Towards the Generation of AMF Configurations from Use Case Maps based Availability Requirements

Jameleddine Hassine[1] and Abdelwahab Hamou-Lhadj[2]

[1] Department of Information and Computer Science
King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia
jhassine@kfupm.edu.sa
[2] Electrical and Computer Engineering Department
Concordia University, Montréal, Canada
abdelw@ece.concordia.ca

Abstract. Dependability aspects, such as availability and security, are critical in the design and implementation of distributed real-time systems. As a result, it is becoming crucial to model and analyze dependability requirements at the early stages of system development life-cycle. The Service Availability Forum (SA Forum) has developed a set of standard API specifications to standardize high-availability platforms. Among these specifications, the Availability Management Framework (AMF) is the service responsible for managing the availability of the application services by handling application redundant components, dynamically shifting a workload of a faulty component to a healthy component. To manage service availability, AMF requires a configuration of the application it manages. This configuration consists of a logical view of the organization of the application's services and components. Recognizing the need to plan for availability aspects at the early stages of system development life-cycle, this paper proposes an approach to map high level availability requirements into AMF configurations. The early availability requirements are expressed in terms of the Use Case Maps (UCM) language, part of the ITU-T User Requirements Notation (URN) standard. Our approach allows for the early reasoning about availability aspects and promotes the portability and the reusability of the developed systems across different platforms.

1 Introduction

Several definitions of availability have been proposed [1, 2, 9, 20, 8, 14]. According to ANSI [1], 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 [9] 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. Wang and Trivedi [20] define the availability as the probability of service provision upon request, assuming that the time required for satisfying each service request is short and negligeable.

Availability requirements can be very stringent as in highly available systems used in telecommunication services (a.k.a. 5 nines (99,999%)). Many proprietary approaches have been proposed to achieve high-availability. However, such solutions hinders the portability of applications from one platform to another. To address this issue, the Service Availability Forum (SA Forum) [18], a consortium of telecommunications and computing companies was created to define and standardize high availability solutions for systems and services. SA Forum [18] supports the delivery of highly available carrier-grade systems through the definition of standard interfaces for availability management [16], software management [5] and several other utility services availability middleware services [17]. SA Forum [18] has developed an Application Interface Specification (AIS), which includes the Availability Management Framework (AMF)[16]. AMF constitutes the core component of the middleware as it is the service responsible for managing the high availability of the services.

An AMF configuration describes an application in terms of logical entities representing services and service provider resources. The application software, managed by AMF, is described by the vendor in the Entity Types File (ETF) [5] in terms of entity prototypes which characterize the deployment options, constraints and limitations of the software. Many attempts to construct AMF configurations from user and vendor requirements, have been addressed in the literature [15, 12, 4, 13, 11]. Salehi et al.[15] have presented a model based approach for generating AMF configurations using UML profiles. The authors have defined a set of transformation rules, expressed in the ATLAS Transformation Language (ATL), to generate AMF configurations from UML model elements representing software entities and configuration requirements. Kanso et al. in [12] and [13] have adopted a code-centric approach. The authors have used a *Configuration Requirements* (CR) artifact to describe AMF middleware requirements for a given application (i.e., ETF types and configuration parameters such as the number of service units (SUs), component service instances (CSIs), etc., provided by a configuration designer), allowing for automatic generation of AMF configuration. In a closely related work Colombo et al. [4] have proposed an approach that aims at producing multiple sets of Configuration Requirements (CR) (resulting in multiple AMF configurations) from User Requirements (UR) and based on a selection mechanism of ETF types [5].

The Use Case Maps (UCM) language, part of the ITU-T User Requirements Notation (URN) standard [10], is a high-level visual scenario-based modeling language that has gained momentum in recent years within the software requirements community. Use Case Maps [10] 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. System non-functional aspects such as *availability* and *security* are often overlooked and underestimated during the initial system design. To address this issue, the UCM language has been extended with availability information in [6] and [7]. These extensions cover the well-known availability tactics, introduced by Bass et al. [3].

Availability requirements modeling and analysis constitute the major motivation of this research. We, in particular, focus on the need to express system availability aspects while assuring portability of applications. This paper serves the following purposes:

- It extends the UCM-based availability annotations introduced in [6] and [7] to accommodate Availability Management Framework (AMF) [16] concepts (e.g., *Service group*, *service unit*, etc.).
- It provides a mapping of the newly introduced UCM-based availability requirements to AMF (Availability Management Framework) [16] concrete APIs.
- It complements the approach introduced in [12]. The configuration requirements (CR) model can be extended and automatically derived from UCM specifications annotated with availability aspects.
- It extends our ongoing research towards the construction of a UCM-based framework for the description and analysis of availability aspects in the very early stages of system development life cycle.

The remainder of this paper is organized as follows. The next section introduces the Availability Management Framework (AMF). Section 3 presents our UCM-AMF configuration generation approach. Use Case Maps availability modeling is provided in Section 4 followed by a discussion in Section 5. An illustrative example is presented in Section 6 to demonstrate the applicability of our approach. Finally, conclusions and future work are outlined in Section 7.

2 The Availability Management Framework (AMF)

The role of AMF is to manage the availability of applications in a clustered environment (note that we use here the term AMF to refer to an implementation of the AMF standard, since AMF is just a specification). To do so, AMF needs a configuration of the components (service providers) and the services.

An AMF configuration consists of a number of logical entities, introduced in the AMF standard [16]. An example is shown in Figure 1. In this figure, we can see that each node has two components grouped in a logical AMF entity called service units (SUs). The services are represented by service instances, also known as component service instances (CSIs). Multiple CSIs can be assigned to the same SU and are grouped in another of AMF entities called service instance (SI).

There are two additional AMF logical entities used for deployment purpose: The cluster and the node. The cluster consists of a collection of nodes under the control of AMF.

AMF supports five different redundancy models, namely, 2N, N+M, N Way, N Way Active, and No-Redundancy. These redundancy models vary in the level of protection they provide. For example, in a 2N model (see Figure 1) there is only

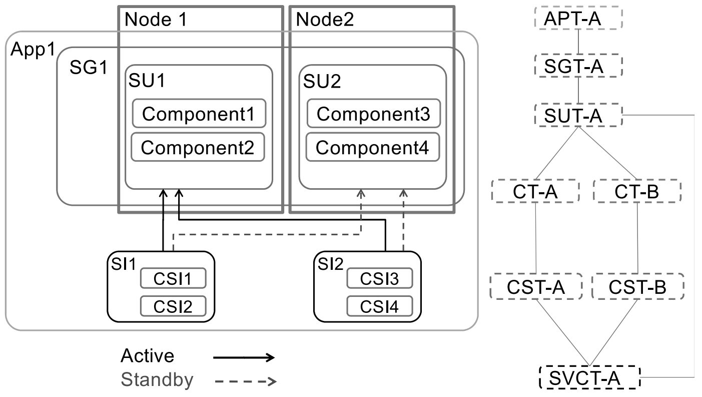


Fig. 1. An example of AMF configuration and its services. The left figure shows a configuration. The right figure shows type of the configuration entity types (taken from [15])

one SU that is active for an SI and another SU that is used as a standby. In the N+M model, N SUs share the active assignments and M share the standbys. Like 2N models, N+M models allow at most one active and one standby assignment for each particular SI.

The set of SUs that follow the same redundancy model are grouped in AMF logical entities called service groups (SGs). The same configuration can have many SGs. For example, some components can be protected using a 2N redundancy model, whereas others (in different SUs) can be protected using an N+M model. Similarly, multiple SGs can be grouped to form an application. A good reference on AMF redundancy models can be found in [11].

When building a configuration, there are several decisions that need to be made. For example, it is important to put highly coupled components in the same SU. In case a failure happens, we can failover the whole SU to recover the service. In addition, there should be a way for AMF to know which CSIs to assign to a specific component depending on whether the component can provide the service or not. Many other similar decisions are needed to produce valid configurations. AMF types aim to do just that.

In AMF, every entity has a type except the cluster and the node. These AMF types are derived from AIS standard, known as the Entity Types File (ETF) [5], which is a file provided by the software vendor to describe the characteristics of the software system that runs under the control of AMF. ETF types should be thought of as power types (or meta types), that describe the possible ways a software system can be deployed on an AMF cluster. Once a configuration is built, only instances of some ETF types are used to construct AMF types. Table 1 describes the ETF types. Each row shows an ETF type, a description, and the AMF entity for which the type derives from that ETF type.

ETF Type	Description	AMF Entity
Component Type (CT)	It describes the component version (used more particularly during upgrades), the component service types that the component of this type can provide, and the component capabilities (how many active and standby CSIs the component of this type can support).	Component
Component Service Type (CST)	It describes the service attributes (e.g., range of IP addresses the component that handles this service can provide).	Component Service Instance (CSI)
Service Unit Type (SUT)	It describes the service type that an SU can provide as well as the set of component types of the components that an SU of this type can contain.	Service Unit (SU)
Service Type (SVCT)	It describes the set of component service types from which its SIs can be built. The service type may limit the number of CSIs of a particular CS type that can exist in a service instance of the service type. It is also used to describe the type of services supported by an SU type.	Service Instance (SI)
Service Group Type (SGT)	It describes the service group. Typical attributes of SGT is the redundancy model (e.g., 2N, N+M, etc.). It also specifies the supported SUTs. In other words, an SG can contain and SU only if its SGT supports the SU's SUT.	Service Group
Application Type	Similar to SGT, an application type defines the SGTs types that are supported by the applications of this type.	Application

Table 1. ETF Types

3 Extending The Use Case Maps Language with AMF Concepts

Figure 2 illustrates our approach for extending Use Case Maps [10] with AMF [16] concepts. Note that the configuration generation process is outside the scope of this paper. Many algorithms, including the work of Kanso et al. [12], and Salehi et al. [15], exist to generate automatically AMF configurations. Our focus is to model AMF concepts using the Use Case Maps language [10]. By doing so, an AMF configuration (at the conceptual level) will always be represented as a UCM map. Note, however, that the AMF standard defines an XML carrier to exchange configurations among tools. This XML representation of AMF configurations can also be generated from UCM (extended with AMF concepts).

The proposed availability extensions are added orthogonally to the UCM specification (functional model and binding architecture). These extensions are modeled using *Metadata* mechanism, which is a mechanism used to support the profiling of the language to a particular domain. *Metadata* are described as name-value pairs that can be used to tag any URN specification or its model elements, similar to stereotypes in UML. *Metadata* instances provide modelers with a way to attach user-defined named values to most elements found in a URN specification, hence providing an extensible semantics to URN. *Metadata* is supported by the jUCMNav tool [19], the most comprehensive URN [10] tool available to date.

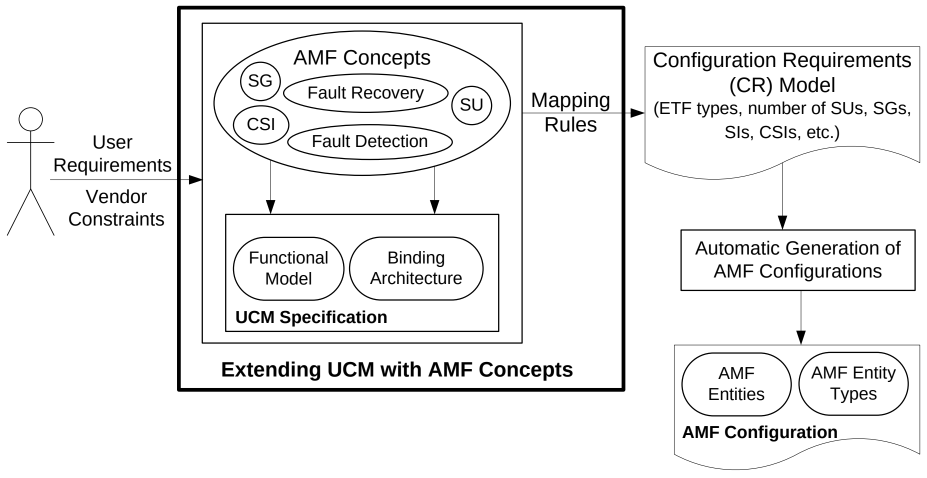


Fig. 2. AMF Configuration Generation Approach

The resulting UCM availability requirements can then be mapped to a configuration requirements (CR) model that describes the used ETF types and configuration parameters such as the number of SUs, CSIs, etc. Finally, based in the AMF requirements model (CR), an AMF configuration model can be generated by leveraging the AMF configuration generation approach proposed by Kanso et al. [12].

4 Use Case Maps Availability Modeling

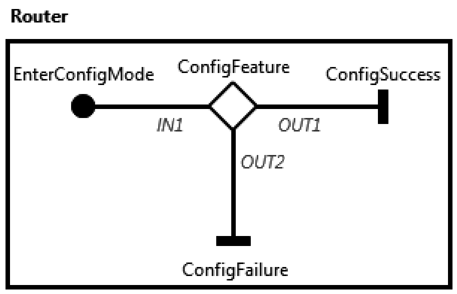
In this section, we introduce the basic Use Case Maps constructs and we present our proposed UCM-based availability extensions to cover error detection (Section 4.3) and recovery (Section 4.4). For a complete description of the Use Case Maps language, interested readers are referred to [10].

4.1 Use Case Maps Functional and Architectural Features

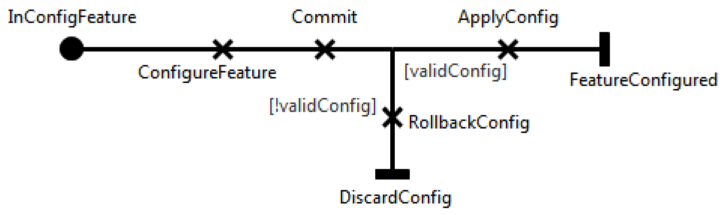
Use Case Maps (UCM) models are expressed by a simple visual notation allowing for an abstract description of scenarios in terms of causal relationships between responsibilities (✕) (e.g., operation, action, task, function, etc.) along paths allocated to a set of components (▭). These relationships are 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.

Figure 3(a) illustrates a UCM scenario for configuring a feature on a router setup. The feature configuration takes place when the router is in configuration mode (i.e., start point *EnterConfigMode*). The configuration steps are embedded within *ConfigFeature* stub, which has two outgoing paths (*OUT1* for successfully configuring the feature and *OUT2* for the rejection of the configuration). Figure 3(b) illustrates the plugin map of the *ConfigFeature* stub. AFter entering the configuration commands of the new feature (i.e., responsibility *ConfigureFeature*) and commit the new changes (i.e., responsibility *Commit*), the new configuration is applied (i.e., responsibility *ApplyConfig*) in case it is a valid config (i.e., condition *validConfig* part of the OR-Fork is true), otherwise the new changes are discraded (i.e., responsibility *RollbackConfig*).



(a) Feature Configuration Scenario



(b) FeatureConfig Stub Plugin

Fig. 3. UCM Scenario: Configure a Feature on a Router

One of the strengths of UCMs resides in their ability to bind responsibilities to architectural components. The default UCM component notation is generic and abstract allowing for representing software entities (e.g., objects, processes, databases, or servers) as well as non-software entities (e.g., actors or hardware).

In the ITU-T standard [10], a UCM component (Figure 4) is characterized by its kind (Team, object, agent, process, actor) and its optional type (user-defined type) and may have several including components (i.e., more than one parent), therefore allowing the capture of several architectural alternatives in one UCM model. A modeler may investigate various allocations of subcomponents to components and reason about trade-offs involving these alternatives.

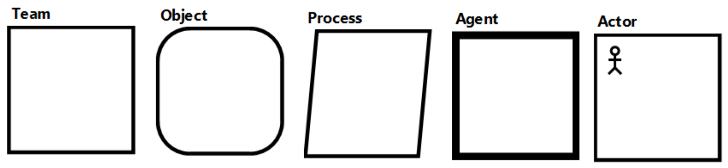


Fig. 4. UCM Components

In this research, we extend the Use Case Maps language with availability annotations. The proposed annotations are inspired from AMF concepts allowing for a smooth mapping of the resulting UCM specifications to AMF configurations. UCM generic components are used extensively to model AMF redundancy aspects, while functional constructs are used mainly to model component service instances (CSIs), modeled as UCM scenario paths.

4.2 Use Case Maps Redundancy Modeling

Error recovery focuses mainly on redundancy modeling in order to keep the system available in case of the occurrence of a failure. To accommodate the mapping to AMF configurations, we introduce five types (user-defined types) of UCM components: *node*, *application*, *serviceGroup*, *serviceUnit*, and *component*. The type is coded as a metadata attribute *Type*.

A UCM component of type *node* is annotated with the following metadata attributes:

- **NodeID**: Used to identify the node.
- **ClusterID**: Specifies the cluster to which the node belongs.

A UCM component of type *application* is annotated with the following metadata attributes:

- **ApplicationID**: Used to identify the hosted application.
- **ClusterID**: Specifies the cluster on which the application is hosted.

A UCM component of type *serviceGroup* is annotated with the following metadata attributes:

– **ServiceGroupID**: Used to identify the group to which a component belongs in a specific redundancy model. That is all components that belong to the same service group can collaborate to protected the offered services.
– **ApplicationID**: Used to specify the application the service group is implementing.
– **RedundancyModel**: Specifies the redundancy type that the service group implements. This attributes takes the following five values: *2N*, *N+M*, *N-Way*, *N-Way-Active*, and *No-Redundancy*.

A UCM component of type *serviceUnit* is annotated with the following metadata attributes:

– **ServiceUnitID**: Used to identify the service unit.
– **ServiceGroupID**: Used to identify the service group to which the service unit belongs.
– **SuActiveRole**: Lists all service instances for which the service unit is in active role.
– **SuStandbyRole**: Lists all service instances for which the service unit is in standby role.
– **SuSpareRole**: Lists all service instances for which the service unit is in spare role.

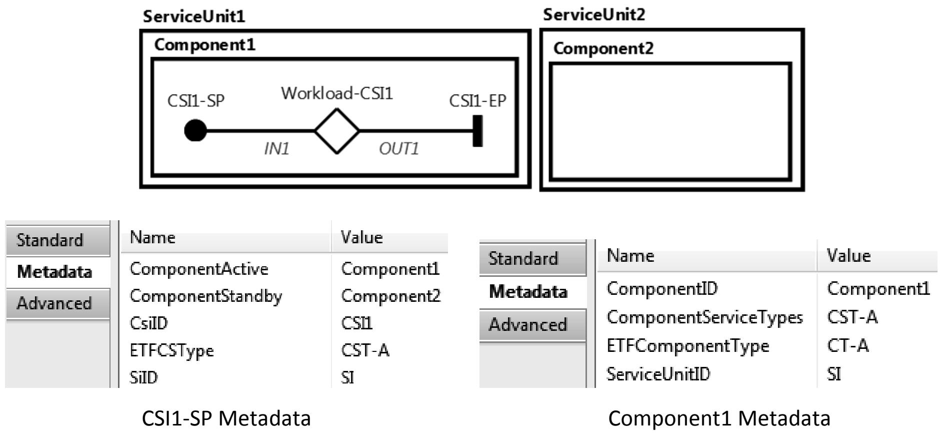
It is worth noting that *SuActiveRole*, *SuStandbyRole*, and *SuSpareRole* represent the "preferred" roles rather than static roles. At runtime, upon failure, roles may change.

A UCM component of type *component* is annotated with the following metadata attributes:

– **ComponentID**: Used to identify the component.
– **ETFComponentType**: Specifies the ETF component type.
– **ServiceUnitID**: Used to identify the service unit to which the component belongs.
– **ComponentServiceTypes**: Defines the list of service types a component can handle.

Service Instances (SIs) have no UCM graphical representation. A service instance is implicitly specified using the set of component service instances (CSIs) assigned to it. A component service instance (CSI), expressed using UCM scenarios (to model the workload), is characterized by a scenario *start point* with the following attributes:

– **CsiID**: Identifies the component service instance (CSI) that the UCM scenario implements.
– **SiID**: Identifies the service instance (SI) to which the CSI belongs.
– **ComponentActive**: Specifies the component for which the CSI is active.
– **ComponentStandby**: Specifies the component for which the CSI is standby.
– **ETFCSType**: Specifies the ETF component service type.
– **ComponentID**: Specifies the potential container of type *component*.



Name	Value
ComponentActive	Component1
ComponentStandby	Component2
CsiID	CSI1
ETFCSType	CST-A
SiID	SI

Name	Value
ComponentID	Component1
ComponentServiceTypes	CST-A
ETFComponentType	CT-A
ServiceUnitID	SI

Fig. 5. Example of a Component Service Instance Representation and its corresponding Metadata

Figure 5 illustrates a UCM architecture describing two service units *ServiceUnit1* and *ServiceUnit2*, which are composed of components *Component1* and *Component2* respectively. The system implements one workload (CSI1), expressed as a UCM scenario path that is enclosed with the active component *Component1* (*Component2* of *ServiceUnit2* being in standby mode). The service group and its redundancy model (i.e., 2N in this case) are not shown. The characteristics of the component service instance *CSI1* are expressed as part of the start point *CSI1-SP* metadata. The UCM plugin bound to the stub "workload-CSI1" contains the functional behavior of the component workload and it is not shown here.

AMF [16] uses a rank-based mechanism to determine whether a service unit is active, standby, or spare. For a detailed description of the ranking system and how the service instances (SIs) are assigned to in-service service units, interested readers are referred to [16].

4.3 UCM Error Detection Modeling

The specification of error detection mechanisms is a key factor in implementing any availability strategy. Error detection modeling involves the specification of liveness requirements (e.g., process heartbeat) and the description of potential errors. In [6] and [7], we have used the UCM comment constructor to describe error detection tactics such as *ping* and *heartbeat*. In this paper, we use metadata attributes and we introduce the concept of component healthcheck; a concept borrowed from the AMF framework. We introduce two types of component healthchecks: *framework-invoked* and *component-invoked*.

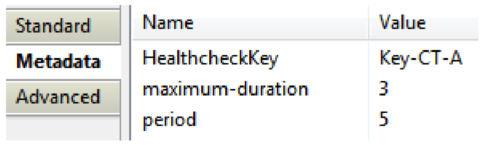
AMF supports the notion of *healthcheck* type, identified by a healthcheck key, that can be associated to a component type. A healthcheck can be invoked by the framework or by the component itself. A healthcheck configuration is composed of two attributes:

– *period*: specifies the period at which the corresponding healthcheck should be initiated. In case the healthcheck is started by the AMF framework and if a process does not respond to a given healthcheck callback (i.e., *saAmfHealthcheckCallback()*) before the start of the next healthcheck period, AMF would not trigger another callback.
– *maximum-duration*: specifies the time-limit after which the AMF framework will report an error on the component. This attribute applies only to framework-invoked healthcheck variant.

The mapping between the UCM-based metadata attributes and AMF configurations is as follows:

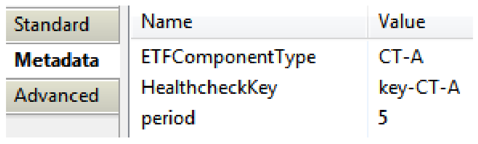
– *period* is mapped to either *saAmfHealthcheckPeriod* (if the healthcheck is configured specifically for the component) or *SaAmfHctDefPeriod* (if the healthcheck is configured for the component type).
– *maximum-duration* is mapped to either *saAmfHealthcheckMaxDuration* (if the healthcheck is configured specifically for the component) or *saAmfHct-DefMaxDuration* (if the healthcheck is configured for the component type).

Figure 6 illustrates two healthcheck descriptions, expressed using URN metadata feature. Both *framework-invoked* (Figure 6(a)) and *component-invoked* (Figure 6(b)) healthchecks use the "*HealthchekKey*" attribute. The component-invoked healthcheck (Figure 6(b)) specifies the type of component (e.g., using the attribute ETFComponentType) that can invoke the check. For the sake of clarity, only healthcheck related attributed are shown in Figure 6(b).



Name	Value
HealthcheckKey	Key-CT-A
maximum-duration	3
period	5

(a) Healthcheck metadata associated with the UCM Specification



Name	Value
ETFComponentType	CT-A
HealthcheckKey	key-CT-A
period	5

(b) Healthcheck metadata associated with a specific UCM component

Fig. 6. UCM-based Healthcheck

Errors are reported to AMF by invoking the *saAmfComponentErrorReport_4()* API function that specifies, amongst others, the erronous component, the absolute time of error reporting (i.e., *errorDetectionTime*), and the recommended recovery action (i.e., *recommendedRecovery*). Section 4.4 discusses how recovery is implemented in Use Case Maps.

4.4 UCM Error Recovery Modeling

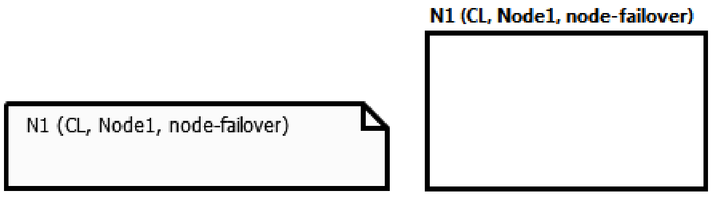
Upon failure detection, AMF would perform an automatic recovery by (1) taking a restart recovery action (restarts the erroneous component or restart all components of the service unit), (2) performing a fail-over (e.g., Standby takes over), (3) restarting the application, or (4) resetting the cluster.

The recovery action can be encoded in component/node/application/cluster definitions using a metadata attribute *RecoveryAction* that may take the following values:

- *component-restart* and *component-failover* for UCM components of type *Component*. These two values are mapped to *SA_AMF_COMPONENT_RESTART* and *SA_AMF_COMPONENT_FAILOVER* respectively in the AMF enumeration *SaAmfRecommendedRecoveryT*. Furthermore, a component fail-over may trigger a fail-over of the entire service unit. Such an option can be defined used the boolean attribute *SUFailOver* (mapped to AMF *saAmfSUFailover* with *SA_TRUE* and *SA_FALSE* as possible values).
- *node-failover*, *node-switchover*, and *failfast* for nodes. These three values are mapped to AMF *SA_AMF_NODE_SWITCHOVER*, *SA_AMF_NODE_FAILOVER*, and *SA_AMF_NODE_FAILFAST* respectively. A detailed description of these three recovery mechanisms under different redundancy models can be found in [16].
- *cluster-reset* for clusters, which is mapped to AMF *SA_AMF_CLUSTER_RESET* enumeration value.
- *app-restart* for application components. The application should be completely terminated first by terminating all its service units. This value is mapped to AMF *SA_AMF_APPLICATION_RESTART* enumeration value.
- *No-recommendation*: The error report does not make any recommendation for recovery. It is mapped to the AMF *SA_AMF_NO_RECOMMENDATION*.

5 Discussion

Our proposed approach relies on extending the Use Case Maps language with AMF related concepts, allowing for the generation of AMF configurations at the early stages of system development process. Most the proposed extensions (e.g., application, node, service group, service unit attributes) are applied at the system architectural level and they are coded as metadata attributes (i.e., they are not represented visually in the UCM specification). Other representation options include the use of:



(a) UCM comments (b) Name overloading

Fig. 7. Other visual UCM-based availability representations

- **UCM comment option:** This option has been used in previous work [6] to add information about availability architectural tactics to a UCM model (see Figure 7(a)). This option is sufficient for visualizing availability attributes 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.
- **Construct name overloading option:** This option attaches availability attributes visually to individual UCM model elements (see Figure 7(b)). However, similarly to the use of UCM comments, this option is informal and cannot be used in automated model analysis.

Contrary to the two options listed above, Our metadata approach formalizes availability attributes, making it easier to use this information in automated model analysis.

6 Illustrative Example

Figure 8 illustrates an example of a UCM system composed of one cluster of two nodes (*Node1* and *Node2*), implementing an application *App* that is composed of one service group *SG*. The relationship between the two nodes and the cluster is described using the metadata attribute *ClusterID*(Figure 9(a)). The service group identifier *SG* and its supported redundancy model *2N* are described using two metadata attributes *ServiceGroupID* and *RedundancyModel* (Figure 9(c)).

The service group *SG* is composed of two service units *SU1* and *SU2*. *SU1* is composed of components *Comp1* and *Comp2*, while *SU2* is composed of components *Comp3* and *Comp4*. The UCM specification defines 4 scenario paths specifying four workloads (referred to as component service instances (CSIs)). These CSIs are grouped into two service instances SI1 and SI2 (not shown graphically) but described using the metadata attribute *SiID* (Figure 10(c)). For example, *CSI1* and *CSI2* are part of service instance *SI1*, while *CSI3* and *CSI4* are part of service instance *SI2*

The UCM shows the preferred active assignment of each component service instance. Since *SU1* has an active assignment with respect to service instances

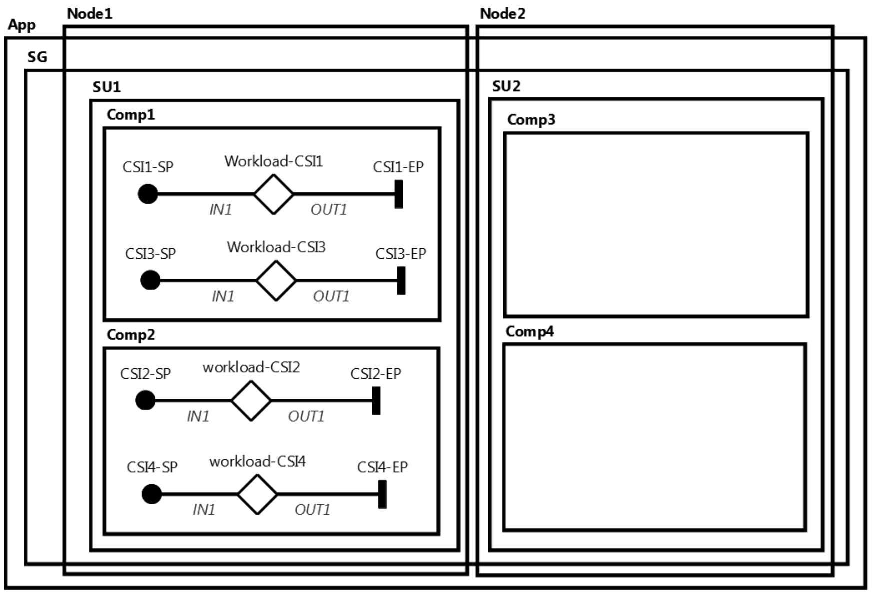
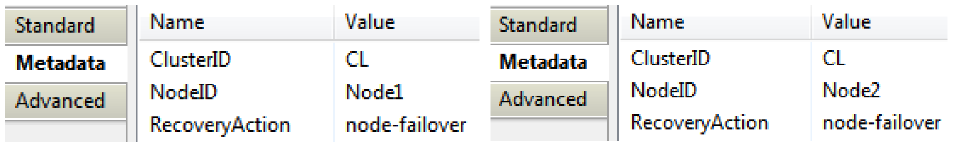


Fig. 8. A UCM Architecture with one SG having two SUs running in 2N redundancy model

SI1 and *SI2*, the four CSIs are described within the *SU1* components *Comp1* and *Comp2*. *SU2* has a standby assignment with respect to service instances *SI1* and *SI2*. Hence, *Comp3* and *Comp4* do not contain any CSI. Figure 10(c) shows the active (using the *ComponentActive* metadata attribute) and standby (using the *ComponentStandby* metadata attribute) assignments of the participating CSIs. For example, *CSI1* and *CSI3* are handled by component *Comp1*, while *CSI2* and *CSI4* are handled by component *Comp2*.

The specification of the healthcheck is exactly the same as in Figure 6(a) (i.e., *framework-invoked* healthcheck), and is hence not repeated here. Recovery actions are expressed in terms of the metadata attribute *RecoveryAction*. In case of a failure, the application should be completely terminated and then started again by first terminating all of its service units and then starting them again. This is depicted in the *RecoveryAction* attribute, being equal to *app-restart*. When an error is identified as being at the node level, all service instances assigned to service units contained in the node are failed over to other nodes (i.e., *RecoveryAction* = *node-failover*). Hence, active components should also fail over to standby components (i.e., *RecoveryAction* = *component-failover*).

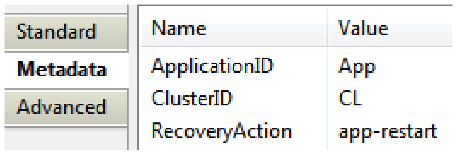
The metadata attributes (describing ETF types, SUs, CSIs, recovery actions, etc.) described in Figures 9 and 10 correspond to the AMF requirements introduced in Figure 1.



Name	Value
ClusterID	CL
NodeID	Node1
RecoveryAction	node-failover

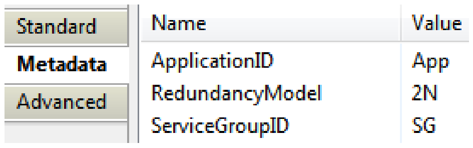
Name	Value
ClusterID	CL
NodeID	Node2
RecoveryAction	node-failover

(a) Node1 and Node2 Metadata Attributes



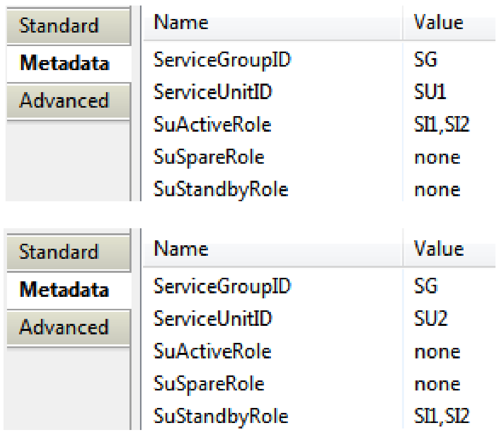
Name	Value
ApplicationID	App
ClusterID	CL
RecoveryAction	app-restart

(b) Application Metadata Attributes



Name	Value
ApplicationID	App
RedundancyModel	2N
ServiceGroupID	SG

(c) SG Metadata Attributes

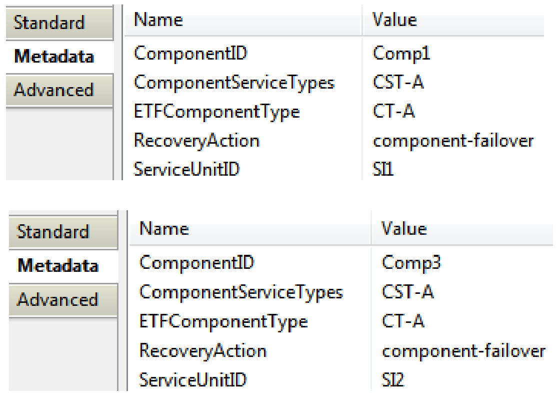


Name	Value
ServiceGroupID	SG
ServiceUnitID	SU1
SuActiveRole	SI1,SI2
SuSpareRole	none
SuStandbyRole	none

Name	Value
ServiceGroupID	SG
ServiceUnitID	SU2
SuActiveRole	none
SuSpareRole	none
SuStandbyRole	SI1,SI2

(d) SU1 and SU2 Metadata Attributes

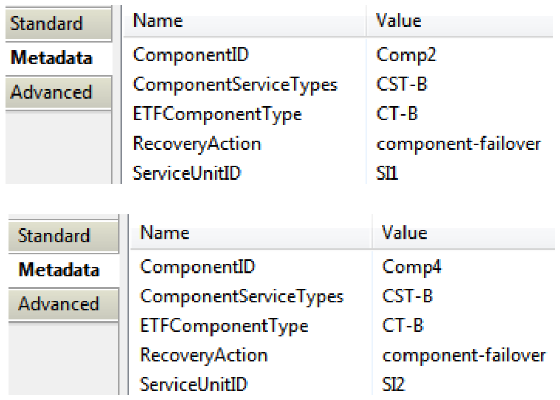
Fig. 9. Metadata Descriptions of the application, the participating nodes and service units



Name	Value
ComponentID	Comp1
ComponentServiceTypes	CST-A
ETFComponentType	CT-A
RecoveryAction	component-failover
ServiceUnitID	SI1

Name	Value
ComponentID	Comp3
ComponentServiceTypes	CST-A
ETFComponentType	CT-A
RecoveryAction	component-failover
ServiceUnitID	SI2

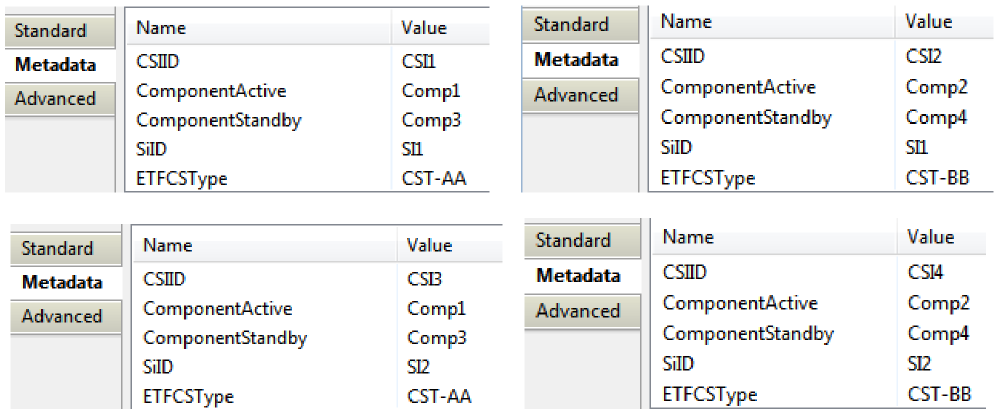
(a) Comp1 and Comp3 Metadata Attributes



Name	Value
ComponentID	Comp2
ComponentServiceTypes	CST-B
ETFComponentType	CT-B
RecoveryAction	component-failover
ServiceUnitID	SI1

Name	Value
ComponentID	Comp4
ComponentServiceTypes	CST-B
ETFComponentType	CT-B
RecoveryAction	component-failover
ServiceUnitID	SI2

(b) Comp2 and Comp4 Metadata Attributes



Name	Value
CSIID	CSI1
ComponentActive	Comp1
ComponentStandby	Comp3
SIID	SI1
ETFCSType	CST-AA

Name	Value
CSIID	CSI2
ComponentActive	Comp2
ComponentStandby	Comp4
SIID	SI1
ETFCSType	CST-BB

Name	Value
CSIID	CSI3
ComponentActive	Comp1
ComponentStandby	Comp3
SIID	SI2
ETFCSType	CST-AA

Name	Value
CSIID	CSI4
ComponentActive	Comp2
ComponentStandby	Comp4
SIID	SI2
ETFCSType	CST-BB

(c) CSI1, CSI2, CSI3, and CSI4 Metadata Attributes

Fig. 10. Metadata Descriptions of the particapting components and their corresponding component service instances

7 Conclusions and Future Work

In this work, we have extended the Use Case Maps language with Availability Management Framework (AMF) related concepts. The use of UCMs (supported by a feature-rich tool, *jUCMNav*) to describe system requirements, extended with AMF concepts, would empower analysis and validation of availability requirements at the very early stages of system development. Furthermore, we have provided a mapping between the introduced UCM-based availability requirements and AMF concepts. The resulting extensions would allow for the generation of AMF configurations from UCM specifications.

As a future work, we plan to investigate the possible integration of the UCM-based extensions (expressed with a metamodel) with a formal representation of AMF concepts, such as a UML profile for AMF.

Acknowledgment

Jameleddine Hassine would like to acknowledge the support provided by the Deanship of Scientific Research at King Fahd University of Petroleum & Minerals (KFUPM) for funding this work through project No. IN111017.

References

1. ANSI/IEEE: Standard Glossary of Software Engineering Terminology, STD-729-1991 (1991)
2. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing **1**(1), 11–33 (2004)
3. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2003)
4. Colombo, P., Salehi, P., Khendek, F., Toeroe, M.: Bridging the gap between user requirements and configuration requirements. In: Engineering of Complex Computer Systems (ICECCS), 2012 17th International Conference on, pp. 13 –22 (2012). DOI 10.1109/ICECCS.2012.11
5. Forum, S.A.: Application Interface Spec. Software Management Framework SAI-AIS-SMF-A.01.02
6. Hassine, J.: Early Availability Requirements Modeling using Use Case Maps. In: 8th International Conference on Information Technology : New Generations (ITNG2011), Modeling and Analysis of Dependable Embedded and Real-time Software Systems Track. 11-13 April 2011, Las Vegas, Nevada, USA, pp. 754–759. IEEE Computer Society (2011)
7. Hassine, J., Gherbi, A.: Exploring Early Availability Requirements Using Use Case Maps. In: I. Ober, I. Ober (eds.) SDL 2011: Integrating System and Software Modeling, *Lecture Notes in Computer Science*, vol. 7083, pp. 54–68. Springer Berlin / Heidelberg (2012)
8. Hatebur, D., Heisel, M.: A Foundation for Requirements Analysis of Dependable Software. In: B. Buth, G. Rabe, T. Seyfarth (eds.) Computer Safety, Reliability, and Security, *Lecture Notes in Computer Science*, vol. 5775, pp. 311–325. Springer Berlin / Heidelberg (2009)

9. ITU-T: E.800: Terms and Definitions related to Quality of Service and Network Performance including Dependability (2008). URL `http://www.itu.int/md/T05-SG02-080506-TD-WP2-0121/en`
10. ITU-T: Recommendation Z.151, User Requirements Notation (URN) (2010). URL `http://www.itu.int/rec/T-REC-Z.151/en`
11. Kanso, A., Khendek, F., Toeroe, M., Hamou-Lhadj, A.: Automatic configuration generation for service high availability with load balancing. Concurrency and Computation: Practice and Experience **25**(2), 265–287 (2013). DOI 10.1002/cpe.2805
12. Kanso, A., Toeroe, M., Hamou-Lhadj, A., Khendek, F.: Generating AMF configurations from software vendor constraints and user requirements. In: International Conference on Availability, Reliability and Security (ARES '09), pp. 454 –461 (2009). DOI 10.1109/ARES.2009.27
13. Kanso, A., Toeroe, M., Khendek, F., Hamou-Lhadj, A.: Automatic generation of AMF compliant configurations. In: T. Nanya, F. Maruyama, A. Pataricza, M. Malek (eds.) Service Availability, 5th International Service Availability Symposium (ISAS), *Lecture Notes in Computer Science*, vol. 5017, pp. 155–170. Springer Berlin Heidelberg (2008). DOI 10.1007/978-3-540-68129-8_13
14. Laprie, J., Avizienis, A., Kopetz, H.: Dependability: Basic Concepts and Terminology. Springer-Verlag, Secaucus, NJ, USA (1992)
15. Salehi, P., Colombo, P., Hamou-Lhadj, A., Khendek, F.: A model driven approach for AMF configuration generation. In: Proceedings of the 6th international conference on System analysis and modeling: about models, SAM'10, pp. 124–143. Springer-Verlag, Berlin, Heidelberg (2011)
16. Service Availability Forum: Application Interface Spec. Availability Management Framework SAI-AIS-AMF-B.04.01
17. Service Availability Forum: Application Interface Spec. Overview SAI-Overview-B.05.03
18. Services Availalbility Forum ^{TM}: SAForum (2010). URL `http://www.saforum.org`
19. jUCMNav v5.2.0: jUCMNav Project (tool, documentation, and meta-model). http://jucmnav.softwareengineering.ca/jucmnav (2013)
20. Wang, D., Trivedi, K.S.: Modeling user-perceived service availability. In: Proceedings of the Second international conference on Service Availability, ISAS'05, pp. 107–122. Springer-Verlag, Berlin, Heidelberg (2005). DOI 10.1007/11560333_10