Modeling Early Availability Requirements Using Aspect-Oriented Use Case Maps

Jameleddine Hassine[1], Gunter Mussbacher[2], Edna Braun[2], and Mohammad Alhaj[3]

[1] Department of Information and Computer Science, KFUPM, Saudi Arabia
jhassine@kfupm.edu.sa
[2] School of Electrical Engineering and Computer Science, University of Ottawa, Canada
{gunterm,ebraun}@eecs.uottawa.ca
[3] Department of Systems and Computer Engineering, Carleton University, Canada
malhaj@sce.carleton.ca

Abstract. Non-functional requirements such as availability, reliability, and security are often crucial in designing and implementing distributed real-time systems. As a result, such non-functional requirements should be addressed as early as possible in the system development life-cycle. The widespread interest in dependability modeling and analysis techniques at the requirements elicitation and analysis stage provides the major motivation for this research. This paper presents a novel approach to describe high-level availability requirements using the Aspect-oriented Use Case Maps (AoUCM) language. AoUCM adds aspects-oriented concepts to the Use Case Maps (UCM) language, part of the ITU-T User Requirements Notation (URN) standard. The proposed approach relies on a mapping of availability architectural tactics to reusable AoUCM models, allowing availability tactics to be encapsulated early in the software development life-cylce. Initial tool support for the resulting availability extensions, is provided by the *jUCMNav* tool. We demonstrate the applicability of our approach using a case study of *Lawful Intercept (LI)*, an IP router feature.

1 Introduction

The Use Case Maps (UCM) language, part of the ITU-T User Requirements Notation (URN) standard [1], is a high-level visual scenario-based modeling language that has gained momentum in recent years within the software requirements community. Use Case Maps can be used to capture and integrate functional requirements in terms of causal scenarios representing behavioral aspects at a high level of abstraction, and to provide the stakeholders with guidance and reasoning about the system-wide architecture and behavior.

The Use Case Maps language, extended with aspect-oriented modeling, resulted in the Aspect-oriented Use Case Maps (AoUCM) language. AoUCM, part of the Aspect-oriented User Requirements Notation (AoURN), supports the modeling of scenario-based, crosscutting concerns during requirements activities, i.e., concerns that are difficult to encapsulate with UCM alone.

System non-functional aspects such as *availability* and *fault tolerance* are often overlooked and underestimated during the initial system design. To address this issue, the UCM language has been extended with availability features in [2] and [3]. In this research, we use the AoUCM language to model the well-known availability tactics, introduced by Bass et al. [4].

The widespread interest in dependability modeling, constitutes the major motivation of this paper. We, in particular, focus on the need to incorporate availability aspects at the very early stages of system development. This work builds upon and extends the work of Hassine and Gherbi [3], and serves the following purposes:

- It describes the availability tactics, introduced by Bass et al. [4], in a well-encapsulated way using the Aspect-oriented Use Case Maps language.
- It introduces an improved Aspect-oriented Use Case Maps language, capable of handling more concisely variations in an aspect such as availability.
- It provides a comparison between our approach and the availability modeling approach introduced in [2] and [3].
- It extends our ongoing research towards the construction of an Aspect-oriented User Requirements Notation (AoURN) framework for the description and analysis of dependability aspects in the very early stages of system development life cycle.

The remainder of this paper is organized as follows. Section 2 introduces the concept of availability and provides an overview of the existing availability description approaches. Section 3 describes the proposed AoUCM-based availability models. A brief discussion of the advantages and shortcomings of the approach is provided in Section 4. A case study of an IP-based router feature, named LI (Lawful Intercept) is presented in Section 5 demonstrating the applicability of our approach. Finally, Section 6 covers conclusions and future work.

2 Availability Requirements

Several definitions of availability have been proposed [5,6,7,8,9]. According to ISO [5], the availability of a system may be defined as the degree to which a system or a component is operational and accessible when required for use. The ITU-T recommendation E.800 [8] defines *availability*, as the ability of an item to be in a state to perform a required function at a given instant of time, or at any instant of time within a given time interval, assuming that the external resources, if required, are provided. Availability has been treated by the field of dependability [6,7,9]. Bass et al. [4] have introduced the notion of tactics as *architectural building blocks* of architectural patterns. The authors [4] have provided a comprehensive categorization of availability tactics based on whether they address fault detection, recovery, or prevention. Figure 1 illustrates these four categories:

1. **Fault Detection** tactics are divided into
   (1) *Ping/Echo* (determines reachability and the round-trip delay through the associated network path),
   (2) *Heartbeat* (reports to system monitor when a fault is incurred), and
   (3) *Exception* (detects faults such as divide by zero, bus, and address faults).
2. **Fault Recovery-Preparation and Repair** tactics are divided into
   (1) *Voting* (A voter component decides which value to take in a redundant environment),
   (2) *Active Redundancy* (called also *hot redundancy*, refers to a configuration where all redundant spares maintain synchronous state with the active node(s)),
   (3) *Passive Redundancy* (called also *warm redundancy*, refers to a configuration where redundant spares receive periodic state updates from active node(s)), and
   (4) *Spare* (called also *cold redundancy*, refers to a configuration where the redundant spares remain out of service until a switch-over or fail-over occurs).
   It is worth noting that the application of one tactic may assume that another tactic has already been applied. For example, the application of *voting* may assume that some form of redundancy exists in the system.
3. **Fault Recovery-Reintroduction** tactics are divided into
   (1) *Shadow* (refers to operating a previously failed component in a *shadow mode* for a predefined duration of time),
   (2) *Rollback* (allows the system state to be reverted to the most recent consistent state), and
   (3) *State Resynchronization* (ensures that active and standby components have synchronized states).
4. **Fault Prevention** tactics include
   (1) *Removal from Service* (refers to placing a system component in an out-of-service state for the purpose of mitigating potential system failures),
   (2) *Transactions* (typically realized using *atomic commit protocols*), and
   (3) *Process Monitor* (monitors the health of a system).

In a closely related work, Scott and Kazman [10] have proposed a refined version of Bass et al. categorization [4]. However, their proposed classification considers tactics that are specific to inter-networking devices like switches and packet routers. Examples of such tactics include *Non-Stop Forwarding* (which maintains the proper functionning of user data plane in case of a failure) and *MPLS ping* (ensures timely ping responses in an MPLS-based network).

In this research, we adopt the more general availability tactics introduced by Bass et al. [4], as a basis for extending the Aspect-oriented Use Case Maps language [11] with availability annotations, allowing us to encapsulate the crosscutting availability concern in scenario models. These tactics have been proven in practice for a broad applicability in different industrial domains.

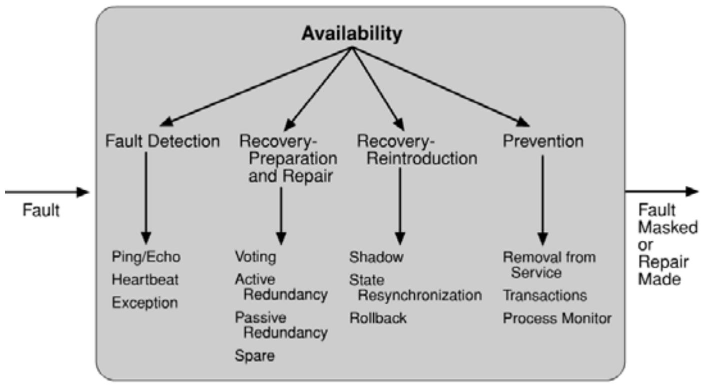


Fig. 1. Availability Tactics [4]

3 Aspect-Oriented Use Case Maps Availability Modeling

In this section, we introduce the AoUCM features and modeling elements that are relevant to our proposed availability extensions. For a complete description of the Aspect-oriented Use Case Maps language, interested readers are referred to [11,12,13,14]. AoUCM builds on the UCM language.

3.1 Use Case Maps

UCMs expressed by a simple visual notation allow for an abstract description of scenarios in terms of causal relationships between responsibilities (×, i.e., the steps within a scenario) along paths allocated to a set of components. These relationships are said to be causal because they involve concurrency, partial ordering of activities, and they link causes (e.g., preconditions and triggering events) to effects (e.g., postconditions and resulting events). UCMs help in structuring and integrating scenarios (in a map-like diagram) sequentially, as alternatives (with OR-forks/joins; —</>—), or concurrently (with AND-forks/joins; ⊣⊏/⊐⊢).

When maps become too complex to be represented as one single UCM, a mechanism for defining and structuring sub-maps becomes necessary. Path details can be hidden in sub-diagrams called plug-in maps, contained in stubs (◇) on a path. A plug-in map is bound (i.e., connected) to its parent map by binding the in-paths of the stub with start points (●) of the plug-in map and by binding the out-paths of the stub to end points (❙) of the plug-in map.

The UCM language supports path elements for failure points (☰), which describe exceptions raised explicitly at a specific point on a path causing the cancellation of the rest of the path (it does not address other concurrent paths). In this research, we annotate responsibilities with failure metadata attributes instead of using failure points (see Section 3.3) to improve the readability of UCM and AoUCM scenario models.

One of the strengths of UCMs resides in their ability to bind responsibilities to architectural components. Several kinds ot UCM components allow system entities (□) to be differentiated from entities of the environment (웃). UCM component relationships depend on scenarios to provide the semantic information about their dependencies. Components are considered to be dependent if they share the same scenario execution path even though no actual/physical connections are drawn between the components.

3.2 Aspect-Oriented Use Case Maps

Aspect-oriented UCM (AoUCM) [11,12,13,14] adds three core aspect-oriented concepts *concerns*, *composition rules*, and *pointcut expressions* to UCM. A concern is a new unit of encapsulation that captures everything related to a particular idea, e.g., availability. AoUCM treats concerns as first-class modeling elements. If a concern is not crosscutting, it can be described with the standard UCM notation. However, if it is crosscutting like availability, then it is best described using the AoUCM notation. Figure 2 presents a simple example with a base concern consisting of two responsibilities *RespA* and *RespB*.

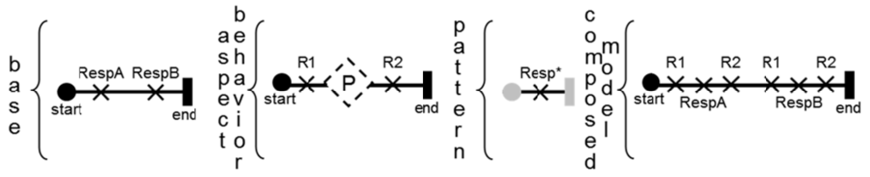


Fig. 2. AoUCM Example

Pointcut expressions are patterns that are specified by an aspect and matched in the base model. The pattern of the shown aspect matches against *Resp**, i.e., *RespA* and *RespB* in the base concern. If a match is found, the aspect is applied at the matched location in the base model. The behavior of an aspect is defined on a standard map. The only difference is that it contains a pointcut stub (◇) that represents the locations matched by the aspect's pattern. The causal relationship between the pointcut stub and the rest of the aspect map defines the *composition rule*. In the example in Fig. 2, *R1* is added before the matched locations, because *R1* occurs before the pointcut stub, while *R2* is added after the matched locations. This results in the composed model shown in Fig. 2. An AoUCM model may also use a replacement pointcut stub (◈) instead of a regular pointcut stub to remove any matched elements from the composed model. Furthermore, an element in the pattern may be defined as a variable ($) which allows the element to be reused in the definition of the aspect behavior.

In the following sections, we use AoUCM to describe high-level availability requirements. We adopt the availability tactics introduced by Bass et al. [4] as a basis for expressing availability requirements with AoUCM, making it possible to model availability as a properly encapsulated crosscutting concern. *Fault Recovery Preparation and Repair* as well as *Fault Recovery Reintroduction* categories (see Fig. 1) are merged to obtain what we refer to as *Fault Recovery.*

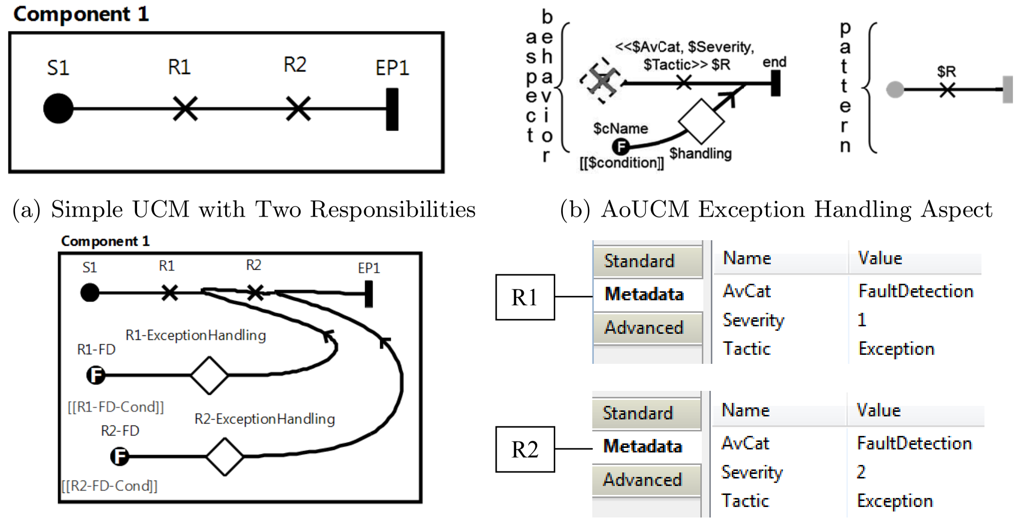
3.3 AoUCM Fault Detection Modeling

The specification of fault detection mechanisms is a key factor in implementing any availability strategy. Fault detection modeling involves the description of potential faults (i.e., *Exception tactic*) and the specification of liveness requirements using ping/echo and heartbeat [4].

Exceptions. *Exceptions* are modeled and handled at the scenario path level. Exceptions may be associated with any responsibility along the UCM execution path. The availability requirements of a responsibility can be modeled using three metadata attributes. The metadata approach allows for a more nuanced description of availability which is not possible with only *failure points.*

1. *AvCat:* Specifies the availability category, if any, that the responsibility is implementing. In the case of exceptions, it is specified as "*FaultDetection*".
2. *Tactic:* Denotes the type of the deployed tactic. In the case of exceptions, it is specified as "*Exception*".
3. *Severity* denotes the severity of the potential fault that might occur as a result of the execution of the responsibility. Three severity levels are considered: "1" (causes the component to stop working), "2" (impacts the component operations), and "3 or higher" (minor fault, not service impacting).

The realization of the exception tactic is assured by the definition of metadata attributes and by the existence of a related exception handling scenario. The actual handling of the exception (through the modeling of a failure scenario) is realized using the fault recovery tactic (Section 3.4). Figure 3(a) illustrates a simple UCM map with a main scenario executing in sequence two responsibilities *R1* and *R2.* Figure 3(c) shows an updated model where both responsibilities implement fault detection exception tactics specified using *AvCat* (i.e., *AvCat* = *FaultDetection*) and *Tactic* (i.e., *Tactic* = *Exception*) attributes. The metadata specifications are shown in Figure 3(d). Responsibility *R1* describes an exception with severity 1 (i.e., *Severity* = 1) while responsibility *R2* describes an exception with severity 2. Exceptions in responsibilities *R1* and *R2* are handled using two failure paths describing the recovery behavior. A failure path starts with a *failure start point* (🅕) [1,11] and a guarding condition (e.g., conditions *R1-FD-Cond* and *R2-FD-Cond* in Fig. 3(c)). In this example, the actual handling of the exceptions takes place within two stubs named *R1-ExceptionHandling* and *R2-ExceptionHandling.* The corresponding plug-in maps *handlingR1* and *handlingR2* are not shown at



(a) Simple UCM with Two Responsibilities

(b) AoUCM Exception Handling Aspect

(c) UCM Exception Handling

R1

Name	Value
AvCat	FaultDetection
Severity	1
Tactic	Exception

R2

Name	Value
AvCat	FaultDetection
Severity	2
Tactic	Exception

(d) Metadata Attributes for R1 and R2 in (c)

Composition Condition	Assignments						Selection of Plug-in Map for
$R =	$AvCat	$Severity	$Tactic	$cName	$condition	$handling	$handling
R1	Fault-Detection	1	Exception	R1-FD	R1-FD-Cond	R1-Exception-Handling	handlingR1
R2	Fault-Detection	2	Exception	R2-FD	R2-FD-Cond	R2-Exception-Handling	handlingR2

(e) Composition Matrix for AoUCM Exception Handling Aspect

Fig. 3. UCM and AoUCM Exception Handling

this point because the details of the exception handling are irrelevant for this discussion on failure detection. The plug-in maps describe fault recovery (Section 3.4). After handling the *R1* and *R2* exceptions, the path continues explicitly with responsibility *R2* and the end point *EP1*, respectively.

As can be clearly seen from Fig. 3(c), the exceptions add significant complexity to the basic scenario in Fig. 3(a). Furthermore, great similarities can be observed for exceptions of responsibilities *R1* as well as *R2*, e.g., both require a failure path. These similarities are captured in the aspect-oriented model for the exception tactic presented in Fig. 3(b). Before explaining the details of the aspect-oriented model, it must be noted that, despite the similarities, there are also many small variations from the failure path of one responsibility to the one of another responsibility (i.e., the guarding condition is different and the actual exception handling is different; the metadata attributes of the responsibility may also be different). Nevertheless, the overall structure is the same. With conventional AoUCM, a separate AoUCM model would have to be created for each combination of variations. However, this does not scale to what is needed for availability tactics. Therefore, the Aspect-oriented Use Case Maps language has been extended with the concept of a *composition matrix* that allows variations to be specified in a concise manner. With this approach, the AoUCM model in Fig. 3(b) describes the generic reusable properties of exception handling, while the composition matrix factors out the adaptation of this generic model to its application context.

The aspect-oriented model in Fig. 3(b) describes a replacement as indicated by the replacement pointcut stub ( ), i.e., the model elements matched by the pattern in Fig. 3(b) are replaced with the model elements that follow the replacement pointcut stub. In this case, the pattern describes a single variable, the responsibility (*$R*). This single responsibility is replaced by itself (*$R* is shown on the path following the replacement pointcut stub which means that the matched responsibility is reused by the aspect) but with several metadata attributes added. In addition, the failure path is added which merges with the path of *$R* after the responsibility. The metadata attributes and several elements of the failure path are also specified with variables (e.g., *$AvCat* and *$condition*). However, these variables are not defined in the pattern, contrary to the *$R* responsibility. This is where the composition matrix comes into play. Any variable that is not bound by the pattern must be defined in the composition matrix as shown in Fig. 3(e).

In this example, the values of these unbound variables depend on the responsibility that is matched by the pattern, i.e., the bound variable. The first column in the composition matrix allows the composition condition to be specified. In this case, *$R* can either be *R1* or *R2*. The second set of columns specify assignments. For example, if *$R* is matched against *R1*, then the metadata variable *$AvCat* needs to be assigned the value *FaultDetection* and the condition variable *$condition* needs to be assigned the value *R1-FD-Cond*. The last column in the composition matrix allows for the specification of a specific plug-in map of the stub defined in the second row of this column (i.e., *$handling* in our case).

For example, when *R1* is matched, then the plug-in map called *handlingR1* must be used as the plug-in map for the *handling* stub. Note that, in addition to the name of the plug-in map, more detailed plug-in bindings can be specified for more complex stubs with several in-paths and out-paths. This, however, is not required for availability tactics. Similarly, regular expressions can be used in the column for the composition conditions, but this is also not required for the example in Fig. 3(b). In general, the composition matrix collects all metadata specifications required for availability in one location that are otherwise spread out over the model as shown in Fig. 3(d), hence allowing all specifications related to availability to be encapsulated in one aspect.

When the availability aspect is applied to the scenario in Fig. 3(a), the composed result is equivalent to the UCM map in Fig. 3(c) including the annotations.

Ping/Echo and Heartbeat Tactics. *Ping/Echo* and *Heartbeat* tactics can be used to determine how long it takes to detect a fault. This can be achieved using the round-trip time and the number of missed pings/heartbeats. In [2] and [3], we have reused the UCM comment constructor to describe ping and heartbeat tactics. In this paper, we use metadata attributes instead. While both alternatives allow for a global description of availability requirements (i.e., attached to the entire UCM model rather than to one specific UCM construct), using metadata provides a structural and more intuitive way of representing attributes. In AoUCM, these global descriptions are attached to the availability aspect instead of the entire UCM model, but their specification otherwise remains the same. For example, a ping initiated by component C1 that must result in an echo response from C2 received within 2ms is specified as metadata *Ping* = "*C1;C2;2*". Similarly, a heartbeat, periodic message exchange such as "I'm alive", that is sent from component C1 towards component C2 with a polling interval of 2000 ms is defined as metadata *Heartbeat* = "*C1;C2;2000*".

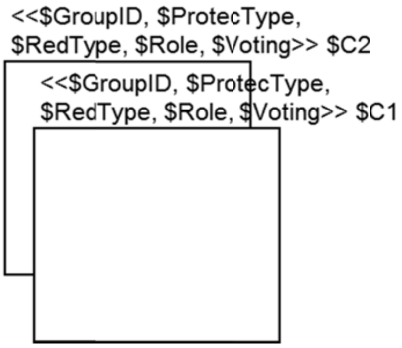
The *ping/echo* and *heartbeat* requirements can also be described using metadata attached to responsibilities (i.e., *AvCat* = "*FaultDetection*"; *Tactic* = "*Heartbeat*). Violations of *ping/echo* and *heartbeat* tactics are handled in scenario paths similar to the exception tactic. If this is the case, then an aspect similar to the AoUCM exception handling aspect is used to define the metadata and failure path for such responsibilities.

3.4 UCM Fault Recovery Modeling

Fault recovery tactics focus mainly on redundancy modeling in order to keep the system available in case of the occurrence of a failure. To model redundancy, we annotate UCM components with the following attributes:

- *GroupID*: A system may have many spatially redundant components of different types. The *GroupID* is used to identify the group to which a component belongs in a specific redundancy model.
- *Role*: Denotes whether a component is in *active* or *standby* role.
- *RedundancyType*: Specifies the redundancy type as *hot*, *warm*, or *cold*.

– *ProtectionType*: The minimal redundancy configuration is to have one active and one redundant node (commonly referred to as *1+1* redundancy). Other redundancy configurations are: *1:N* (refers to a configuration where one spare is used to protect multiple active nodes) and *M:N* (refers to a configuration where multiple spares are used to protect multiple active nodes).
– *Voting*: A boolean variable describing whether a component plays a voting role in a redundancy configuration.



<<$GroupID, $ProtecType, $RedType, $Role, $Voting>> $C2
<<$GroupID, $ProtecType, $RedType, $Role, $Voting>> $C1

(a) AoUCM Redundancy Aspect

Composition Condition	Assignments				
$C1 = ; $C2 =	$GroupID	$ProtecType	$RedType	$Role	$Voting
RP1;	G1	1+1	Hot	Active	false
RP2	G1	1+1	Hot	Stdby	false

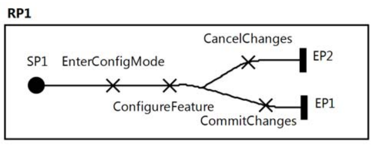
(b) AoUCM Composition Matrix

Fig. 4. AoUCM Redundancy Aspect with Composition Matrix

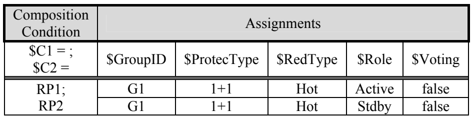
Since component redundancy has to be described visually and the involved redundant components share the same scenario path, an elegant way to illustrate such a configuration is to use overlapping components. Note that the current URN standard [1] does not allow overlapping components (while the *jUCMNav* tool [15] does support such a feature).

Figure 4 illustrates an example of a system with two components *RP1* and *RP2* participating in a 1+1 hot redundancy configuration. *RP1* is in active role while *RP2* is in standby role. None of these two components is taking part in a voting activity (i.e., Voting: *false*).

The presented redundancy annotations deal with the static description of component availability requirements. The operational implications of such availability requirements, in case of failure for instance, can be described using the

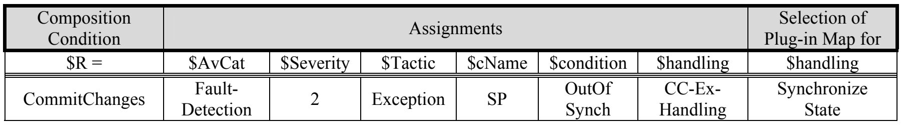


(a) Feature Configuration Example



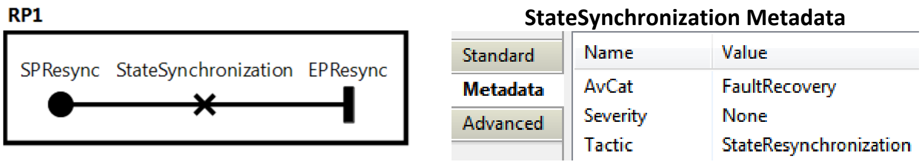
Composition Condition	Assignments				
$C1 = ; $C2 =	$GroupID	$ProtecType	$RedType	$Role	$Voting
RP1; RP2	G1	1+1	Hot	Active	false
	G1	1+1	Hot	Stdby	false

(b) Redundancy AoUCM Composition Matrix



Composition Condition	Assignments						Selection of Plug-in Map for
$R =	$AvCat	$Severity	$Tactic	$cName	$condition	$handling	$handling
CommitChanges	Fault-Detection	2	Exception	SP	OutOf Synch	CC-Ex-Handling	Synchronize State

(c) Exception Handling AoUCM Composition Matrix



Name	Value
AvCat	FaultRecovery
Severity	None
Tactic	StateResynchronization

(d) Synchronize State Plug-in Map and Metadata Attributes

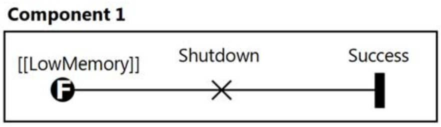
Fig. 5. Implementation of the State Resynchronization Tactic with AoUCM

UCM scenario path. Hence, reintroduction tactics such as *Shadow*, *State Resynchronization*, and *Rollback* can be described using the metadata attributes associated to responsibilities. Typically, these tactics are used in the exception handling scenario in response to fault detection, i.e., they are used in the stub in the exception handling aspect as discussed in Section 3.3.

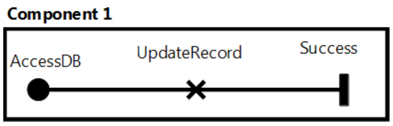
Figure 5 illustrates a feature configuration scenario on a dual route processor (RP) system. The configuration of a new feature may result in having the active and the standby route processors (respectively *RP1* and *RP2*) in Out-of-Sync state (e.g., configuration is not applied to the standby RP). The detection of such a situation would trigger an exception path (i.e., precondition *OutOfSych* is satisfied) and causes both RPs to synchronize again (using responsibility *StateSynchronization*). The basic scenario is defined in Fig. 5(a). Figures 5(b) and 5(c) show the composition matrices for the redundancy aspect and the exception handling aspect, respectively. The former insures that two redundant components, *RP1* and *RP2*, are defined using the redundancy aspect from Fig. 4. The latter composition matrix is for the exception handling aspect in Fig. 3(b). As stated earlier, the composition matrix defines which plug-in map to use. In this case, it is the *Sychronize State* plug-in map shown in Fig. 5(d) which may even be provided as a predefined specification of the state synchronization tactic as part of the AoUCM availability aspect. Since this plug-in map and its responsibility *StateSynchronization* are part of the availability aspect, the required metadata attributes for the responsibility are specified directly for the responsibility.

3.5 UCM Fault Prevention Modeling

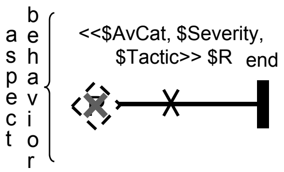
Annotations presented in Sections 3.3 can be used to accommodate this category. Indeed, responsibilities can be annotated with availability metadata attributes specifying a removal from service property, transactions properties, and process monitoring properties. For example, Fig. 6(a) illustrates a UCM scenario describing the placement of a component in an out-of-service state by shutting it down (i.e., responsibility *Shutdown*) to prevent potential system failures in case the component is running low on memory, and Fig. 6(b) provides a scenario of updating a database record using a two-phase-commit type of transaction (a.k.a. *2PC*). Failing to ensure the two phase commit requirement, would trigger an implicit rollback to undo the record update (not shown in the figure). The aspect in Fig. 6(c) uses the same pattern as specified in Fig. 3(b), but the aspect behavior is simpler as it only replaces the matched responsibility with the same but annotated responsibility. The composition matrix of the fault prevention aspect in Fig. 6(c) ensures that the responsibilities are annotated accordingly.



(a) Removal From Service Tactic



(b) Transactions Tactic



Composition Condition	Assignments		
$R =	$AvCat	$Severity	$Tactic
Shutdown	FaultPrevention	None	RemovalFromService
Update Record	FaultPrevention	None	Transactions

(c) AoUCM Fault Prevention Aspect with Composition Matrix

Fig. 6. AoUCM Fault Prevention Modeling

4 Discussion

Our proposed approach relies on a mapping of availability architectural tactics proposed by Bass et al. [4] to generic, reusable AoUCM models, which allows the tactics to be encapsulated at the early phases of the software development life-cycle. In previous work [2], the use of comments is suggested to add information about the same availability tactics to a UCM model [3]. This paper uses metadata to describe availability tactics with an aspect-oriented approach but the same metadata could also be added directly to a conventional UCM model. These three options (comments, UCM metadata, AoUCM metadata) share the same goal of modeling the availability tactics at the early phases of development, which allows earlier detection of design errors and helps modelers to select between different design alternatives. We briefly compare the advantages and shortcomings of these three approaches.

1. **Comment-based option:** This option is sufficient for visualizing availability tactics in a model but does not lend itself to further analysis, because the availability information is captured in a non-formalized way. Another disadvantage of this approach is that comments cannot be attached to individual UCM model elements but only to UCM maps. The availability tactics, however, require information to be attached to individual UCM model elements. This option is hence not further considered.
2. **UCM metadata-based option:** This option formalizes availability information in metadata, making it easier to use this information in automated model analysis. However, availability information is not encapsulated well in one location in the model but rather distributed over the whole model, making maintenance, reuse, and evolution of the availability information more difficult.
3. **AoUCM metadata-based option:** This option also uses metadata but localizes the availability information in one availability aspect (which may be sub-divided into smaller availability aspects dealing specifically with exception handling, fault prevention, etc.). Previous comparisons of UCM and AoUCM models [13] indicate that AoUCM models exhibits better modularity, reusability, and maintainability than UCM models. Essentially, a larger vocabulary size (because aspects are introduced) is trade-off against better separation of concerns, less coupling, and stronger cohesion. These results also apply to the availability aspect. The effects are less pronounced for simple metadata assignments which could be maintained through additional tool management features without the need for aspect-oriented techniques. However, the availability tactics of exception detection and handling can greatly benefit from an aspect-oriented approach because of the high number of model elements affected by these tactics. Hence, the model can be simplified significantly. The addition of *composition matrices* further improves the handling of small variations in an aspect such as availability. The common, generic, reusable part of an availability tactic can be captured concisely in a rather simple AoUCM aspect, while the variations are factored out into the composition matrices. A disadvantage of the AoUCM metadata-based option is that the AoUCM model is more fragmented and an automated composition is required to merge the availability aspect into the system model. However, these disadvantages can be alleviated by available tool support with the jUCMNav editor [15] which helps modelers with the navigation through AoUCM models and the composition of AoUCM models.

5 Case Study: Lawful Intercept (LI) Feature

In this section, we illustrate our proposed approach using a case study of a Lawful Intercept (LI) feature, running on a Cisco CRS-1 router[1]. LI allows service providers to meet the requirements of law enforcement agencies (e.g., state and federal police, intelligence agencies, and independent commissions against corruption) to provide authorized interception of VoIP and data traffic at content IAP (intercept access point) routers.

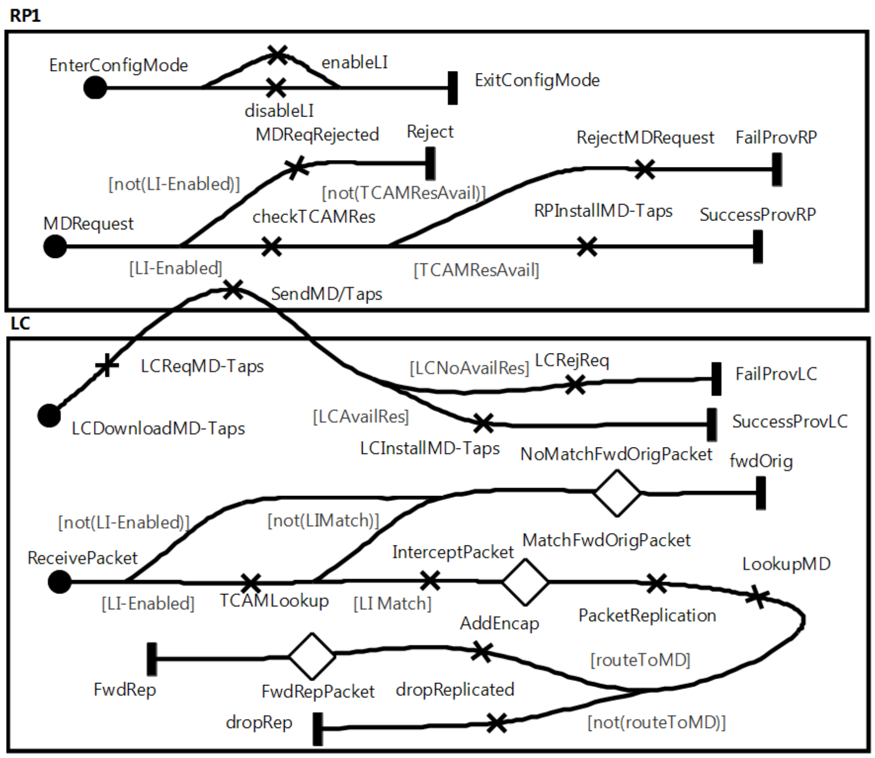
A minimal CRS-1 router architecture consists of one or many route processors (RP) cards (that provide route processing, alarm, fan, and power supply controller function), one or many ingress line cards (that process incoming traffic), one or many egress line cards (that process outgoing traffic), and a switch fabric (receives user data from ingress cards and performs the switching necessary to route the data to the appropriate egress cards). Since LI is an ingress feature (i.e., applied on ingress line cards), its description is abstracted from the switch fabric and the egress line cards. LI is described using an AoUCM scenario model (Figure 7) bound to an architecture composed of two Route Processors (RP) in a hot redundancy mode (RP1 is active role while RP2 is the standby) and one ingress line card (LC). The specification of the redundancy is exactly the same as in Fig. 5, and is hence not repeated here.

In a typical operation, a lawfully authorized intercept request is provisioned by the MD (Mediation Device) on the content IAP (Intercept Access Point), which is the device within the network used to intercept the targeted information. The IAP is responsible for identifying the IP traffic of interest and forwarding it to the MD, while remaining undetectable by the intercept subject. In a typical operation, the router (e.g., Cisco CRS-1) is the content IAP.

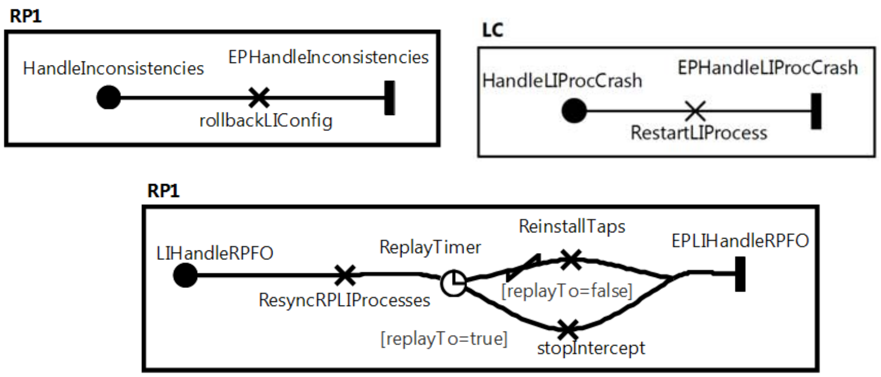
Figure 7(a) provides a high level description of basic LI scenarios. The Mediation Device (MD) crafts an interception request based on the content to be collected and sends it to the router. Upon reception (start point *MDRequest*), the MD request may be rejected (*MDReqRejected*) in case LI is disabled on the router, otherwise a check whether TCAM (Ternary content-addressable memory) resources are available (*checkTCAMRes*) is performed. Requested MD/Tap entries are programmed (*RPInstallMD-Taps*) when TCM resources are available, and they are rejected (*RejectMDRequest*) otherwise. Enabling and disabling LI may cause unexpected configuration inconsistencies. This exception is handled with the exception handling aspect introduced earlier. The composition matrix entries for *enableLI* and *disableLI* state that the *Rollback-LIConfiguration* plug-in map handles the exception by reverting back to the last consistent configuration, i.e., a rollback occurs as described by responsibility *rollbackLIConfig* and its composition matrix entry in Fig. 8.

The composition matrix also shows that there is a risk of an RP failover (RPFO), when installing new MD/taps in the TCAM (*RPInstallMD-Taps*) and the exception handling is defined on the *Resynchronize-Process-RPLI* plug-in

[1] `www.cisco.com/en/US/prod/collateral/routers/ps5763/prod_brochure0900aecd800f8118.eps`



(a) Lawful Intercept High Level Description



(b) Rollback-LIConfiguration, Restart-Process-LI, and Resynchronize-Process-RPLI Plug-in Maps

Fig. 7. Lawful Intercept AoUCM Modeling

Composition Condition	Assignments						Selection of Plug-in Map for
$R =	$AvCat	$Severity	$Tactic	$cName	$condition	$handling	$handling
enableLI	Fault-Detection	1	Exception	Handle-Incon-sistencies	Config-Inconsistencies-eLI	CI-Ex-Handling	Rollback-LIConfiguration
disableLI	Fault-Detection	1	Exception	Handle-Incon-sistencies	Config-Inconsistencies-dLI	CI-Ex-Handling	Rollback-LIConfiguration
RPInstallMD-Taps	Fault-Detection	1	Exception	LIHandle-RPFO	RPFO	RPFO-Ex-Handling	Resynchronize-Process-RPLI
LCInstallMD-Taps	Fault-Detection	1	Exception	HandleLI-ProcCrash	LIProcessCrash	LIPC-Ex-Handling	Restart-Process-LI

Composition Condition	Assignments		
$R =	$AvCat	$Severity	$Tactic
rollbackLIConfig or ResyncRPLIProcesses or RestartLIProcess	FaultRecovery	None	Rollback

Fig. 8. Lawful Intercept Exception Handling and Fault Recovery AoUCM Composition Matrices

map. At any point in time, the MD has the responsibility to detect the loss of the taps. LI uses a replay timer, an internal timeout that provides enough time for MD to re-provision tap entries while maintaining existing tap flows. It resets and starts on the active RP when an RPFO takes place. After replay timeout (the zigzag path leaving the timer in Fig. 7(b)), interception stops on taps that are not re-provisioned.

In the ingress LC, we distinguish two scenarios. The first one deals with downloading MD and Tap entries from the RP and installing them (*LCInstallMD-Taps*) on the LC if resources are available, otherwise the download request is rejected (*LCRejReq*). The second scenario deals with the interception of traffic. Packets received while LI is disabled are forwarded to their destination (stub *NoMatchFwdOrigPacket*). Packets matching Tap entries are intercepted (*InterceptPacket*), then forwarded to their original destinations (stub *MatchFwdOrigPacket*), and replicated (*PacketReplication*). If there is a valid route to MD (*LookupMD*), packets will be encapsulated (*AddEncap*) and sent to the MD (stub *FwdRepPacket*), otherwise the replicated packets are dropped (*dropReplicated*).

The LI process may crash while installing new MDs/Taps. This is again described with the exception handling aspect (Figure 7(b)). The composition matrix entry for *LCInstallMD-Taps* states that the exception handling is defined on the *Restart-Process-LI* plug-in map which restarts the process (*RestartLIProcess*).

6 Conclusions and Future Work

In this work, we have incorporated availability information at the very early stages of system development with the help of an aspect-oriented approach. To this end, we have extended the Aspect-oriented Use Case Maps (AoUCM) language (which is based on the ITU-T User Requirements Notation (URN) standard) with availability metadata covering the well known availability tactics proposed by Bass et al. [4]. AoUCM adds aspect-oriented concepts to UCM which allow the availability architectural tactics to be encapsulated at the early phases of system development. Availability tactics typically need to be applied to numerous locations in a system. A characteristic of availability tactics is that each time a slightly different availability tactic needs to be applied. Therefore, we have extended the AoUCM language with the notion of *composition matrices* which enables these small variations of the availability tactics to be specified concisely. We envision that the proposed composition matrix is useful for other concerns in addition to availability. For future work, we plan to investigate which concerns could benefit from composition matrices. We also aim to study how to map AoUCM availability concepts into the Service Availability Forum's Availability Management Framework configurations. Our goal is to allow for the early reasoning about availability aspects and promote the portability and the reusability of the developed systems across different platforms.

References

1. International Telecommunication Union: Recommendation Z.151 (10/12), User Requirements Notation (URN) – language definition (2012), `http://www.itu.int/rec/T-REC-Z.151/en`
2. Hassine, J.: Early Availability Requirements Modeling using Use Case Maps. In: 8th International Conference on Information Technology – New Generations, ITNG 2011, pp. 754–759. IEEE Computer Society (2011)
3. Hassine, J., Gherbi, A.: Exploring Early Availability Requirements Using Use Case Maps. In: Ober, I., Ober, I. (eds.) SDL 2011. LNCS, vol. 7083, pp. 54–68. Springer, Heidelberg (2011)
4. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley (2003)
5. ISO/IEC/IEEE: 24765:2010(E) - Systems and software engineering – vocabulary, pp. 1–418 (2010), `http://dx.doi.org/10.1109/IEEESTD.2010.5733835`
6. Avizienis, A., et al.: Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing 1(1), 11–33 (2004)
7. Hatebur, D., Heisel, M.: A Foundation for Requirements Analysis of Dependable Software. In: Buth, B., Rabe, G., Seyfarth, T. (eds.) SAFECOMP 2009. LNCS, vol. 5775, pp. 311–325. Springer, Heidelberg (2009)
8. International Telecommunication Union: Recommendation E.800 (09/08) Definitions of terms related to quality of service (2008), `http://www.itu.int/rec/T-REC-E.800/en`
9. Laprie, J., Avizienis, A., Kopetz, H.: Dependability: Basic Concepts and Terminology. Springer (1991)
10. Scott, J., Kazman, R.: Realizing and refining architectural tactics – Availability. Carnegie Mellon University – Software Engineering Institute (2009), `http://www.sei.cmu.edu/library/abstracts/reports/09tr006.cfm`
11. Mussbacher, G.: Aspect-oriented user requirements notation. Ph.D. thesis, University of Ottawa (2010), `http://lotos.csi.uottawa.ca/ucm/pub/UCM/VirLibGunterPhDThesis/Aspect-Oriented_User_Requirements_Notation.pdf`

12. Mussbacher, G., Amyot, D.: Extending the user requirements notation with aspect-oriented concepts. In: Reed, R., Bilgic, A., Gotzhein, R. (eds.) SDL 2009. LNCS, vol. 5719, pp. 115–132. Springer, Heidelberg (2009)
13. Mussbacher, G., Amyot, D., Araújo, J., Moreira, A.: Requirements Modeling with the Aspect-oriented User Requirements Notation (AoURN): A Case Study. In: Katz, S., Mezini, M., Kienzle, J. (eds.) Transactions on AOSD VII. LNCS, vol. 6210, pp. 23–68. Springer, Heidelberg (2010), `http://dx.doi.org/10.1007/978-3-642-16086-8_2`
14. Mussbacher, G., et al.: AoURN-based modeling and analysis of software product lines. Software Quality Journal 20(3-4), 645–687 (2012), `http://dx.doi.org/10.1007/s11219-011-9153-8`
15. jUCMNav v5.2.0: jUCMNav Project (tool, documentation, and meta-model) (2013), `http://jucmnav.softwareengineering.ca/jucmnav`