JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 7, No. 6, July–August 2008

Modeling Interdependent Concern Behavior Using Extended Activity Models

Mark Strembeck, New Media Lab, Institute of Information Systems, Vienna University of Economics and BA
Uwe Zdun, Distributed Systems Group, Information Systems Institute, Vienna University of Technology

Software engineering considers many assets relevant for developing a software system, ranging from requirements to source code. In this context, a concern is a particular goal, concept, or area of interest that needs to be considered throughout a number of these assets. Even though the concerns in a software system usually have many interdependencies among each other, specifying the interdependent behavior of concerns is not a focus of today's (concern) modeling approaches. In this paper, we present an approach to model interdependent concern behavior using extended UML2 activity models. Within these concern activity models, we directly support the separation of interdependent concerns. In addition, we provide bindings of the concern activity models to UML class and interaction models to enable a detailed specification of concern behavior.

1 INTRODUCTION

A *concern* is a particular goal, concept, or area of interest [11]. Depending on the modeling perspective and the type of development project, concerns can be in arranged in different categories. For example, one can distinguish the *core concerns* of a software system, such as payment processing in a credit card payment system, from *system-level concerns*, such as logging, transactions, authentication, or performance (see [11]). A successful approach to manage complexity when dealing with multiple concerns is the *separation of concerns* (see, e.g., [5]) which describes the process of breaking a software system down into distinct, consumable concerns that overlap as little as possible.

Most software engineering paradigms explicitly support separation of concerns. For example, object-orientation separates concerns into classes and methods; service-oriented computing separates concerns into services and operations; model-driven software development (MDSD) [17] separates concerns into various model types, defined using meta-models, and domain-specific languages (DSL) based on them; aspect-orientation (AO) [10] separates concerns into aspects. Some concern separation techniques even offer a multi-dimensional separation of concerns (see, e.g., [21]). For example, in aspect-orientation, object-oriented techniques are frequently applied for the main dimension of concern separation (often focusing on the core concerns), and aspects are used for separating tangled or cross-cutting concerns (often the system-level concerns are represented as

Cite this document as follows: Mark Strembeck, Uwe Zdun: *Modeling Interdependent Concern Behavior Using Extended Activity Models*, in Journal of Object Technology, vol. 7, no. 6, July–August 2008, pages 143–166,
http://www.jot.fm/issues/issues_2008_04/article5

aspects). A similar multi-dimensional separation of concerns can also be achieved using MDSD techniques (see, e.g., [23]).

Modeling of interdependent concern behavior refers to a common problem in all mentioned concern separation techniques. For instance, in AO, modeling interdependent concern behavior refers to modeling the aspect interactions, defined through pointcuts and the weaving algorithms of the aspect weaver. In MDSD, modeling interdependent concern behavior refers to defining the model integration across model types (especially different meta-models) and DSLs (for instance defined through template languages or transformation languages) as well as the code generation process.

It is not trivial though to *model interdependent concern behavior* because the semantics of concern separation techniques are typically intricately tangled in different model types and particularities of the environment (like an aspect weaver or a code generator). Moreover, as modeling interdependent concern behavior is only one facet of modeling "concerns" in general, a modeling approach for interdependent concern behavior should be well integrated with other modeling techniques, so that it can be used together with existing tools and approaches.

In this paper, we propose an extension to the UML2 to support modeling interdependent concern behavior. An overview of our approach is given in Section 2, the details are specified in Sections 3, 4, and 5. In particular, our approach introduces extensions of UML activity diagrams to model *concern behavior* in general, and *interdependent concern behavior* in specific (explained in Section 3). In addition, we provide a straightforward binding of our extended activity models to classes that implement the concerns (see Section 4) and UML interaction models that specify detailed invocation sequences for the concerns (see Section 5). As an UML extension, the approach can also easily be integrated with other UML-based approaches for modeling separated concerns, such as [1, 7, 22, 26] for example.

We demonstrate our approach on a case study (see Section 6) where we modeled interdependent concerns in a domain-specific language for role-based access control (RBAC). Subsequently, Section 7 gives a comparison of our approach to related work, before Section 8 concludes the paper.

2 APPROACH OVERVIEW AND BACKGROUND

In general, our approach consists of three principle steps, and Figure 1 shows an overview of the models produced in the approach. The main steps are:

- *Define concern activities*: We use extended activity diagrams to model interdependent concern behavior (see Section 3).
- *Map concern activities to classes*: Each concern activity is associated with a specific class[1] implementing this particular concern (see Section 4).

[1]Note that UML2 classes can model composite structures and that UML2 components are defined as

- *Refine concern activities via interaction models*: Interaction models are used to refine the concern activities. In particular, they define the behavior of the classes implementing the concern activities (see Section 5).

These steps are repeated to refine the corresponding models until the software engineers conducting the process define the models as complete.

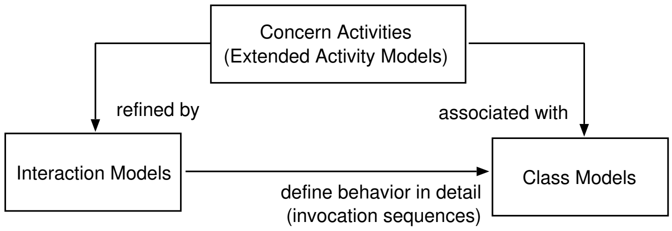


Figure 1: Relations of the models produced in the approach

All modeling steps of our approach are supported via extensions of UML2. In general, the UML standard [15] defines two extension options: a) an extension of the language meta-model, which means a definition of new elements for the UML family of languages; b) a profile specification, which is in essence a set of stereotypes, tag definitions, and constraints that are based on existing UML elements with some extra semantics according to a specific domain. In this paper, we use both extension mechanisms to extend the UML meta-model with new modeling constructs. We also apply the object constraint language (OCL) [14] to define the necessary constraints for the newly defined stereotypes and meta-classes to formalize their semantics. OCL constraints are the primary mechanism for traversing UML models and specifying precise semantics on metaclasses and stereotypes.

The bindings between the more detailed interaction and class diagrams, on the one hand, and the more high-level control flow in the activity diagrams, on the other hand, are needed to provide a consistent and comprehensive concern specification. They also provide the foundation for traceability between the different models (see, e.g., [6, 16]) and for the integration in software tools. We chose this approach to modeling concerns, because the activity diagrams – as primarily behavioral models – help us to focus on concern behavior and interdependencies. Technical details regarding the realization of concerns are externalized into the class diagrams, interaction diagrams, and model bindings. The detailed specification is necessary especially to get all the details into the models that are needed for tools based on these models, such as code generators. In addition, the separation of high-level control flow models from technical detailed models that refine them has a number of other benefits: The high-level models provide an overview that is useful for communicating with non-technical stakeholders such as business or domain experts, whereas the details of the class and interaction models are needed by technical experts. It is then fairly easy to change or exchange technical details in the separated implementation models (which is a common procedure, e.g. due to technology changes) while the

a subtype of the "Class" type (for details see [15]). Therefore, our "concern activities" can be associated with a single class, a structured class, or a component (consisting of several classes) which implements this particular concern.

typically more stable control flow models – that represent the main (business) logic of the application – can stay unaffected.

3 UML EXTENSION FOR CONCERN ACTIVITIES

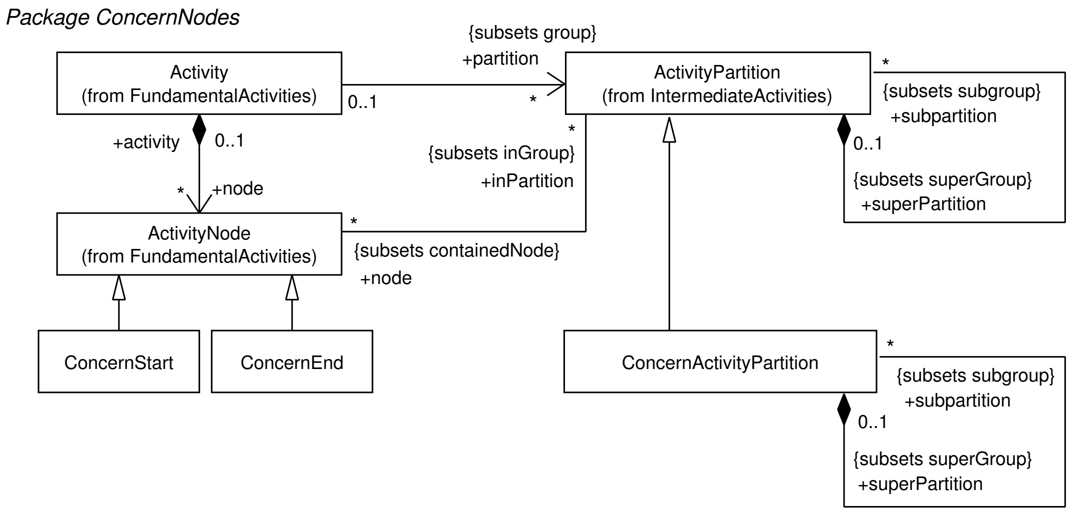


Figure 2: UML Meta-model extension for ConcernNodes

As mentioned above, we model interdependent concern behavior as part of the activity diagrams describing the system's behavior. That is, we primarily model interdependent concerns from a control-flow-oriented (or "process-oriented") perspective. Furthermore, to support separation of concerns when specifying multiple concerns and their interdependencies in the same model, we use activity sub-partitions as a simple, yet effective, means to distinguish the parts of an activity diagram that model a specific concern from the parts of the activity diagram modeling other concerns or concern-independent parts.

We define the new package *ConcernNodes* as an extension to the UML2 meta-model (see Figure 2). In particular, we introduce two new nodes as subclasses of the UML2 ActivityNode meta-class (from the FundamentalActivities package, see [15]), and a new type of partition as subclass of the ActivityPartition meta-class (from the IntermediateActivies package, see [15]).

A *ConcernStart* node is an ActivityNode that can be used in an UML activity diagram to indicate that the concern referred to via the name of the ConcernStart node has intercepted the control flow at this point. All steps in an activity diagram between a ConcernStart and the corresponding ConcernEnd ActivityNode (referred to via the same name for the ConcernEnd as used for the ConcernStart node) are modeling the respective concern. In other words: *ConcernEnd* is an ActivityNode that can be used in an UML activity diagram to indicate that the interception of the control flow by the corresponding concern has ended. ConcernStart and ConcernEnd nodes are always included in a ConcernActivityPartition. The following OCL invariant [14] formally defines this constraint:

```
context Activity
inv: self.node->forAll(n |
     if n.oclIsKindOf(ConcernStart) or
        n.oclIsKindOf(ConcernEnd) then
     n.inPartition->forAll(p |
       p.oclIsKindOf(ConcernActivityPartition))
     endif)
```

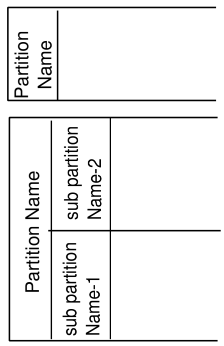
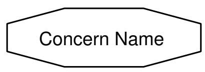
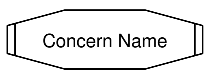
Node Type	Notation	Explanation & Reference
ConcernActivityPartition		A ConcernActivityPartition is represented by two parallel lines (also called swimlane notation), either horizontal or vertical, and a name labeling the partition in a box at one end. Each ConcernActivityPartition may have an arbitrary number of sub partitions. See ConcernActivityPartition from ConcernNodes and ActivityPartition from IntermediateActivities.
ConcernStart		Each ConcernStart node is represented by an octagonal frame including the name of the corresponding node. A ConcernStart node indicates that the concern "Concern Name" has intercepted the control flow at this point. All subsequent steps in the Activity Diagram until a ConcernEnd Activity with the same "Concern Name" is reached are handled by the concern "Concern Name". A ConcernStart node is a specialized UML2 ActivityNode that models the start activity of a particular concern. See ConcernStart from ConcernNodes and ActivityNode from FundamentalActivities.
ConcernEnd		Each ConcernEnd node includes the name of the corresponding node and is represented by an octagonal frame with an additional vertical line on the left hand and the right hand side of the octagon. A ConcernEnd node indicates that the interception of the control flow by the concern "Concern Name" has ended. A ConcernEnd node is a specialized UML2 ActivityNode that models the end activity of a particular concern. See ConcernStart from ConcernNodes and ActivityNode from FundamentalActivities.

Figure 3: Notation elements for ConcernNodes

It is also possible for another concern to intercept the control flow between a ConcernStart and ConcernEnd. This way, we can model concern interdependencies and nested concerns. Modeling of interdependent concerns is directly supported because each ConcernActivityPartition may include sub-partitions, as defined in Figures 2 and 3.

To guarantee a proper nesting of concern nodes, the start and the corresponding end node must be contained in the same partition. Moreover, each (sub-)partition (each swimlane) in a ConcernActivityPartition must contain only ConcernStart and ConcernEnd nodes of one and the same concern. Therefore, we demand that the ConcernStart and the ConcernEnd node within a particular ConcernActivityPartition must have the same name[2] – as defined through the following OCL invariants:

```
context ConcernActivityPartition
inv: self.node->forAll(n:ConcernStart |
       n.name.notEmpty())
inv: self.node->forAll(n:ConcernEnd   |
       n.name.notEmpty())
inv: self.node->forAll(n1:ConcernStart |
       self.node->forAll(n2:ConcernEnd |
         n1.name = n2.name))
```

Because this constraint defines invariants for the ConcernActivityPartition, it holds for all sub-partitions of type ConcernActivityPartition as well (see also Figures 2 and 3).

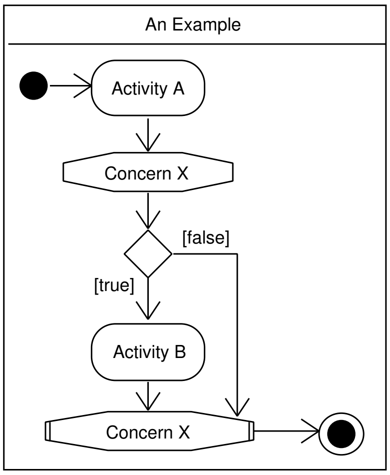


Figure 4: Example diagram with ConcernNodes

Figure 4 shows an example activity diagram with ConcernNodes: after Activity A is completed, Concern X intercepts the control flow. If the subsequent condition evaluates to “true” the control flow proceeds with Activity B. If, however, the condition evaluates to “false”, the corresponding ConcernEnd node is reached and the activity sequence ends.

To model concern interdependencies, we use sub-partitions to clearly separate the different concerns while specifying the complete activity sequence in a consistent, integrated

[2]A UML ActivityNode is a NamedElement (see [15]). Therefore, ConcernStart and ConcernEnd are also NamedElements. In UML, two instances of the NamedElement type may co-exist within a namespace if they are distinguishable. Two NamedElements are *distinguishable* if they a) have unrelated types, i.e. if no direct or transitive subtype/supertype relation (in the sense of `oclIsTypeOf` and `oclIsKindOf`) between their types exists, or b) they have related types but different names (see [15]). ConcernStart and ConcernEnd are unrelated types, i.e. they share the same supertype (ActivityNode) but there is no subtype/supertype relation between ConcernStart and ConcernEnd. Therefore, a ConcernStart and a ConcernEnd node can share the same name and can legally co-exist in the same ConcernActivityPartition.

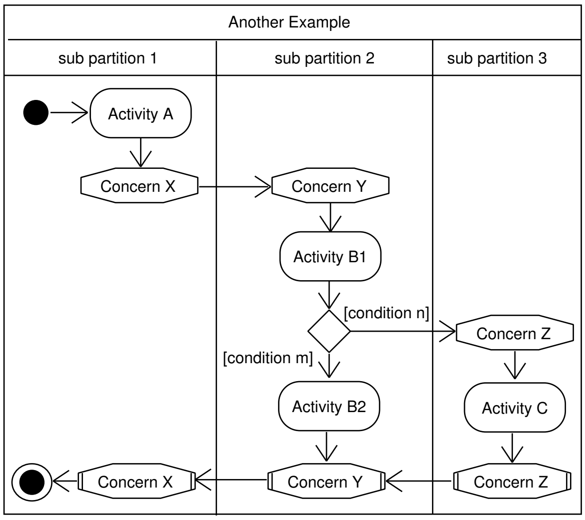


Figure 5: Example with sub partitions and nested concerns

model. In other words, because each concern is modeled via its own sub-partition, it is, on the one hand, easy to examine the different concerns individually, and, on the other hand, we can analyze concern interdependencies directly using one and the same activity model. Figure 5 depicts an example of three interdependent concerns X, Y, and Z, each modeled in its own sub-partition.

4 INTEGRATION WITH STRUCTURE MODELS

To define bindings from ConcernNodes to corresponding classes that implement the behavior of the respective ConcernNodes we define the `concernSpec` and `concern` stereotypes (see Figure 6).

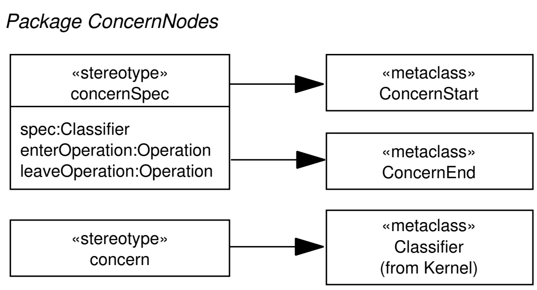


Figure 6: Stereotypes to integrate ConcernNodes with Classes

The `concernSpec` stereotype extends the ConcernStart and ConcernEnd metaclasses introduced in Section 3. The `concernSpec` stereotype defines three properties, and two of these properties especially make use of the fact that UML2 activity models have a token semantics (see below). In particular, an ActivityNode is executed when all required tokens were accepted at the *incoming* ActivityEdge(s) of the respective node. When an

ActivityNode finishes execution, tokens are offered to one or more of its *outgoing* ActivityEdge(s) (for details see [15]):

- The `spec` property refers to the class[3] or an interface that implements the behavior of one particular ConcernNode, i.e. the `spec` property includes the name of a corresponding class or interface defined in a UML structure model (see below).
- The `enterOperation` property defines which operation of the class or interface (referred to via the `spec` property) is invoked as soon as the corresponding ConcernNode is entered (i.e. when all required tokens were accepted at the *incoming* ActivityEdge(s) of this particular ConcernNode).
- The `leaveOperation` property defines which operation of the class or interface (referred to via the `spec` property) is invoked as soon as the corresponding ConcernNode is left (i.e. when a token is offered to one or more of the *outgoing* ActivityEdge(s) of this particular ConcernNode).

The definition of a `concernSpec` for a ConcernNode is optional (depending on the intended use of the corresponding model). However, as mentioned above, if a `concernSpec` is defined, the `spec` property must either refer to a class or to an interface. Moreover, it must include the `spec` property and at least one of the `enterOperation` or `leaveOperation` properties, as defined through the following OCL constraints:

```
context concernSpec
inv: self.spec.oclIsKindOf(Class) or
     self.spec.oclIsKindOf(Interface)
inv: self.spec.notEmpty() and
     (self.enterOperation.notEmpty() or
      self.leaveOperation.notEmpty())
```

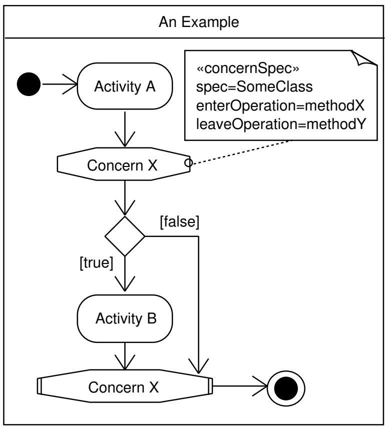


Figure 7: A ConcernNode with a concernSpec stereotype

[3]As mentioned above (see Section 2): by referring to UML2 classes the `spec` property can also refer to a structured class or a component (which may consist of several classes).

In addition to concernSpec, we define the concern stereotype (see Figure 6) which extends the UML metaclass Classifier (cf. [15]). We require that an Interface or Class which realizes the behavior of a ConcernNode must be typed with the concern stereotype[4]:

```
context concernSpec
inv: concern.base_Classifier->exists(c|
        c.name = self.spec and
        (c.oclIsKindOf(Class) or
         c.oclIsKindOf(Interface))
     )
```

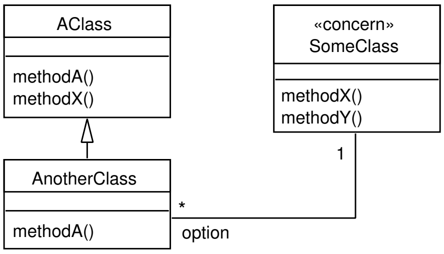


Figure 8: A Class with a concern stereotype

Figure 7 shows the example from Figure 4 extended with a concernSpec for the ConcernStart node of "Concern X". The spec property of this concernSpec defines that the behavior of the respective ConcernStart node is implemented by a class called SomeClass. Furthermore, it defines methodX of SomeClass as the enterOperation and methodY of SomeClass as the leaveOperation for this node. The class model in Figure 8 includes the respective SomeClass class, stereotyped with ≪*concern*≫. As can be seen in the example figure, a class (or component) implementing a concern is part of an ordinary class diagram and may have association and/or inheritance relationships to other classes. These other classes are often implementing parts of a concern, and the class stereotyped by ≪*concern*≫ only is a Facade or Interface to the concern implementation. In any case, we just need to tag a class (or component) with the ≪*concern*≫ stereotype in order to use an existing class as a concern spec.

5 INTEGRATION WITH INTERACTION MODELS

In addition to the extensions introduced above, we need a means to describe concern interactions (invocation sequences, return values, etc.) on a detailed level, if necessary. Therefore, we define an additional stereotype that allows for an integration of concern activities with UML Interaction models.

[4]Again: since the concern stereotype can be applied to UML classes, the same stereotype can also be applied for structured classes and components. Moreover, the concern stereotype can be attached to UML interfaces. Thereby we achieve additional flexibility because interfaces are abstract entities which are implemented by one or more class, structured class, or component, and interfaces can be referenced by component ports (for details see [15]).

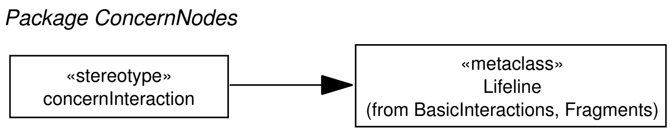


Figure 9: Stereotype to integrate ConcernNodes with UML Interaction models

In particular, the `concernInteraction` stereotype extends the UML metaclass Lifeline (from the BasicInteractions Package, see [15]) to enable bindings between ConcernNodes and interaction participants in corresponding UML Interaction models (see Figure 9). Using Interaction models, we can now model the detailed invocation sequences that occur when (interdependent) concerns (implemented via certain classes) are executed at runtime. The following OCL constraints define that a `concernInteraction` lifeline must refer to the class defined in the corresponding `concernSpec`, or to a class implementing the interface referred to via the respective `concernSpec` (making sure that we specify the behavior of the correct class).

```
context concernSpec
inv: concernInteraction.base_Lifeline->forAll(ll |
       if self.spec.oclIsKindOf(Class)
       then
          self.spec.role->exists(r | r.lifeline = ll)
       endif

       if self.spec.oclIsKindOf(Interface)
       then
          self.spec.interfaceRealization->exists(ir |
            ir.implementingClassifier.oclIsKindOf(Class)
            and
            ir.implementingClassifier.role->exists(r | r.lifeline = ll)
          )
       endif
)
```

Moreover, we define two additional OCL constraints on `concernInteraction`. The first constraint specifies that if the `concernSpec` defines an `enterOperation`, this operation must occur as a message of the respective lifeline, and it must be the first operation called on that lifeline (be it synchronous or asynchronous):

```
context concernSpec
inv: concernInteraction.base_Lifeline->forAll(ll |
      if (self.enterOperation.notEmpty())
      then
         let mos : MessageOccurrenceSpecification =
             ll.interaction.fragment->select(f |
               f.oclIsKindOf(MessageOccurrenceSpecification)
               and f.covered = ll)->first()
         in
           mos.notEmpty() and
           mos.message.notEmpty() and
           mos.message.oclIsKindOf(Operation) and
           mos.message = self.enterOperation and
           (mos.message.messageSort = #syncCall or
           mos.message.messageSort = #asyncCall)
      endif)
```

The second OCL constraint defines that if the respective lifeline contains a message that corresponds to a `leaveOperation` of the respective `concernSpec`, this operation must be the last (synchronous or asynchronous) operation called on that lifeline (note that a "reply" message does not model an operation call, for details see [15]):

```
context concernSpec
inv: concernInteraction.base_Lifeline->forAll(ll |
      if (self.leaveOperation.notEmpty())
      then
         let mos : MessageOccurrenceSpecification =
             ll.interaction.fragment->select(f |
               f.oclIsKindOf(MessageOccurrenceSpecification)
               and f.covered = ll
               and f.message.notEmpty()
               and (f.message.messageSort = #syncCall or
                    f.message.messageSort = #asyncCall))->last()
         in
           mos.notEmpty() and
           mos.message.notEmpty() and
           mos.message.oclIsKindOf(Operation) and
           mos.message = self.leaveOperation
      endif)
```

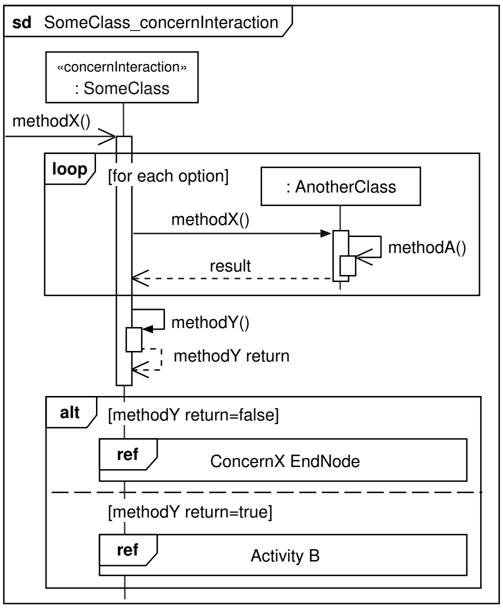


Figure 10: An Interaction diagram modeling the invocation sequence in the ConcernX start node - implemented by the `SomeClass` class

Figure 10 depicts an interaction diagram modeling the detailed invocation sequence that is performed when the control flow defined in Figure 7 reaches the start of `Concern X`. In particular, it shows that an object of `SomeClass` (see Figure 8) receives an invocation of `methodX`. Subsequently, it triggers the execution of `methodX` in each associated `AnotherClass` object before executing `methodY`. Depending on the return value of `methodY` the control flow either proceeds with the execution of the `Concern X` end node or with `Activity B` (cf. Figures 7 and 10).

6 CASE STUDY: MODELING INTERDEPENDENT CONCERNS IN AN RBAC DSL

In this section, we demonstrate our approach on a non-trivial case where we defined interdependent concerns in a DSL for the specification of role-based access control (RBAC) policies. To ease the understanding of the example, we briefly introduce some essential RBAC terms first: In the RBAC context, an access control *subject* is an active entity (e.g. a human user or a software agent) that is (or should be) able to access objects (e.g. files or hardware resources as a printer or a network card for example) in a particular information system. Each subject has a number of *roles* that are assigned to this subject. Moreover, *permissions* are assigned to roles, and permissions can be associated with *context constraints* (cf. Figure 11). Context constraints define predicates that must evaluate to "true" in order to grant a certain access request. They allow for the consideration of context information in access decisions and enable the definition of additional conditions on permissions, like time constraints for example (for details see [19]).

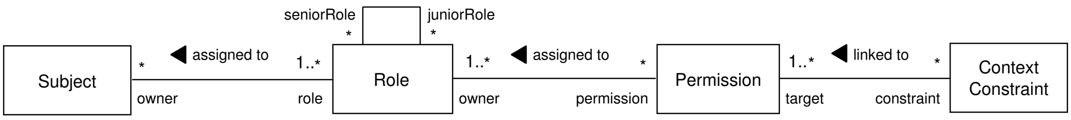


Figure 11: Basic elements of the role-based access control DSL

Our RBAC DSL provides the functionality of the xoRBAC component (see [13, 19]) as an expressive language that separates the different concerns in this component. To implement the RBAC DSL, we defined a domain-specific weaver component that is capable to weave the different concerns according to domain-specific restrictions. The weaver component provides functions for role-to-subject assignment and revocation, as well as corresponding functions for permission-to-role assignment and revocation, and for linking and unlinking permissions and context constraints. Moreover, it allows to generate new role, permission, or context constraint classes at runtime (see also [20]).

On the source code level, each model element of the role-based access control DSL (in essence these are: subject, role, permission, and context constraint) is represented via a class or class hierarchy, and the definition of individual elements is separated from the classes and hierarchies representing other domain-specific model elements. It is, however, not trivial to achieve this goal since the different concerns are interdependent, and these concern interdependencies can hardly be modeled using standard modeling constructs. For the specification of the DSL, we therefore used activity models with ConcernNodes that specifically focus on the *separation of concerns* in the RBAC DSL.

Modeling the Authorization Decision Concern

Figure 12 depicts an activity diagram that shows the primary control flow for authorization decisions. Furthermore, it shows the interdependencies of the Authorization concern, the Role concern, the Permission concern, and the ContextConstraint concern.

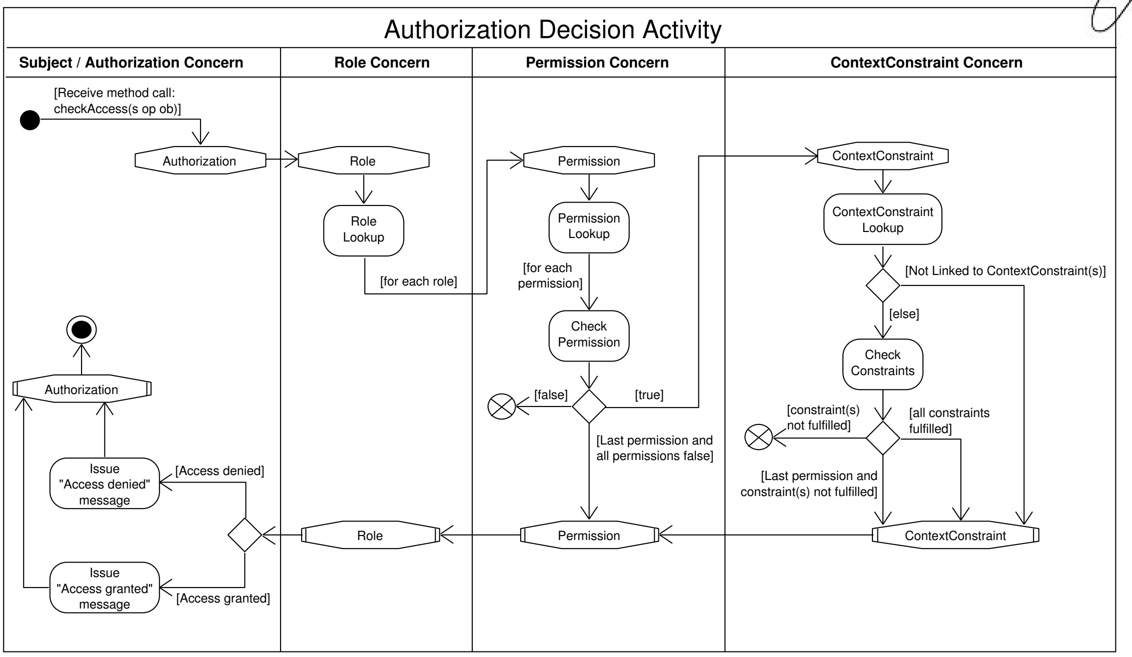


Figure 12: Control flow of the Authorization concern and corresponding nested concerns

The `checkAccess` operation provides a central functionality of the RBAC DSL and is applied to 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` (see Figure 12). After receiving a `checkAccess` message, the Authorization concern is responsible to make an access decision for the corresponding access request. In order to reach this decision, it has to interact with the Role concern. The Role concern then performs a role lookup procedure to determine the roles assigned to the respective subject. Subsequently, the Permission concern takes over to check the permissions which are assigned to the corresponding role objects. However, as mentioned above, to grant a certain access request it is not sufficient to own the appropriate permission - all context constraints associated with the corresponding permission must be fulfilled at the same time. Thus, if a certain permission actually grants the access request (indicated by returning "true") the ContextConstraint concern intercepts the control flow to check the constraints linked to this particular permission object (cf. Figure 12).

Figure 13 shows an excerpt of Figure 12 that includes ≪*concernSpec*≫ stereotypes for the ConcernStart nodes of the `Role` and the `Permission` concerns. In particular, these `concernSpecs` define that the behavior of the corresponding concerns is implemented through the `Role` and `Permission` classes, respectively. Moreover, the `enterOperation` of both concerns is implemented via the `checkAccess` operation of the corresponding classes.

Figure 14 depicts an excerpt of the RBAC DSL class model (for the purposes of this paper we show only an excerpt of the class model, for details see [19]). It indicates how the `Subject`, `Role`, and `Permission` classes, that implement the corresponding

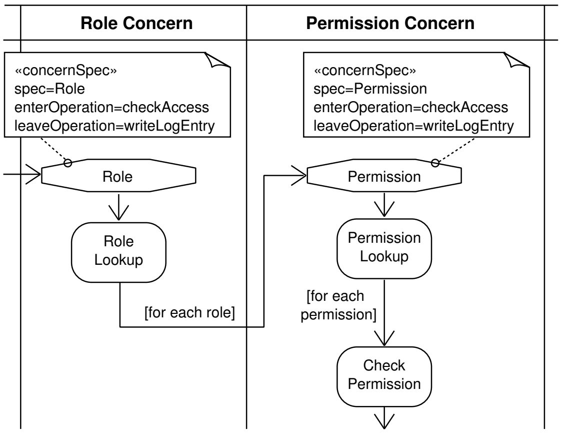


Figure 13: The Role and Permission Concerns with concernSpec stereotypes

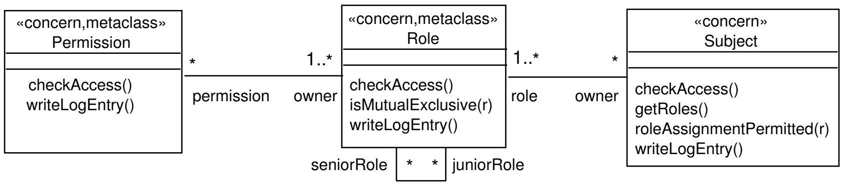


Figure 14: Excerpt of the RBAC DSL class model

concerns in the DSL, are interconnected.

In Figure 15, we see an interaction diagram that models the invocation sequence in the `Role` concern. After receiving a `checkAccess` message, the corresponding `Role` object invokes the `checkAccess` method on the permission objects assigned to this particular role. If one of the permissions grants the access by returning "true", the `Role` object immediately stops checking its permissions (see the "break" InteractionOperator in Figure 15), writes a corresponding log entry, and returns the access decision.

Modeling the Role-to-Subject Assignment Concern

Subject-to-Role Assignment deals with the procedure of associating roles with subjects, and, similar to the Authorization concern, the Assignment concern of the RBAC DSL consists of several interdependent concerns (see Figure 16). In particular, RBAC directly supports the "separation of duty" concept (see, e.g., [3, 4, 18]) which directly affects subject-to-role assignment. In access control, separation of duty constraints enforce conflict of interest policies. Conflict of interest arises as a result of the simultaneous assignment of two mutual exclusive roles (or permissions) to the same subject. Mutual exclusive roles (or permissions) result from the division of powerful rights or responsibilities to prevent fraud and abuse. An example is the common practice to separate the "controller" role and the "chief buyer" role in medium-sized and large companies.

When a `roleSubjectAssign` message is received, the Assignment concern inter-

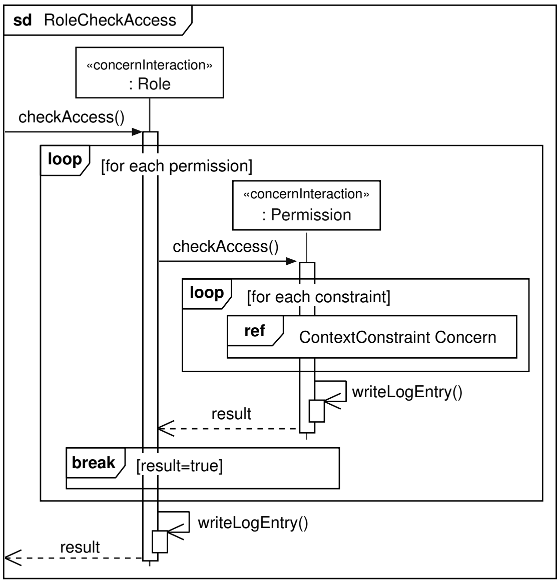


Figure 15: Interaction model for the Role concern

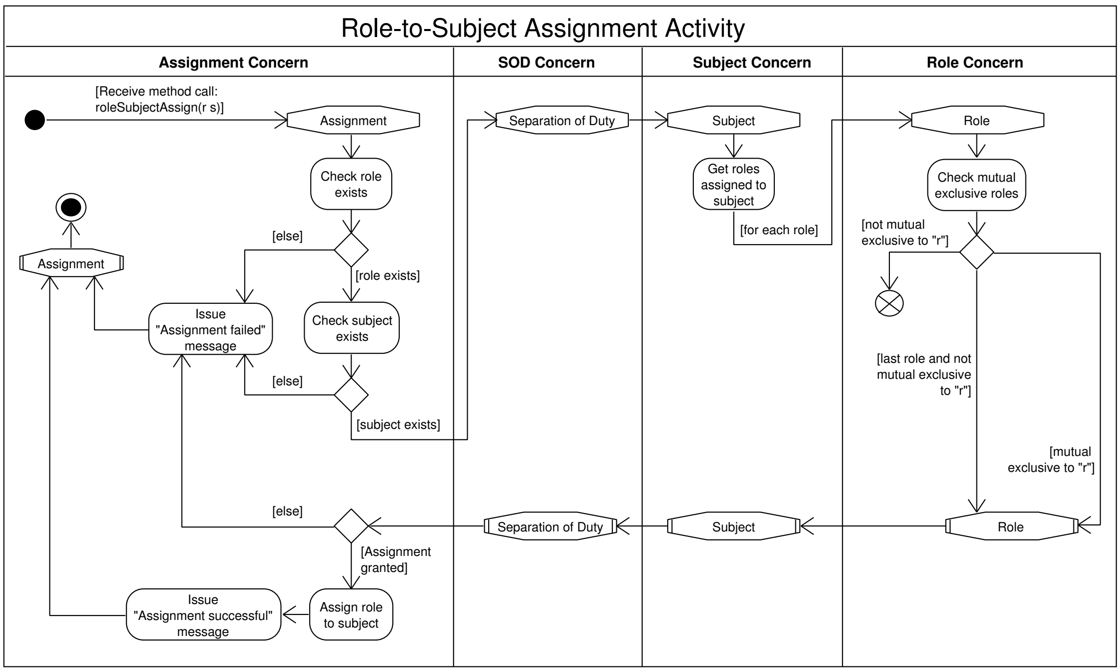


Figure 16: Control flow the Role-to-Subject Assignment Concern

cepts the control flow and first checks if the corresponding role and subject actually exist (cf. Figure 16). Subsequently, the Separation of Duty (SOD) concern interacts with the Subject and Role concerns to determine if the corresponding role can legally be assigned to the respective subject. In case the role to be newly assigned is defined as mutual exclusive to one of the roles already owned by the subject, the assignment is denied.

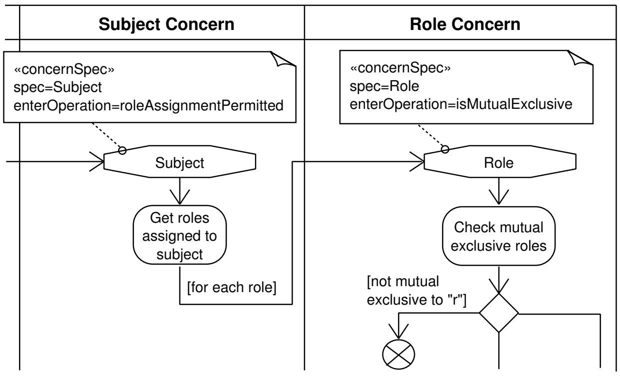


Figure 17: The Subject and Role Concerns with concernSpec stereotypes

Figure 17 depicts an excerpt of Figure 16 that includes ≪*concernSpec*≫ stereotypes for the ConcernStart nodes of the Subject and Role concerns. These stereotypes define that the `enterOperation` of the Subject concern is implemented through the `roleAssignmentPermitted` method of the `Subject` class and the `enterOperation` of the Role concern is implemented via the `isMutualExclusive` method of `Role` class (see also Figure 14).

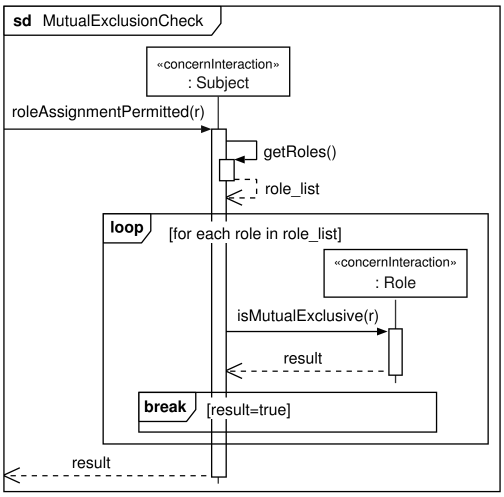


Figure 18: Interaction model for the Subject concern

The interaction model shown in Figure 18 specifies the invocation sequence of the Subject concern in detail. After receiving a `roleAssignmentPermitted` message the corresponding Subject instance invokes the `getRoles` method to obtain a list of all roles currently assigned to this particular subject. Subsequently, it calls the isMutualExclusive method for each of these roles. In case one of the roles returns "true" as result of the isMutualExclusive call (indicating it is in fact mutual exclusive to the role to be newly assigned), the subject stops checking its roles (see "break" InteractionOperator in Figure 18) and the assignment is denied (see also Figure 16).

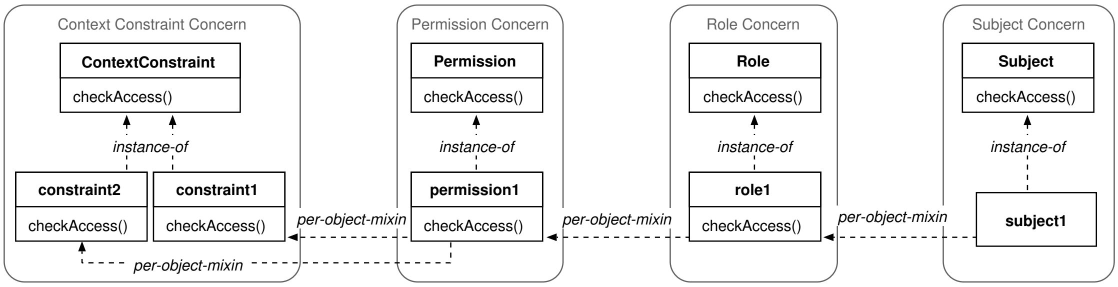


Figure 19: Example of a composed executable model

Finally, Figure 19 sketches an example of a composed runtime model generated from our RBAC 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. In our implementation each of these assignment relations is realized through a transitive mixin relation (as explained in [20, 27]). However, other implementation techniques such as aspects or generated classes could have been used equivalently.

7 COMPARISON TO RELATED WORK

Tarr et al. introduced the concept of multi-dimensional separation of concerns [21], which aims at a separation of arbitrary kinds of concerns. That is, a modeler does not need to decompose different concerns of a system along a single dimension and neglect other dimensions. They use so called hyperslices to model concerns in different dimensions and these hyperslices are composed in hypermodules. This concept necessarily includes interdependencies among the concerns, such as overlapping, nested, or interacting concerns. Our approach supplements the work of Tarr et al. with a concept for modeling interdependent concern behavior using Activity Diagrams as the primary model type, as well as Class and Interaction Diagrams to specify the details of each concern.

A number of other authors have proposed approaches to model mostly structural facets of (multi-dimensional) concern separation, especially in the field of aspect-orientation. For instance, in [26] we have proposed an approach for modeling the evolution of aspect configurations using model transformations. Barros and Gomes [1] use UML2 activity diagrams to model crosscutting in aspect-oriented development. Via an UML profile they define a new composition operation called "activity addition". Activity additions are used for weaving a crosscutting concern in a model. In particular, they define two stereotypes to mark certain nodes in activity diagrams that define the so called interface nodes which are then used to merge two or more activity diagrams, and the so called subtraction nodes which define what nodes need to be removed from a given activity diagram. Hence, Barros and Gomes use a similar approach to model concerns via activity diagrams. However, they do not focus on modeling interdependent concern behavior and do not provide bindings to other models that specify concern implementations in detail.

Han et al. [7] present an approach to support modeling of AspectJ language features to narrow the gap between implementations based on AspectJ and the corresponding models. Mahoney and Elrad [12] describe a way to use statecharts and virtual finite state machines to model platform specific behavior as crosscutting concerns. They especially plan to evaluate the effectiveness of their approach in a model driven development context. Tkatchenko and Kiczales [22] present another approach to model crosscutting concerns. They extend the UML with a joinpoint model, advice and inter-type declarations, and role bindings. Moreover, they provide a weaver to process the corresponding extensions. Jezequel et al. [9] represent crosscutting behavior using contract and aspect models in UML. They model contracts using UML stereotypes, and represent aspects using parameterized collaborations equipped with transformation rules expressed as OCL constraints. In particular, OCL is used in the transformations for navigating instances of the UML meta-model.

Each of the above mentioned approaches provides some type of modeling support for multiple concerns that might be cross-cutting – as their primary focus is on aspect-oriented systems. However, none of these approaches supports modeling concern interaction based on the behavior of interdependent (multi-dimensional) concerns, which is the major focus of our concern modeling approach.

Czarnecki and Antkiewicz propose an approach to model variants of behavioral models [2]. In particular, they use feature models to describe the possible variants of UML activity diagrams. Model templates specify the possible composition of a system's features. Furthermore, they use a special-purpose tool to instantiate the model templates from a feature configuration. Hence, the feature models are akin to our approach in the sense that they also lead to a separation of concerns. Feature modeling, however, concentrates on modeling variants of behavioral models, whereas our approach is intended to model the control flow together with concern interdependencies, as well as the connection to implementation classes for the concerns.

The Concern Manipulation Environment [8] supports the composition of software systems from concern models. It produces directives that can be applied to compose arbitrary object-oriented structures. The main focus of concern composition in the Concern Manipulation Environment is to support a wide variety of aspect-oriented approaches. The Concern Manipulation Environment supports various relationships among concerns, such as overlap and cross-cut. Due to the focus on aspect-oriented approaches, the concern models in the Concern Manipulation Environment are closer to aspects or feature models than to our more general behavioral models based on Activity Diagrams.

A number of other authors and tools provide explicit representations of multiple interacting concerns (and aspects) using MDSD concepts. Voelter summarizes the best practices that have been developed in this area in software patterns form [23]. Our ap-

Table 1: Overview and comparison of related work

Approach	*Modeling of multiple concern dimensions*	*Behavioral concern models*	*Structural concern models*	*Modeling behavioral concern interdependencies*	*Separation of high-level and detailed concern models*	*Separation of different concerns in behavior models*
Toward Support for Crosscutting Concerns in Activity Diagrams [1]	supported	supported	not supported	not supported	not explicitly supported	not supported
Mapping Features to Models: A Template Based Approach [2]	supported	supported	not explicitly supported	supported	not explicitly supported	not explicitly supported
Towards Visual AspectJ [7]	supported	not explicitly supported	supported	not explicitly supported	not explicitly supported	not supported
Concern Manipulation Environment [8]	supported	not supported	supported	supported	not explicitly supported	not explicitly supported
From Contracts to Aspects [9]	supported	not supported	supported	not supported	not explicitly supported	not supported
Aspect-Oriented Statecharts and Virtual Finite State Machines[12]	not explicitly supported	supported	not supported	not supported	not explicitly supported	not supported
Multi-Dimensional Separation of Concerns [21]	supported	not explicitly supported	supported	not explicitly supported	not explicitly supported	not explicitly supported
Modeling Crosscutting Structure [22]	supported	supported	supported	not explicitly supported	not explicitly supported	not explicitly supported
Patterns for Handling Cross-Cutting Concerns [23]	supported	not explicitly supported	supported	not explicitly supported	not explicitly supported	not explicitly supported
Concern Activities (our approach)	supported	supported	supported	supported	supported	supported

proach follows these best practices in a way using a control flow model as the primary model to interconnect other model types. For example, this practice can also be used for process-driven SOA models, as described in detail in [24].

Table 1 summarizes the comparison to related work in terms of the major features of our approach. In the table, we use the term “supported”, if the (research) papers and additional materials (e.g. available from related web pages) about the other approaches explictly describe a support for the respective feature; “not supported” is used, when we did not find information that the approach explicitly supports the feature. Finally, “not explicitly supported” is used, when we did not find information that a particular approach explicitly supports a certain feature, but other concepts that are available from the approach can be used to (straightforwardly) build some support for the respective feature.

Our approach, as well as most of the related approaches mentioned above, supports modeling of multiple concern dimensions – even though each approach uses different abstractions and mechanisms. A key contribution of our approach is that it supports the modeling of interdependent concern behavior together with the structure of concern models and the detailed specification of interaction sequences. None of the related approaches explicitly supports all of these facets. However, many related approaches can be extended with the missing model types. For instance, it would be possible to create feature models on top of class diagrams, to extend the approach in [2]. This can for instance be done using the a similar binding between the structural and behavioral models as used in our approach.

Our approach explicitly models the interactions between concerns through interception of the control flow by concern start and end nodes. Some other approaches offer explicit support for modeling concern interaction, either through composition primitives on the structural models or through feature composition in feature models (cf. Table 1).

None of the related approaches supports a separation of abstraction levels – ranging from the specification of concern behavior in activity diagrams to low-level concern implementation specifications in structural and interaction models. However, because most approaches mentioned above directly support multiple concern dimensions, these approaches could be extended with support for concern levels. Our approach additionally provides a binding between the different concerns levels – which is needed for tools such as code generators. Moreover, our approach supports the separation of interdependent concerns in models through activity sub-partitions (swimlanes). Though activity swimlanes are a standard means for structuring behavioral models (especially in activity models) this is not explicitly supported by the related approaches, most of them, however, could be extended with the same concept. Finally, our approach can be directly integrated with model transformation diagrams (see [25]) to model dynamic changes in concern behavior.

With our work we aimed to define an approach that is well integrated with the UML (as it is the de-facto standard for software systems modeling) and combines the strengths of related approaches while providing an integrated, easy-to-use, and easy-to-understand modeling approach for interdependent concerns. Note that, due to the focus of this paper, Table 1 merely summarizes features that refer to modeling support for interdependent concern behavior and blanks out other modeling concepts additionally supported by the different approaches.

8 CONCLUSION

We introduced an approach to model interdependent concern behavior. While our general concepts to modeling interdependent concern behavior are independent of a certain modeling language, we chose the UML to demonstrate the approach because it is a standardized and well-known language. The approach provides an extension to UML2 activity models where concern start and concern end nodes are applied to model how different concerns intercept the control flow in a system. Each concern is modeled via an own activity (sub-)partition, and each activity model may include an arbitrary number of activity partitions. Hence, modelers can examine the different concerns individually by focusing on a single partition. On the other hand, our approach also supports the examination of concern interdependencies in the same activity model by inspecting all (sub-)partitions of the respective activity model.

In addition, we provide a straightforward integration with the classes (or components) that implement the concerns and with interaction models that specify invocation sequences in detail. The high-level concern activity models provide an overview that is useful for communicating with non-technical stakeholders such as business or domain experts. The more low-level class and interaction models are needed by technical experts. Using these different abstraction levels it is fairly easy to change or exchange low-level concern details in class and interaction models, while the typically more stable control flow models, that represent the main application logic, can remain unaffected. The integration of high-level concern activity models with class and interaction models is useful, for example, in the context of MDSD approaches and code generation techniques.

Furthermore, as each of our modeling constructs is defined as UML2 extension, the approach can be applied to supplement other UML-based approaches and can be integrated in UML-based software tools.

References

[1] J. Barros and L. Gomes. Towards the Support for Crosscutting Concerns in Activity Diagrams: a Graphical Approach. In *Proc. of the AOSD Modeling with UML Workshop*, October 2003.

[2] K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *Proceedings of 4th International Conference on Generative Programming and Component Engineering (GPCE 2005)*, Sep/Oct 2005.

[3] D. Ferraiolo, J. Barkley, and D. Kuhn. A Role-Based Access Control Model and Reference Implementation within a Corporate Intranet. *ACM Transactions on Information and System Security (TISSEC)*, 2(1), February 1999.

[4] D. Ferraiolo, R. Sandhu, S. Gavrila, D. Kuhn, and R. Chandramouli. Proposed NIST Standard for Role-Based Access Control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3), August 2001.

[5] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 1991.

[6] O. Gotel and A. Finkelstein. An analysis of the requirements traceability problem. In *Proc. of the IEEE International Conference on Requirements Engineering (ICRE)*, 1994.

[7] Y. Han, G. Kniesel, and A. Cremers. Towards Visual AspectJ by a Meta Model and Modeling Notation. In *Proc. of the International Workshop on Aspect-Oriented Modeling*, March 2005.

[8] W. Harrison, H. Ossher, S. Sutton, and P. Tarr. Supporting aspect-oriented software development with the concern manipulation environment. *IBM Systems Journal*, 44(2), 2005.

[9] J. Jezequel, N. Plouzeau, T. Weis, and K. Geihs. From Contracts to Aspects in UML Designs. In O. Aldawud, G. Booch, S. Clarke, T. Elrad, W. Harrison, M. Kande, and A. Strohmeier, editors, *Aspect-Oriented Modeling with UML*, Enschede, The Netherlands, April 2002.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP)*. LCNS 1241, Springer-Verlag, June 1997.

[11] R. Laddad. *AspectJ in Action, Practical Aspect-Oriented Programming*. Manning Publications Co., 2003.

[12] M. Mahoney and T. Elrad. Modeling Platform Specific Attributes of a System as Crosscutting Concerns using Aspect-Oriented Statecharts and Virtual Finite State Machines . In *Proc. of the International Workshop on Aspect-Oriented Modeling*, March 2005.

[13] 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.

[14] OCL 2.0 Specification. http://www.omg.org/technology/documents/formal/uml.htm, May 2006. Version 2.0, formal/06-05-01, The Object Management Group.

[15] OMG Unified Modeling Language (OMG UML): Superstructure. http://www.omg.org/technology/documents/formal/uml.htm, November 2007. Version 2.1.2, formal/2007-11-02, The Object Management Group.

[16] B. Ramesh and M. Jarke. Toward reference models for requirements traceability. *IEEE Transactions on Software Engineering (TSE)*, 27(1), January 2001.

[17] T. Stahl and M. Voelter. *Model-Driven Software Development*. J. Wiley and Sons Ltd., 2006.

[18] 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.

[19] 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.

[20] M. Strembeck and U. Zdun. Definition of an Aspect-Oriented DSL using a Dynamic Programming Language. In *Proc.of the Workshop on Open and Dynamic Aspect Languages (ODAL)*, March 2006.

[21] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, May 1999.

[22] M. Tkatchenko and G. Kiczales. Uniform Support for Modeling Crosscutting Structure. In *Proc. of the 8th International Conference on Model Driven Engineering Languages and System (MoDELS)*. LNCS 3713, Springer Verlag, October 2005.

[23] M. Voelter. Patterns for handling cross-cutting concerns in model-driven software development. In *Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPlop)*, July 2005.

[24] U. Zdun and S. Dustdar. Model-Driven Integration of Process-Driven SOA Models. *International Journal of Business Process Integration and Management (IJBPIM)*, 2(2), 2007.

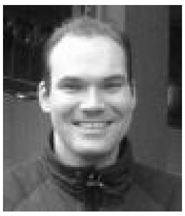
[25] U. Zdun and M. Strembeck. Modeling Composition in Dynamic Programming Environments with Model Transformations. In *Proc. of the 5th International Symposium on Software Composition*. LNCS 4089, Springer-Verlag, March 2006.

[26] U. Zdun and M. Strembeck. Modeling the Evolution of Aspect Configurations using Model Transformations. In *Proc.of the Linking Aspect Technology and Evolution Workshop (LATE)*, Bonn, Germany, March 2006.

[27] U. Zdun, M. Strembeck, and G. Neumann. Object-based and class-based composition of transitive mixins. *Information and Software Technology*, 49(8), August 2007.

ABOUT THE AUTHORS

Mark Strembeck is currently working as an assistant professor in the Institute of Information Systems at the Vienna University of Economics and BA, Austria. Mark received two diploma degrees in information systems from the University of Essen (Germany) in 1998 and 1999 respectively, and a doctoral degree in information systems from the Vienna University of Economics and BA in 2003. His research interests include access control, role engineering, distributed systems, services computing, model-driven development, language engineering, and the modeling and management of dynamic software systems. Mark can be reached at strembeck@acm.org.



Uwe Zdun is currently working as an assistant professor in the Distributed Systems Group at the Vienna University of Technology, Vienna, Austria. Prior to that, Uwe has worked as an assistant professor in the Institute of Information Systems at the Vienna University of Economics and BA, Vienna, Austria. He received his doctoral degree from the University of Essen in 2002. His research interests include software patterns, software architecture, language engineering, SOA, distributed systems, and object-orientation. Uwe has published in numerous conferences and journals, and is co-author of the books "Remoting Patterns" (J. Wiley & Sons) and "Software-Architektur" (Elsevier/Spektrum). He is (co)author of open-source software systems, such as Frag, Extended Object Tcl (XOTcl), Leela, ActiWeb, and many other software systems. Uwe served as conference chair for EuroPLoP 2005, program chair for EuroPLoP 2006, is the European Editor of the Transactions on Pattern Languages of Programming (TPLoP) journal published by Springer, and is Associate Editor-in-Chief for the IEEE Software magazine. Uwe can be reached at zdun@infosys.tuwien.ac.at