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.

Previous work showed that most of the current methods for implementing DSMLs are closed with respect to changes in their semantics. For instance, van Deursen pointed that extensible DSL compilers and interpreters have been little explored and Mernik stated that "building DSLs [...] in an extensible way" is an open problem. To support the need for extensible modeling languages, note that even UML2 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. 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, 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-
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-fits-all 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].

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.

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 3.

3.1 Using Groovy to Implement DSMLs

Groovy [3, 13] 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.

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 arguments 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. 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.

Listing 1: Closures in Groovy

```
// creating a closure
aClosure = {x->
  println "hello "+x
}
def m(Closure c) {
  c("world") // executing the closure
}
// passing the closure as parameter
m(aClosure)
```

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.

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.

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.

```
// 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"
```
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 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, 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, are responsible for handling the execution 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 5 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 \( D \) is bound to the default semantics represented by the meta-object \( z \). Every new domain object that is created, e.g., \( a \), runs under the default semantics. The semantic adaptation defines new semantics for domain type \( D \). In the upper right quadrant, a new meta-object \( y \) is defined to represent the new semantics and \( D \) 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 \( z \).

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 6 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 5, 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 6. 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 6, 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 7 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.
A meta-object for the new subclass is automatically created in the line 14. The automatic creation of a meta-object for each new domain type to create a new meta-object allows leveraging two key Groovy features used in listing 8.

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 9 to 10 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 9.

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

2. The automatic creation of a meta-object for each new subclass.
Listing 7: Method-Level Adaptation

def handleEvent(event) {
    /∗ cf. variation point handleEvent ∗/
}
def transitionSelection(event) {
    /∗ cf. variation point transitionSelection ∗/
}

Listing 8: Semantic Module Adaptation

def validate(event):
    if event == 'error':
        raise Exception('Error occurred during booking flight')
    return True

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 23 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 [6], i.e., it is a DSML interpreter realized by a set of domain classes implementing the domain semantics in their methods. Kermeta [22] 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 multi-

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 [10] 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