Visualizing Early Aspects with Use Case Maps

Gunter Mussbacher[1], Daniel Amyot[1], Michael Weiss[2]

[1] SITE, University of Ottawa, 800 King Edward, Ottawa, ON, K1N 6N5, Canada
{gunterm, damyot}@site.uottawa.ca
[2] School of Computer Science, Carleton University, 1125 Colonel By Drive,
Ottawa, ON, K1S 5B6, Canada
weiss@scs.carleton.ca

Abstract. Once aspects have been identified during requirements engineering activities, the behaviour, structure, and pointcut expressions of aspects need to be modeled unobtrusively at the requirements level, allowing the engineer to seamlessly focus either on the behaviour and structure of the system without aspects or the combined behaviour and structure. Furthermore, the modeling techniques for aspects should be the *same* as for the base system, ensuring that the engineer continues to work with familiar models. This paper describes how, with the help of Use Case Maps, scenario-based aspects can be modeled at the requirements level unobtrusively and with the same techniques as for non-aspectual systems. Use Case Maps are a visual scenario notation under standardization by the International Telecommunication Union. With Use Case Maps, aspects as well as pointcut expressions are modeled in a visual way which is generally considered the preferred choice for models of a high level of abstraction.

Keywords: Aspect-Oriented Requirements Engineering, Use Case Maps, Scenario Notations, User Requirements Notation.

1 Introduction

Aspects [26] expand object-oriented software development by adding means for encapsulating requirements-level concerns (also called requirements units, e.g. features, use cases, etc). Generally, such encapsulation cannot be achieved with classes/objects alone as "the units of interest in the requirements domain are fundamentally different from the units of interest in object-oriented software" [21]. In object-oriented software, this manifests itself in scattering (parts of a concern are scattered over many classes) and tangling (one class contains parts of many different concerns). Aspects address these problems (see [25] for an example on how use cases can be encapsulated throughout the whole software development process). Note that although this paper focuses on object-oriented techniques, similar arguments in support of aspects can be made for other development paradigms. As Use Case Maps (UCMs) also have been used in the context of various software development paradigms (see section 2.1 for examples), aspect-oriented UCMs (AoUCM) are applicable to a wide variety of development paradigms.

Use Case Maps (UCMs) [38] are part of the User Requirements Notation (URN) [39], a standardization effort of the International Telecommunication Union (ITU). The Use Case Maps notation is a visual scenario language that focuses on the causal flow of behavior superimposed on a structure of components. UCMs depict the interaction of architectural entities while abstracting from message and data details. UCMs have been used to drive performance analysis, scenario interaction detection, testing activities, and the evaluation of architectural alternatives at a very early stage in the software development process. In addition to UCMs, URN contains a second language for goal modeling and the description of non-functional requirements (GRL – the Goal-oriented Requirement Language [37]), making it the first standardization effort to address non-functional requirements explicitly in a graphical way.

This paper describes the first step in unifying URN concepts and aspects concepts, both on a notational level as well as on a process level, in order to take advantage of mutual benefits. Aspects can improve the modularity, compositionality, reusability, scalability, and maintainability of URN models. Considering the strong overlap between non-functional requirements and concerns encapsulated by aspects, aspects can help bridge the gap between goals (non-functional requirements described with GRL) and operational scenarios (the UCM scenarios which describe how a goal is achieved). On the other hand, aspects can benefit from a standardized way of modeling non-functional requirements (and therefore concerns) with URN.

The aforementioned first step endeavors to unify UCMs and aspects by using existing notational elements of UCMs to describe aspect-oriented concepts. Aligning UCM and aspect concepts makes it possible to visually describe aspect-oriented models with UCM models (as long as the aspect models are scenario-based). Furthermore, it allows aspect concepts to be used as first class modeling elements when building UCM models.

The remainder of the paper gives an overview of UCMs in section 2.1 and an overview of modeling techniques for early aspects in section 2.3. Section 3 describes how aspects can be modeled with UCMs at the requirements level and explains extensions to the URN metamodel in order to accommodate aspect-oriented UCMs. Section 4.1 discusses a matching algorithm for aspect-oriented UCMs and section 4.2 a composition algorithm for aspect-oriented UCMs. Section 4.4 talks about additional features and tool support for them. Throughout the paper, a hotel reservation system is used as an example to illustrate standard UCMs in section 2.2, aspect-oriented UCMs in section 3.5, and the composed system in section 4.3. The paper ends with a conclusion and a discussion of future work in section 5.

2 Background

This section provides an overview of Use Case Maps and modeling techniques for scenario/use case-based approaches to aspect-oriented requirements engineering.

2.1 Use Case Maps

Use Case Maps (UCMs) [38] are an integral part of the International Telecommunication Union's (ITU) effort to standardize the *User Requirements Notation* (URN) [39]. UCMs are a visual scenario notation for the description of functional requirements and, if desired, high-level design. Paths describe the causal flow of behavior of a system (e.g. one or many use cases). Optionally, paths are superimposed over components which represent the architectural structure of a system (e.g. classes or packages). UCMs abstract from the details of message exchange and communication infrastructures while still showing the interaction between architectural entities. As UCMs integrate many scenarios and use cases into one combined model of a system, it is possible to reason about undesired interactions between scenarios [4], analyze performance implications [34,35], and drive testing efforts based on UCM specifications [6]. As UCMs show architectural structures, various architectural alternatives can be analyzed [7,8,40]. Over the last decade, UCMs have successfully been used for service-oriented, concurrent, distributed, and reactive systems such as telecommunications systems [3,8], e-commerce systems [5], agent systems [23], operating systems [16], and health information systems [1]. UCMs have also been used for business process modeling [40].

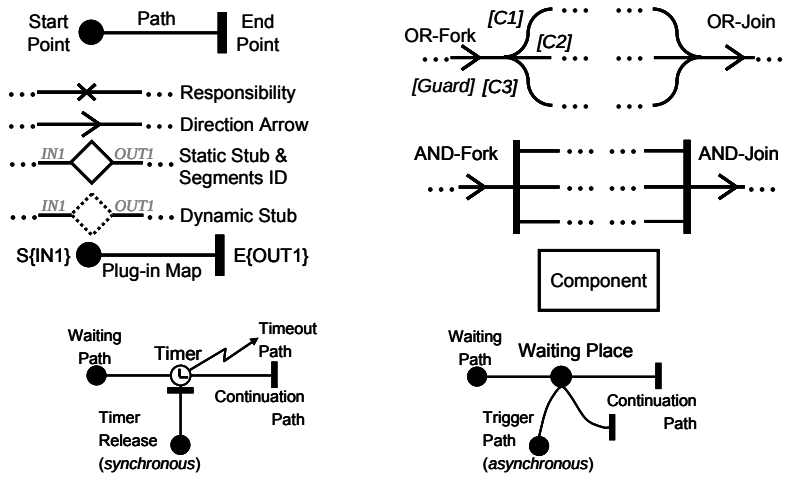

Fig. 1. Basic Elements of UCM Notation

The basic elements of the UCM notation are shown in Fig. 1. A *map* contains any number of *paths* and structural elements (*components*). Paths express causal sequences and may contain several types of path nodes. *Responsibilities* describe required actions or steps to fulfill a scenario. *OR-forks* (possibly including guarding *conditions*) and *OR-joins* are used to show alternatives, while *AND-forks* and *AND-joins* depict concurrency. Loops can be modeled implicitly with OR-joins and OR-forks. As the UCM notation does not impose any nesting constraints, joins and forks can be freely combined and a fork does not need to be followed by a join.

UCM models can be decomposed using *stubs* which contain sub-maps called *plug-ins*. Plug-in maps are reusable units of behavior and structure. *Plug-in bindings* connect in-paths and out-paths of stubs with start and end points of a plug-in map, respectively (see Fig. 2 (a) for an example; note that the arrows have been added to this and following figures to clearly indicate plug-in bindings for the UCM model – UCM editing tools do not display such arrows but manage plug-in bindings much more concisely). Stubs without plug-in bindings for start or end points can be shown visually without an in-path or out-path, respectively (see Fig. 2 (b)). A stub may be *static* which means that it can have at most one plug-in, whereas a *dynamic* stub may have many plug-ins which may be selected at runtime. A *selection policy* decides which plug-ins of a dynamic stub to choose at runtime.

Map elements which reside inside a component are said to be *bound* to the component. Components have various types and characteristics (not discussed in this paper) and can contain sub-components.

Other notational elements of UCMs are *timers* and *waiting places*. A timer may have a *timeout path* which is indicated by a zigzag line. A waiting place denotes a location on the path where the scenario stops until a condition is satisfied. If an endpoint is connected to a waiting place or a timer, the stopped scenario continues when this end point is reached (synchronous interaction). Asynchronous, in-passing triggering of waiting places and timers is also possible. End points can also be connected to start points as shown in Fig. 2 (c) to indicate simple sequences of paths. A more complete coverage of the notation elements is available in [17,19,38].

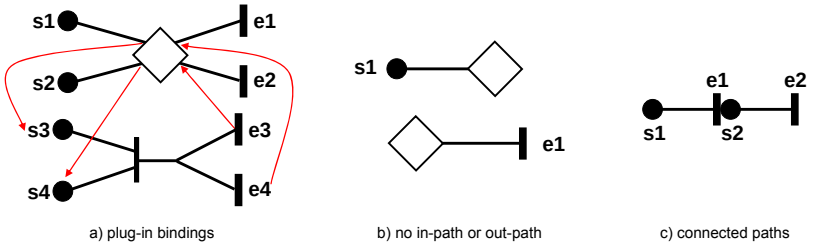

Fig. 2. Connecting Stubs, Plug-in Maps, and Paths

UCMs also support the definition of scenarios. For each choice point in the UCM model (e.g. an OR-fork), Boolean expressions are defined for all alternatives. A scenario describes a specific path through the UCM model (only one alternative at a choice point is taken) by initializing the variables used in the Boolean expressions. For each scenario, its pre-conditions and post-conditions are additionally specified as well as its start points and expected end points.

UCMs share many characteristics with UML activity diagrams but UCMs offer more flexibility in how sub-diagrams can be connected, how sub-components can be represented, and how dynamic responsibilities and dynamic components (not shown here) can be used to capture requirements for agent systems. UCMs also integrate a simple data model, performance annotations, and a simple action language used for analysis. However, activity diagrams have better support for object flows and a better integration with the rest of UML.

On the other hand, UCMs are integrated with goal-oriented models described with the *Goal-oriented Requirement Language* (GRL), the second modeling notation in the URN. With URN, the goals of stakeholders and the system are modeled on GRL graphs, showing the impact of often conflicting goals and various alternative solutions proposed to achieve the goals. Softgoals (which relate to non-functional requirements), hard goals (which relate to functional requirements), and solutions impact each other in either a negative or positive way. The solutions from the GRL graphs are refined by the UCM model which provides the structural and behavioral details of scenarios for the solution.

Two editing tools for UCMs have been developed at Carleton University and the University of Ottawa. The tools make it possible to create, maintain, analyze, and transform UCM models. The original UCM Navigator (UCMNAV) [27] is only a UCM tool whereas the new Eclipse-based jUCMNav [33] is a true URN tool that offers GRL modeling in addition to UCM modeling.

2.2 Use Case Maps Example

This section discusses a simplified example which further illustrates the UCM notation. Given the following requirements, the basic UCM model in Fig. 3 can be created consisting of two use cases:

R1. Users shall be able to make reservations.
R2. System shall authenticate users before any access to a reservation (such as making, canceling, ...).
R3. System shall add failed reservation attempts to a waiting list from which they will be taken when another reservation is canceled in the future.

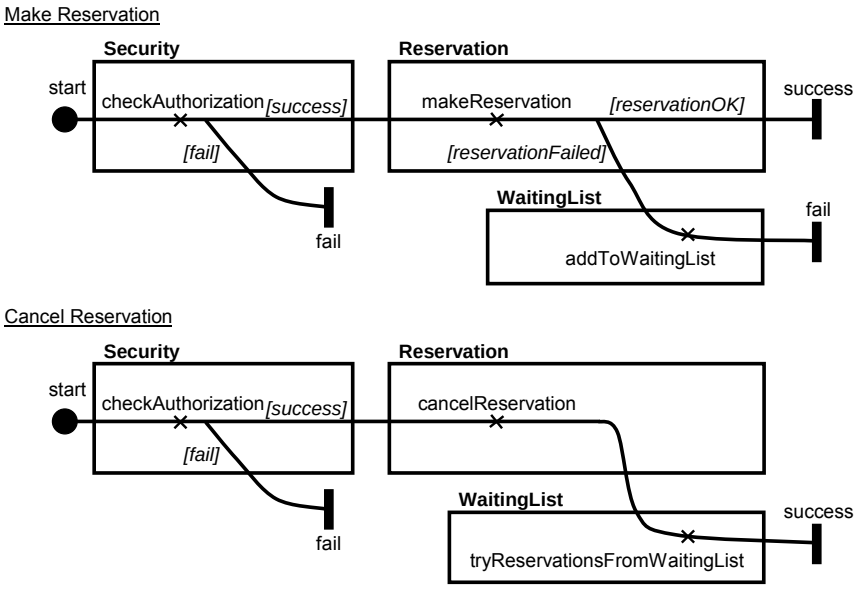

Fig. 3. Basic UCM Reservation Use Cases

In the make reservation use case, the system first checks for authorization and only if the user is authorized is the reservation made. If the reservation is not successful, the reservation is added to a waiting list. In the cancel reservation use case, the system again checks for authorization and only if the user is authorized is the cancellation performed. After the cancellation, the system tries to fulfill a reservation from the waiting list. Note that the components in this case correspond to objects, but this is just one of the possible interpretations of a UCM component. Considering that there exist other non-functional requirements (such as security) that need to be added to the use cases and considering that there exist other features that interact with making and canceling reservations (such as waiting lists) and that need to be added to the use cases, there is clearly a need for better modularization. A more advanced UCM model which extracts the common behavior into a plug-in map is shown in Fig. 4.

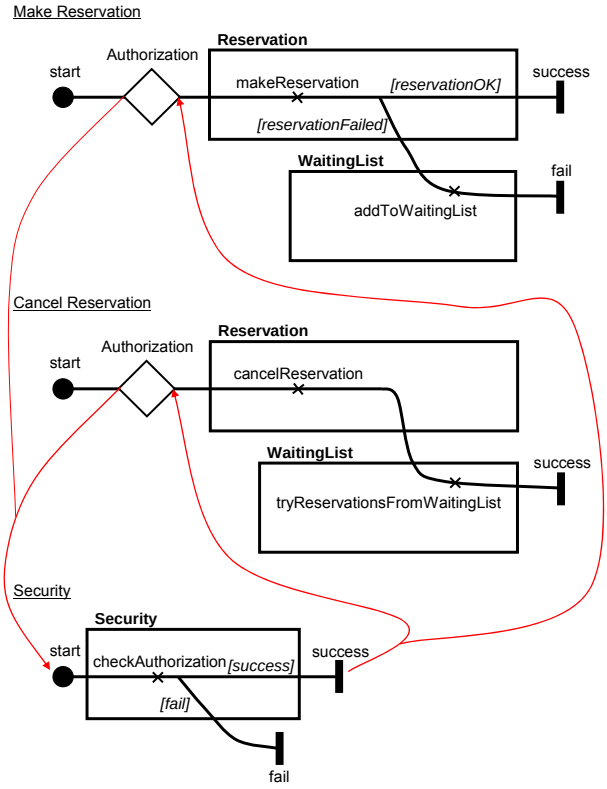

Fig. 4. Advanced UCM Reservation Use Cases – First Attempt

Even though stubs structure the UCM model considerably better, the cancel and make reservation use cases still include descriptions of non-related behavior (the stubs related to authorization and the responsibilities related to waiting lists). A further improvement of the UCM model in Fig. 4 is shown in Fig. 5. Now, each individual plug-in map only deals with one concern. Top-level maps (often called root maps) describe how the different concerns are composed together. However, root maps often

turn out to be rather complicated. Scalability is also an issue as, for example, the Authorization stub has to be added explicitly to each root map. Furthermore, the description of making a reservation is in one plug-in map only because no other use case exists that needs to be interleaved with making a reservation. Fig. 5 only shows the situation where behavior has to be added before or after making and canceling reservations. Often however, use cases are interleaved, causing the behavior for making a reservation to be split up into several maps (see Fig. 6 - the disjoint plug-in maps in Make Reservation Part 1 and Part 2 only contain parts of the behavior for making a reservation). This makes it much harder to understand and maintain individual use cases. Aspect-oriented UCMs (AoUCM) address these problems.

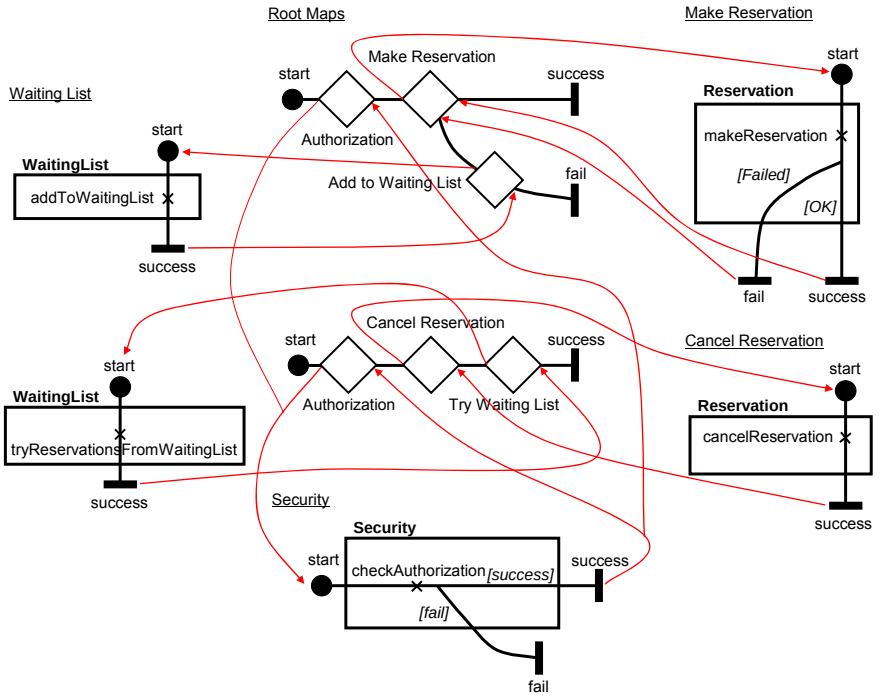

Fig. 5. Advanced UCM Reservation Use Cases – Second Attempt

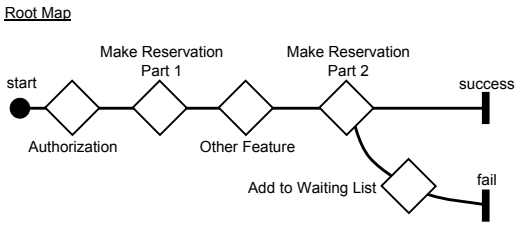

Fig. 6. Description of Make Reservation is spread out over multiple disjoint plug-in maps

2.3 Modeling Techniques for Aspect-Oriented Requirements

About a decade ago, *aspect-oriented programming* (AOP) [26] introduced a new way of structuring software systems. With this new modularization, it is possible to address problems of object-oriented software engineering that occur because the units of interest to the requirements engineer cannot readily be encapsulated with object-oriented units [21]. This results in *scattering* (parts of a requirements unit are scattered over many classes) and *tangling* (one class contains parts of many different requirements units). This problem has also been referred to as the tyranny of the dominant decomposition [36], as a chosen modularization technique (e.g. objects) inevitably will cause unwanted side-effects in the software design (e.g. scattering and tangling). The UCM model in Fig. 5 exhibits signs of scattering and tangling. The Authorization stub is scattered over multiple root maps and various concerns are tangled in each of the root maps. Examples for requirements units (or *concerns*) for which aspects provide a better encapsulation than objects are authorization/authentication, caching, concurrency management, debugging, distribution, logging, testing, transaction management, or even a feature or use case [25].

Initially, aspect-orientation focused on the implementation level leading to an ever-growing number of tools that provide extensions for aspects to major programming languages [9,10]. Essentially, aspects identify locations in a program (called *joinpoints*) through parameterized expressions (called *pointcuts*). Aspects also specify behavior (called *advice*) which will be inserted into the specified locations. As advices may change the behavior of already existing structural entities thus violating proper object-oriented modularization, entities external to an aspect are referenced in a structured way (with the help of *intertype declarations*). Note that we are using AspectJ terms [14] but that the concepts also apply to other flavors of aspect-oriented programming.

Aspect-oriented modeling (AOM) or *early aspects* aims to apply aspect-oriented concepts earlier in the software development life cycle in order to manage more effectively concerns at the requirements and architecture stages. Many approaches to *aspect-oriented requirements engineering* (AORE) are described in a recent survey [20], grouped into viewpoint, goal, scenario/use case, concern, and component-based approaches. As our technique is more closely related to the group of scenario/use case-based approaches, we will briefly review this group in more detail.

In *Aspect-Oriented Software Development (AOSD) with Use Cases* [25], Jacobson and Ng view a well-written use case as a concern and add the notion of *pointcuts* to the traditional use case approach. Pointcuts in one use case reference extension points in other use cases in a textual way. The traditional usage of extension points is slightly altered as they do not directly reference a use case anymore but only identify a step in the use case where an extension may occur. Furthermore, a new kind of use case, the *infrastructure use case*, is used in addition to the traditional *application use cases*. It describes scenarios required to address non-functional requirements. Infrastructure use cases do not reference other use cases directly but pointcuts in a generic *perform transaction use case*. This particular use case models abstractly all types of interactions of an actor with the system, providing a generic description of

the system. The perform transaction use case is eventually mapped to traditional use cases, effectively weaving aspect behaviour described by infrastructure use cases into application use cases.

In *Scenario Modeling with Aspects* [41], Whittle and Araújo use UML sequence diagrams to describe non-aspectual scenarios and *interaction pattern specifications* (IPS) to describe aspectual scenarios. IPS are very similar to sequence diagrams but allow the definition of roles for classifiers, messages, and parameters. Binding the roles in IPS to elements of sequence diagrams produces a composed system, which is then translated into state machines for validation. Alternatively, the sequence diagrams and IPS are first both translated into state machine representations (finite state machines and *state machine pattern specifications* (SMPS), respectively) and then composed together at the state machine level [13] with the same binding technique. SMPS are very similar to state machines but also contain roles identified in IPS. In both cases, the binding is specified textually and identifies explicitly elements to be bound. On one hand, this allows for a very flexible composition. On the other hand however, the explicit binding may cause problems with scalability.

The *Aspectual Use Case Driven Approach* is the third major research direction discussed for scenario/use case-based approaches in [20]. Moreira *et al.* [12,29] propose to add extensions to UML use case and sequence diagrams in order to visualize how crosscutting non-functional requirements are linked to functional requirements expressed by use case diagrams or sequence diagrams. Non-functional requirements are captured with the help of templates. [28] builds on this work, extending the set of use case relationships to include “constrain”, “collaborate”, and “damage” relationships and making use of *activity pattern specifications* (APS). The new relationships describe how one use case impacts another (restricting it, contributing positively to it, or contributing negatively to it). APS extend UML activity diagrams by allowing the specification of roles similar to IPS [41] and SMPS [13]. APS are used to describe use cases in more detail. Various activity diagrams are composed by composition rules which are similar to the binding in [13,41].

Rashid *et al.* [32] describe an approach for conflict identification and resolution for aspectual requirements. While the example presented in [32] uses a viewpoint-based approach to requirements engineering, it is argued that the technique can also be applied to other requirements engineering approaches including scenarios/use cases. The topic matter, however, is orthogonal to our proposed technique.

Araújo and Coutinho [11] discuss aspects in a viewpoint-based requirements engineering approach that also includes use cases. Non-functional requirements (defined with templates) and use cases are linked to viewpoints. Use cases that are included by or extend more than one use case or that crosscut several viewpoints are called *aspectual use cases*. This approach does not discuss composition of aspectual use cases with other use cases but focuses more on how to extend the work in [32] for conflict resolution.

Barros and Gomes [15] apply aspect-orientation to UML activity diagrams. The approach is based on an additional composition operation called *activity addition* which allows the fusing of stereotyped nodes in one activity diagram with nodes in another. Stereotyping is effectively used as a pointcut expression, identifying

explicitly nodes in another activity diagram for behavior merging. Zdun and Strembeck [42] extend UML activity diagrams with nodes for start and end points of aspects in order to visualize aspects in a composed system.

In the UCM community, the applicability of UCMs to model aspects was identified very early on by Buhr [18] but received little attention since then with the exception of de Bruin and van Vliet [22]. The approach by de Bruin and van Vliet adds *Pre stubs* and *Post stubs* for each location on a UCM that requires a change. The stubs allow behaviour to be added before or after the location by plugging *refinement maps* into the stubs. Components on UCMs are identified by a *Name:Type pair*. A refinement map can be placed in a Pre or Post stub only if the component type on the refinement map matches the component type to which the Pre or Post stub is bound.

Defining and representing aspects must consider a number of factors. It should be easy to switch from traditional modeling to aspect-oriented modeling. Preferably, there should not be a difference and the same modeling language should be used for both in order to avoid having to learn yet another modeling language. It should be possible to define aspects without influencing the base model. This is a crucial point of aspect-orientation as the base model must not be polluted by aspect-specific information. Breaking the modeling paradigm is best avoided, and therefore aspects, including advice, pointcut expressions, and intertype declarations should be modeled using the same modeling paradigm (e.g. without the use of graphical and purely textual representations at the same time). Pointcut expressions should be parameterized to avoid scalability issues. For scenario/use case-based aspect techniques to be effective at the requirements phase, the employed technique should be at the right abstraction level where message or data details of interactions are irrelevant. The composition technique should be flexible and exhaustive in that it allows all frequently encountered compositions to be expressed.

None of the scenario/use case-based approaches to aspect-oriented requirements engineering mentioned in this section excels in all of these factors (see section 5 for a discussion on this). Aspect-oriented UCMs (AoUCM) aim to address this shortcoming.

Furthermore, the group of goal-based approaches to aspect-oriented requirements engineering is also of interest in the context of an aspect-oriented URN, e.g. work on aspects and the *i** framework [2]. As GRL borrows many concepts from the i* framework, we believe that aspects can also be added to the GRL part of URN and that there is a need to synchronize aspect-oriented GRL models with aspect-oriented scenario models expressed with UCM. Gross and Yu [24] present an initial attempt at adding aspects to GRL. They introduce the concept of an intentional aspect as an abstraction for the design of aspects, the composition of actors and aspects, and linking aspects to implementation artefacts. Actors and intentional aspects encapsulate goals and alternative solutions of system modules and aspects, respectively. When actors and aspects are composed, their contributions to common non-functional requirements are merged. The main objective of the approach is to provide traceability between aspects and goals, and aspects and implementation artefacts. However, composition of actors and aspects is not presently supported by tools. Alencar *et al.* [2] suggest three rules to identify crosscutting concerns in i* models. Based on the findings, the i* model is restructured in an aspect-oriented way using a new notational element for aspects (a star). Both proposals are limited to

adding aspects to goal models, and do not consider linking goal models to scenario models such as UCM models.

3 Aspect-Oriented Use Case Maps (AoUCM)

In order to unify aspect concepts with UCM concepts, the first step is to define a joinpoint model for UCMs. Furthermore, advice, intertype declarations, and pointcuts have to be defined in UCM terms. We will approach these tasks initially in an informal way. At the end of this section, however, the concepts mentioned above will be defined precisely in the URN metamodel. While previous work [30] reported only on early results, the remaining sections of this paper give a much more in-depth introduction to aspect-oriented UCMs (AoUCM). In summary, the following questions need to be answered:

- What are good joinpoints in UCMs?
- How is advice specified for an aspect?
- How are intertype declarations specified for an aspect?
- How are pointcuts specified and how are pointcuts and advice linked?

3.1 Joinpoint Model

In aspect-oriented programming, a joinpoint is a point in the dynamic flow of the program such as a method call, the execution of a method, the initialization of an object, set methods, get methods, or exception handling. Hence, a joinpoint has a behavioral dimension as well as a structural dimension. For use cases, Jacobson and Ng define a joinpoint as a step in a flow of the use case. In UCM terms, this translates directly into responsibilities (behavioral dimension) optionally bound to components (structural dimension). Responsibilities are just one kind of path node. Therefore, any path node could be a joinpoint (i.e. a location on the path). Defining any path node as a joinpoint gives the most flexibility to the requirements engineer without significantly increasing the complexity of AoUCM or requiring additional modeling constructs. Fig. 7 shows UCM path nodes such as start points, dynamic and static stubs, responsibilities, OR-forks and OR-joins, AND-forks and AND-joins, waiting places, timers, and end points. Each of these path nodes is a joinpoint. The small diamonds in Fig. 7 do not indicate joinpoints but insertion points. An aspect can insert behavior at an insertion point, i.e. either before or after joinpoints. An insertion point is associated with exactly one joinpoint. Some joinpoints such as start and end points can only have two insertion points, one before and one after the joinpoint. Other joinpoints such as stubs, forks, and joins can have more than two insertion points. The component in Fig. 7 indicates that the joinpoint model does not concern itself only with behavioral specification but optionally also with structural specifications.

The small light-shaded diamonds indicate that more than one insertion point exist *before* or *after* the path node (i.e. joinpoint). Note that even though the figure only shows at the most three insertion points for one joinpoint, any joinpoint with three insertion points may also have more than three insertion points. The joinpoint model,

however, does not need to concern itself with the number of insertion points. The joinpoint model simply identifies all path nodes as joinpoints. We will see in section 3.4 how it is possible to reference each of these insertion points individually.

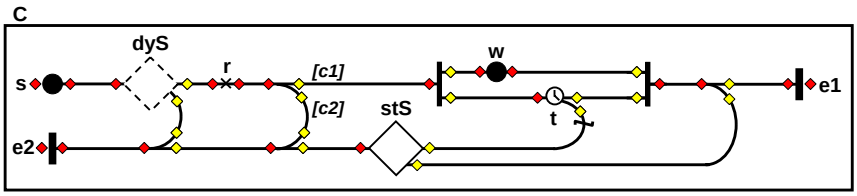

Fig. 7. Joinpoints and Insertion Points in UCMs

3.2 Advice Map

An advice describes the behavior of an aspect triggered in a certain situation. Intertype declarations identify the structural entities that either provide the advice or contribute to the advice. Describing behavior in UCMs is straightforward as UCMs are meant to do exactly that: describe behavior with paths on top of a structure of components. The semantic meaning of components in UCMs is cast very wide – anything that can provide a service is a component from classes to actors to roles. Therefore, structural entities identified by intertype declarations can certainly be described with UCM components. The resulting map is called *advice map*. An advice map does not differ syntactically from a non-aspectual map. Both describe behavior with a path, and both use components to indicate who is responsible for providing the behavior. The difference manifests itself in terms of how this map fits into the overall system. This is explained in section 3.4.

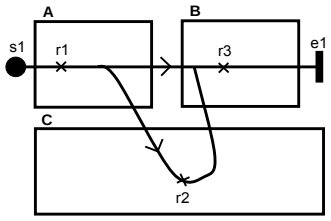

Fig. 8. Advice Map with Path and Components

Fig. 8 shows the description of an advice that requires r1 from component A, responsibility r2 from component C, and responsibility r3 from component B. All three components are contributing to the advice. There are three cases:

a. An advice may use an already existing responsibility.
b. An advice may add an aspect-specific responsibility to an already existing component.
c. An advice may define an aspect-specific responsibility for an aspect-specific component.

These three cases can of course be automatically identified by looking at the usage of the responsibilities and components in the whole UCM model, but there is no standard visual representation that indicates which case applies. Therefore, the differentiation between these three cases is a matter of naming conventions. For example, A could be called A (Shared) to indicate case (a), B could be called B (Extended) to indicate case (b), and C could be called C (Owned) to indicate case (c).

3.3 Pointcut Map

A pointcut defines a set of joinpoints either explicitly or in a parameterized way, thereby defining the structural context and behavioral context for the execution of an advice. In aspect-oriented programming, pointcuts can be rather lengthy and complex multi-line expressions. In aspect-oriented use case modeling, pointcuts reference extension points in a different use case. Both use text as the means to describe pointcuts. Considering the visual character of UCMs, a visual representation for pointcuts is a natural choice. More importantly, using a visual representation instead of defining separate textual representations of pointcuts avoids a modeling paradigm break. Therefore, pointcuts are defined on a UCM called *pointcut map* (see Fig. 10 and Fig. 11 for examples).

On first look, there is no difference between a pointcut map and other UCMs. Looking more closely, however, the pointcut map represents a partial map which identifies joinpoints when matched against all other maps in the UCM model. The parameterization of pointcuts is achieved by *wildcards* in the names of UCM elements on the pointcut map. Any of the named elements on a pointcut map may contain the wildcard * or logical expressions. The pointcut expressions in Fig. 9 match against a) all start points starting with s, b) all responsibilities, c) all waiting places named ready or starting with w, and d) all components starting with A. These examples are not complete pointcut maps – in fact, the first three examples are not even valid UCMs. They only illustrate expressions which may be used on a pointcut map. Fig. 10 and Fig. 11 show complete pointcut maps.

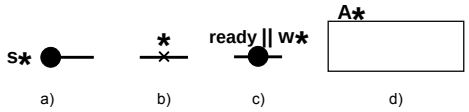

Fig. 9. Four Examples of Visual Pointcut Expressions

As mentioned previously, pointcut maps are partial maps. This is evident in the usage of start and end points on the pointcut map. Start or end points without a name denote only the start or end of the partial map and are therefore not matched against start or end points on other maps in the UCM model (see gray path nodes in Fig. 10). For example, Fig. 10 (a) matches against all maps with a responsibility r. Fig. 10 (b) matches against all maps with a start point s, followed by responsibility r, and followed by an end point e. Fig. 10 (c), on the other hand, matches against all maps with a start point s followed by responsibility r.

Note that it is possible to use unnamed start and end points as markers for the start and end of the pointcut expression because the usage of unnamed start and end points in standard UCMs is strongly discouraged. Scenario definitions and plug-in bindings also prefer named start and end points.

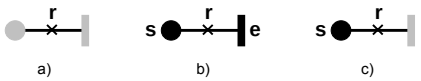

Fig. 10. Three Pointcut Maps

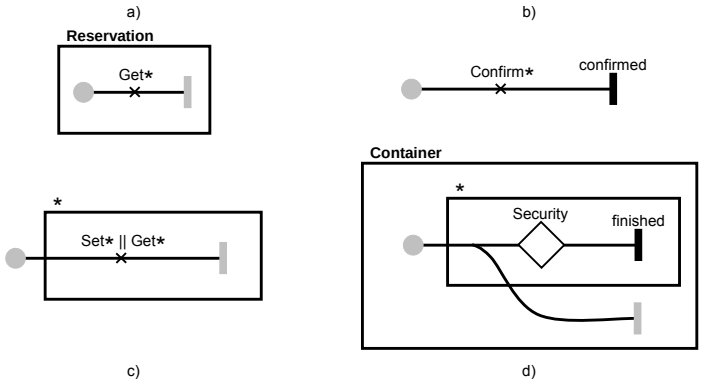

Fig. 11. More Examples of Pointcut Maps

Finally, the location of start and end points is also important for the meaning of the pointcut map. Fig. 11 (a) matches all maps with a responsibility starting with Get and bound to the component Reservation. Fig. 11 (b) matches all maps with a responsibility starting with Confirm. The responsibility must be immediately followed by an end point called confirmed. The responsibility and the end point may or may not be bound to a component. Fig. 11 (c) matches all maps with a responsibility starting with Set or Get and bound to any component as the first path node of the component (because the start point is outside the component). Finally, Fig. 11 (d) matches all maps with an OR-fork bound to any component inside component Container as the first path node. The OR-fork must be immediately followed on one branch by a static stub called Security and an end point called finished, and followed by nothing on the other branch before exiting the component (because the end point is outside the inner component).

In other words, if a path on the pointcut map crosses the boundary of a component because of the location of a start or end point, then the path in the matching UCM will also have to cross the boundary of the matching component. Note that the start and end points that are not matched are shown in gray in Fig. 11 and in Fig. 12. See Table 1 and Fig. 12 for a summary of all cases. The base maps and pointcut maps in Fig. 12 assume that after the start points all maps are identical. Therefore, whether a base map matches a pointcut map is solely dependent on the locations and the names of the start points. Note that the same matching rules apply to end points.

Table 1. Matching Rules for Start Points on Pointcut Maps and Base Maps

Do the base map and the pointcut map match?	Base map's start point [1] is *named* and *inside* component (Fig. 12 (a))	Base map's start point is *named* and *outside* component (Fig. 12 (b))	Base map's start point is *named differently* and *inside* component (Fig. 12 (c))	Base map's start point is *named differently* and *outside* component (Fig. 12 (d))
Pointcut map's start point [1] is *named* and *inside* component (Fig. 12 (e))	**yes**	**no** [2]	**no** [3]	**no** [3]
Pointcut map's start point is *named* and *outside* component (Fig. 12 (f))	**no** [2]	**yes**	**no** [3]	**no** [3]
Pointcut map's start point is *unnamed* and *inside* component (Fig. 12 (g))	**yes**	**yes** [4]	**yes**	**yes** [4]
Pointcut map's start point is *unnamed* and *outside* component (Fig. 12 (h))	**no** [5]	**yes**	**no** [5]	**yes**

1) The same reasoning applies to end points.
2) The pointcut expression stipulates that there has to be a start point with name abc inside or outside component abc. Therefore, there is no match if the start point in the base map is **not** inside or outside component abc, respectively.
3) There is no match in this case because the names do not match.
4) This pointcut expression does not require the path to cross component abc, but it also does not exclude it. Therefore, the base map is matched although the start point is outside of component abc.
5) This pointcut expression requires the path to cross the component because the start point is outside of component abc. Therefore, the start point in the base map cannot be inside component abc.

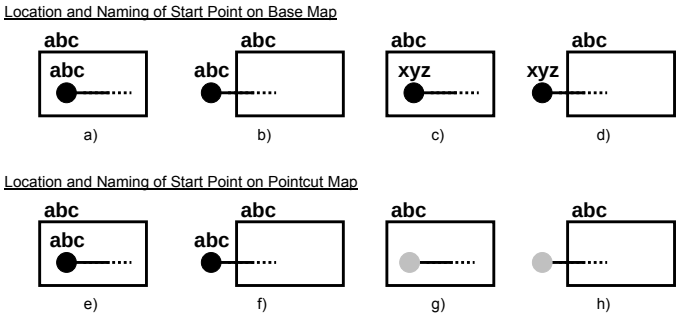

Fig. 12. Location and Naming of Start Points on Pointcut Maps and Base Maps

3.4 Advice Map Revisited

At this point, the advice defined on advice maps still needs to be woven into the base system with the help of the pointcuts defined on pointcut maps. Somehow, advice and pointcuts need to be linked. Advice may be executed before, after, or around joinpoints identified by many pointcuts. To achieve this, a dynamic stub called the *pointcut stub* is added to the advice map introduced in section 3.2. The plug-ins of the pointcut stub are the pointcut maps discussed in section 3.3. For example, the pointcut map in Fig. 11 (d) could be plugged into the pointcut stubs below by binding the one in-path to the one start point, one of the out-paths to one of the end points, and the other out-path to the other end point.

By keeping advice and pointcut expressions on separate UCMs (advice maps and pointcut maps, respectively), it is possible to reuse advice and pointcut expressions separately. For example, the same advice can be reused in a different UCM model with different pointcut maps plugged into the pointcut stub. Similarly, the same pointcut map can be used for different aspects.

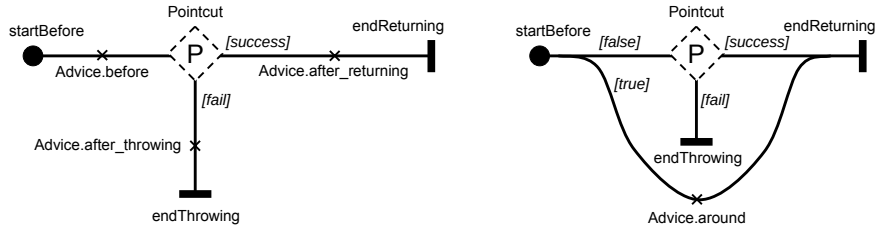

Fig. 13. Pointcut Stub

The two advice maps in Fig. 13 show how advice and pointcuts are linked to each other. The requirements engineer can understand in one glance the relationship of the advice to the base system due to the visual representation. For example, the left map shows advice being executed before and after the pointcuts specified on the pointcut maps bound to the pointcut stub. The right map shows very clearly that advice is being executed *around* the specified pointcuts, allowing a situation to be modeled which occurs frequently in aspect-oriented modeling. The aspect in the right map overrides the behavior of the base. In more detail, the left map shows that before the specified pointcuts the responsibility Advice.before is executed and after the specified pointcuts the responsibility Advice.after_returning is executed in the success case and the responsibility Advice.after_throwing is executed in the fail case. The right map shows that the path elements defined by the specified pointcuts are never executed in the composed system because the [false] branch is never taken. The [true] branch is always taken and therefore the responsibility Advice.around is executed instead of the path elements matched by the pointcuts.

Note that many more composition rules (such as concurrency, loops, and interleaving) than the ones mentioned here (before/after/around) can easily be modeled with AoUCM (see [31] for details).

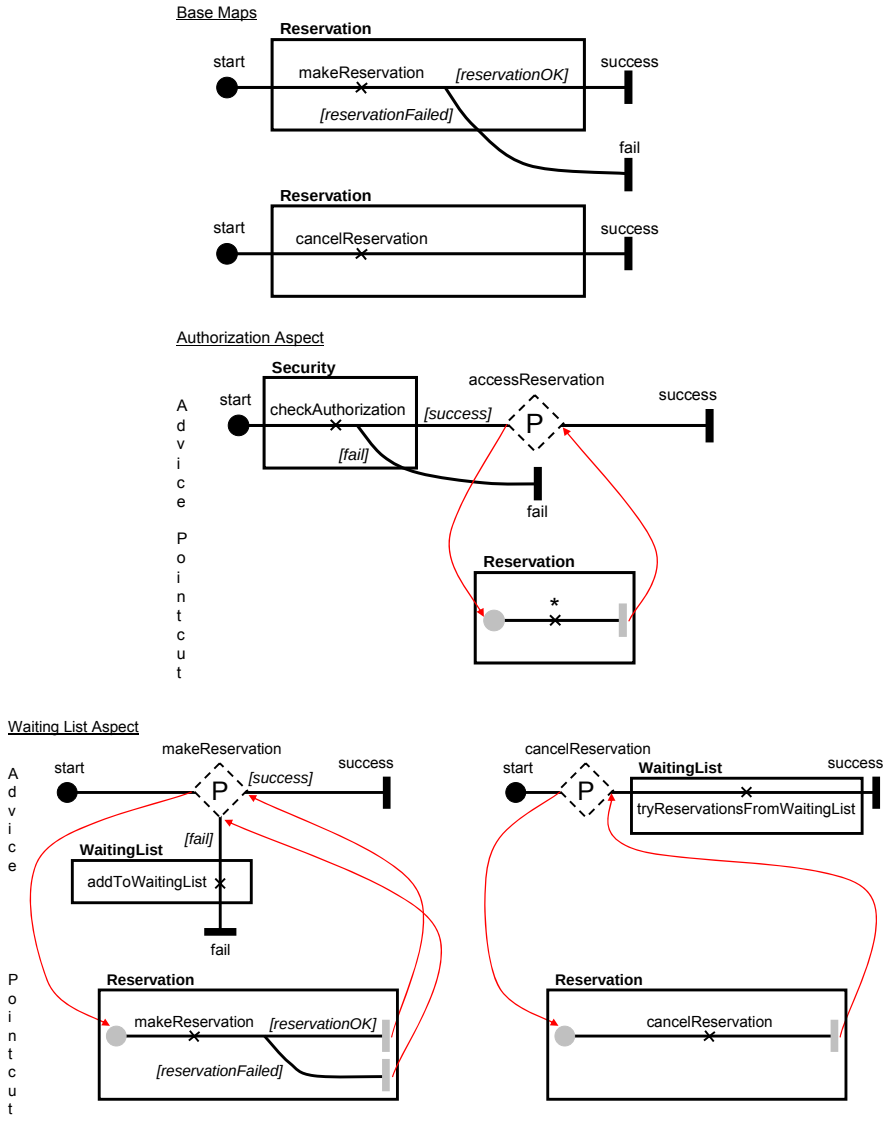

Fig. 14. Aspect-Oriented UCM Model for Reservations

There are a number of observations to be pointed out regarding the use of the pointcut stub. First, the number of in-paths and out-paths for the pointcut stub is flexible. This makes it possible to take into account path nodes with more than one before or after location (e.g. stubs, forks, and joins – see section 3.1). Advice can be specified individually for such locations. Second, the pointcut stub is a dynamic stub which may contain multiple plug-in maps, each of which may describe a different pointcut expression. Therefore, as many different pointcut expressions as necessary can be described for the aspect. Finally, each advice shown on the maps in Fig. 13 is

very simple as it consists only of one responsibility. In reality, the responsibility may be replaced by much more complicated advice maps (e.g. see Fig. 8 and imagine a pointcut stub added somewhere on the path).

3.5 Aspect-Oriented Use Case Maps Example

This section revisits the reservation example from section 2.2 and models it with AoUCMs. In Fig. 14, the base maps show the use cases for making and canceling a reservation. Two aspects are defined that describe authorization and waiting list behavior, respectively. The authorization aspect contains one advice map and one pointcut map. For the waiting list aspect, two advice maps and two pointcut maps are defined. The plug-in bindings are indicated by arrows. The authorization aspect adds an authorization check (checkAuthorization) before any responsibility in the Reservation component, while the waiting list aspect adds failed reservations to a waiting list (addToWaitingList) or tries to fulfill a reservation from the waiting list if a cancellation occurs (tryReservationsFromWaitingList).

The pointcut map for authorization matches any access to the Reservation component. Therefore, the responsibility in the authorization pointcut map is matched against makeReservation and cancelReservation in the base maps. Note that it is irrelevant for the match that the make reservation map contains two end points and the cancel reservation map only one because the end points (as well as the start points and the OR-fork) are not matched. The pointcut map requires only the responsibilities to be matched.

The pointcut maps for the waiting list match the makeReservation responsibility with the makeReservation responsibility in the base maps, the OR-fork including conditions with the OR-fork and conditions in the base maps, and the cancelReservation responsibility with the cancelReservation responsibility in the base maps.

3.6 URN Metamodel and Aspects

For the purpose of this paper, we are focusing on a subset of the UCM portion of the URN metamodel (Fig. 15) [33,38]. A UCMmap consists of component references (ComponentRef) and PathNodes, reflecting structure and behavior, respectively. Components may contain other components as well as path nodes. There are a great number of different kinds of path nodes but only the Stub is of greater interest with regard to aspects. Stubs may contain plug-in maps and PluginBinding specifies how a stub and a plug-in map are connected.

In order to accommodate the new concepts introduced by aspects, the URN metamodel needs to be extended. The new concepts are advice map, pointcut map, pointcut stub, and joinpoint. UCMadviceMap and UCMpointcutMap specialize the UCMmap class whereas PointcutStub and AspectStub specialize the Stub class. Joinpoint is a new class associated with PathNode since each path node can be a joinpoint. To be exact, this is not true for path nodes such as direction arrows which

only serve as visual aids. Therefore, the association between PathNode and Joinpoint is optional (0..1).

The association between PathNode and Joinpoint indicates an additional relationship, showing that path nodes on pointcut maps are matched against joinpoints in the base model. Note that Joinpoint instances and their associations are created only at run time and are therefore not part of a URN source model.

Finally, the AspectStub is required for visualizing the composed system (see section 4.2) and there is an association between aspects to capture any precedence relationships. This is required for conflict resolution of aspects trying to insert behavior at the same insertion point (see section 4.4).

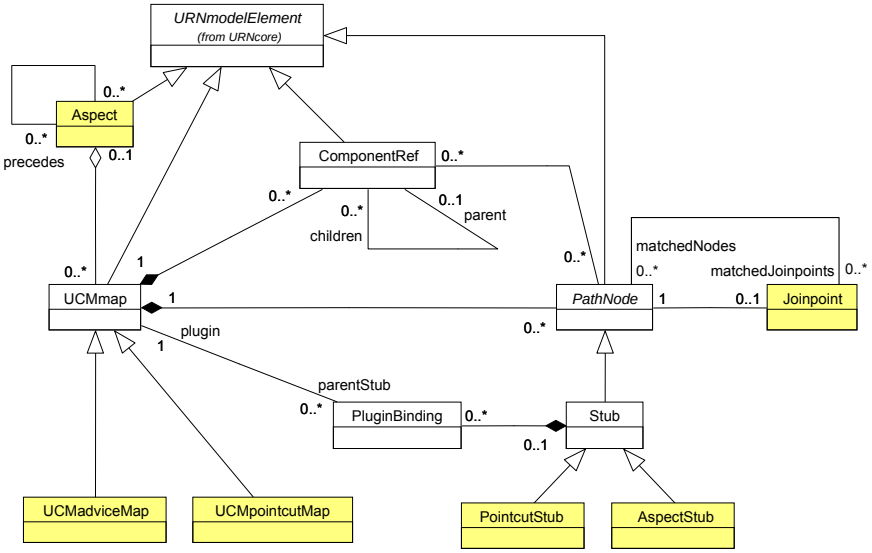

Fig. 15. Extended URN Metamodel

With the help of these new classes in the URN metamodel, an aspect can now be defined. An Aspect contains zero or more UCMmaps, some of which are UCMadviceMaps. Only UCMadviceMaps contain zero or more PointcutStubs. A UCMpointcutMap plugs only into a PointcutStub. An AspectStub contains only one or more UCMadviceMaps. Finally, only the PathNodes of a UCMpointcutMap match zero or more Joinpoints. These constraints could be expressed with OCL in the aspect-oriented URN metamodel.

4 Algorithms for Tool Support

Current URN tool support with jUCMNav allows advice maps, pointcut maps, and pointcut stubs including the binding to pointcut maps to be defined in the UCM model. This does not require any additional features to be implemented as the

standard features of jUCMNav are sufficient (anything discussed in section 3 can be done with jUCMNav right now).

Full support for aspect-oriented UCMs (AoUCM), however, requires additional features in order to facilitate the matching of pointcut maps to other maps in the UCM model, to indicate on base maps whether a path node is advised, to indicate on advice maps which base maps are being advised by the aspect, to compose aspect maps and base maps into a complete view of the system, and to switch back and forth between the visualization of the base system with aspects and the composed system. Algorithms for these additional features have been implemented and tested to verify the feasibility of this approach and will be released soon into the official version of jUCMNav.

4.1 Matching Algorithm

The matching algorithm compares the static structure of a pointcut map with all other maps in the UCM model except for other pointcut maps. The matching algorithm establishes a mapping from each path node on the pointcut map to path nodes in the UCM model. A successful match requires a mapping of each path node on the pointcut map to exactly one path node in the UCM model. As the pointcut map may be matched several times by the UCM model, the result contains a list of mappings for each matched instance of the pattern described by the pointcut map. Several path node types such as direction arrows as well as stubs including the start and end points of its plug-ins are not relevant to the matching algorithm and are therefore not mapped. These path node types are called map “white space”.

The matching algorithm is a recursive algorithm that begins at a start point of the pointcut map and at each step scans the next relevant path node following the current path node on the pointcut map (i.e. map “white space” is skipped). If multiple branches leave or enter the current path node, all branches are considered in one step. The following step will then continue to consider all branches at once. If the matching algorithm finds a matching path node in the UCM model for the initial path node of the pointcut map, the matching algorithm scans the base UCM model and the pointcut map in parallel. At each step, the algorithm tries to match all next path nodes of the pointcut map against all next path nodes in the UCM model. This match requires all possible permutations to be considered. For example, let us assume that the matching algorithm has mapped an OR-fork from the pointcut map to an OR-fork in the UCM model and both OR-forks have two branches. Then, the first branch of the OR-fork on the pointcut map can be matched either against the first or second branch of the OR-fork in the UCM model. The same applies to the second branch. If more than one permutation can be matched against the path nodes on the pointcut map, the matching algorithm will continue to explore recursively each matched permutation as an individual match candidate.

At each step, new mappings are added to the result if the matching is successful. If the matching is not successful, the mappings established for the current match candidate are discarded and the candidate is not pursued further. The matching, however, continues with all other still valid permutations and their corresponding mappings. Matching may not be successful in one of the following cases:

- The next path node of the pointcut map cannot be matched against the next path node in the base UCM model (in case only one branch and therefore only one next path node has to be considered).
- The set of next path nodes of the point cut map cannot be matched against the set of next path nodes in the base UCM model (in case several branches have to be considered).
- The next path node or set of path nodes can be matched but a new mapping contradicts the already established mappings.

Finally, the matching algorithm takes cycles on the pointcut maps into account in order to avoid infinite loops by not further considering an already visited path node on the pointcut map. See Appendix A: Matching Algorithm for more details on the matching algorithm.

The following criteria are taken into account to decide whether a path node in the pointcut map matches another path node. The names and types of the path nodes must match. The component names of the path nodes must match as well as the location of the path node in the component (first, last, or any path node in the component). Furthermore, the names of conditions have to be matched and the type of branch (e.g. timeout branch) has to be matched.

When matching a set of next path nodes on the pointcut map against another set of next path nodes, each matching pair of path nodes must fulfill the criteria listed in the previous paragraph. In addition, the current implementation requires the number of path nodes to be the same in each set for the match to be successful. Alternatively, the matching algorithm could provide an option to relax this requirement for any path node with multiple outgoing or incoming branches by allowing the set corresponding to a path node on the pointcut map to be smaller than the other set. For example, this would allow matching an OR-fork with two branches on the pointcut map against an OR-fork with two or more branches in the UCM model as long as two of these branches can be matched.

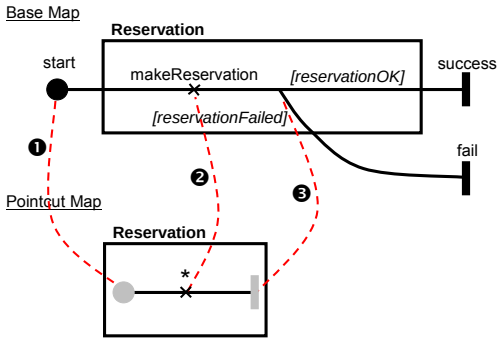

Fig. 16. Example of a mapping between pointcut map and base map

Fig. 16 shows an example of a mapping that was established by the matching algorithm. The responsibilities are mapped to each other since the wildcard matches any name. Mappings 1 and 3 contain unnamed start and end points. In terms of the matching algorithm, one can think of unnamed start and end points as free matches.

They can be matched with anything as long as they are matched to the path nodes that are closest to the other mappings.

While the mapping of the actual pointcut expression (in this case only the responsibility – see mapping 2) ensures that the pattern described by the pointcut map exists in the base map, the other two mappings (1 and 3) turn out to be the useful in terms of providing the additional features mentioned at the beginning of section 4 and discussed in greater detail in sections 4.2 and 4.4. Mappings 1 and 3 are so important because they help determine the joinpoints and insertion points associated with the joinpoints. The joinpoint in Fig. 16 is the responsibility in the base map.

As mentioned earlier in the bulleted list, contradictory mappings are the third reason for unsuccessful matches. One such case is illustrated in Fig. 17 (a). After matching the start point in the pointcut map to `s1` in the base map (see mapping 1) and the OR-fork in the pointcut map to the OR-fork in the base map (see mapping 2), the next step attempts to match the successors of the OR-fork in the pointcut map with the successors of the OR-fork in the base map. In both cases, the successors are OR-joins (note that the OR-fork in the pointcut map also has two successors – one for each branch, but the two successors happen to be the same OR-join). For each successor individually, a match is possible (see mappings 3 and 4), but these two mappings contradict each other since the same path node in the pointcut map cannot be mapped to two different path nodes in the base map (or vice versa). Fig. 17 (a) shows a contradiction occurring at the same step. Fig. 17 (b), on the other hand, shows a similar contradiction that appears at different steps of the matching algorithm (i.e. the new mapping 6 contradicts mapping 4 established by a previous step of the matching algorithm).

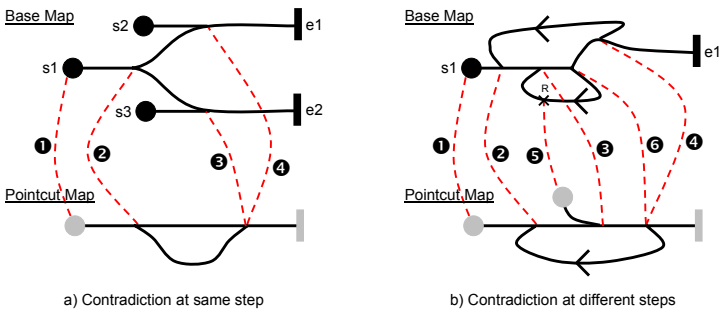

Fig. 17. Contradictory Mappings

Note that the mappings of one pointcut map to a base map may overlap with mappings established for another pointcut map. Overlapping mappings, however, do not conflict with or contradict each other. See section 4.4 for more details.

4.2 Composition Algorithm

Now that the joinpoints have been identified, the composition strategy for an AoUCM model is fairly straightforward. Given joinpoints, base maps, advice maps, pointcut maps, and plug-in bindings, the composed system is realized by adding *aspect stubs*

to the base maps. One such stub links to the appropriate part of an advice map. The insertion points for the static stubs on the base maps are defined by the mappings of the joinpoints found by the matching algorithm. The plug-in bindings for the inserted stubs are also derived from the mappings of the joinpoints.

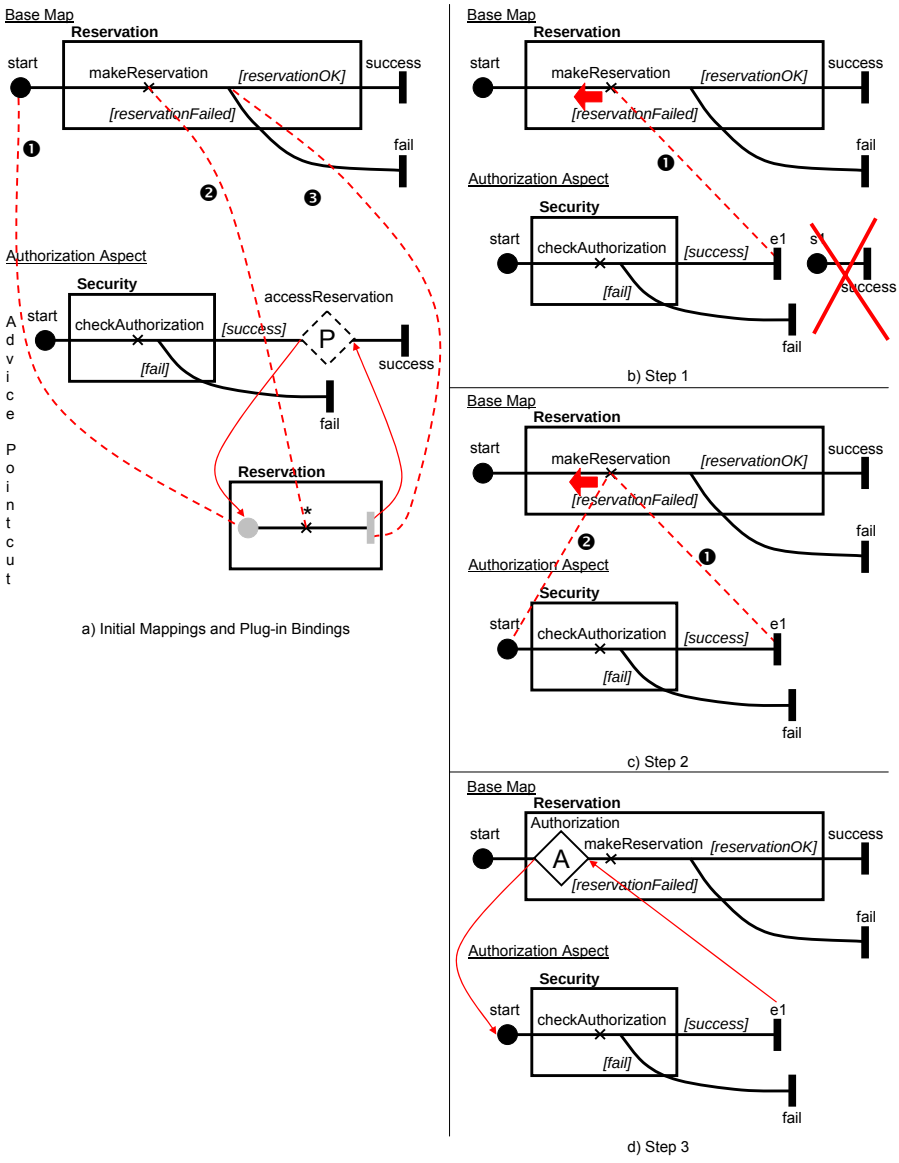

Fig. 18. Composition of Aspect and Base Map

For an example of the composition algorithm, recall the authorization aspect from Fig. 14 and the mappings from Fig. 16 (repeated in Fig. 18 (a)). Note the joinpoint

(the responsibility in the base map). The right side of Fig. 18 shows the three steps involved in composing the system.

Step 1 removes the pointcut stub from the advice map while retaining the mappings to the base map. The removal of the pointcut requires new start and end points to be inserted into the advice map (`s1` and `e1` in Fig. 18 (b)). Empty paths on the advice map are then removed from the advice map (see crossed out path in Fig. 18 (b)) because no new behavior is added by empty paths. In this example advice is only added before the pointcut and therefore only one disjoint path exists after all empty paths are removed. If advice is added before and after the pointcut, then several disjoint paths exist after removing all empty paths.

An in-path of the pointcut stub turns into a new end point. The mapping for a new end point is found by following the plug-in binding of its associated in-path to the start point on the pointcut map. The mapping of the path node after this start point is retained (see mapping 2 in Fig. 18 (a) and mapping 1 in Fig. 18 (b)). An out-path of the pointcut stub turns into a new start point. The mapping for a new start point is found by following the plug-in binding of its associated out-path to the end point on the pointcut map. The mapping of the path node before this end point is retained. Note that if a pointcut stub contains multiple pointcut maps, several mappings are retained.

In addition, the direction towards the next closest mapping is also retained (see the large arrow called the *closest-mapping arrow* in Fig. 18 (b)). The closest-mapping arrow points towards the path node in the base model identified by the mapping from the unnamed start or end point on the pointcut map (depending on which plug-in binding was followed from the pointcut stub). This is especially important for path nodes in the base map that can have more than one path node as successor or predecessor as the following or preceding mapping clearly identifies the successor or predecessor, respectively.

Step 2 scans the path on the advice map to find a corresponding start point for each new end point and a corresponding end point for each new start point. If no or more than one such start or end point can be found, the advice map is malformed and composition cannot proceed. The path is scanned backwards for new end points and forward for new start points. Once a start or end point is found, the same mapping as for the corresponding new end point or new start point, respectively, is created (see mapping 2 in Fig. 18 (c)).

All that is left to do in step 3 is to insert the stub in the base map. The insertion point is the one associated with the joinpoint and the one towards which the closest-mapping arrow identified in step 1 is pointing. The inserted stub is bound to the same component as the joinpoint (in the example in Fig. 18, this path node is identified by the mapping in Step 1). Note that it is possible to specify with the UCM notation whether components on a plug-in map are contained in the component of the corresponding stub. In all presented examples, the components on a plug-in map are not contained in the component of the stub. Finally, the mappings are converted into plug-in bindings and thus, the aspect has been woven into the base map. Several stubs may have to be inserted for one advice map (e.g. if the same pointcut expression is matched successfully against many base maps or if one pointcut stub contains multiple pointcut maps that cause multiple successful matches). See Appendix B: Composition Algorithm for more details on the composition algorithm.

There are a number of special cases that have to be taken into account by the composition algorithm: a) insertion points before a start point or after an end point and b) around advice. Fig. 7 shows that insertion points exist before a start point or after an end point. These insertion points are required for named start or end points on a pointcut map which are matched against start or end points in the base map, respectively. At step 1 of the composition algorithm, named start points or named end points in pointcut maps receive special treatment when establishing the closest-mapping arrow. The closest-mapping arrow of a start point in the base map that is mapped to a named start point in the pointcut map always points to the insertion point before the start point. The closest-mapping arrow of an end point in the base map that is mapped to a named end point in the pointcut map always points to the insertion point after the end point. Consequently, a stub has to be inserted in the base map either before the mapped start point or after the mapped end point. A stub, however, cannot be simply inserted before a start point or after an end point. Additional start and end points are required as illustrated in Fig. 19 (see s1, e1, s2, and e2). Note that any plug-in bindings of the mapped start or end point have to be transferred to the new start or end point, respectively (e.g. from abc to s1 or xyz to e2 in Fig. 19).

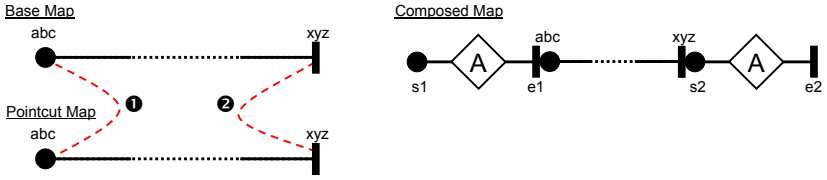

Fig. 19. Composition with Named Start and End Points

The second special case involves around advice. Fig. 20 introduces a new feature for the reservation example. Instead of using a waiting list, a reservation at a partner hotel is attempted and the reservation is added to the waiting list only if the reservation at the partner hotel also fails. In terms of the composition algorithm, the only differences to the example in Fig. 18 occur at steps 1 and 3. In step 1, the composition algorithm notices that both, a new start point and a new end point, exist in one path on the advice map (see s1 and e1 in Fig. 20 (b)). This indicates around advice. Because of that, the stubs inserted into the base map in step 3 are marked with A↓ and A↑ in order to highlight that the path segment between the two stubs may be skipped. In the example in Fig. 20, this occurs if the reservation at the partner hotel is successful. The stubs A↓ and A↑ represent the entrance and exit of a tunnel underneath the base map that can be used to circumvent the path segment between the two stubs.

The around advice in Fig. 20 is a weak kind of around advice as the base behavior is not overridden all the time by the aspect. Strong around advice can be indicated even more clearly. Let us assume that the book with hotel partner feature is replacing the waiting list behavior. This is certainly not good customer service but let us ignore this for one moment for the sake of this example. Then, the aspect could be described as in Fig. 21 (a). The [true] branch is always taken regardless of whether the reservation at the partner hotel was successful or not. The [false] branch is never taken, therefore causing the waiting list behavior to be replaced.

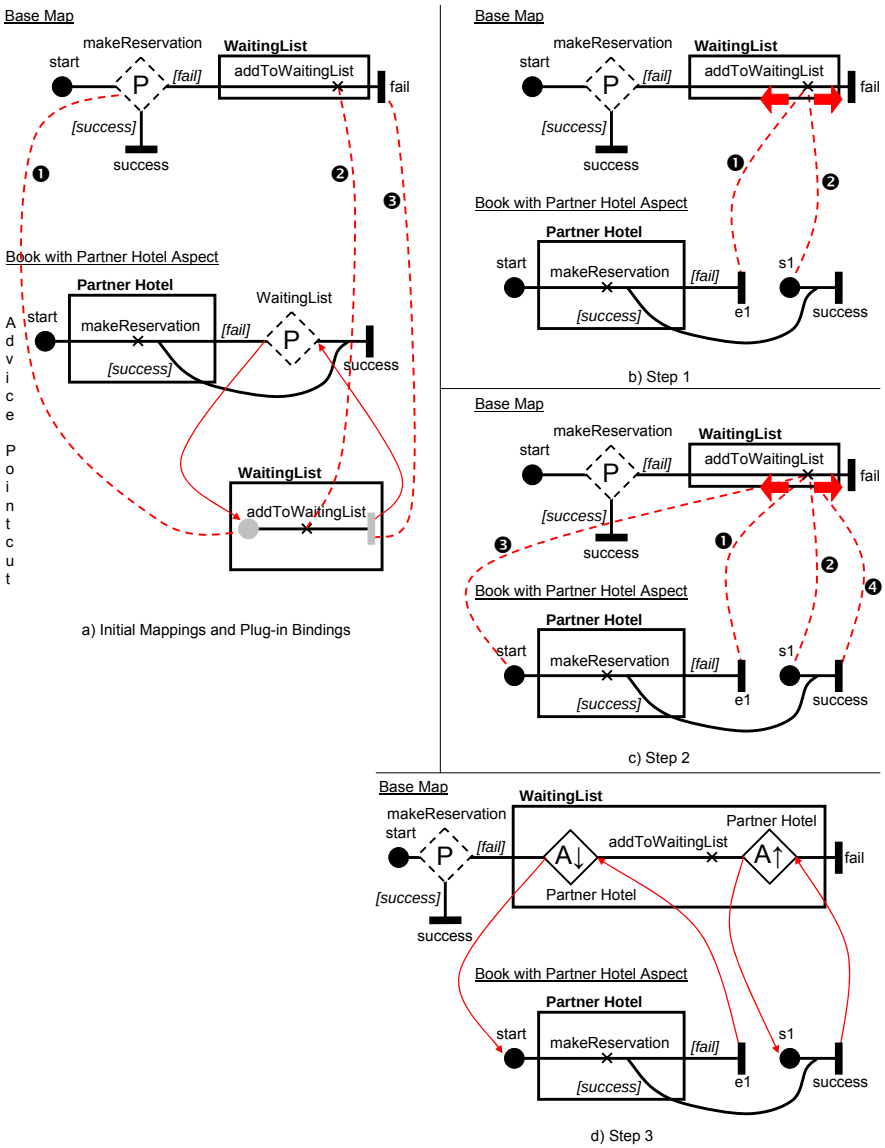

Fig. 20. Composition with Around Advice

The resulting composed system uses stubs without in-paths or out-paths as in Fig. 21 (b) to highlight that the behavior between the stubs is replaced by the aspect. The difference for the composition algorithm lies in step 1 where, in addition to identifying new start and end points on the same path on the advice map, the algorithm also performs a scan of the advice map that reveals the existence of a [false] branch leading directly to the pointcut stub. This indicates a strong around advice. In step 3, the stubs without in-paths or out-paths are used and additional start and end

points have to be added for the path segment between the two Partner Hotel stubs (s1 and e1 in the base map of Fig. 21 (b)).

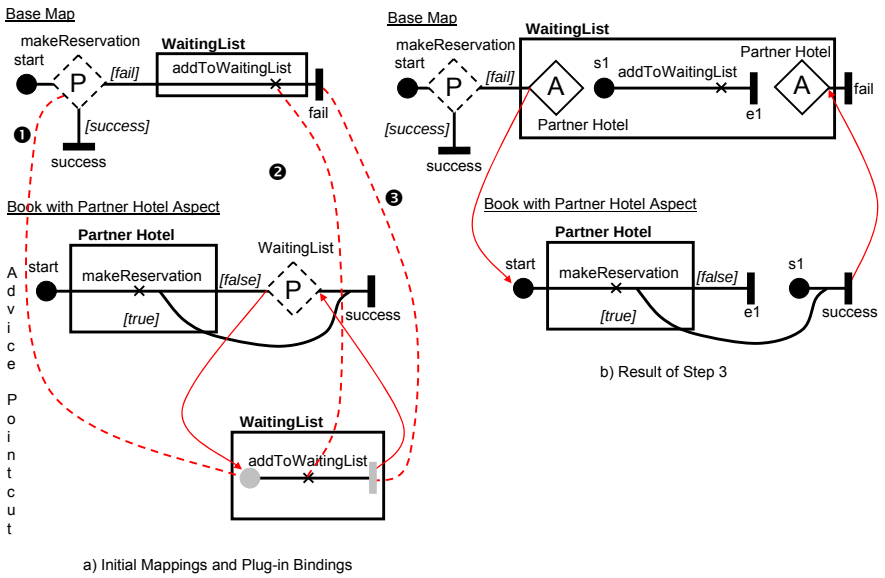

Fig. 21. Composition with Strong Around Advice

Note that the replaced path between the Partner Hotel stubs in Fig. 21 (b) can optionally be deleted from the base map by the composition algorithm as the path does not serve any purpose in the composed system anymore. It is also possible to merge the two Partner Hotel stubs into one stub. This, however, can only be done if the stubs are on the same map. The example in Fig. 21 shows the general case where the two stubs could be on different maps and merging is not possible.

Besides illustrating around advice, this example also shows that aspects can be defined on other aspects because the base map in Fig. 20 is an advice map and is only a base map relative to the book with hotel partner aspect.

4.3 Complete Example of Composed System

The complete composed system is shown in Fig. 22. It is very similar to the non-aspectual UCM model in Fig. 5 except that no root maps are used but the impact of other concerns is shown directly on the maps for making and canceling a reservation. Multiple disjoint plug-in maps for making or canceling a reservation are therefore avoided. The authorization stubs appear in making and canceling a reservation but they are added automatically by the composition algorithm. Therefore, they are less of a concern in terms of scalability and maintainability of the model.

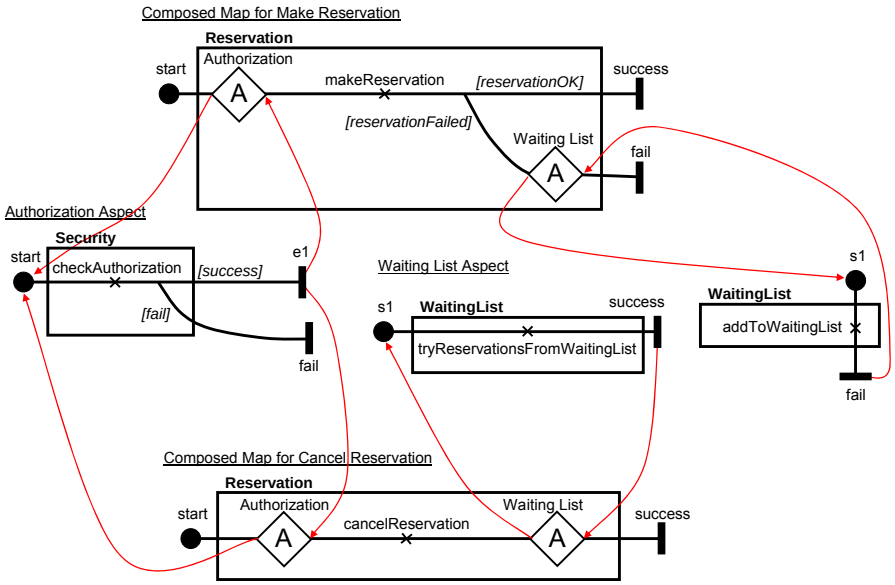

Fig. 22. Composed System

4.4 Beyond Matching and Composition

The joinpoints and mappings identified as a result of the matching algorithm are the base of further functionality besides composition. For example, it is possible now

a. to easily indicate on UCMs whether a path node is advised,
b. to easily indicate on advice maps which path nodes on which UCMs are being advised by the aspect,
c. to warn the requirements engineer if multiple aspects are advising the same path node (A simple way of indicating this is to add a dynamic stub to the base map instead of the static stub shown in the examples. The dynamic stub contains a plug-in map for each aspect and a conflict resolution mechanism based on simple precedence rules is used. A precedence rule states that aspect A has to precede aspect B. If no precedence rule is specified, the order is chosen randomly.), and
d. to warn the requirements engineer if the same pointcut advises the same path node in multiple ways (For example see Fig. 23, given a pointcut map containing only an OR-fork with two branches, the OR-fork can be matched against all OR-forks in the UCM model in two ways. The first branch of the OR-fork in the UCM model can be mapped to the first or second branch of the OR-fork on the pointcut map and the second branch of the OR-fork in the UCM model can be mapped to the second or first branch of the OR-fork on the pointcut map, respectively. If advice is being added before this pointcut, then the same advice would be added twice at the same insertion point – once for each matched permutation of

the OR-fork branches. If multiple matches of the same pointcut map are not desired, the requirements engineer can use the warning to modify the pointcut map in order to resolve the multiple matches.).

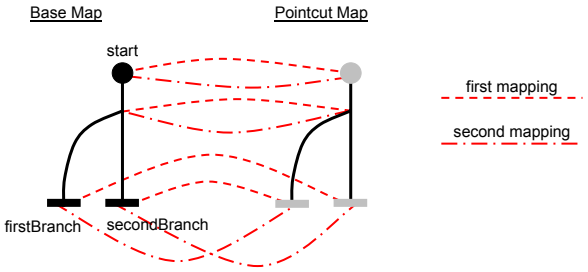

Fig. 23. Multiple Mappings of Pointcut Map

Overlapping mappings from two pointcut maps to a base map do not conflict with or contradict each other because only the mappings that identify joinpoints are important. If the latter mappings overlap, then case (c) occurs for which a conflict resolution mechanism exists.

A phenomenon that is called emergent behavior in the feature interaction community can also be observed when aspects and the base are composed together, possibly resulting in unexpected behavior. Undesired emergent behavior occurs when an aspect and the base work well individually but not when combined together. The feature interaction research carried out for UCMs does apply to AoUCM and can be used to deal with conflicts caused by undesired emergent behavior. The UCM scenario definitions can be used to describe the pre-conditions and post-conditions as well as start and expected end points of aspects and base behavior alike. After composing the system, the scenario definitions can be checked to ensure that no pre-conditions and post-conditions have been violated and all expected end points are reached. Essentially, UCM scenarios allow a high-level test suite to be built for the UCM model.

Finally, the descriptions of the matching and composition algorithms in the previous sections only deal with one pointcut map. In order to compose a UCM model containing many aspects, one needs to ensure that each pointcut map is matched against the same base model. When a pointcut map is matched against the base model, the identified joinpoints can be indicated in the original model because this does not influence the matching algorithm for the next aspect. Composition, however, does change the structure of the UCM model and therefore influences the matching algorithm. Therefore, the composition algorithm first creates a copy of the existing UCM model which can then be safely changed by the composition algorithm. When the next aspect is added incrementally to the system, the original base model is used to identify the joinpoints for the new aspect. Based on the joinpoints and the original base model, the composition algorithm then makes further changes to the copy of the base model in order to weave the new aspect into the system. The order in which aspects are dealt with, however, is irrelevant for the matching and composition algorithms presented in this paper.

5 Conclusion and Future Work

This paper is the first to discuss in detail aspect-oriented UCMs (AoUCM) and algorithms for tool support. It shows how to define aspects with UCMs, presents an algorithm for matching pointcut expressions (defined by pointcut maps) against maps in the UCM model, presents an algorithm for composing and visualizing aspects and the base model together, and extends the URN metamodel with aspect concepts. Compared to other scenario-based approaches to aspect-oriented requirements engineering, AoUCM do not require any new notational concepts, aspects can be modeled unobtrusively, and everything can be modeled visually for an aspect, including parameterized pointcut expressions.

No new notational concepts. AoUCM make use of the same set of modeling elements as traditional UCMs, making it easier to switch from traditional modeling to aspect-oriented modeling. Most significantly, stubs are used to link advice and pointcut expression as well as to visualize the composed system. Jacobson and Ng [25] add the concept of pointcut to use case modeling and change the meaning of extension points. Moreira *et al.* [12,28,29] require several extensions to UML diagrams in order to visualize aspects. Zdun and Strembeck [42] add start and end nodes for aspects. Whittle and Araújo [13,41], de Bruin and van Vliet [22], as well as Barros and Gomes [15], however, do not require changes to modeling notations (note that stereotyping does not really change the modeling notation).

Modeled unobtrusively. AoUCM allow aspects to be defined without influencing the base model as parameterized pointcut expressions are linked with the base through a matching algorithm. This is a crucial point of aspect-orientation often referred to as obliviousness as the base model must not be polluted by aspect-specific information. Jacobson and Ng violate this point by requiring extension points to be defined in the base. Similarly, de Bruin and van Vliet require Pre and Post stubs to be added to the base model. All other techniques mentioned in the paragraph above, however, model aspects also unobtrusively. Note that the approach by Zdun and Strembeck is not applicable to this category because it is not concerned with defining aspects but visualizing the composed system only.

Visual aspects (including parameterized pointcuts). AoUCM can model visually every part of an aspect including parameterized pointcuts, therefore avoiding a modeling paradigm break. Visual models are usually the preferred choice at higher levels of abstractions. The parameterization is achieved through the use of wildcards but this use of text is minimal. Parameterization of pointcut expressions is important to address scalability issues. Jacobson and Ng use textual expression to define parameterized pointcut expressions. Note that the extend relationship for use case diagrams is a visual representation of a pointcut but does not contain enough information and therefore has to rely on the textual representation. The binding rules used by Whittle and Araújo and composition rules used by Moreira *et al.* are represented in a textual way. Furthermore, these rules explicitly link one element with another and do not allow parameterized expressions. Barros and Gomes also use a textual representation of pointcuts and also explicitly link nodes in UML activity diagrams, not allowing parameterized expressions. de Bruin and van Vliet define aspects (refinement maps) in a visual way but use limited type matching to merge

behavior and structure from the aspect with the base model. Zdun and Strembeck's approach again is not applicable to this category.

In previous work, Mussbacher *et al.* [31] compared composition techniques of several scenario-based approaches to aspect-oriented requirements engineering and concluded that AoUCM have a **flexible and exhaustive composition technique** with significant advantages over the other approaches mentioned in this section.

In terms of abstraction levels, UCMs are at the same level as the techniques used by Jacobson and Ng, Barros and Gomes, and Zdun and Strembeck. Note that the examples in Barros and Gomes represent rather low-level control flow but that activity diagrams can also be used to describe high-level workflow. UCMs abstract from message and data details. UCMs can therefore be used earlier than message-based behavioral models but also contain more information than UML use case diagrams. Therefore, UCMs are at a **higher level of abstraction** than the work by Whittle and Araújo which is at the message/state machine level. Moreira *et al.* make use of some models that are at the same and some models that are at a lower level of abstraction than UCMs.

Furthermore, UCMs **model the whole system** making it possible to reason about interactions between various use cases or scenarios. UCMs have already been used not only for scenario interaction detection but also for performance analysis and testing purposes. By applying these research results to the composed UCM model containing the base and aspects, it is possible to achieve greater confidence in the model at a very early stage in the development. This is an advantage over all other techniques mentioned in this section as these techniques model scenarios in isolation.

This paper reports on the first results of a much larger research goal. In the long term, we plan to also extend the GRL part of URN with aspects and synchronize the extensions to the GRL part with the extensions to the UCM part. The matching and composition algorithms have been implemented and tested and will soon be available in an official release of the jUCMNav tool. A case study of a non-trivial e-commerce application is also underway which we hope will further illustrate the benefits of our approach. UCMs have already been used to model systems of significant sizes. Initial results suggest that AoUCM further improve the scalability of UCMs. Finally, some aspects inherently exist in UCMs. As UCMs abstract from message and data details, a simple path going from one component to another may represent a very complex interaction between these two entities, possibly involving multiple message exchanges. Aspects could define such interactions and would allow UCMs to more easily be moved forward to the next abstraction level.

Acknowledgments. This research was supported by the Natural Sciences and Engineering Research Council of Canada, through its programs of Discovery Grants and Postgraduate Scholarships, and by the Ontario Research Network on e-Commerce.

References

1. Abdelaziz, T., Elammari, M., and Unland, R.: Visualizing a Multiagent-Based Medical Diagnosis System Using a Methodology Based on Use Case Maps. *Multiagent System Technologies* (editors Lindemann-v. Trzebiatowski, G., Denzinger, J., Timm, I.J., and Unland, R.), LNCS 3187, Springer, pp 198-212 (September 2004)
2. Alencar, F., Moreira, A., Araújo, J., Castro, J., Silva, C., and Mylopoulos J.: Using Aspects to Simplify *i** Models. *14th IEEE International Requirements Engineering Conference (RE 06)*, Minneapolis, USA (September 2006)
3. Amyot, D. and Logrippo, L.: Use Case Maps and LOTOS for the Prototyping and Validation of a Mobile Group Call System. *Computer Communication*, Vol. 23(12), pp 1135-1157 (July 2000)
4. Amyot, D., Charfi, L., Gorse, N., Gray, T., Logrippo, L., Sincennes, J., Stepien, B., and Ware, T.: Feature Description and Feature Interaction Analysis with Use Case Maps and LOTOS. *Feature Interactions in Telecommunications and Software Systems VI*, Glasgow, Scotland, UK, IOS Press, pp 274-289 (May 2000)
5. Amyot, D., Roy, J.-F., and Weiss. M.: UCM-Driven Testing of Web Applications. *SDL 2005: Model Driven* (editors Prinz A., Reed R., and Reed J.), LNCS 3530, Springer, pp 247-264 (June 2005)
6. Amyot, D., Weiss, M., and Logrippo L.: UCM-Based Generation of Test Purposes. *Computer Networks*, Vol. 49(5), pp 643-660 (December 2005)
7. Amyot, D.: Introduction to the User Requirements Notation: Learning by Example. *Computer Networks*, Vol. 42(3), pp 285-301 (21 June 2003)
8. Andrade, R.: Applying Use Case Maps and Formal Methods to the Development of Wireless Mobile ATM Networks. *Lfm2000: Fifth NASA Langley Formal Methods Workshop*, Williamsburg, Virginia, USA, pp 151-162 (June 2000)
9. *AOSD Community Wiki – Research Projects*. http://aosd.net/wiki/index.php?title=Research_Projects (accessed February 2007)
10. *AOSD Community Wiki – Tools for Developers*. http://aosd.net/wiki/index.php?title=Tools_for_Developers (accessed February 2007)
11. Araújo J. and Coutinho, P.: Identifying Aspectual Use Cases Using a Viewpoint-Oriented Requirements Method. *Early Aspects 2003: Aspect-Oriented Requirements Engineering and Architecture Design*, Workshop of the 2nd International Conference on Aspect-Oriented Software Development (AOSD), Boston, USA (March 2003)
12. Araújo, J. and Moreira, A.: An Aspectual Use Case Driven Approach. *VIII Jornadas de Ingeniería de Software y Bases de Datos (JISBD 2003)*, Alicante, Spain (November 2003)
13. Araújo, J., Whittle, J., and Kim, D.: Modeling and Composing Scenario-Based Requirements with Aspects. *Proceedings of the 12th IEEE International Requirements Engineering Conference (RE 04)*, Kyoto, Japan, IEEE CS Press, pp 58-67 (Sep. 2004)
14. *AspectJ web site*. http://www.eclipse.org/aspectj/ (accessed February 2007)
15. Barros, J.-P. and Gomes, L.: Toward the Support for Crosscutting Concerns in Activity Diagrams: a Graphical Approach. *Workshop on Aspect-Oriented Modelling* (held with UML 2003), San Francisco, California, USA (October 2003)
16. Billard, E.A.: Operating system scenarios as Use Case Maps. *Fourth International Workshop on Software and Performance (WOSP 2004)*, Redwood Shores, California, USA, pp 266-277 (January 2004)
17. Buhr, R.J.A. and Casselman, R.S.: *Use Case Maps for Object-Oriented Systems*. Prentice-Hall (1995)
18. Buhr, R. J. A.: A Possible Design Notation for Aspect Oriented Programming. *Aspect-Oriented Programming Workshop at ECOOP'98*, Brussels, Belgium (July 1998)
19. Buhr, R.J.A.: Use Case Maps as Architectural Entities for Complex Systems. *IEEE Transactions on Software Engineering*, Vol. 24(12), pp 1131-1155 (December 1998)

20. Chitchyan, R. et al.: *Survey of Analysis and Design Approaches*. AOSD-Europe Report ULANC-9 (May 2005), http://www.aosd-europe.net/deliverables/d11.pdf (acc. Feb. 2007)
21. Clarke, S. and Baniassad, E.: *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison-Wesley (2005)
22. de Bruin, H. and van Vliet, H.: Quality-Driven Software Architecture Composition. *Journal of Systems and Software*, Vol. 66(3), pp 269-284 (June 15, 2003)
23. Elammari, M. and Lalonde, W.: An Agent-Oriented Methodology: High-Level View and Intermediate Models. *1st International Workshop on Agent-Oriented Information Systems (AOIS)*, Heidelberg, Germany (June 1999)
24. Gross, D., and Yu, E.: Dealing with System Qualities During Design and Composition of Aspects and Modules: An Agent and Goal-Oriented Approach. *Proceedings of the 1st International Workshop on Traceability in Emerging Forms of Software Engineering, Automated Software Engineering Conference*, Edinburgh, U.K., pp. 1-8 (October 2002)
25. Jacobson, I. and Ng, P.-W.: *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley (2005)
26. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J.: Aspect-Oriented Programming. *ECOOP'97–Object Oriented Programming, 11th European Conference*, LNCS 1241, Springer, pp 220-242 (June 1997)
27. Miga, A.: *Application of Use Case Maps to System Design with Tool Support*. MEng thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada (October 1998), http://www.UseCaseMaps.org/tools/ucmnav (accessed February 2007)
28. Moreira, A. and Araújo, J.: Handling Unanticipated Requirements Change with Aspects. *Proceedings of the 16th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, Banff, Canada (June 2004)
29. Moreira, A., Araújo, J., and Brito, I.: Crosscutting Quality Attributes for Requirements Engineering. *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, Ischia, Italy, ACM Press, pp 167-174 (July 2002)
30. Mussbacher, G., Amyot, D., and Weiss M.: Visualizing Aspect-Oriented Requirements Scenarios with Use Case Maps. *International Workshop on Requirements Engineering Visualization (REV 2006)*, Minneapolis, USA (September 11, 2006)
31. Mussbacher, G., Amyot, D., Whittle, J., and Weiss M.: Flexible and Expressive Composition Rules with Aspect-oriented Use Case Maps (AoUCM). *10th International Workshop on Early Aspects (EA 2007)*, Vancouver, Canada (March 13, 2007)
32. Rashid, A., Moreira, A., and Araújo, J.: Modularisation and Composition of Aspectual Requirements. *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, Boston, USA, ACM Press, pp 11-20 (March 2003)
33. Roy, J.-F. Kealey, and Amyot, D.: Towards Integrated Tool Support for the User Requirements Notation. *SAM 2006: Language Profiles - Fifth Workshop on System Analysis and Modelling*, Kaiserslautern, Germany, LNCS 4320, Springer, pp 183-197 (May 2006), http://www.softwareengineering.ca/jucmnav (accessed February 2007)
34. Scratchley, W.C. and Woodside, C.M.: Evaluating Concurrency Options in Software Specifications. *7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, College Park, Maryland, USA, pp 330-338 (October 1999)
35. Siddiqui, K.H. and Woodside, C.M.: Performance aware software development (PASD) using resource demand budgets. *Workshop on Software and Performance (WOSP)*, Rome, Italy, pp 275-285 (July 2002)
36. Tarr, P., Ossher, H., Harrison, W., and Sutton, S.M.: N degrees of separation: Multidimensional separation of concerns. *Proceedings of the 21st Intl. Conference on Software Engineering (ICSE 99)*, IEEE, Los Angeles, ACM press, pp 107-119 (May 1999)
37. *URN - Goal-oriented Requirement Language (GRL)*, ITU-T Draft Recommendation Z.151. Geneva, Switzerland (Sep. 2003), http://www.UseCaseMaps.org/urn (accessed Feb. 2007)

38. *URN - Use Case Map Notation (UCM)*, ITU-T Draft Recommendation Z.152. Geneva, Switzerland (September 2003), http://www.UseCaseMaps.org/urn (accessed Feb. 2007)
39. *User Requirements Notation (URN) – Language Requirements and Framework*, ITU-T Recommendation Z.150. Geneva, Switzerland (February 2003), http://www.itu.int/ITU-T/publications/recs.html (accessed February 2007)
40. Weiss, M. and Amyot, D.: Business Process Modeling with URN. *International Journal of E-Business Research*, Vol. 1(3), pp 63-90 (July-September 2005)
41. Whittle, J. and Araújo, J.: Scenario Modelling with Aspects. *IEE Proceedings – Software*, Vol. 151(4), pp 157-172 (August 2004)
42. Zdun, U. and Strembeck, M.: Modeling the Evolution of Aspect Configurations using Model Transformations. *Proceedings of the Linking Aspect Technology and Evolution Workshop (LATE)*, Bonn, Germany (March 2006)

Appendix A: Matching Algorithm

A high level summary of the algorithm follows, showing how one pointcut map is matched against the UCM model. In addition to the two operations matchPointcutMap and match, two more, rather straightforward operations were implemented. The first operation decides, given two path nodes, whether the two path nodes match according to the criteria mentioned in section 4.1. It is used by matchPointcutMap. The second operation finds, given two sets of path nodes, all permutations that match elements in one set against elements in the other. It is used by match. Note that a) each element in one set has to be matched against exactly one element in the other set and vice versa, and that b) the second operation makes use of the first to match individual elements.

Operation: matchPointcutMap
Input: UCMPointcutMap pointcutMap, UCMmodel baseUCM
Output: MappingsList
Exception: NoMatchFound (if match is unsuccessful)

```
begin // matchPointcutMap
    MappingsList resultMappingsList = Ø
    PathNode initPointcutNode = pointcutMap.getInitialPathNode()
    foreach PathNode pn in baseUCM
        if initPointcutNode matches pn then
            MappingsList firstMapping = Ø
            add new Mapping(initPointcutNode, pn) to firstMapping
            try
                // Match next path nodes of initPointcutNode against next path nodes of
                // pn and return matchList (which contains firstMapping plus
                // new mappings found by match()).
                MappingsList matchList = match(initPointcutNode, pn, firstMapping)
                add matchList to resultMappingsList
            endtry
            catch NoMatchFound
                // Do nothing and continue for loop (this gives initPointcutNode a
                // chance to be matched against other path nodes in the UCM model).
            endcatch
        endif
```

```
    endforeach // PathNode
    if resultMappingsList == Ø then
        throw new NoMatchFound() // No mapping found at all.
    endif
    return resultMappingsList
end // matchPointcutMap

Operation:    match
Input:        PathNode pointcutNode, PathNode baseNode,
              MappingsList currentMappings
Output:       MappingsList
Exception:    NoMatchFound (if match is unsuccessful)

begin // match
    if pointcutNode does not have any next path nodes then
        return currentMappings // Stops recursion.
    endif
    // Else, there are still path nodes to match.
    MappingsList finalMappingsList = Ø
    // In order to match the next path nodes, all permutations of these path nodes
    // have to be considered. Hence the need for the first for loop!
    // The resulting permutations contain all possible mappings between all next
    // path nodes of pointcutNode and all next path nodes of baseNode.
    // These mappings can be accessed with getMappings() for each
    // permutation. A match is successful if at least one permutation can be
    // matched recursively.
    Find all matching permutations of baseNode's next path nodes
    foreach permutation p in the found permutations
        MappingsList permutationMappings = Ø
        // The following if statement catches an invalid permutation because the
        // mappings for the permutation contradict already established mappings.
        // Therefore, discard this permutation and move on to the next.
        if currentMappings contradict p.getMappings()
            then continue with next loop iteration
        endif
        copy currentMappings to permutationMappings
        add p.getMappings() to permutationMappings
        try
            MappingsList mergeResult = Ø
            // For each mapping in the current permutation a recursive match has to
            // be attempted. Hence, the need for the second for loop! There are two fail
            // cases: 1) the recursive match cannot find any matching path elements
            // (match() is not successful) or 2) the merging of all mappings causes
            // contradictory mappings (merge is not successful).
            foreach mapping pointcutNode2 to baseNode2 in p.getMappings()
                // Match recursively next path nodes of pointcutNode2 against next
                // path nodes of baseNode2 and return mappingsList (which contains
                // permutationMappings plus new mappings found by match()).
                // Match() throws NoMatchFound exception if not successful.
                MappingsList recursionResult = match(pointcutNode2, baseNode2,
                                                     permutationMappings)
                // At this point, the results need to be merged. This is necessary
```

```
            // because in each pass of the for loop a branch is explored recursively
            // and a match is only found if the results from all branches together
            // make sense (i.e. they do not contradict each other). Merge also
            // throws NoMatchFound exception if not successful.
            merge recursionResult with mergeResult
        endforeach // mapping
        add mergeResult to finalMappingsList
    endtry
    catch NoMatchFound
        // Do nothing and continue for loop (this gives the next permutation a
        // chance).
    endcatch
  endforeach // permutation
  if finalMappingsList == Ø then
    throw new NoMatchFound()
  endif
  return finalMappingsList
end // match
```

Appendix B: Composition Algorithm

A high level summary of the algorithm follows, showing how one aspect is woven into the base model. The result of the algorithm is a list of UCMs that were changed by the composition of the given aspect. In addition to the operation composeAspect, three more operations were implemented: removePointcutStubs, scan, and insertStub. Descriptions of these operations can be found in the comments below.

```
Operation:    composeAspect
Input:        Aspect aspect, UCMmodel baseUCM, MappingsList mappingsList
Output:       UCMmapList
Exception:    MalformedAdviceMap, CompositionNotRequired

begin // composeAspect
  UCMmapList updatedMaps = Ø
  foreach UCMadviceMap am of aspect
    // Removing all pointcut stubs from the advice map results in one or more
    // disjoint paths on the advice map. In-paths are replaced by end points.
    // Out-paths are replaced by start points. Empty paths are deleted before
    // proceeding. For each new start and end point on the advice map, the
    // following is created based on mappingsList:
    // a) mapping(s) to the base model
    // b) reference(s) to the closest path node in the base model with a mapping
    //    from the pointcut map (not necessary for start/end points mapped to
    //    named start/end points on pointcut map)
    // In the simplest case, there is only one mapping and one reference because,
    // there is only one pointcut map that is matched against one base map.
    // However, several mappings and references may have to be established
    // because one pointcut map may be matched against many base maps and
    // also because one pointcut stub may contain several pointcut maps.
    // In addition, a path that contains both (new start and end points) is marked
```

```
        // as around advice. An around advice that contains a false branch leading
        // directly to a new end point is marked as strong around advice.
        MappedUCMmap disjointPaths = am.removePointcutStubs(mappingsList)
        // Only consider this advice map if it was possible to match the pointcut
        // expression (in this case a mapping to the base model exists)
        if disjointPaths.getNumberOfMappedPathNodes() > 0 then
            add disjointPaths to updatedMaps
            foreach mapped PathNode pn of disjointPaths
                // A mapped path node is either a new start point or a new end point.
                // Scan disjointPaths to find the corresponding end or start point,
                // respectively. In case of a new start point, DisjointPath is scanned
                // forward. In case of a new end point, DisjointPath is scanned
                // backwards. Scan throws a MalformedAdviceMap exception if no or
                // more than one corresponding path node is found.
                PathNode correspondingPathNode = disjointPaths.scan(pn)
                // Remembering the correspondingPathNode for each pn essentially
                // duplicates the mapping from pn to the path node in the base model
                // for the correspondingPathNode.
                pn.addCorrespondingPathNode(correspondingPathNode)
            endforeach // PathNode
            foreach mapped PathNode pn of disjointPaths
                // Go through all mappings and references created by
                // removePointcutStubs
                foreach Mapping m of pn
                    // Insert a stub at the path node in the base model identified by m
                    // (see (a) above). The insertion point is on the path segment
                    // towards the referenced path node of pn for the mapping m (see
                    // (b) above). Plug-in bindings are also established from the stub to
                    // pn and to pn.getCorrespondingPathNode.
                    // If a stub has already been inserted at the same location, a new
                    // stub is not inserted but the existing stub is made dynamic (if it is
                    // static) and only a new plug-in map is added.
                    // If the path of pn is marked as around advice, then the inserted
                    // stub is labeled with up and down indicators.
                    // If the path of pn is marked as strong around advice, then the out-
                    // path or in-path of the inserted stub is removed (if pn is an end or
                    // start point, respectively). The corresponding plug-in binding is also
                    // removed.
                    UCMmap updatedBaseMap = m.getBaseMap().insertStub(m)
                    add updatedBaseMap to updatedMaps
                endforeach // Mapping
            endforeach // PathNode
        endif
    endforeach // UCMadviceMap
    if updatedMaps == Ø then
        // No composition occurred because no mappings were established.
        throw new CompositionNotRequired()
    endif
    return updatedMaps
end // composeAspect
```