Definition of an Aspect-Oriented DSL using a Dynamic Language

Mark Strembeck, Uwe Zdun
Institute of Information Systems, New Media Lab
Vienna University of Economics, Austria
{mark.strembeck|uwe.zdun}@wu-wien.ac.at

ABSTRACT

We present an approach to define an aspect-oriented DSL using a dynamic language. In particular, we describe an extensible aspect-oriented DSL for role-based access control and its implementation. Furthermore, we show how a dynamic pointcut language can be used to compose the different elements of our DSL. We implemented our approach using the XOTcl scripting language. The general approach, however, can be realized using any other dynamic language as well.

1. INTRODUCTION

Domain-specific languages (DSL) are "small" languages that are particularly expressive in a certain problem domain. Recently, in the area of model-driven software development and related research areas (see for instance [11, 4, 1, 3]), DSLs are used as languages which represent the abstractions familiar to domain experts (so-called domain modeling languages). A DSL may either have a textual or graphical representation. This concrete syntax of the DSL is mapped to an abstract syntax which is defined by the formal language model. In the model-driven approach, the semantics of the DSL are defined using model transformation and code generation. That is, a generator translates the DSL into an executable representation, according to the models and meta-models, and the semantics of the model elements.

Sometimes DSLs define semantics that are aspect-oriented in their nature. Consider, for instance, a DSL for defining role-based access control policies[1]. In this context, an access control subject has a number of roles that are assigned to the subject. Moreover, permissions are assigned to roles, and permissions can be associated with context constraints (see [13]). This basic model is shown in Figure 1. Each of the elements in the model is represented via classes or a class hierarchy, and the definition of individual elements is independent of the other classes and hierarchies. It is, however, not trivial to achieve this goal since the context constraint concerns cross-cut the permission concerns, which again cross-cut role concerns, and the roles cross-cut concerns in the subjects. To avoid tangled code in the DSL definition and in the code written in the DSL, we introduce an aspect-oriented definition of the role-based access control DSL. From an aspect-oriented point of view, roles can be interpreted as aspects of subjects, permissions as aspects of roles, and context constraints as aspects of permissions.

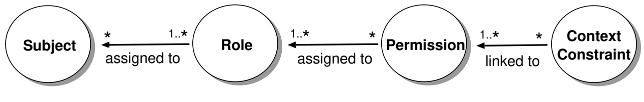

Figure 1: High-level model for role-based access control rights

In this paper, we want to explore the combination of the DSL concept and aspect-oriented programming concepts. We believe the requirements of the role-based access control DSL are quite typical for an aspect-oriented DSL. Unfortunately, these requirements are hard to implement with many existing AOP frameworks:

- The AOP framework must be able to depict "aspects of aspects". In the access control example, and many other examples, the aspects are related to other aspects which must be reflected by the AOP framework because the DSL user must be able to define and control how the aspects interact.
- The pointcut language of the AOP framework must be extensible, so that we can define new domain-specific pointcuts, which can be exposed to the DSL.
- The AOP framework and its programming language must allow for the extension with new elements or to define new DSL language elements, i.e. their syntax and semantics. Moreover, the AOP framework should be able to directly generate an executable representation from a specification written in the DSL.
- In the access control example, and many other examples, the aspects must be dynamic. A recompilation for changing roles, permissions, or constraints is not feasible.

To meet these requirements we especially need an open aspect language that is dynamic and able to handle "aspects of aspects". In this paper, we will use the scripting language XOTcl [8] as an open aspect language. Please note that we use XOTcl just for exploration of the concepts. The general concept of an aspect-oriented DSL is not depending on this specific language. In particular, we will describe the aspect-oriented DSL that provides the same functionality as the xoRBAC tool [5, 13]. Our aspect-oriented DSL for RBAC, however, separates the different concerns in a more efficient way and thus results in a more comprehensible and better maintainable implementation that allows for a straightforward evolution of xoRBAC.

2. EXTENDED OBJECT TCL (XOTCL) AS AN OPEN ASPECT LANGUAGE

Before showing how to realize an aspect-oriented DSL, we briefly explain how the scripting language XOTcl [8, 14] can be used as an open, dynamic aspect language. Like most scripting languages, XOTcl can be extended with new language elements. Thus, it is a good starting point to rapidly define a DSL. In addition, XOTcl supports the dynamic composition of aspects. That is, the XOTcl interpreter receives symbolic invocations that are indirected to the actual implementations of all objects in the system. The interpreter can dynamically intercept any message in the call flow when it is dispatched. At this point, the aspect are applied.

The idea of applying aspects as dynamic message interceptors on top of a (given) interpreter architecture is quite simple: we specify

[1] This is used as a running example in this paper.

all calls that are in focus of an aspect as criteria for the message interceptor, and let the interpreter execute this message interceptor every time such messages are called. In this way we can implement any aspect that relies on message exchanges. To receive the necessary information for dealing with the invocations, the message interceptor should be able to obtain the message context to find out which method was called on which object (the callee). Often the calling object and the respective method are required as well. Introspection options are used to obtain structure information via reflection.

XOTcl provides *mixin classes* [6] as a dynamic message interceptor implementation. In XOTcl any "ordinary" class can be registered as a mixin. The predefined `instmixin`[2] method accepts a list of classes to be registered as per-class mixins, whereas the predefined `mixin` method registers classes as per-object mixins.

XOTcl mixins may be dynamically added and removed at any time. To keep track of these dynamic relationships, `info instmixin` and `info mixin` provide introspection functions for mixins. Thus, at runtime one can always determine the current mixins of an object or class (see also [8, 14]).

For instance, consider the following XOTcl code (corresponding to one of the introductory AspectJ examples):

```
Class Point
...
Class PointAssertions
PointAssertions instproc assertX x {
 if {$x <= 100 && $x >= 0} {return 0}
 return 1
}
PointAssertions instproc setX x {
 if {[my assertX $x]} {
   puts "Illegal value for x"
 } else {
   next
 }
}
Point instmixin add PointAssertions
```

At first, the corresponding code for the class and the aspect (here also implemented as a class) is defined. Then we dynamically register one of these classes as an instance mixin (a class-based message interceptor) for all points; thus all calls to the method `setX` are intercepted by the `PointAssertion` mixin's same-named method `setX`.

There are two common ways to ensure the non-invasiveness of aspects (i.e. the obliviousness property in the terminology of Filman and Friedman [2]) when using mixins:

- Mixins can be applied to a superclass or interface, and are automatically applied to all subclasses in the class hierarchy. Thus, developers of subclasses can be oblivious to the aspect.
- A mixin can be registered for a set of classes using introspection options (aka reflection). For instance, one can apply a mixin for all class names starting with `Point*`. This way mixins can be applied in a non-invasive way for any kind of criteria (pointcuts) that can be specified using the dynamic introspection options of XOTcl.

The first variant was demonstrated in the previous code example. An example for the second variant is shown in the code below. We use introspection options to get all classes defined in the system and check whether they match `Point*`.

```
# Pointcut definition based on introspection
foreach p [Object getAllSubclasses] {
  if {[string match $p ::Point*]} {
    # Mixin registrations for weaving the mixin aspect
    $p instmixin add PointAssertion
  }
}
```

[2]Please note that "instmixin" is a short form of "instance mixin", meaning that a corresponding mixin is applied for all instances of the class the mixin was registered for. XOTcl uses a similar naming convention for methods: a method applied to instances of a class is called "instproc" which is a short form of "instance procedure".

The instruction `next` is responsible for forwarding the invocation. It thus handles (non-invasive) ordering of the message interceptors in a chain. Thus, the placement of the `next` instruction enables us to implement before, after, or around behavior of the message interceptor.

Besides mixin classes, XOTcl provides another message interceptor, called the *filter*. In contrast to mixin classes which only intercept specific methods, a filter can automatically intercept any invocation sent to an object, class, or class hierarchy. Filters are described in detail in [7].

In contrast to AspectJ, we do not have to "introduce" the method `assertX` on `Point` (in the example above) using an intertype declaration, as the mixin shares its object identity with the class or object it extends. However, in other cases we might want to change the class structure. In XOTcl a new method can be defined at any time (because all XOTcl structures are fully dynamic). Such dynamics require introspection options to ensure that we do not violate some architectural constraints by re-structuring the architecture. For instance, in the example above we can first perform a runtime check that there is no method `assertX` defined for `Point` yet, before we introduce it:

```
if {[Point info instprocs assertX] == ""} {
 Point instproc assertX x {
   if {$x <= 100 && $x >= 0} {return 0}
   return 1
 }
}
```

3. TRANSITIVE MIXINS IN XOTCL

In XOTcl, "aspects of aspects" can be modeled using transitive mixins. For example, consider a situation in which a class `PCM_2` is used as a per-class mixin, and we want to define an aspect for this mixin class. The aspect is implemented in a class `TMix_1`[3]. Consider further that is aspect `TMix_1` itself should have another aspect `TMix_2` (see Figure 2). The original composition of mixins should stay unaffected by the addional aspects.

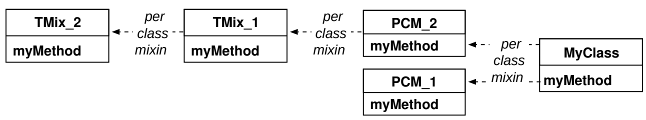

Figure 2: Example of transitive per-class mixins

In XOTcl this is solved by adding the corresponding per-class mixins to the method resolution order of the affected mixin. After weaving the mixins as aspects, the configuration in Figure 2 is generated. This configuration means that all per-class mixins of the mixin itself (and their superclasses) are searched before the method resolution order proceeds to the next mixin, resulting in chain of mixins that is visited in a transitive fashion (see also Figure 2). This scheme is applied recursively, because mixins might themselves have per-class mixins, which again might have per-class mixins, and so on.

4. ROLE-BASED ACCESS CONTROL DSL

The foundation of our aspect-oriented DSL for role-based access control (RBAC) consists of subjects, roles, permissions, and context constraints. Each of these basic elements is implemented via an own class that defines the specific functions of the corresponding concept. Some of these classes can be used as the root of a complex class hierarchy. These classes represent orthogonal concerns that we like

[3]Note that we use "TMix" as an abbreviation for "transitive mixin" in this example.

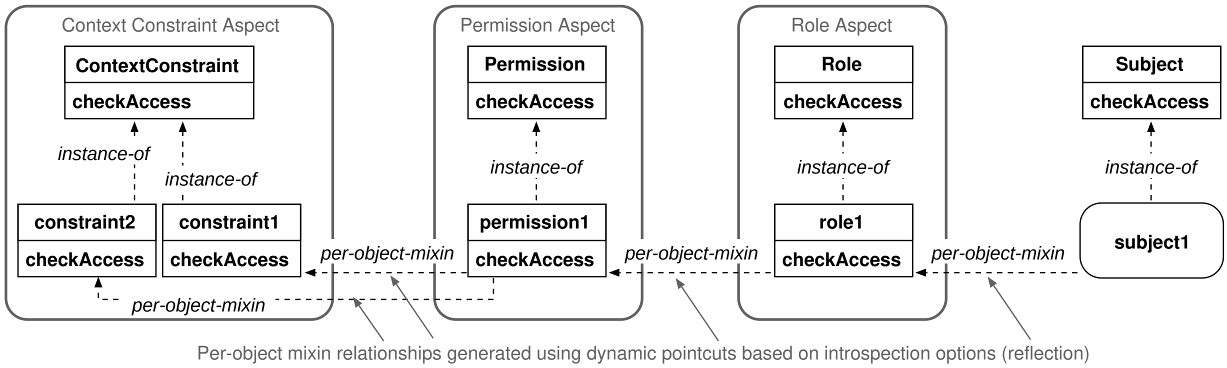

Figure 3: Example of an executable model generated form the DSL

to define independently from each other, and compose as aspects of each other (as outlined in the example in Section 1).

To implement an aspect-oriented DSL, we need to define a domain-specific aspect weaver, which is capable to weave aspects according to domain-specific constraints, and which realizes a pointcut language offering domain abstractions. For this task, we define the new class `RBACAspectWeaver`. Below we show an excerpt of the methods ("instprocs") of this class, which are mapped to DSL instructions:

```
Class RBACAspectWeaver
RBACAspectWeaver instproc roleSubjectAssign {r s}
RBACAspectWeaver instproc roleSubjectRevoke {r s}
RBACAspectWeaver instproc permRoleAssign {p r}
RBACAspectWeaver instproc permRoleRevoke {p r}
RBACAspectWeaver instproc linkCtxConstraintToPerm {cc p}
RBACAspectWeaver instproc unlinkCtxConstraintFromPerm {cc p}
...
RBACAspectWeaver instproc allSubjectInstances {}
RBACAspectWeaver instproc allRoleInstances {}
RBACAspectWeaver instproc allPermissionInstances {}
RBACAspectWeaver instproc allContextConstraintInstances {}
...
RBACAspectWeaver instproc allSubjectsOwningRole {r}
RBACAspectWeaver instproc allRolesAssignedToSubject {s}
...
RBACAspectWeaver instproc checkAccess {s op ob}
...
```

Our RBAC DSL weaver provides functions for weaving role to subject assignment and revocation (`roleSubjectAssign` and `roleSubjectRevoke`), as well as corresponding weaving functions for permission to role assignment and revocation, and for linking and unlinking permissions and context constraints. Moreover, it offers different introspection options that allow to define domain-specific, dynamic pointcuts on all instances of the basic DSL elements (`allSubjectInstances`, `allRoleInstances` etc.), as well as on specific DSL elements (e.g. `allSubjectsOwningRole`). The `checkAccess` function is applied to define pointcuts that check if a certain access can be granted or must be denied, i.e. if a certain subject `s` is allowed to perform operation `op` on object `ob`.

An example of a domain-specific weaving function is `roleSubjectAssign`:

```
RBACAspectWeaver instproc roleSubjectAssign {r s} {
  if {[my existSubject $s]} {
    if {[my existRole $r]} {
      if {[my ssdConstraintsAllowRSA $r $s]} {
        if {[my roleMaxSubjectCardinalityAllow $r]} {
          $s mixin add $r
          return 1
        } else { return 0 }
      } else { return 0 }
    } else { return 0 }
  } else { return 0 }
}
```

In the formal definition of this weaving function, first, we have to make sure that the respective role and subject exist (calls of `existSubject` and `existRole`). Subsequently, we check if the assignment of this particular role to this particular subject can be granted with respect to the static separation of duty constraints on roles that are in effect at this very moment (for details see [12]). If so, we further check that the maximum subject cardinality defined on the role is not yet reached. In case all checks are passed, we register the role as a new mixin for our subject.

Next, we give an example of an unweaving function, namely the revocation of a permission from a role. Again, we first have to assure that the corresponding role and permission exist. Then, we check that the minimum owner cardinality for this particular permission is not violated if we revoke the permission. Finally, we can delete the permission from the mixin list of the respective role (see source code below).

```
RBACAspectWeaver instproc permRoleRevoke {p r} {
  if {[my existRole $r]} {
    if {[my existPermission $p]} {
      if {[my permMinOwnerCardinalityAllow $p]} {
        $r mixin delete $p
        return 1
        }
      } else { return 0 }
    } else { return 0 }
  } else { return 0 }
}
```

In addition to the weaving and unweaving functions, used for assigments and revocations, our aspect-oriented DSL for RBAC offers various introspection functions that are used as elements of pointcuts in the DSL. Below we exemplary describe the `allSubjectsOwningRole` function which uses XOTcl reflection options to determine all subjects that own a given role. After checking if the respective role exists, the function checks for each subject if this particular role is assigned to the subject, i.e. if the role is registered as a mixin on the corresponding subject. All subjects owning the role are written to a list which is returned as the function result.

```
RBACAspectWeaver instproc allSubjectsOwningRole {r} {
  if {[my existRole $r]} {
    foreach s [my allSubjectInstances] {
      if {[$s ismixin $r]} {
        lappend roleOwners $s
      }
    }
    if {[info exists roleOwners]} {
      return $roleOwners
    } else { return "" }
  } else { return "" }
}
```

Now that we have defined our DSL's weaving functions and pointcut elements, we can use XOTcl as a dynamic pointcut language to define domain-specific pointcuts based on the different introspection options (using XOTcl reflection). Below we show two example pointcuts. The first pointcut matches all permissions starting with an "A" and links the context constraint `cc_A` to each of these permissions. The second pointcut matches all roles of type `StudentRole` and assigns the permission `get_exam` to each of these roles.

```
# Instantiate a domain-specific weaver
RBACAspectWeaver aw

# Instantiate two aspects
aw createPermission get_exam
aw createContextConstraint cc_A

# Pointcut definition
foreach p [$aw allPermissionInstances] {
  if {[string match $p ::A*]} {
    # Use the domain-specific weaving function
    # to weave the advice (implemented as a mixin)
    aw linkContextConstraintToPerm cc_A $p
  }
}

# Pointcut definition
foreach r [$aw allRoleInstances] {
  if {[$r isType StudentRole]} {
    # Use the domain-specific weaving function
    # to weave the advice (implemented as a mixin)
    aw permRoleAssign get_exam $r
  }
}
```

Figure 3 depicts a composed class model (i.e. an executable model in XOTcl generated from the DSL). In particular, the role `role1` is assigned to a subject `subject1`. Again, there is a permission assigned to `role1`, and the permission is linked to two context constraints `constraint1` and `constraint2`. Each of these assignment relations is realized through an XOTcl mixin relation. In this way, we are not only able to define aspects on objects and classes, but also to define aspects on aspects. This specification of aspects on aspects is realized via transitive mixins as outlined in Section 3.

The user of the DSL only uses the domain-specific pointcuts and weaving functions to compose the aspects. That is, the user only sees the domain-oriented view, not the technical details of the mixin and introspection model used internally. The weaver automatically realizes a mixin chain from these definitions.

5. RELATED WORK

JAC [10] provides a way to define DSLs for configuring aspects. Like many other application server AOP frameworks, JAC makes use of metadata configurations. In JAC the metadata language can be extended by the user: operations of the aspect component can be provided as Command implementations and invoked from the configuration file. This way each aspect can define its own configuration language. For instance, JAC predefines an authentication aspect component which offers domain-specific functions like `addTrustedUser` to configure the aspect. In [9] Zhang et al. describe how they extended their role slice approach to support something they call dynamic permissions. These dynamic permissions consider certain runtime information, esp. the state of related class instances, when making an access decision. However, they do not use an aspect-oriented RBAC DSL to define access control policies nor do they use a dynamic pointcut language.

6. CONCLUSION

In this paper, we presented an aspect-oriented DSL for role-based access control that provides all functions of the xoRBAC component. However, in comparison to the xoRBAC component our DSL is aspect-oriented in nature and offers a strict separation of concerns between the basic language elements of the DSL (especially subjects, roles, permissions, and context constraints). Moreover, we used XOTcl as a dynamic pointcut language to weave the different aspects.

Our approach allows for a straightforward evolution of the DSL and all of its language features. The approach is not limited to the domain of role-based access control, of course. In principle, it is applicable to arbitrary application domains where we first define a domain-specific language which is then mapped to a concrete implementation, e.g. an XOTcl implementation. Subsequently, we use a dynamic pointcut language (for example XOTcl including its rich introspection/reflection features) to compose the different elements. Note that the XOTcl language was used primarily for demonstration purposes and that the general approach can of course be realized with other dynamic languages.

7. REFERENCES

[1] S. Dmitriev. Language oriented programming: The next programming paradigm. Onboard Magazine, http://www.onboard.jetbrains.com/is1/articles/04/10/lop/index.html, October 2004.

[2] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *OOPSLA Workshop on Advanced Separation of Concerns*, Minneapolis, USA, October 2000.

[3] M. Fowler. Language workbenches: The killer-app for domain specific languages? http://www.martinfowler.com/articles/languageWorkbench.html, June 2005.

[4] J. Greenfield and K. Short. *Software Factories: Assembling Applications with Patterns, Frameworks, Models & Tools*. J. Wiley and Sons Ltd., 2004.

[5] G. Neumann and M. Strembeck. Design and Implementation of a Flexible RBAC-Service in an Object-Oriented Scripting Language. In *Proc. of the 8th ACM Conference on Computer and Communications Security (CCS)*, November 2001.

[6] G. Neumann and U. Zdun. Enhancing object-based system composition through per-object mixins. In *Proceedings of Asia-Pacific Software Engineering Conference (APSEC)*, Takamatsu, Japan, December 1999.

[7] G. Neumann and U. Zdun. Filters as a language support for design patterns in object-oriented scripting languages. In *Proceedings of COOTS'99, 5th Conference on Object-Oriented Technologies and Systems*, pages 1–14, San Diego, California, USA, May 1999.

[8] G. Neumann and U. Zdun. XOTcl, an object-oriented scripting language. In *Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference*, Austin, Texas, USA, February 2000.

[9] J. Pavlich-Mariscal, L. Michel, and S. Demurjian. Role Slices and Runtime Permissions: Improving an AOP-based Access Control Schema . In *Proc. of the International Workshop on Aspect-Oriented Modeling*, October 2005.

[10] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: a flexible framework for AOP in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, Kyoto, Japan, Sep 2001.

[11] T. Stahl and M. Voelter. *Modellgetriebene Software Entwicklung*. D.Punkt, 2005.

[12] M. Strembeck. Conflict Checking of Separation of Duty Constraints in RBAC - Implementation Experiences. In *Proc. of the Conference on Software Engineering (SE 2004)*, February 2004.

[13] M. Strembeck and G. Neumann. An Integrated Approach to Engineer and Enforce Context Constraints in RBAC Environments. *ACM Transactions on Information and System Security (TISSEC)*, 7(3), August 2004.

[14] Extended Object Tcl (XOTcl) Homepage. http://www.xotcl.org, 2006.