. Comparing to our method, the DSML programmer has to understand not only the interpreter of the DSML but also the composition operator of pseudomonads.

Ramsey [26] described the implementation of the Lua scripting language as an embedded interpreter in Objective Caml. While Ramsey implements a general-purpose language (Lua) interpreter, our approach targets domain-specific interpreters in order to design them as extensible.

The initial method of embedding DSMLs by Hudak [15] does not consider the issue of semantic adaptations. Similar to our work, polymorphic embedding [14] enables several interpretations of a DSML program by employing a similar architecture for the DSML implementation that separates the language interface and the domain metamodel. However, polymorphic embedding does not support a meta-level architecture allowing DSML programs to change their semantics in a fine-grained and application context-specific manner during their execution.

Reflection and Meta-Object Protocols.

Meta-interfaces have been implemented for various languages, e.g., 3-KRS [21], CLOS [17], Smalltalk [10]. Meta-object protocols (MOPs) provide "interfaces to the language that give users the ability to incrementally modify the language's behavior and implementation" [17]. MOPs are *open implementations* [16] of (object-oriented) general-purpose languages. Compile-time MOPs have been provided for popular compiled languages OpenC++ [4] and OpenJava [29]. MOPs have been adopted in dynamic scripting languages, such as Ruby [27] and Groovy [5]. Using the above MOPs for extending DSML semantics have not been addressed.

There are no methods for DSML implementation available, that derive a MOP for the implemented DSML. The approach proposed in this paper is generic for class-based languages. Other dynamic languages that come with a MOP can be used to provide a flexible DSML semantics as presented in this paper.

xPico [11] allows to extend the syntax and semantics even at runtime by reflectively manipulating the AST at well-defined adaptation points. The idea to use reflection and the targeted flexibility is similar to our approach. Although xPico allows syntactic variability, the semantic adaptation of xPico is limited, as explicit adaptation points must be defined to allow extensibility. The problem with the xPico approach is that it does not provide an adequate meta-interface and provides only access to the AST but not to domain abstractions. However when implementing a DSML for multiple users, it is impossible to envision every adaptation point at design-time of the DSML semantics. On the contrary, our approach permits unanticipated adaptation points, i.e., every domain class method is a *latent* adaptation point.

A domain-specific meta-object protocol for distributed environment, called diMOP, has been presented in [20]. This design-time MOP is used to specify behavioral characteristics, such as non-functional concerns, at the design level in extended UML diagrams. However, the focus of this paper is the language level and executable meta-objects. The idea of having a domain-specific meta-object protocol is an interesting one. The diMOP is only a MOP for one domain, while every DSML implemented following our approach exposes a domain-specific MOP.

6. SUMMARY AND FUTURE WORK

In this paper, we have presented a method for implementing DSMLs that support semantic adaptations that may be application-specific and may occur as late as during the execution of DSML programs. The proposal leverages meta-objects [16] in the context of domain-specific modeling languages. Also, we have elaborated on an instantiation of the method in the Groovy programming language in the context of state machines. Although not shown in this paper, our solution is applicable for DSMLs with more complex sets of domain concepts (and language constructs), such as workflow languages, aspect languages, and others [6, 7].

As usual for dynamic approaches, there is a trade-off between adaptability and statically checked correctness. Our approach supports a maximal adaptability and may suffer from possible correctness issues. For instance, DSML programmers who override part of the default DSML semantics might violate contracts and responsibilities that are implicit in the DSML design. These limitations will be addressed in future work. For instance, semantic adaptations may require adaptations in several domain classes and operations performed in concert. Our future work will also explore explicit contracts that can be checked at runtime to ensure the semantic consistency of adaptations.

7. REFERENCES

[1] D. Batory, B. Lofaso, and Y. Smaragdakis. Jts: Tools for implementing domain-specific languages. In *Conference on Software Reuse*, pages 143–53, 1998.

[2] A. Charfi, T. Dinkelaker, and M. Mezini. A Plug-in Architecture for Self-Adaptive Web Service Compositions. In *Proceedings of the 2009 IEEE International Conference on Web Services*, pages 35–42. IEEE Computer Society, 2009.

[3] F. Chauvel and J.-M. Jézéquel. Code Generation from UML Models with Semantic Variations Points. In L. Briand and C. Williams, editors, *UML MoDELs*, volume 3713 of *LNCS*, pages 54–68, Montego Bay, Jamaica, October 2005. Springer Verlag.

[4] S. Chiba. A metaobject protocol for C++. In *OOPSLA*, pages 285–299. ACM Press New York, NY, USA, 1995.

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

[6] T. Dinkelaker. Versatile language semantics with reflective embedding. In *Proceedings of the 2009 OOPSLA Doctoral Symposium*, 2009.

[7] T. Dinkelaker, M. Eichberg, and M. Mezini. An architecture for composing embedded domain-specific languages. In *Proceedings of AOSD*, 2010.

[8] T. Ekman and G. Hedin. The jastadd extensible java compiler. In *Proceedings of OOPSLA'2007*, 2007.

[9] J. Estublier, G. Vega, and A. D. Ionita. Composing domain-specific languages for wide-scope software engineering applications. In *Proceedings of MODELS/UML*, 2005.

[10] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley, Boston, MA, USA, 1983.

[11] S. Gonzalez, W. De Meuter, and V. Brussel. Domain-Specific Language Definition Through Reflective Extensible Language Kernels. In *Workshop on Reflectively Extensible Programming Languages and Systems (at GPCE)*, 2003.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass, 2000.

[13] G. Hedin and E. Magnusson. JastAddŮan aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.

[14] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of DSLs. In *Generative Programming and Component Engineering (GPCE'08)*, pages 137–148. ACM New York, NY, USA, 2008.

[15] P. Hudak. Modular domain specific languages and tools. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[16] G. Kiczales. Beyond the Black Box: Open Implementation. *IEEE Software*, 13(1):8–11, 1996.

[17] G. Kiczales, J. d. Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[18] D. König and A. Glover. *Groovy in Action*. Manning, 2007.

[19] T. Kosar, P. Martínez López, P. Barrientos, and M. Mernik. A preliminary study on various implementation approaches of domain-specific language. *Information and software technology*, 50(5):390–405, 2008.

[20] J. Lee, S. Min, and D. Bae. Aspect-Oriented Design (AOD) Technique for Developing Distributed Object-Oriented Systems over the Internet. In *International Computer Science Conference*. Springer, 1999.

[21] P. Maes. *Computational Reflection*. PhD thesis, Vrije Universiteit Brussel, 1987.

[22] M. Mernik and A. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)*, 37(4):316–344, 2005.

[23] P. A. Muller, F. Fleurey, and J. M. Jézéquel. Weaving executability into object-oriented meta-languages. In *Proceedings of MODELS/UML 2005*, 2005.

[24] N. Nystrom, M. Clarkson, and A. Myers. Polyglot: An extensible compiler framework for java. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 138–152. Springer, 2003.

[25] OMG. UML 2.0 superstructure. Technical report, Object Management Group, 2004.

[26] N. Ramsey. Ml module mania: A type-safe, separately compiled, extensible interpreter. *Electronic Notes in Theoretical Computer Science*, 148(2):181–209, 2006.

[27] Ruby programming language. `http://www.ruby-lang.org/`.

[28] G. L. Steele. Building interpreters by composing monads. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 472–492. ACM New York, NY, USA, 1994.

[29] M. Tatsubori, S. Chiba, M. Killijian, and K. Itano. OpenJava: A class-based macro system for Java. *Reflection and Software Engineering*, 1826, 2000.

[30] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

[31] J. van Gurp. *Variability in software systems: the key to software reuse*. PhD thesis, Blekinge Institute of Technology, 2000.

[32] P. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming (LFP'90)*, pages 61–78, 1990.

[33] J. Åkesson, T. Ekman, and G. Hedin. Development of a modelica compiler using jastadd. *Electronic Notes in Theoretical Computer Science*, 203(2):117 – 131, 2008. Workshop on Language Descriptions, Tools, and Applications (LDTA 2007).