

An Approach for Aspect-Oriented Use Case Modeling

Stéphane S. Somé
School of Information Technology and
Engineering (SITE)
University of Ottawa
800 King Edward, Ottawa, Canada
ssome@site.uottawa.ca

Pauline Anthonysamy
School of Information Technology and
Engineering (SITE)
University of Ottawa
800 King Edward, Ottawa, Canada
panthony@site.uottawa.ca

ABSTRACT

A use case model is a specification of a system's requirements consisting in use cases, actors and relationships. A use case captures stakeholders concerns as required interactions between a system and its actors. Use case models may however include concerns that crosscut across several use cases. We propose an `<<aspect>>` relation for the modularization and composition of these crosscutting concerns. The composition approach is formally established by mappings to Petri nets, and is implemented as an extension to an existing use case modeling tool.

Categories and Subject Descriptors

D.2.1 [Software Engineering]: Requirements/Specification—*Mehodologies, Tools*

General Terms

Languages

Keywords

Concerns, Use Cases, Petri nets, UML

1. INTRODUCTION

Aspect-Oriented Software Development (AOSD) suggests the identification, isolation and modularization of concerns in the early stages of the development process including the requirements stage [7]. The ability to isolate and modularize concerns (requirements) helps requirements modelers focus on concerns separately, helps avoid tangled representation in the requirements specification and enables requirements evolution. However, an aspect-oriented requirements modeling approach should be such that the interactions among independently defined crosscutting requirements can be visualized and validated early. Failure to do so may result in serious difficulties at later development stages. It is therefore necessary to also provide a composition mechanism that

allows the entire set of requirements to be visualized and validated as a whole.

The use case approach is one of the common requirement modeling technique. Use cases capture requirements as description of interactions involving systems and their environments. Each use case is “the specification of a set of actions performed by a system, which yields an observable result that is, typically, of value for one or more actors or other stakeholders of the system” [12]. In a previous work [15], we proposed an approach to support use case based requirements engineering. We capture use cases in a restricted natural language with formal semantics and automatically generate executable state machines [16]. The generated state machines are used as prototypes for requirements validation by simulation. The approach is supported by a tool called Use Case Editor (UCed) [3]. In this paper, we extend our previous work to support modeling aspects in use case specifications. The relation between use cases and aspects has been discussed before. For instance, Jacobson and Ng [10] noticed a close relation between use cases and aspects as each use case typically crosscuts a set of components and usually involves re-occurring concerns. We propose a new approach for modeling aspects in a use case specification. We first observe the inadequacy of UML use case relations `<<include>>` and `<<extend>>` for modeling re-occurring concerns as aspects and propose an extension of use case models with a specific `<<aspect>>` relationship. Then we present a composition mechanism allowing obtaining a global behavior model from use cases integrating all independently defined concerns.

The remainder of this paper is organized as follows. In the next section, we present background material on use case modeling and introduce an example to motivate and illustrate our approach. We discuss crosscutting concerns related to use cases in section 3 and the limitations of traditional UML relationships. In Section 4 we introduce an `<<aspect>>` relationship with elements for modeling concerns in use cases. We also describe concerns composition in term of Petri nets. In section 5, we position our works in relation related to ours. Finally, section 6 concludes the paper and discusses some future works.

2. USE CASE MODELING

A UML use case model includes use cases, actors, relationships between actors and use cases, and relationships between use cases. The three types of relationships between use cases are use case inclusion (`<<include>>` relation), use case extension (`<<extend>>` relation) and use case ge-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EA'08, May 12, 2008, Leipzig, Germany.

Copyright 2008 ACM 978-1-60558-032-6/08/05 ...\$5.00.

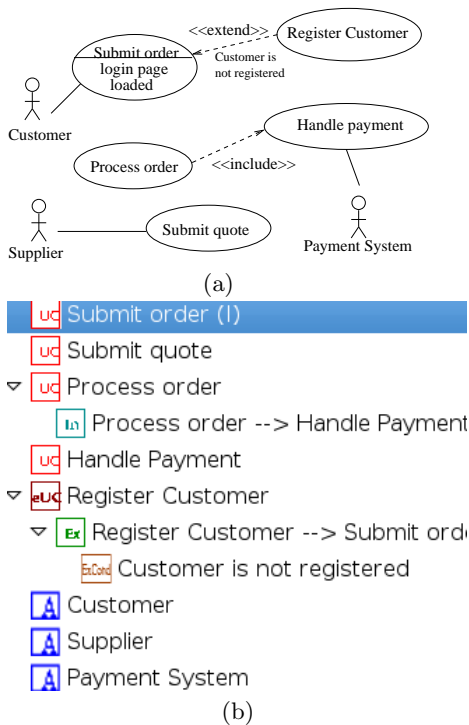


Figure 1: UML Use Case diagram for an order processing system and the corresponding representation in the UCed tool.

neralization/specialization. An $\ll include \gg$ relationship $uc_{base} \times uc_{inc}$ denotes the inclusion of use case uc_{inc} as a sub-process of use case uc_{base} (the base use case). An *extend* relationship $uc_{ext} \times econd \times epoints \times uc_{base}$ denotes an extension of a use case uc_{base} with the addition of “chunks” of behaviors defined in an *extension use case* uc_{ext} . These chunks of behaviors are included at specific places in the base use case called *extension points* (*epoints*). Each extension is realized under a specific condition (*econd*). A use case generalization/specialization relation defines an inheritance relation between a more general use case and a more specific use case. The more specific use case is a special case of the more general use case. A UML use case diagram is a graphical depiction of a use case model. Figure 1-(a) shows a use case diagram for an *Order Processing* system. The corresponding representation of the use case model in the UCed tool is shown in Figure 1-(b). UCed uses a “tree” representation for use case models such that properties attached to a relation appear as children of that relation.

Use case interaction details are usually captured at the requirement engineering stage as a text in natural language. In order to support automated synthesis of state models from use cases, we formalized use case description by defining an abstract syntax, a concrete syntax based on restricted natural language and by providing Petri nets based semantics to use cases [16]. We distinguish *normal use cases* from *extend use cases*. While a normal use case specifies complete interaction sequences (resulting in a use case goal being fulfilled or abandoned), an extend use case consists in a set of behavior *chunks* intended to extend other use cases. Figure 2 shows a normal use case (“Submit order”) and an extend use case (“Register Customer”). Formally, a normal use case

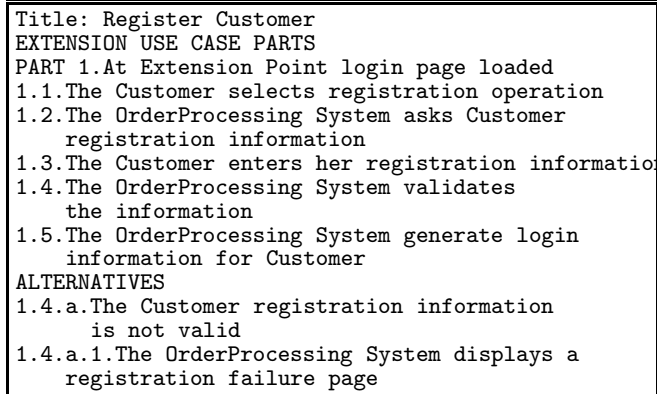
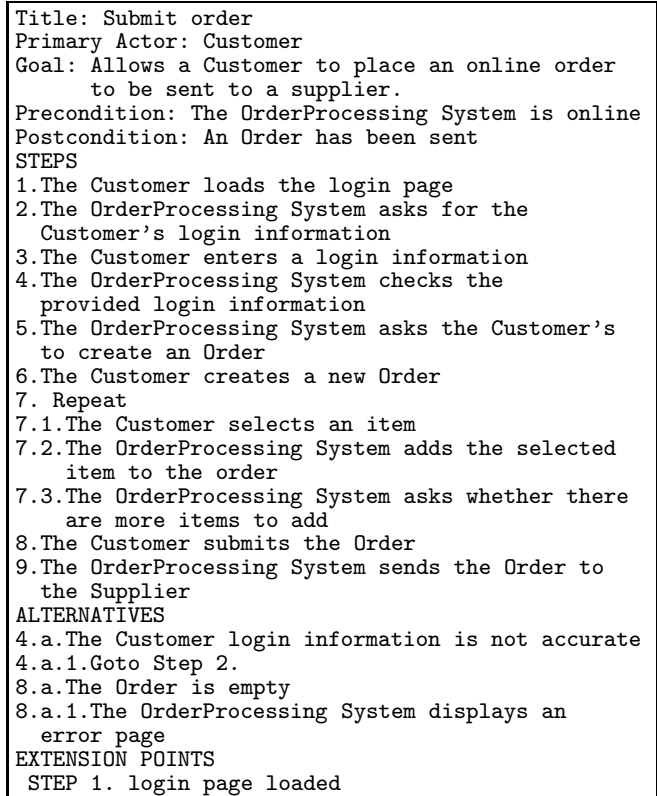


Figure 2: Details of use cases “Submit order” and “Register Customer”.

is a tuple $[Title, PrimaryActor, Goal, Precondition, Steps, Alternatives, ExtensionPoints]$. *Steps*, *Alternatives* and *ExtensionPoints* are sets of steps, alternatives and extension points respectively. Each step may correspond to a goto, a repeat, a use case inclusion or an operation initiated by an actor or the system. Each alternative is a triple $[StepRef, AltCond, Steps]$ with *StepRef* a possibly null reference to a step and *AltCond* a condition (possibly based on a timeout). Each extension point is a pair $[StepRef, ExtensionPLabel]$ consisting of a reference to a step and a label. A extend use case is a pair $[Title, Parts]$ with *Parts* a set of parts. Each part is a tuple $[ExtensionPLabel, Steps, Alternatives]$. A concrete natural language is used for operations and conditions in steps¹. Operations are written as active verb sentences

¹The description of the concrete natural language is beyond

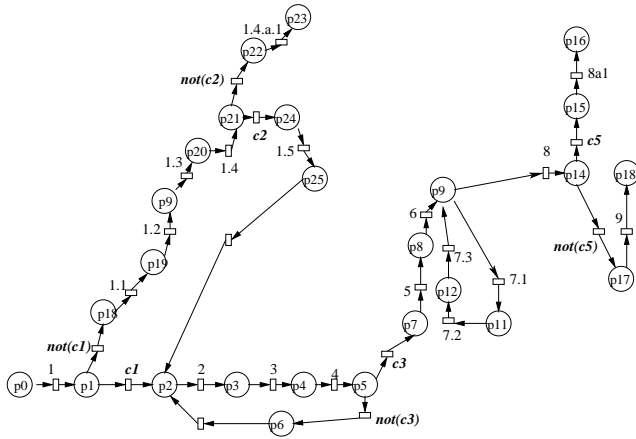


Figure 3: Petri nets corresponding to use case “Submit order” with extend use case “Register Customer”. We use the corresponding step numbers as labels for transitions. $c1$ is condition “Customer is registered”, $c2$ is condition “Customer registration is valid”, $c3$ is condition “Customer login information is accurate”, $c4$ is condition “Customer has more items to add” and $c5$ is condition “Order is empty”.

and conditions as predicative sentences. The UCed tool validates use cases against a *domain model* where domain entities including operations are defined [15]. A validated use case consists in object instantiated from the metamodel and links to operation instances in the domain.

Use cases execution semantics are expressed in the Petri nets formalism [13]. A Petri net is a triple $[P, T, F]$ with: P a finite set of places, T a finite set of transitions, $F \subseteq (P \times T) \cup (T \times P)$ a flow relation. Figure 3 shows a Petri net corresponding to use case “Submit order” extended with extend use case “Register Customer”. Use case semantics are the basis for a UML Statechart synthesis algorithm from use cases [16] implemented in UCed. Given a use case model, the algorithm consists first in the derivation of Petri nets corresponding to each use case with a consideration of $\ll include \gg$ and $\ll extend \gg$ relations. Subsequently, a global UML Statechart model that integrates all sequentially related use cases is created from the Petri nets [16].

3. USE CASES AND CONCERNS

A use case is a specification of a set of actions (behaviors) performed by a system in order to achieve a certain *goal* for actors or other stakeholders of the system. In order to fulfill its primary goal, a use case typically involves pieces of behavior related to other identifiable functional or non-functional *sub-goals*. These pieces of behavior are often invoked at several places in several use cases. For instance, the primary goal of use case “Submit order” shown in Figure 2 is expressed as “allows a Customer to place an online order to be sent to a Supplier”. For security reasons, it might be required that the Customer be redirected to a *timeout page* anytime he/she is prompted for an entry by the system and do not respond in a timely manner. Another re-

the scope of this paper. The interested reader is referred to [3, 15, 16].

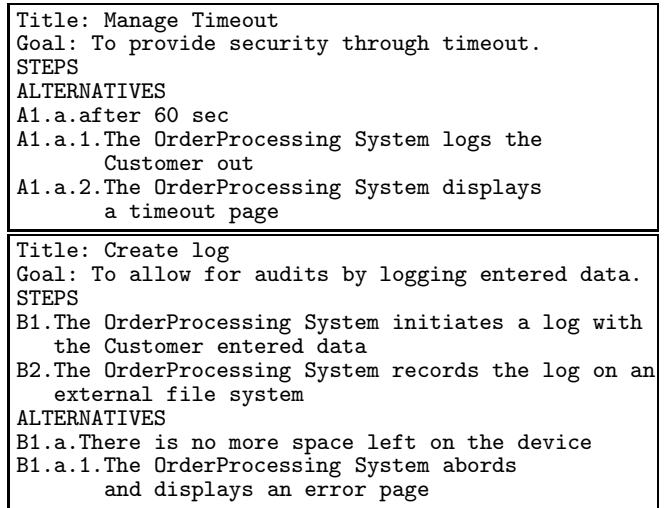


Figure 4: Use cases “Manage timeout” and “Create log”.

quirement related this time to auditing might ask for every entry provided to be *logged*. These two goals: “providing security through timeout” and “allowing for audits by logging” are concerns separate from the main goal of the use case. They are also likely to re-occur in different other use cases. Ideally, the different concerns should be kept separate for a better readability, maintainability and re-usability of use cases. Figure 4 shows two use cases capturing the *timeout* and *logging* goals.

The UML use case relations $\ll include \gg$, $\ll extend \gg$ and inheritance do not appear appropriate for modeling re-occurring concerns such as *timeout management* and *logging* as identified above.

- The $\ll include \gg$ relationship specifies an explicit inclusion of a use case as part of another use case. A use case inclusion is analogous to a subroutine call in a programming language. In the above example, the modeler would have to explicitly state “Manage timeout” inclusion after steps 2, 5, 7.3 and 1.2 of use cases “Submit order” and “Register Customer”. The same process would have to be repeated for every use case added to the model. This would result in concerns entanglement and negatively affect readability and maintainability.
- The $\ll extend \gg$ relation is based on an identification of *extension points* in an extended use case and references to these extension points in the relation (or the extending use case). In the above example, extension points would have to be created for steps 2, 5, 7.3 and 1.2 and referred to in use case “Manage timeout”. As observed in section 5, the need to explicitly define and relate to extension points has the effect of limiting flexibility and reusability.
- Inheritance is not appropriate here as the use cases do not satisfy a generality/specificity rule.

More generally, we can observe that as for AOP, *quantification* - the ability to specify locations for concern inclusion using quantified statements and *obliviousness* - the ability

to specify re-occurring concerns without changes to affected concerns [8] are desirable properties that can not be fully obtained with UML use case relationships². We propose an approach for the specification of crosscutting concerns more in line with *aspect-orientation*. This approach satisfies quantification and obliviousness better. Because crosscutting concerns apply across use case models, constructs for such concerns should allow to transparently refer to a set of affected use cases. A clear benefit to that is the ability to extend the use case model by adding use cases without the need to modify other use cases.

4. ASPECT-ORIENTED CONCERNS MODELING IN USE CASES

We propose to extend use case description with constructs for the expression and composition of crosscutting concerns. Some of our constructs are inspired from *AspectJ* [1], one of best known AOP approach. However, differently to *AspectJ*, we adopt a *symmetric* model where all concerns (including crosscutting concerns) may be extended. The relevant terminology for specifying aspects in the *AspectJ* approach include *advices*, *joinpoints* and *pointcuts*. In this section, we define advices, joinpoints and pointcuts in the context of use cases.

4.1 Advice use cases

We introduce the notion of *advice use cases* to capture crosscutting concerns. Any normal use case may be used as an advice use case. Additionally, we allow use cases that do not adhere strictly to normal use cases *well-formness* rules as stated in the UML Specification [12], to be used as advice use cases.

According to the UML Specification, a normal use case yields an observable result that is, typically, of value for one or more actors or other stakeholders of the system. Normal use cases are complete end-to-end behavior descriptions geared toward actors or other stakeholders goals. Any normal use case is initiated by an actor and should be meaningful on its own. Since crosscutting concerns often relate to sub-goals, we allow advice use cases that transgress normal use cases well-formness rules. An advice use case may not be meaningful on its own as it may need to be interpreted within the context of a base use case. We also allow advice use cases to be initiated by the system itself and to be incomplete. For instance, Figure 4 describes two advice use cases that would not qualify as normal use cases according to the UML Specification. Use case “Manage Timeout” has an empty set of Steps while use case “Create log” is initiated by the system. Alternative *A1.a.* defined in use case “Manage Timeout” is meaningful only when attached to a step defined in another use case.

Although advice use cases may deviate from normal use cases well-formness rules, there is no difference at the syntactic level. All advice use cases are described according to the abstract and concrete syntax introduced in Section 2. Having advices in the same notation as normal use cases emphasizes the fact that they express concerns as any other use case. As a consequence, advices could also be advised by other crosscutting concerns.

²Obliviousness is desirable at the description stage. However, all concerns need to be made explicit in a composite model. We propose Petri nets generation for that purpose.

<code><pointcut_expression></code>	<code>::=</code>	<code><step_pointcut></code> <code><operation_pointcut></code> <code><alternative_pointcut></code> <code><condition_pointcut></code>
<code><operation_pointcut></code>	<code>::=</code>	<code><pointcut_type></code> “operations” <code><operation_specs></code>
<code><operation_specs></code>	<code>::=</code>	<code><operation_spec></code> [“,” <code><operation_specs></code>]
<code><pointcut_type></code>	<code>::=</code>	“after” “before” “around” “concurrent”

Figure 5: BNF grammar excerpt for pointcut expressions.

4.2 Joinpoint model for use cases

Any element related to behavior description in a use case is a possible *joinpoint*. According to our abstract syntax, use case description elements include: *steps*, *operations*, *alternatives*, *extension points* and *conditions*. These elements are all possible joinpoints. It is noteworthy that some elements such as *steps* are specific to use cases and are therefore not well suited as joinpoints for crosscutting concerns over several use cases. Elements such as conditions and operations are better suited to that purpose. By using use case description elements as joinpoints, we satisfy obliviousness as no specific description needs to be added to use cases for crosscutting concerns.

4.3 Pointcut expression

Traditional AOP considers a computer program as a set of “instructions” which execute sequentially according to control structures. A pointcut expression in that context, specifies a set of joinpoints (program instructions) and how advices are composed relatively to these joinpoints. *AspectJ* distinguishes three ways for advice composition *before*, *after* or *around* a joinpoint. We draw similarities between computer programs and use cases. We formally interpret a use case description as a set of sequential “events” with some control flow [16]. Because of this similarity, we consider the same composition types (*before*, *after*, *around*) for use cases as those defined by *AspectJ*. Additionally, we define a type for the concurrent composition (*concurrent*).

Figure 5 shows an excerpt of a BNF grammar for pointcut expressions. The grammar as the remainder of this paper, focuses on pointcuts based on operation joinpoints. An operation specification (`<operation_spec>` in the grammar) may include wildcards (“*”). This allows to identify operation joinpoints using pattern matching. For instance, “Customer*” refers to all operations from entity “Customer”, “* ask*” refers to all operations from any entity with the word “ask” as part of the operation name. The use of wildcards provides a quantification mechanism that makes joinpoints identification more flexible in situations where a large set of joinpoints needs to be referred to or when the extend of joinpoints is unpredictable.

4.4 Aspect relation

We define an `<<aspect>>` relation to link advices to base use cases and to specify related pointcuts. More formally, an `<<aspect>>` relationship $uc_{adv} \times a_{cond} \times A_{pcuts} \times BaseUCs$ denotes that the advice use case uc_{adv} is weaved to base use cases $BaseUCs$ according to pointcuts A_{pcuts}

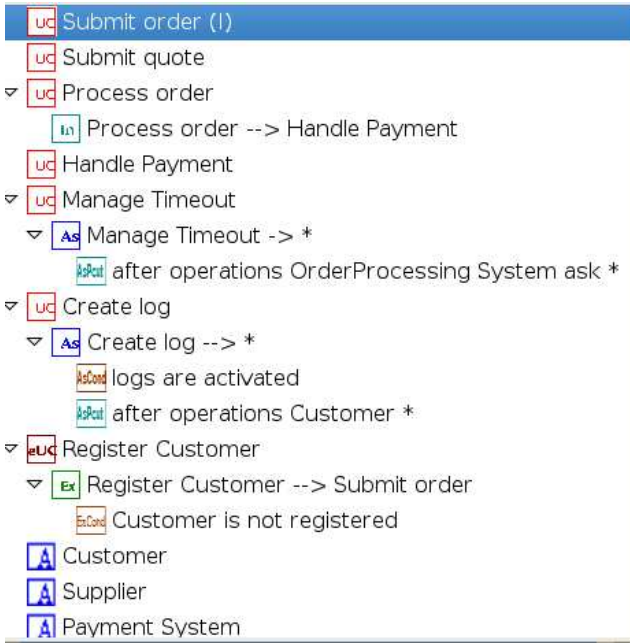


Figure 6: UCed representation of a use case diagram with `<<aspect>>` relations.

when condition *acond* holds. Notice that differently to `<<include>>` and `<<extend>>` relations, the `<<aspect>>` relation cardinality is one to many. This is a reflection to the fact that crosscutting concerns typically affect several use cases. The cardinality makes graphical UML use case diagrams poorly suited for `<<aspect>>` relationships. Linking advice use cases to each affected use cases would create cluttered diagrams. UCed “tree” representation provides a more convenient way to express `<<aspect>>` relationships. Figure 6 shows UCed rendering of the order processing use case model with `<<aspect>>` relations. The model includes two advice use cases “Manage Timeout” and “Create log”. The target use cases of an `<<aspect>>` relationship may be designated using a wildcard expression. For instance, “* order*” corresponds to a set of use cases which names include “order”. This expression would match use cases “Submit order” and “Process order” in the use case model in Figure 1. Use cases may also be matched based on description elements “Title”, “Primary Actor”, “Goal”, “Precondition” and “Postcondition”. For instance, expression “Primary Actor Customer” matches all use cases which Primary Actor as specified in the description is “Customer”.

In Figure 6, both advice use cases are linked to all the other use cases but themselves (an advice use case is always excluded from its set of related use cases to avoid infinite inclusion). “Manage Timeout” applies after each OrderProcessing operation with a name starting with “ask” while, “Create log” applies after every Customer’s operation. The later `<<aspect>>` relationship is constrained by condition “logs are activated”. Notice that the correctness of pointcut expressions in relation to requirements is based on the right identification of the affected operations. It is assumed here that system operations with verb “ask” correspond to prompts for data entry and that every Customer’s operation involves data entry. New use cases could be added

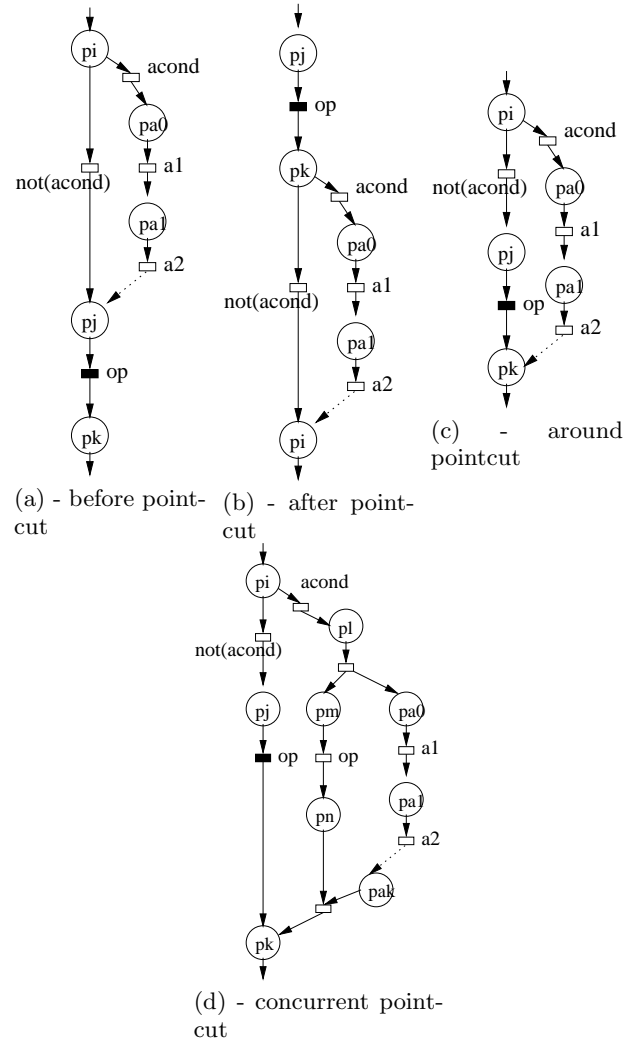


Figure 7: Weaving based on the different types of operation pointcut expressions.

to the model with no modification to advice use cases or `<<aspect>>` relationships as long as these use cases consistently refer to the same operations. UCed *domain model* helps in that respect as it is a central location where all operation in a use case model are defined. The domain model that serves as a dictionary for use cases validation is also useful to ensure a consistent use of operation names.

4.5 Mapping to Petri nets

The behavior defined in an advice use case is weaved with the behavior defined in an affected base use case at the Petri net level. Given an aspect relation $uc_{adv} \times acond \times Apcuts \times BaseUCs$, the weaving process augments the Petri nets corresponding to each use case in *BaseUCs* (obtained by applying the algorithm in [16]) according to the pointcut expressions in *Apcuts*.

Figure 7 shows the basic weaving of advice use cases according to the different types of operation pointcuts. Use cases to Petri nets mapping is such that each reference to an operation corresponds to a transition. We suppose *op* is the operation referred to as joinpoint in each pointcut ex-

capture non-functional crosscutting constraints. They are linked to base use cases using the `<<constrain>>` relationship. A similar approach is discussed in [17], where “*cross-cutting use cases*” are proposed to capture crosscutting concerns in use case models and a `<<crosscut>>` relationship used to link these concerns to base use cases. Information related to the composition of concerns is kept separately in a table. Both the `<<constrain>>` and `<<crosscut>>` relationship have a one to one cardinality. Additionally, use case step numbers are used as joinpoints in [17], which further limits flexibility as step numbers are specific to use cases and are susceptible to change when use cases are edited.

A language construct called *AspectU* is introduced in [14] to model crosscutting concerns in use cases. *AspectU* is proposed as an aspect-oriented extension at the use case level similar to *AspectJ* used at the code level. It defines an aspect entity, which includes pointcuts and advices. Pointcuts refer to named steps and extensions in use cases as joinpoints. Advices consist of steps and extensions specified to be weaved after, before or around joinpoints. In addition to *AspectU*, another construct called *AspectSD* is proposed at the sequence diagram level to enable the translation of *AspectU* constructs to *AspectJ*. *AspectSD* is used as an intermediate language to facilitate this transformation. Differently to [14], our approach is based on existing use case modeling constructs such as relationships and use cases. We believe the adoption of aspect based use case modeling would be easier with concepts familiar to use case modelers.

Several related scenario/use case based approaches including [11, 5, 9] differ from ours in the notation used for use cases. An extension to UML activity diagrams is used in [11]. In [5], aspectual scenarios are modeled using *Interaction Pattern Specifications* (IPSS) and non-aspectual scenarios are described using UML sequence diagrams. Finally, Use Case Maps (UCMs) are used in [9].

6. CONCLUSIONS

This paper presented an approach for modeling use case based requirements with aspect-oriented techniques. Cross-cutting requirements are modeled in addition to functional requirements by extending traditional use case model with a new relationship stereotyped as `<<aspect>>`. In order to improve flexibility, the cardinality of the `<<aspect>>` relationship is one to many, joinpoints are use case description elements and quantification is used to identify use cases and joinpoints. Use cases are composed with aspect use cases by a transformation to Petri nets and UML state machines. The whole approach is automated and tool supported by adding extensions to UCED, an existing use case modeling tool. UCED includes facilities for simulation of generated state machines and test generation. The simulation of a use case model with aspectual components allows early validation of the interaction of crosscutting concerns. Test cases derived from state models that integrate concerns ensure a broad coverage of the system’s requirements.

Four types of composition (before, after, around and concurrent) are considered in this work, we will investigate other types of composition in our future work. We will also consider more elaborate joinpoints than simple use case elements. For instance, some concerns may only apply based on the occurrence of certain sequences of operations. Finally, we plan to seek a convenient way for integrating the `<<aspect>>` relationship to UML graphical use case dia-

grams. One solution that will be explored is the use of use case generalizations.

7. REFERENCES

- [1] AspectJ Project. <http://www.eclipse.org/aspectj/>.
- [2] Petri nets tools database. <http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html>.
- [3] Use Case Editor (UCED) toolset. http://www.site.uottawa.ca/~ssome/Use_Case_Editor_UCED.html.
- [4] J. Araújo and A. M. D. Moreira. An Aspectual Use-Case Driven Approach. In *VIII Jornadas Ingeniería del Software y Bases de Datos (JISBD 2003)*, pages 463–468, 2003.
- [5] J. Araújo, J. Whittle, and D.-K. Kim. Modeling and Composing Scenario-Based Requirements with Aspects. In *Proceedings of the 12th IEEE International Requirements Engineering Conference (RE’04)*, 2004.
- [6] R. Chitchyan, A. Rashid, P. Rayson, and R. Waters. Semantics-based composition for aspect-oriented requirements engineering. In *AOSD ’07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 36–48. ACM, 2007.
- [7] R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M. P. Alarco, J. Bakker, B. Tekinerdogan, S. Clarke, and A. Jackson. Survey of Analysis and Design Approaches. AOSD-Europe-ULANC-9, May 2005.
- [8] R. Filman and D. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. Workshop on Advanced Separation of Concerns, OOPSLA 2000, October 2000.
- [9] M. Gunter, A. Daniel, and W. Michael. Visualizing aspect-oriented requirements scenarios with use case maps. In *First International Workshop on Requirements Engineering Visualization*, 2006.
- [10] I. Jacobson and P.-W. Ng. *Aspect-Oriented Software Development with Use Cases*. Addison Wesley, 2005.
- [11] A. M. D. Moreira and J. Araújo. Handling unanticipated requirements change with aspects. In *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE’2004)*, pages 411–415, 2004.
- [12] OMG. UML 2.0 Superstructure, 2003. Object Management Group.
- [13] C. A. Petri. *Communication with Automata*. PhD thesis, Technische Universität Darmstadt, 1962.
- [14] J. Sillito, C. Dutchyn, A. Eisenberg, and K. DeVolder. Use Case Level Pointcuts. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP’04)*, 2004.
- [15] S. Somé. Supporting Use Cases based Requirements Engineering. *Information and Software Technology*, 48(1):43–58, 2006.
- [16] S. S. Somé. Petri Nets Based Formalization of Textual Use Cases. Technical Report TR-2007-11, SITE, University of Ottawa, 2007.
- [17] G. Sousa, S. Soares, P. Borba, and J. Castro. Separation of crosscutting concerns from requirements to design: Adapting the use case driven approach. In *Proceedings of the Early Aspects Workshop at AOSD 2004*, 2004.