Dynamically Linked Domain-Specific Extensions for Advice Languages

Tom Dinkelaker Mira Mezini

Darmstadt University of Technology
Hochschulstr. 10, 64289 Darmstadt, Germany
{dinkelaker,mezini}@informatik.tu-darmstadt.de

Abstract

Domain-specific aspect languages allow defining aspects for a certain problem domain faster and easier by providing specialized expressivity and by reducing the complexity of the language interface. However, providing domain-specific aspect languages is a rather complex task. With current approaches only specialists can build new domain specific aspect languages; in doing so they have to replicate large parts of the tool set. In this paper, we have extended a general-purpose aspect language to support embedded domain-specific syntax in the advice language. The approach has several advantages. First, it allows reusing a large part of existing tools and infrastructure. Second domain-specific extensions can be defined in separated modules, which can be dynamically linked into the advice language; these modules can be inherited from, they can be refined from existing implementations, and can be composed to support abstractions from different domains.

Categories and Subject Descriptors D.2.3 [**Software Engineering**]: Coding Tools and Techniques

Keywords Domain-specific Aspect Languages, Embedded Domain-specific Languages.

1. Introduction

The aspect-oriented language community has addressed special problem domains using domain-specific aspect languages (DSAL) from the beginning [17]. The idea is to provide special domain abstractions in an AO language in order to simplify the implementation of aspects[1][2]. Current research on DSAL focuses on providing new DSALs implemented from scratch or by using specialized compilers [13], program transformation [18], and designated execution environments [15] that weave in aspects defined in a DSAL. Current DSALs provide good support for particular domains, such as remote invocation, error handling [18], and advanced transactions [14]. Frameworks have been proposed that allow introducing new keywords [13] and that allow defining templates for aspect definitions [16].

A problem with most existing DSAL approaches is that they do not facilitate the refinement and composition of the domain-specific extensions. As a consequence, significant parts of the implementation efforts have to be duplicated when creating new DSALs and large parts of the companying tools have to be replicated for each DSAL. Today's approaches can only be used by developers that are experts in building parsers, compilers, or execution environments. This strongly restricts their applicability for ordinary DSAL users that are only familiar with the principles of AOP.

An alternative approach to support domain-specific abstractions are embedded domain-specific languages (EDSLs) [8][9][10], which is a well-known technique in several languages. In this paper we show that EDSLs can also be used to rapidly prototype domain-specific extensions (DSX) for aspect languages. However, the approach cannot be used with popular AOP tools, as the latter do not support embedding DSLs.

To address this problem, our ongoing research is focused on answering the following questions: *Can we provide a generic approach to provide new and extensible implementations of domain-specific aspect languages using the embedded languages approach?* And, *can we provide a language framework that can directly be used by language designer to tailor a domain-specific aspect language for the user's domain?* The above questions have to be answered twice for domain-specific aspect languages - once for *pointcut language* and once for the *advice language*. In this paper the focus is on embedding DSXs in advice languages.

The basic idea is to extend aspect languages with support for embedding DSL. We have built an extension to AspectJ [1] that allows implementing advice in the Groovy scripting language [1]. Groovy has been chosen because of its very good support for embedding DSLs. Our approach allows AOP developers to design new domain-specific languages for user's needs and for special domains that have not been considered so far. Thereby, AOP developers are guided by a well-defined recipe for writing a DSX. The approach has several advantages. First, a DSX can concisely be defined in only one class. Second, the approach allows developer to extend and refine existing DSX implementations, and it supports ad-hoc composition of DSXs.

The reminder of the paper is structured as follows. Sec. 2 gives a short introduction to Groovy. Sec. 3 elaborates on a pattern for implementing EDSL and shows how to implement an EDSL in Groovy. Sec. 4 presents our approach for building DSXs and Sec. 5 shows several applications of our prototype. Sec. 6 gives implementation details. Sec. 7 concludes the paper.

2. Introduction to Groovy

Groovy [1] is a dynamic scripting language based on Java technology. At runtime, right before execution, Groovy code is compiled to Java *bytecode* that is interpreted by a Java VM. Groovy provides a sophisticated set of language features. However, it is out of the scope of this paper to give a full introduction to Groovy. Therefore, we refer to [1] and discuss only the features relevant for this work. Groovy has a Java-like syntax, but in contrast it is a *pure* object-oriented language, i.e., everything is an object. Primitives are *auto-boxed* and the language supports both *duck typing* and *strong typing*. Groovy has special built-in language support for selected types. Groovy provides so-called **`GString`**s in which one can directly access variables such that **`println "Value of x is $x"`** will print out the value of **`x`** inside the string.

One can statically construct Lists and Arrays listing their elements **`[elem`$_1$`,…, elem`$_N$`]`**, HashMaps using **`[key`$_1$`:val`$_1$`,…, key`$_N$`:val`$_N$`]`**, and regular expression can be defined using the following syntax: **`/<RegEx>/`**.

Further, Groovy supports overloading certain operators for arbitrary classes. Operators are automatically mapped to method calls, e.g., the usage of the "**`+`**" operator in **`a+b`** is automatically mapped to the call **`a.plus(b)`**. Thus, one can overload the "**`+`**" operator by overwriting the **`plus()`** method. Another special type is closure. A Groovy *closure* is a first-class entity that can be used to defer the evaluation of a piece of code. A closure block is defined using curly brackets. E.g., **`Closure cl = {x -> x*x}`**, defines a closure that takes the (untyped) parameter **`x`** and returns its square value. Closure instances like **`cl`** can be referenced and passed to methods as parameters. Later, the closure **`cl`** can be evaluated by invoking **`cl.call(5)`**, which will return **`25`**. By default, Groovy closures do not come with designate environment. Nonetheless, we can bind a context to a Groovy closure by setting a *delegate object*. E.g., for **`{y+z}.delegate = [y:1,z:2]`**, the delegate HashMap will receive all property accesses (and method calls): it uses the property name as the key and returns the associated value.

The collection types define certain methods that take a closure as an argument and apply it to each element in order to iterate over contained elements, e.g., **`list.each {e -> print "$e "}`** prints all elements of the list. Groovy defines a *meta-object protocol* (MOP), which can be used for *pretended method calls* (and *pretended properties*), which means that code may invoke a method on an object that is not defined in its class. As method calls are first handled by the MOP, they can be forwarded to other methods or objects. Groovy defines a number of rules about the meaning of pretended calls for certain classes.

3. Embedded Domain-Specific Languages

Object-oriented scripting languages can be used to implement DSLs using a special implementation technique that embeds the domain-specific language into another language. A DSL implemented using this approach is called an *embedded domain-specific language* (EDSL) [8]. By reusing the features of the *host language* in which the EDSL is embedded, the EDSL implementation approach promises to faster provide a powerful language than following the traditional approach. The traditional approach requires the developer to provide parsers, compilers, and development tools; in contrast, new EDSLs can partially reuse the parsers, compilers and development tools of their host languages.

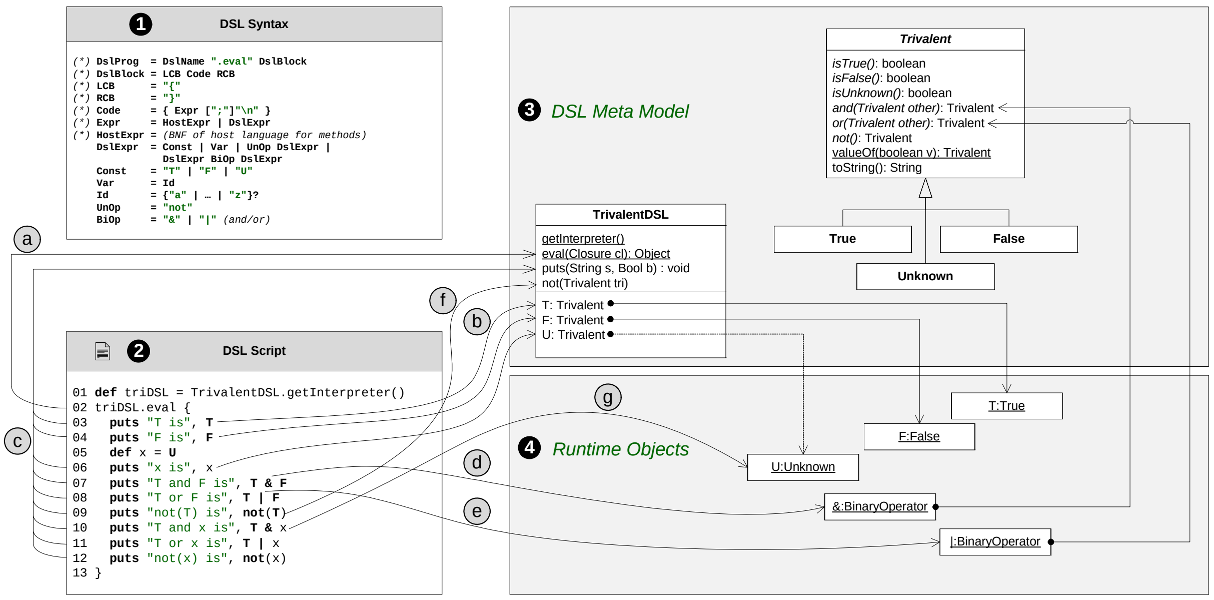
Implementing EDSL is a well-known technique in dynamic languages, such as Ruby [10][12] or Groovy [1]. For example, Groovy strongly utilized a family of DSLs in the Grails Web framework [1]. In these scripting languages, one can write domain-specific code in the host language syntax. Domain-specific literals and operators that are not defined in the host language are dynamically mapped to property accesses and method calls on an *interpreter instance*, which is at the heart of the EDSL implementation. This exploits the fact that code is evaluated right before executing it. One can write code that uses a *domain operation* not defined for the host language. Note that, there is no compiler that rejects the usage of such an operation because it is not yet defined. Instead, the operation is interpreted as a method call to an undefined method. The MOP particularly intercepts calls to *missing* methods. This gives the opportunity to redirect property accesses and operator calls to the interpreter instance that understands the domain operations and evaluates these operations according to domain semantics.

3.1 A Pattern for Building EDSLs

The design and implementation of EDSLs strongly depends on the host language, thus one needs a good development description at the finger tips. There are numberless informal sources that use EDSLs in object-oriented scripting languages [10][1][12], but to the best of our knowledge no EDSL pattern description is available that describes the implementation tasks in scripting languages from grammar to code. Thus, we describe the development tasks for the "EDSL pattern" as a set of instructions one can follow. This pattern applies to any host language that provides the following features: (1) o*bject-oriented constructs* that allow a modularized implementation of the domain logic, (2) c*ode blocks* or closures as first-class values to support nested structures in DSL code, (3) a *meta-object protocol* that allows intercepting pretended property accesses and method calls and that can be used to *delegate* method calls, and (4) a *flexible* and *non-intrusive syntax* that allows using undefined methods in code and that allows omitting unnecessary details for better designing the concrete syntax of the DSL. These requirements are met by Groovy but also by other object-oriented scripting languages such as Ruby.

The instructions for implementing an EDSL start directly from the DSL syntax, which is given in BNF form or as a list of keywords. In the recipe for the EDSL pattern, one follows the subsequent list of steps:

1. For each object type in the domain create a *domain class* T_k
   - Consider reusing an existing type from the host language or from a library.
   - Create attributes that describe the domain object.
   - Create getters and setters to access the attributes
   - Create a method for each operation on this type,
   - All domain classes form a *domain meta-model.*
2. Create an *interpreter class E* (for each EDSL/DSX).
   - The class implements a special interface with two methods: **`getInterpreter()`** and **`eval()`**. The method **`getInterpreter()`** instantiates a new DSL interpreter. The method **`eval()`** takes a closure as a parameter and returns an **`Object`**. The closure contains DSL statements that are evaluated and the resulting value is returned.
   - The interpreter class defines several domain literals and domain operations.
   - The interpreter extensively uses domain types from the domain meta-model to implement the semantics of the domain,
   - The interpreter may hold state, e.g., use a HashMap to simulate a heap of domain objects.
3. Add a property to *E* for each *domain literal* of the BNF. This forms a new literal keyword in the DSL.
   - The property is named after the keyword identifier used in the BNF.
   - The property has a type T_i.
4. Add a method to *E* for each *domain operation* of the BNF. This forms a new operation keyword in the DSL.
   - The method is named after the keyword identifier used in the BNF.
   - For an N-ary operator, the method takes N parameters.
   - The method returns a type T_j.
5. Add a method to *E* for each *nested structure* of the BNF.
   - A *nested structure* in the BNF is also defined in form of a method that takes a closure as the last parameter. This method is responsible to handle the code contained in the closure, e.g., it evaluates the closure immediately.



```
(*) DslProg  = DslName ".eval" DslBlock
(*) DslBlock = LCB Code RCB
(*) LCB      = "{"
(*) RCB      = "}"
(*) Code     = { Expr [";"]"\n" }
(*) Expr     = HostExpr | DslExpr
(*) HostExpr = (BNF of host language for methods)
    DslExpr  = Const | Var | UnOp DslExpr |
               DslExpr BiOp DslExpr
    Const    = "T" | "F" | "U"
    Var      = Id
    Id       = {"a" | … | "z"}?
    UnOp     = "not"
    BiOp     = "&" | "|" (and/or)
```

```
01 def triDSL = TrivalentDSL.getInterpreter()
02 triDSL.eval {
03   puts "T is", T
04   puts "F is", F
05   def x = U
06   puts "x is", x
07   puts "T and F is", T & F
08   puts "T or F is", T | F
09   puts "not(T) is", not(T)
10   puts "T and x is", T & x
11   puts "T or x is", T | x
12   puts "not(x) is", not(x)
13 }
```

Figure 1. Overview of the Execution of the embedded DSL Pattern in Groovy

The above steps can be adopted in case of special needs, however, the process may never violate a syntax rule of the host language, e.g., use a host language keyword as an identifier in the new DSL. In such cases, the concrete syntax of the DSL must be modified.

3.2 Implementing embedded DSLs in Groovy

In the following, we elaborate on how the instructions can be followed in Groovy. We exemplify the pattern by implementing a simple DSL for *trivalent logic,* which besides *true* and *false* has a third state *unknown*. Fig. 1 shows the different artifacts that are used to implement this EDSL. The syntax of the EDSL (index 1 in Fig. 1) is compatible with Groovy syntax.

In Fig. 1, at index 3, the domain meta-model for the trivalent DSL is given. A domain meta-model defines the *domain types* and an *interpreter class*. Domain types define the semantics of domain objects. They are modeled as ordinary classes in the host language. In this example, to follow step 1 of the recipe, we define an abstract class that represents the trivalent states. The methods of this abstract class **`Trivalent`** are implemented by each of the derivates **`True`**, **`False`**, and **`Unknown`**. The derivates encapsulate the logic for operations on types.

The **`TrivalentDSL`** interpreter, whose implementation is shown in Fig. 2, is used to evaluate DSL code and to handle domain objects. The class provides a method **`getInterpreter()`** (Fig. 2, line 3) to create a new instance for interpreting DSL code. With such an interpreter instance, the **`eval()`** method (line 8) is used to execute DSL code, which is passed to **`eval()`** as a **`Closure`**. The interpreter sets itself to be the *delegate* of the closure (lines 9-10)[1]. This way, the interpreter instance receives all operations during the execution of the DSL code (starting at line 11). Lines 15 to 17 define the domain literals of the trivalent DSL. Next, in lines 20-22 the **`not()`** domain operator is defined that simply forwards to the corresponding domain types. In lines 24-27, the domain operation **`puts()`**is defined. We do not have to implement the operators "**&**" and "**|**". Groovy automatically maps them to calls to **`and()`**, respectively **`or(),`** on the objects to which the operators are applied, thus they are directly mapped to the methods of the domain types. Therefore, the domain operations on types can be defined in the meta-model.

In the following, we use the small program given at index 2 of Fig. 1 to illustrate the dynamics of executing code written in the trivalent logic DSL. The objects of the domain meta-model classes that are instantiated when the DSL program at index 2 is executed are shown at index 4.

In the DSL program at index 2 line 1, an instance of the DSL interpreter class is retrieved. In the next line, we call **`eval()`** on the interpreter instance to evaluate the following DSL code block (lines 3-12). The trivalent DSL has three domain literals: **`T`** references an instance of **`True`** from the meta-model, **`F`** references **`False`**, and **`U`** references the state **`Unknown`**. The reader familiar with Groovy will notice that we reuse Groovy syntax and language features in DSL code. In line 5, an untyped variable **`x`** is defined and the trivalent **`U`** is assigned to **`x`**. Using Groovy variables in DSL code is possible, because domain types are modeled as ordinary Groovy classes. Further, we make usage of the fact that the operators "**&**" and "**|**" are mapped to **`and()`** and **`or()`** in Groovy. Operators return a reference to a domain object, hence we can continue working with result values in DSLs. In addition to this, the trivalent DSL defines two domain operations: the binary operator **`puts`**,which prints out a **`String`** (this type is reused from the host language) followed by a **`Trivalent`**. Also, the DSL defines the unary operator **`not()`** that takes a trivalent and returns its complement. We do not go into the details for lines 3 to 9, as more interesting are lines 10 to 12 where **`x`** is *unknown*. The semantics of *unknown* is that whenever the result of an operation depends on the *unknown* value, the result will be *unknown*. For this reason, the result of line 10 is **`U`**. In contrast, in case of line 11, the result does not depend on **`x`** because the other value is **`T`**, therefore "**|**", resp. **`or()`**, will return **`T`**. Again, in line 12 the complement depends on unknown, hence must be unknown.

When evaluating the example DSL code from Fig. 1 (index 2, lines 3-12), the domain operation **`puts()`** results in a pretended method call on the interpreter instance. Right before the call, its

[1] Recall: A delegate is an object that receives all property accesses and method calls the closure does not understand.

```
01 public class TrivalentDSL implements DSL {
02
03   static DSL getInterpreter() {
04     DSLCreator.getInterpreter(
05       new TrivalentDSL());
06   }
07
08   Object eval(Closure cl) {
09     cl.delegate = this;
10     cl.resolveStrategy = DELEGATE_FIRST;
11     return cl.call();
12   }
13
14   /* Literals */
15   Trivalent T = new True();
16   Trivalent F = new False();
17   Trivalent U = new Unknown();
18
19   /* Operations */
20   Trivalent not(Trivalent tri) {
21     return tri.not();
22   }
23
24   void puts(String str,
25             Trivalent tri) {
26     println "$str $tri";
27 }  }
```

Figure 2. DSL **i**nterpreter class for the trivalent logic.

parameters must be evaluated. Using the literal **`T`** as the second parameter results in a pretended property access on the interpreter instance that returns an instance of **`True`**, this is indicated by index b. Next, the call to **`puts()`** is executed with the evaluated parameter (index c). Line 4 is analog to line 3. In line 5, an untyped Groovy variable is defined that refers to an instance of **`Unknown;`** that value is printed out in line 6. In line 7, "**`T & F`**" is mapped by Groovy to a call "**`T.and(F)`**" (index d) because this operator is directly defined on **`Trivalent`** there is no need to define an operator method **`and()`** in the DSL. At runtime, **`T.and(F)`** with result in **`Trivalent`** value that again can be referenced and which can further be used in calculations. In the same way, "**`T | F`**" is mapped to "**`T.or(F)`**" (index e). In contrast, in line 9, "**`not(T)`**" is invoked on the interpreter instance (index f). The remaining lines are just repetitive.

3.3 Reasoning on the EDSL Approach

Using the EDSL approach to define extensions makes it is easy to later extend the DSL syntax: We inherit from an existing interpreter class and add new properties and methods for additional keywords. In particular, we do not need to create and maintain the code of *abstract syntax tree* nodes. By reusing the Groovy syntax, the interpreter instance can directly work on the types of the domain meta-model. Because we directly work with references, we also do not need to resolve identifiers. An important detail makes the EDSL approach more agile. Most interpreters and compilers are implemented with a large loop around a switch-case-block that entangles the logic of every expression type. Hence, whenever, we add a new literal or operator there is a missing case in this badly maintainable switch-block, thus the interpreter will break.

In the EDSL approach, this logic is separated into designated methods, which can easily be added or removed. Further, we can even refine the interpreter using inheritance and OO method dispatch will take care of finding the correct method implementation that handles an expression of a certain type.

```
// (A) SampleUseGroovy.aj
01 public aspect SampleUseGroovy {
02
03   before(): execution(* *.*(..))  {
04     groovy (thisJoinPoint) {
05       println "Hello $thisJoinPoint"
06 } } }

// (B) SampleUseDSL.aj
01 public aspect SampleUseDSL {
02
03   before(): execution(* *.main(..)) {
04     groovy () {
05       def triDSL;
06       triDSL=TrivalentDSL.getInterpreter()
07       triDSL.eval {
08         def expr = T & (F | U)
09         puts "Trivalent expression: ", expr
10 } } } }
```

Figure 3. Usage of (a) a Groovy code block, and (b) an EDSL.

4. Embeddeding DSLs into AspectJ Advice

Most popular AOP languages, in particular AspectJ, do not provide the necessary features to implement EDSLs following the pattern presented in Sec. 2. A solution would be to write a new aspect language from scratch that supports the EDSL pattern. However, we would have to implement features and tools available for general-purpose aspect languages.

We suggest a more pragmatic techniques to extend general purpose aspect languages with support for embedding domain-specific extensions (DSXs)) into advice: We have implemented a simple extension of AspectJ that allows implementing advice in Groovy; using EDSLs in pointcuts is part of our ongoing research, The possibility to use Groovy in advice blocks, brings in the necessary language features to embed DXSs into AspectJ advice.

Fig. 3 (a) shows a simple aspect that uses the proposed extension. Aspects and pointcuts are defined in AspectJ syntax. In contrast, the advice can be implemented in Groovy syntax. The advice body uses the new keyword **`groovy`** to directly use Groovy code for implementing the advice. In line 4, the **`groovy`** keyword indicates that a Groovy code block will follow. Next, all variables that should be accessible from Groovy are selected through a comma-separated list that is defined between the parenthesis; in this example, we only want to access **`thisJoinPoint`**. Finally, the Groovy code block is enclosed in curly brackets (lines 4 to 6). When the advice is executed at runtime, the extension will automatically bridge to advice execution in Groovy. The most important advantage in terms of flexibility comes from the Groovy language features that open us the possibility to embed DSLs. Still, our extension comes with further support that helps integrating with Groovy, e.g., the variables that are automatically made available to advice. The direct integration and the conventions how this language bridge is designed distinguish the extension from using the Java 6 scripting package.

Now that we understand how Groovy can be used in advice implementations, we can discuss how DXSs can be embedded into advice code. Fig. 3 (b) shows an aspect that uses the **`TrivalentDSL`** from Fig. 2. The **`SampleUseDSL`** aspect uses the **`groovy`** keyword to escape to Groovy syntax. In line 6, the advice obtains an interpreter instance of the trivalent DSL. In the following line, we pass a DSL code block to the **`eval()`** method of the interpreter. This code block allows directly writing DSL expressions into advice, such as **`T & (F | U)`** in line 8.

Note that, the embedded DSL is used as a domain specific extension (DSX) that is *dynamically linked* into the advice language

```
01 public aspect PolicyGeneratorAspect {
02
03   pointcut serviceSelection() :
04     execution(* Registry.find(..));
05
06   before() : serviceSelection() {
07     String policy = null;
08     groovy(policy) {
09       policyDSL = new PolicyDSL();
10
11       policyDSL.eval {
12         asserts = confidentiality &
13           integrity & token(SAML);
14         policy = convertToPolicy(asserts);
15       }
16     }
17     //access result in policy …
18 } }
```

Figure 4. EDSL for generating security policies in advice.

at runtime. The concrete interpretation is performed by the **`triDSL`** instance that receives all pretended property accesses and method calls that are triggered from the DSL code. We could easily change the semantics of DSL code, e.g., by replacing the **`TrivalentDSL`** with a refined version that interprets the *unknown* value by default as *false*. The usage of Groovy in advice alone and the simple example DSL probably will not convince the reader of the approach so far. For this reason, we provide a number of more sophisticated applications in the following.

5. Applications

The following example applications have been inspired from related work. They exemplify the possibilities to define similar abstractions for a DSAL by using our extension.

A DSAL for generating Security Policies. In previous work [7], we have proposed a domain-specific aspect language to enforce security policies in *composite applications*. One task in the enforcement was to generate an XML representation of the security requirements of a composite application in form of a WS-SecurityPolicy policy [3]. The generated policy is used in a dynamic *policy negotiation process* between the composite and the services that are called from the composite. When a composite application communicates with an external Web service, first a policy is generated that represents the requirements of the composite. Next, the generated policy is *matched* against the WS-SecurityPolicy policy attached to that Web service. Such a match determines whether the requirements of both the composite and the Web service can be fulfilled and whether communication can be established.

We have implemented a new DSX for generating WS-SecurityPolicy using the EDSL approach. The resulting DSAL could be used in the course of [7]. This would release security developers from processing XML to generate security policies. The DSX defines the keywords **`confidentiality`**, **`integrity`**, and **`token`** (authentication) to declare high-level security requirements, in terms of WS-SecurityPolicy: *security assertions*. **`confidentiality`**, **`integrity`** are domain literals in the policy DSL. In contrast to the other keywords, **`token`** is defined as a unary domain operation that is parameterized with a *token type*, e.g. the parameter **`SAML`** selects a certain token format to be used. The aspect defined in Fig. 4 uses this DSL to generate the policy XML artifact. In DSL code, different assertions can conveniently be combined using the operators "**`&`**" (and) and "**`|`**" (or). Finally, the policy can be converted to a string representation using the domain operation **`convertToPolicy()`**, resulting in the following XML snippet:

```
<Policy><ExactlyOnce><All>
  <Confidentiality>...</Confidentiality>
  <Integrity>...</Integrity>
  <SecurityTokenRefe-
rence>...<saml>...</saml>...
  </SecurityTokenReference>
</All></ExactlyOnce></Policy>
```

Note that, the actual specification of the security requirements boils down to the high-lighted lines 12 and 13, and that there are no XML processing details in the code.

For this DSL, the policy meta-model defines classes for each specific **`Assertion`** type: **`ConfidentialityAssertion`**, **`IntegrityAssertion`**, **`TokenAssertion`**, and so on. This domain types are then used by the policy interpreter. The EDSL implementation **`PolicyDSL`** defines the properties **`confidentiality`**, **`integrity`**, and **`SAML`** and the methods **`token()`** and **`convertToPolicy()`**. When executing the DSL block, the properties defined for assertion keywords reference corresponding instances of the policy meta-model, e.g., for **`confidentiality`**, an instance of the class **`ConfidentialityAssertion`** is referenced. The meta-model classes of assertions can be combined because they overwrite the operator methods **`and()`** and **`or()`**. When combining assertions, a special *combinator assertion* is returned according to the WS-Policy framework [1], i.e., **`and()`** returns an **`AllAssertion`** for which all contained assertions must be fulfilled, similarly, **`or()`** returns an **`ExactlyOnceAssertion`**. The embedded DSL for policies and the integration of its domain meta-model allows conveniently constructing an XML structure directly in an advice without dealing with technical details and by hiding the details of XML representations for each assertion type.

Composition of two DSXs. In practice it is often the case that problem domains overlap, e.g., the assertions of the PolicyDSL can be used as elements in the domain of the *set theory*. In such a case, instead of building a new DSX that combines the two domains, it is desirable to reuse the two existing DSXs for policies and sets in order to combine them into one. To show how several DSXs can be combined in our approach, we have implemented another reusable DSX that allows performing set operations from *set theory* on **`Collection`** types, called **`SetsDSL`**. Among other things, this DSL defines the operations **`union()`**, **`intersect()`**, **`difference()`**, and **`powerSet()`** to construct new sets and subsets. We have defined additional functions that perform operations on elements of sets: **`conjunction()`** uses "**`&`**" to conjunct all elements, **`disjunction()`** uses "**`|`**", and **`convertToDNF()`** converts a set of subsets to *disjunctive normal form*.

```
01 before() : serviceSelection() {
02   String policy = null;
03   groovy(policy) {
04     policyDSL = new PolicyDSL();
05     listSetsDSL = new ListSetsDSL();
06     combinedDSL = DSLCreator.
07         getCombinedInterpreter(
08         policyDSL,listSetsDSL);
09
10     combinedDSL.eval {
11       choices = [confidentiality,
12                  integrity, token(SAML)];
13       allSubsets = powerSet(choices);
14       allCombis = convertToDNF(allSubsets);
15       policy = convertToPolicy(allCombis)
16     }
17   }
18   //access result in policy …
19 }
```

Figure 5. Composition of DSXs in advice.

The combined logic of both DSXs – the PolicyDSL and SetsDSL – can be efficiently used to construct a policy that allows several policy alternatives. Therefore, we would like to use the types from the policy domain, namely assertions, in the domain of sets, so that assertions are simply handled as elements of sets. A good example in the domain of [7] is generating a policy that allows all possible assertion combinations for a set of selected assertions, which represents the available security features for one partner. The code in Fig. 5 exemplifies how a composition of DSXs can be used in order to quickly derive such a policy that accepts all possible assertion combinations. In line 4 and 5, the two interpreters of both DSLs are retrieved. In line 6-8, we then combine the two interpreters using the **`DSLCreator`**. An interpreter instance is created that in particular understands the keywords of both DSXs. Next, we can use this combined interpreter to evaluate code that use operations from both domains. In lines 11 to 12, we construct a set of assertions. In line 13, we calculate all possible subsets of this set. In line 14, we create all possible assertion combinations from the set by building the disjunctive normal form of the in **`allSubsets`** contained subsets and elements, and finally in line 15 the generated policy is made accessible to the AspectJ advice parts by assigning it to the Groovy code block parameter **`policy`** from line 3. Note that, the operation **`convertToDNF()`** internally uses the **`conjunction()`** and **`disjunction()`** operations defined in **`SetsDSL`**, which apply "**&**" and "**|**" to the elements. The most important fact at this point is that the *ad-hoc polymorphism* of Groovy, maps the operations **`and()`** and **`or()`** to the implementations defined in the **`PolicyDSL`**. Therefore, **`conjunction()`** and **`disjunction()`** can be applied to arbitrary objects that define **`and()`** and **`or()`**. This detail allows semantic and reusable compositions of different DSX in our approach to be easily achieve at virtually no additional costs.

Control Flow Abstractions in DSALs. Control flow abstractions can reduce the complexity of aspects by providing constructs to set up a predefined control flow. A sophisticated example for this kind of abstraction in DSALs is presented in [14], in which particularly an approach is proposed that helps solving the problem of *tangled aspect code*. That is fragments of aspect code corresponding to a "sub-concern" that is tangled with code of other "sub-concerns" in advice code. The KALA language [14] uses "begin", "commit" and "abort" blocks to separately declare transactional properties for each sub-concern in advanced transaction management. Later, the KALA weaver composes all sub-concerns and weaves them the resulting transactional demarcation into a transaction's functional code.

```
01 groovy(thisJoinPoint) {
02   def atxDSL = AtxDSL.
03     getInterpreter(thisJoinPoint);
04
05   atxDSL.eval {
06     begin {
07       dep(ConcernA, ad, self);
08       dep(ConcernA, wd, self);
09       dep(ConcernA, scd, self);
10     }
11     commit { terminate(self); }
12     abort { terminate(self);
13 } } }
```

Figure 6. Advanced Transaction Sub-Concern.

Our solution can also be used to define similar domain-specific control flow abstractions as in KALA. For demonstration purposes we have implemented a simplified but similar realization of the "begin", "commit", and "abort" blocks from KALA. Because we are only interested in the language abstractions KALA provides, we neither fully re-implement all features in KALA nor realized a transaction monitor. When using our approach sub-concerns can be defined using ordinary aspects. Fig. 6 shows an advice excerpt that is executed before a *transactional method* so that transactional properties for the executed method are defined. In lines 2-3, an EDSL for the advanced transaction models is instantiated that defines the KALA keywords: **`begin`**, **`commit`**, and **`abort`**, which define control flow abstractions. Further, the EDSL defines the domain literals and operations for the other KALA statements that in particular define transactional properties (dep, alias, terminate, and so on). Note that, we initialize the interpreter with context information about **`thisJoinPoint`**, so that defined properties can be related back to the currently executed method. The keywords such as **`begin`** are implemented as domain operations that take a closure as the only parameter. The closure is then stored in a list associated with the join point. Note that, other sub-concerns are defined in similar aspects that declare different transactional properties. In our approach, e.g., before actually executing the transactional method, a special aspect collects all "begin" closures from the list that is associated with the current join point. The aspect executes the collected closure in order, which results in the necessary demarcation code.

6. Implementation

We have implemented the prototype as a *source code transformer* that transforms aspects with **`groovy`** statements to code compilable by existing tools. Before compilation, the transformer produces two kinds of outputs for each DSAL aspect file. On the one hand, for each Groovy block a Groovy *advice script* is produced that contains the advice code. On the other hand, for each DSAL aspect, a corresponding aspect in *pure* AspectJ is written, which contains *hooks* that replace the Groovy code blocks and that bridge to the Groovy advice script at runtime. When compiling, AspectJ aspects are woven into the program using the AspectJ compiler. At runtime, the Groovy advice scripts are executed using the Groovy engine.

In Fig. 7, we show the transformed aspect from Fig. 3 (b). For transformed aspects, pointcuts declaration stays the same. Only the advice code blocks in which Groovy advice is implemented are replaced. In Fig. 7 line 4, first a HashMap is created that contains key-value pairs for each parameter to be made available to the Groovy advice. The **`GroovyShell`** and **`Script`** in lines 5 to 8 are necessary to execute the Groovy advice code. In line 9, the advice file is executed as a Groovy script. The advice script returns a **`GroovyObject`** that holds a closure with the previously extracted advice code. Line 10 invokes **`run(context)`**, what

```
public aspect SimpleAspect {
  ...
  before() : pcexpr() {
    //setup context HashMap with parameters
    GroovyShell gshell = new GroovyShell();
    try {
      Script script = gshell.parse(
        new File("<advicescript>.groovy"));
      GroovyObject a = script.run();
      a.invokeMethod("run",context);
    } catch (Exception e) { ... } } }
```

Figure 7. AspectJ-to-Groovy Language Bridge (in AspectJ).

will finally call the advice closure containing the EDSL code. When evaluating the advice closure, an access to a non-local variable, such as **`thisJoinPoint`** in Fig. 6 line3, is treated as an access to an advice variable in the context. Such an access results in a pretended property access on the **`context HashMap`**. The map then uses the variable identifier as the key to lookup if there is a variable defined and returns the corresponding value.

Note that the hooks for around advice are slightly different than that for before or after advice. The AspectJ **`proceed()`** statement is wrapped in an *anonymous inner class* instance that is stored in the context HashMap, before bridging to Groovy. Because the inner class instance is stored in the map with the key "proceed", Groovy advice then can *proceed* join points simply by invoking **`proceed.call(<params>)`**. Another difference in the hooks for around and after returning advice is that they must take care of possible returning parameters.

Composition of several DSXs that are provided as EDSLs is possible using the designated **`DSLCreator`** tool provided in our approach. With this, one can instantiate a combined interpreter by invoking the **`getCombinedInterpreter()`** method and passing two EDSL interpreter instances to it. This method returns a *combined interpreter instance*. Using the MOP, the combined interpreter intercepts all properties accesses and method calls that are made in DSL code and automatically dispatch them to one of the combined EDSLs instances. Internally the combined interpreter holds a list of DSL interpreters to be combined. When receiving a property access or method call, the combined interpreter looks up whether the property or method is defined in one of the interpreters in the ordered list. If that is the case, it dispatches the access or call to that instance.

7. Conclusion

In this paper, we have presented an extension for AspectJ to support embedding DSLs into advice code. The approach follows a set of instructions to implement a domain-specific extension that can be dynamically linked to use DSL code in advice. Extensions once implemented can be refined and composed. The simplicity with which new domain-specific advice languages can be created supports idea to embed DSL into aspect language. However, the current prototype can only embed DSLs into advice. It can only build limited domain abstractions in pointcuts using the *abstract pointcuts* of AspectJ. The prototype cannot be used to create domain-specific aspect languages that need sophisticated abstractions in the pointcut language. This limitation is mainly because we are using AspectJ for the static weaving of aspects. Therefore, our ongoing research focuses on how to use embedded DSLs in pointcut languages.

Acknowledgments

We would like to thank Christian Hofer and Karl Klose for the valuable comments and discussions.

References

[1] AspectJ homepage. http://www.eclipse.org/aspectj/.

[2] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. Griswold. An overview of AspectJ. In *Proceedings of the ECOOP 2001*, Budapest, Hungary, 2001.

[3] WS-SecurityPolicy, http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ws-securitypolicy-1.2-spec-cs.pdf.

[4] Groovy Homepage, http://groovy.codehaus.org/.

[5] D. König. Groovy in Action. Manning Publications, New York, Januar, 2007.

[6] G. Laforge and J. Wilson. Tutorial: Domain-Specific Languages in Groovy. QCon 2007 Conference, London, 2007. http://glaforge.free.fr/groovy/QCon-Tutorial-Groovy-DSL-2-colour.pdf

[7] T. Dinkelaker, A. Johnstone, Y. Karabulut and I. Nassi. Secure Scripting Based Composite Application Development: Framework, Architecture and Implementation. In *CollaborateCOM 2007*, White Plains, NY, October 2007.

[8] P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv. 28*, December 1996.

[9] P. Hudak. Modular Domain Specific Languages and Tools. In *Fifth International Conference on Software Reuse ICSR'98*, 1998.

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

[11] Ruby homepage. http://www.ruby-lang.org/

[12] J. Cuadrado and J. Molina. Building Domain-Specific Languages for Model-Driven Development. In *Software, IEEE*, Vol. 24 , No. 5, p. 48-55, September 2007.

[13] P. Avgustinov, A. Christensen, L. Hendren, S. Kuzins, J. Lhotak, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam and J. Tibble. abc : An extensible AspectJ compiler. In *Transactions on Aspect-Oriented Software Development I*, Vol. 3880 of LNCS, Springer, Berlin, Heidelberg, Februar 2006.

[14] J. Fabry. Modularizing Advanced Transaction Management - Tackling Tangled Aspect Code. *PhD thesis*, Vrije Universiteit Brussel, 2005.

[15] J. Fabry, E. Tanter and T. D'Hondt. ReLAx: Implementing KALA over the Reflex AOP Kernel. In *Workshop DSAL'07*, Vancouver, Britisch Columbia, Canada, March 2007.

[16] C. Lopes and T. Ngo. The Aspect Markup Language and its support of Aspect Plugins. *ISR Technical Report UCI-ISR-04-8*, University of California, Irvine, 2004.

[17] C. Lopes. D: A Language Framework For Distributed Programming. PhD thesis, College of Computer Science of Northeastern University, 1997.

[18] A. Bagge and K. Kalleberg. DSAL= library+notation: Program In *First Domain-Specific Aspect Languages Workshop*, 2006.