Model Driven Upgrade Campaign Specification Generation and Evaluation

Oussama Jebbar

A Thesis in the Department of Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements for the Degree of Master of Applied Science in Software Engineering

At Concordia University

Montreal, Quebec

Canada

December, 2016

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: Oussama Jebbar

Entitled: Model Driven Upgrade Campaign Specification Generation and Evaluation

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. B. Desai Chair

Dr. E. Shihab Examiner

Dr. J. Rilling Examiner

Dr. F. Khendek Supervisor

Dr. M. Toeroe Supervisor

Approved by

Chair of Department or Graduate Program Director

Dean of Faculty

Date

Abstract

Model Driven Upgrade Campaign Specification Generation and Evaluation

Oussama Jebbar

High availability is an important non-functional requirement for carrier grade services. The applications/systems providing and protecting such services undergo frequent upgrades which makes meeting this requirement very challenging. A system upgrade is the migration process from the system's current configuration to a new one. This migration may include configuration changes, installation and removal of software, etc.. The Service Availability Forum (SAF) published a set of standards that describe a high availability enabling middleware for Commercial-off-the-shelf (COTS) components based systems. In such a middleware, the Software Management Framework (SMF) is the service responsible for orchestrating the upgrades. These upgrades are performed according to a road map called an upgrade campaign specification. The Availability Management Framework (AMF) is another service defined in the SAF standards and which is responsible of managing the availability of the services and the service providers. To take a SAF compliant system from one configuration to another, one has first to come up with an upgrade campaign specification for that purpose. Moreover, there are multiple upgrade campaign specifications that can take the system from the same source configuration to the same target configuration, but they differ in the duration they take and the service outage they may induce. Designing an upgrade campaign specification for a SAF compliant system is not a straight forward process. Indeed, this is an error prone task that becomes more challenging when the system and the set of changes to perform get larger. Besides, selecting which upgrade campaign specification to apply among all the valid ones is either expensive (running the same upgrade campaign specification on a replica of the real system), or tedious (evaluating, comparing and selecting upgrade campaign specifications manually).

In this thesis we propose automation as a solution to ease and minimize human intervention in the design and evaluation of upgrade campaign specifications. We devise a model driven approach to automatically generate upgrade campaign specifications. Our approach consists of several activities in order to ensure the SAF compliance of the gener-

ated upgrade campaign specification on one hand. On the other hand, we use the dependencies between system components to apply a set of rules that can improve the quality of the upgrade campaign specification by avoiding some of the unnecessary service outage. These rules include rules to order changes to be performed as well as a set of heuristics that make use of the dependencies. We also address the upgrade campaign evaluation related issues. We extend an existing discrete event systems based simulation approach for upgrade campaign evaluation. We expose the limitations of a random simulation as its results are unreliable for comparison. To overcome these limitations we define best case and worst case scenarios that we use to guide upgrade campaign simulations to see how the upgrade campaigns perform in edge cases. We also devise a method for upgrade campaign specification selection/elimination based on applicability checks according to two criteria: the maintenance window, and the acceptable outage during this window. Finally, we implemented prototypes for upgrade campaign specification generation and evaluation.

Acknowledgments

I would like to thank my family for their everlasting and unconditional love, support, and encouragement.

I would like to express my gratitude to my supervisors Dr. Ferhat Khendek and Dr. Maria Toeroe for their patience, support, and their valuable feedback that inspired me during this thesis. Thank you very much for granting me the opportunity to benefit from this extremely enlightening and gratifying experience within your research team.

I would like to thank my colleagues in MAGIC team for the good memories.

I would like to thank Concordia University, Ericsson Canada, and NSERC for the funding and the facilities they put under my disposal to achieve this work.

Finally, I would like to thank the “gang”, Amine, Amine, Abdel, and Mehran for the memorable adventures, and for …

Table of Contents

List of Figures

List of Tables

Chapter 1 - Introduction

This chapter explains briefly the context of the thesis. It introduces concepts such as High Availability (HA), system upgrade, and Service Availability Forum (SAF) [1]. It also presents the motivations behind the thesis as well as its contributions.

1.1. High availability & upgrade

Availability is defined as "the proportion of time when a system is in a condition that is ready to perform the specified functions" [2]. The availability of a system depends on the reliability of its components and the required time to repair them in case of failure. The metrics that are commonly used to evaluate these two attributes are the Mean Time To Fail (MTTF) and Mean Time To Repair (MTTR). System availability is defined as follows [2]:

$$Availability = \frac{MTTF}{MTTF+MTTR} \qquad \text{eq 1-1}$$

There are several levels of availability. Systems that provide carrier grade services must to be highly available, meaning that their availability should be at least 99.999% (also referred to as five nines). In other words, the services provided by such system cannot be interrupted for more than 5.26 minutes per year. In order to enable such availability, one should properly deal with the two types of downtime:

- ✓ Unplanned downtime: downtime caused by components' failures.
- ✓ Planned downtime: downtime caused by system upgrades and maintenance.

There are several principles, if followed, can help tackling these challenges such as:

- ✓ Thoroughly choosing the components that will compose the system. The components should be extensively tested, and should undergo enough verification, validation, and benchmarking to ensure that they are reliable enough to provide highly available services.
- ✓ Enabling error detection. The system should be able to identify components that are unhealthy, erroneous and not able to provide the service.
- ✓ Enabling error repair. This can be achieved by using repairable components to build the system.
- ✓ Enabling fault tolerance. Fault tolerance is defined as *"enabling a system to continue its normal operation in the presence of faults…without human intervention"* [2], this can be achieved by implementing means for error detection and service recovery. Service recovery can be implemented either using redundancy so that the service can still be provided even with a presence of a faulty component as

the system has duplicates that can provide the same service, or by using components that can be repaired within specific deadlines and thus providing the required availability.

The aforementioned principles, when applied, can only make the system capable of handling the unplanned downtime but still they do not provide any viable solution to cope with the planned downtime. As introduced earlier, the planned downtime is the downtime caused by system upgrades. System upgrades are performed via an upgrade campaign which represents the process of migration from the current configuration of the system to a new configuration called the target configuration. This process of migration includes the configuration changes that need to be done, the software that needs to be installed, the software that needs to be removed, as well as how every set of required changes have to be done and their ordering. Upgrading systems that are deployed in a redundant manner can be done in different ways. While some upgrade campaign designers might choose to upgrade all the elements that are redundant of each other at the same time (if they aim at saving time for example), some others prefer to upgrade them one at a time in order to minimize the service disruption as the service can still be provided while the system is under upgrade. This can be achieved by making sure that at any moment during the upgrade campaign, and for a given service provided by the system, there is at least one entity or subsystem able to provide that service and that entity is not undergoing an upgrade. To automate this process and make it easier and more manageable, one need to dispose of an upgrade engine which should have the capability of performing these kinds of changes on the system's entities as well as communicating with the deployed components either directly or through a third party.

1.2. Service Availability Forum

Different software vendors used to implement different availability management solutions for their products. These solutions use to be proprietary, and sometimes target specific domains (availability management for a DBMS, a VM, a network, etc.) which hampers portability of their software products.

The Service Availability Forum (SAF) [1], is a consortium of telecommunication and computing companies whose goal is to publish and maintain open specification standards that define a set of middleware services enabling high availability for Commercial-Off-The-Shelf (COTS) components based systems. Hence, software vendors are only required to implement appropriate SAF defined APIs in order to incorporate high availability into their products.

SAF specifications cover a wide spectrum of system related aspects:

- ✓ Layers-wise: SAF standards cover both lower and upper layers of a system. The Hardware Platform Interface (HPI) defines the utilities used to monitor and con-

trol the hardware. The Application Interface Specification (AIS) covers the services and interfaces used to manage upper layers of system (Platform, Software, etc.).

- ✓ Services-wise: the AIS standard defines several services and interfaces used for multiple purposes such as availability management (AMF), configuration management (IMM), software management (SMF), communication(Event EVT), state synchronization(Checkpoint CKPT), etc.

This work relates mainly to the services used for availability management, the Availability Management Framework (AMF) [3], and software management, the Software Management Framework (SMF) [4], and to some extent configuration management, the Information Model Management (IMM) [5].

1.3. Motivations and Contributions

As introduced earlier, an upgrade campaign is the process of migrating a system from its current configuration to a new configuration. In a SAF compliant environment, the system is composed of COTS components which are managed by AMF [3]. In order to manage the availability of these components, AMF abstracts them into logical entities which are described in the AMF configuration. A subset of these AMF managed components is what basically composes the target of a SAF compliant upgrade campaign. This upgrade campaign is carried out by SMF [4] following a roadmap called the Upgrade Campaign Specification. The Upgrade Campaign Specification is an XML file that specifies partially ordered sets of changes that compose the upgrade procedures. For a change to be correctly performed, a set of components may be prevented from providing the service which may cause service interruption during the execution of the upgrade campaign.

A typical process of upgrading a system would consist of, first designing one or multiple upgrade campaign specifications that can take the system from the source configuration to the target configuration. Before executing the upgrade campaign, the administrator checks the applicability of the upgrade campaign specifications in order to rule out the ones that are not applicable, and pick one for execution. The applicability check in this thesis goes along two dimensions:

- ✓ Execution time-wise: the administrator is granted a maintenance time window. A maintenance window is a period for which some service under-provisioning or disruption can be tolerated. The tolerable under-provisioning/disruption is known as the allowed outage associated with this maintenance window. Both the maintenance windows and their associated allowed outages are usually agreed on in the Service Level Agreements (SLAs) between service consumers and service providers. In order to comply with the SLA, the administrator should perform the upgrade and bring back the system into a fully operational state within the maintenance window. The expected execution time of the upgrade campaign specification should fit within this maintenance window.

- ✓ Outage-wise: the upgrade campaign should not interrupt services that are not allowed to be interrupted (compliance with SLAs). The induced outage should stay within the allowed outage. The allowed outage is a set of services and the maximum duration of interruption that can be tolerated for each service in a given maintenance window.

The design of upgrade campaign specifications and their applicability checks are mandatory to upgrade a SAF compliant system in order to meet the availability requirement. While performing these tasks, one should properly consider some system aspects such as the dependencies between service providers, the dependencies between services, and the relationships between the services and the service providers. Handling these aspects becomes harder as the systems get larger and more complex, which makes manually performing these tasks tedious, error prone, and time consuming.

In order to ease the design of upgrade campaign specifications and make it more efficient, we propose in this thesis a model driven approach to automatically generate a valid upgrade campaign specification. We define a valid upgrade campaign specification as one that is syntactically correct, and conforms to the XSD specified in the SMF specification [4].

In this approach we build upon the work in [6] as it allows to compare two AMF configurations and generate the set of changes to be performed to move from one configuration to another. We take this set of changes, and start by building a SAF compliant upgrade campaign specification. This is done by creating an upgrade procedure in the upgrade campaign specification for every change that needs to be done. This upgrade campaign specification undergoes a first refinement in which we group changes that target logically related entities into the same procedures. We propose a set of rules to perform such a grouping of changes. The upgrade campaign specification undergoes yet another refinement that takes into consideration the dependencies between system components to: 1) properly order the changes by applying the rules proposed in [55, 57]; and 2) apply some heuristics that may improve the quality of the upgrade campaign specification (minimizing the outage it may induce and the time it may take).

The applicability check as previously described requires an outage-wise and time-wise evaluation of upgrade campaign specifications. In this thesis, we build upon the work in [7] as it proposes a simulation based framework that enables performing such evaluation on upgrade campaign specifications. We defined the best case and worst case execution scenarios of an upgrade campaign in a SAF environment in order to overcome some limitations of the work in [7]. We also defined a method that, based on the results of the best case and worst case evaluations, can help select some upgrade campaign specifications, eliminate others, and identify the ones that can be further optimized to be applicable in a given situation (a given maintenance window and acceptable outage).

Finally we have developed a prototype for the upgrade campaign specification generation and we integrated it with an implementation of the work in [6]. We also implemented a prototype for best case and worst case evaluation of upgrade campaign specifications.

1.4. Organization of the thesis

The rest of this document is organized as follows. Chapter 2 gives a background about SAF standards, namely AMF, SMF and IMM, as well as an overview of Model Driven Engineering (MDE), Epsilon, DEVS formalism, and a review of the related work. In Chapter 3 we describe our model driven upgrade campaign specification generation approach. In Chapter 4 we present our contributions for upgrade campaign evaluation. Before concluding, Chapter 5 discusses briefly the prototypes.

Chapter 2 - Background

This chapter introduces the SAF specifications, in particular the Information Model Management (IMM), AMF, and SMF. It also introduces Model Driven Engineering (MDE). It presents the tools used for prototypes implementation, namely DEVS formalism and Epsilon environment. Finally, it discusses the related work.

2.1. Information Model Management

The Information Model Management (IMM) is the service responsible for maintaining the integrated information of the SAF compliant system [2]. The information model holds both configuration and runtime information required and used by all the SAF middleware provided services. It can also be used by other applications to store their own application-specific information. IMM manages the access to the information in the information model through the read, write or modify operations. Access to an object is usually done via its Distinguished Name (DN). In the SAF context there are two types of names that follow the following rules:

- ✓ Relative Distinguished Name (RDN): a name given to the object in the format specified in the SAF standards for the instances of the entity to which that object belongs.
- ✓ Distinguished Name (DN): which consists of the RDN of the object followed by the DN of its parent. If the object has no parent the RDN and DN are identical.

IMM exposes an API for external software to use in order to access the information model. The IMM service also interacts internally with all the services that compose the SAF compliant middleware.

2.2. Availability Management Framework

The Availability Management Framework (AMF) [3] is a service defined by SAF. It is responsible for the management of the resources used to provide services. This management uses a model, the AMF configuration, which captures the following aspects of a system:

- ✓ A static description of the system which describes the resources that the system has, the services it provides, and a mapping that associates each service with the resources that can provide it.
- ✓ A dynamic representation of the system that continuously represents the runtime state of the system. This runtime state reflects the presence of enough service providers to ensure the services' high availability among other things. In addition, it also keeps record of the locations where the services are being provided, as well as the eligibility and readiness of service providers to provide the services. This

information is captured using either some runtime objects that AMF instantiates to track this information, or state models that are associated with the configured AMF entities.

During the lifetime of the system, configuration changes (by the administrator or some third party), errors and failures can take place. For AMF to be able to properly manage the availability of the services and the resources, the AMF service should be aware of all of these factors that lead to changes on the dynamic description of the system. In the following subsections we will introduce the concepts used in the AMF configuration as well as the state models associated with them and which are relevant to this work.

2.2.1. AMF configuration

The AMF configuration is a representation of the system through a set of AMF defined logical entities and their relations. This configuration is stored and managed by another SAF service, the Information Model Management (IMM) [5]. AMF uses its content to manage the resources accordingly.

In addition to AMF entities, AMF also uses AMF entity types to describe the system by associating AMF entity types with AMF entities. AMF entity types usually hold the common attributes for the AMF entities associated with them. In the rest of this subsection we will introduce the AMF entities that compose the AMF configuration as well as the types associated with them if any.

2.2.1.1. Component and Component Type

The component is the smallest building block of a system and that can provide a service. It is also considered the smallest fault zone on which AMF performs fault detection, isolation and repair. This kind of control, among others, can be done either directly via the AMF API or indirectly through the Command Line Interface (CLI) or both. We distinguish between many categories of components from different perspectives:

- ✓ Compliance to the standard: a component can be SA-aware if it incorporates HA by implementing the AMF API, thus becoming directly managed by AMF. Or Non-SA-aware when it does not implement the AMF API and it is managed by AMF via the CLI or a third party.
- ✓ Dependency between the lifecycle and the service provisioning: a component is said to be pre-instantiable if it can be instantiated without being assigned a workload to handle. Similarly, a component is said to be non-pre-instantiable if it only gets instantiated when it is assigned a workload to handle. Note that according to the AMF specification [3] all SA-aware components are pre-instantiable.
- ✓ AMF management: a component that is SA-aware and directly managed by AMF is called regular SA-aware component. An SA-aware component can be managed directly by AMF with a life-cycle management customization through a third party. This deployment pattern was introduced by AMF as the container-contained pattern. In this deployment pattern the lifecycle management of an SA-

aware component, the contained, is customized through a regular SA-aware component called the container. Moreover, the container and the contained should share the same host. Note also that since AMF specification [3] introduces some container level repair actions it may be safe to say that a container is considered as a fault zone as well. In addition to the container-contained pattern, AMF introduces yet another deployment pattern called the proxy-proxied. In this deployment pattern, AMF uses a regular SA-aware component called the proxy to manage a Non-Sa-aware component called the proxied. Unlike the container and the contained, the proxy and the proxied do not necessarily have to share the same host. The last category from this perspective is the Non-Sa-aware Non-proxied components that AMF manages only through the CLI. Note that according to [3], all the components of this category are non-pre-instantiable.

The component type holds the common attributes for components running the same software such as paths to lifecycle operations scripts and the component category. The component type also holds default values of lifecycle and management operations timeouts that the configuration designer can override at component level. .

2.2.1.2. Component Service Instance and Component Service type

Component Service Instance (CSI) is a named set of attributes that configure a component for a service it is capable of providing [2], thus becoming AMF's mechanism to control service provisioning without being aware of what is being provided as a service. Accordingly, a Component Service type defines the names of the attributes that need to be configured for each of the CSIs of this Component Service type. A component can play either the role of an active service provider (actually providing the service), or the role of a standby or have no role depending on the HA state of the component for that CSI (section 2.2.2). When a component has a given role for a CSI we say that this component has an assignment for that CSI. In order to enable this kind of management, a CS type should be associated with the component types that support it (can be assigned an HA state for it). This association defines the component capability model that applies to that component type for this CS type. The component capability model defines the multiplicities of active and standby assignment that a component of a given component type can provide for a CSI of a given CS type. AMF supports the following types of capability models [3]:

- x_active_and_y_standby: components that have this capability model can be active for up to x CSIs and standby for up to y CSIs at a time.
- x_active_or_y_standby: components that have this capability model can be either active for up to x CSIs or standby for up to y CSIs at a time.
- 1_active_or_y_standby: components that have this capability model can be either active for one CSI or standby for up to y CSIs at a time.
- 1_active_or_1_standby: components that have this capability model can be either active for 1 CSI or standby for 1 CSI.

- x_active: components that have this capability model can never be standby, and can be active for up to x CSIs at a time.
- 1_active: components that have this capability model can never be standby, and can only be active for one CSI at a time.

The x and y in the names of the capability models are configured as attributes of an association between a component type and a CS type. These attributes can be overridden by the attributes of an association between a component of that type and the CS type if any.

2.2.1.3. Service Unit and Service Unit type

Components only handle units of workload as CSIs which may not necessarily coincide with the functionalities that the system is required to provide. Therefore, AMF introduces the Service Unit (SU), which is a logical entity that groups components able to handle CSIs that compose one or more desired functionality. An SU is also considered a fault zone and it is the smallest and only entity that gets duplicated in order to provide redundancy and thus insure the availability of the services. It is also worth noting that from a naming perspective, an SU is the parent entity of all the components that compose it. The Service Unit type contains all the common attributes of the SUs that share that type. Amongst the most relevant is the list of component types that compose the SUs of that SU type as well as a minimum and maximum number of components of each of those component types. In addition, the SU type also holds the Service types that the SUs of this type can provide. The concept of Service type will be introduced in the next subsubsection.

2.2.1.4. Service Instance and Service type

The CSIs provided by the components of the same SU when combined can make one or more desired functionality, also called in the context of AMF Service Instance (SI). A SU that can provide a SI can play either the role of active or standby for that SI. This role is based on the roles the components of that SU play for the CSIs that compose that SI. Some of the most relevant attributes of an SI are the preferred number of active assignments and the preferred number of standby assignments which specify the number of assignments both active and standby that this SI should have in order to insure its high availability. In addition, the list of dependent SIs that depend on a given SI is also considered a valuable attribute. An SI (dependent) is said to depend on another SI (sponsor) if the dependent cannot be assigned unless the sponsor is already assigned. AMF configuration enables the specification of the dependency between SIs that is marked with an attribute called tolerance time. The tolerance time specifies the maximum amount of time that the dependent SI can still be assigned without the sponsor SI being assigned. Similarly to SUs, an SI is considered the parent entity of all the CSIs composing it.

SIs are typed using Service types. The common SI characteristic that the Service type holds is the list of CS types that compose a SI of that type, as well as the maximum number of CSIs of every CS type.

2.2.1.5. Service Group and Service Group type

A Service Group (SG) is the logical entity that groups all the SUs protecting the same set of SIs. In order to provide redundancy, an SG should have at least two SUs so that if one fails the other can take over the service provisioning. The configuration of an SG drives how AMF manages the SUs that compose this SG. It specifies the preferred number of in-service SUs (SUs that can provide the service), the preferred number of standby SUs, the preferred number of active SUs as well as the maximum number of SIs that can be assigned to an SU as active or standby. From a naming perspective, an SG is the parent entity of all the SUs that compose it.

Similarly to the other entity types, the SG type holds the common attributes of several SGs, including the types of SUs that can compose the SGs of this type. In addition, the way AMF manages the redundancy of an SG is also configured at the SG type level. In the AMF specification [3], the different ways of managing the redundancy are called redundancy models. These redundancy models differ on the number of SIs that can be assigned to an SU at time, the roles an SU can play at a time (active or standby), the number of SUs allowed to play each role, and the preferred number of assignments (active and standby) of each SI they can support. AMF introduces five redundancy models as follows:

- 2N redundancy model (Figure 2-1): in this redundancy model:
  - At most one SU should be active for all the SIs protected by the SG.
  - At most one SU should be standby for all the SIs protected by the SG.
  - Each SI can have at most one active assignment and at most one standby assignment.

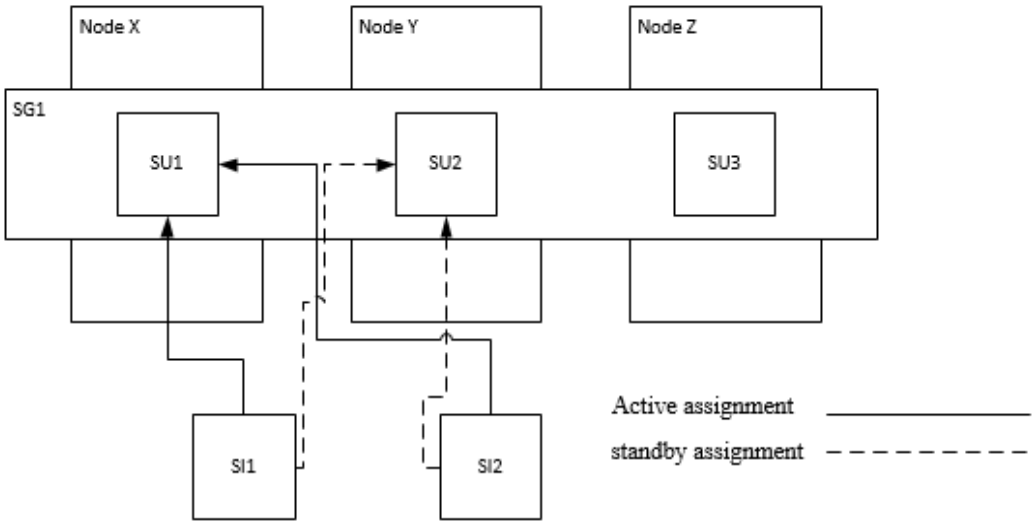


Figure 2-1: example of 2N redundancy model

- N+M redundancy model (Figure 2-2): this redundancy model is different than the 2N in a way that it allows for multiple SUs of the same SG to have the same but only one role (active or standby) at a time but not for the same SI. It can be defined as follows:
    - An SU can be either active for all the SIs assigned to it or standby for all the SIs assigned to it at a time.
    - At any given time, for each SI the SG should have at most one active SU and at most one stand by SU.

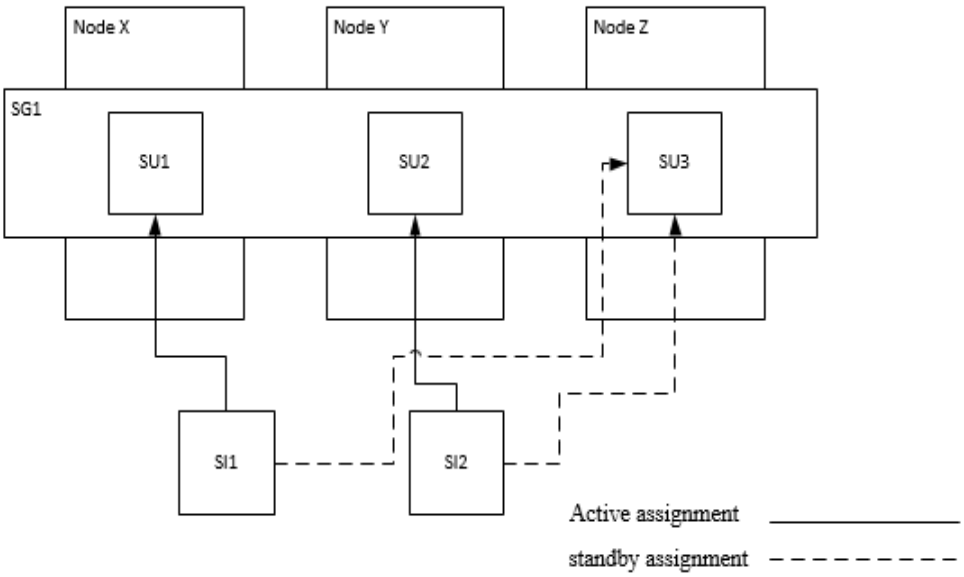


Figure 2-2: example of N+M redundancy model

- N-Way redundancy model (Figure 2-3): this redundancy model is more flexible than the N+M, as it allows the same SU to have two different roles at a time but not for the same SI. Moreover, in this redundancy model an SI can be configured to have zero or many standby assignments unlike the redundancy models listed previously. The definition of the N-Way redundancy model goes along the following lines:
    - An SU can be active for some SIs and standby for some other SIs at the same time.
    - For each SI we should have at most on active SU and zero or many standby SUs. The number of standby SUs is configured at SI level as the preferred number of standby assignments.

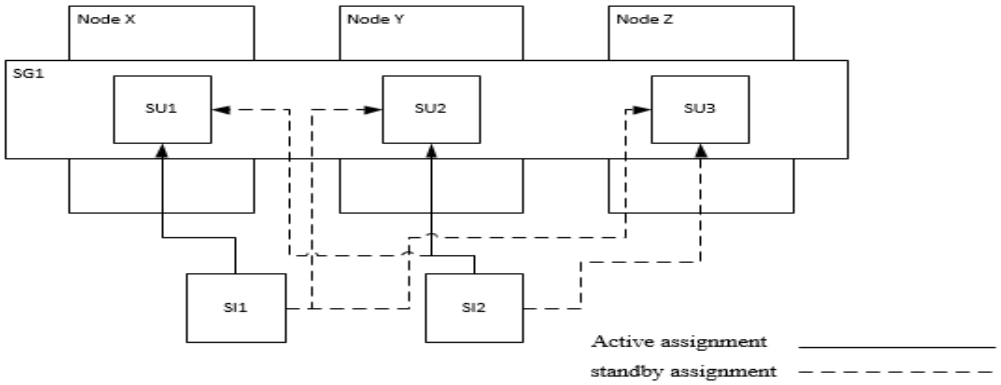


Figure 2-3: example of N-Way redundancy model

- N-Way Active redundancy model (Figure 2-4): in this redundancy model SUs are only allowed to be active for all the SIs assigned to them, and there are no standby SUs. As an SI is allowed to have one or many assignments (to different SUs), the number of active assignment for each SI is configured at SI level as the preferred number of active assignments.

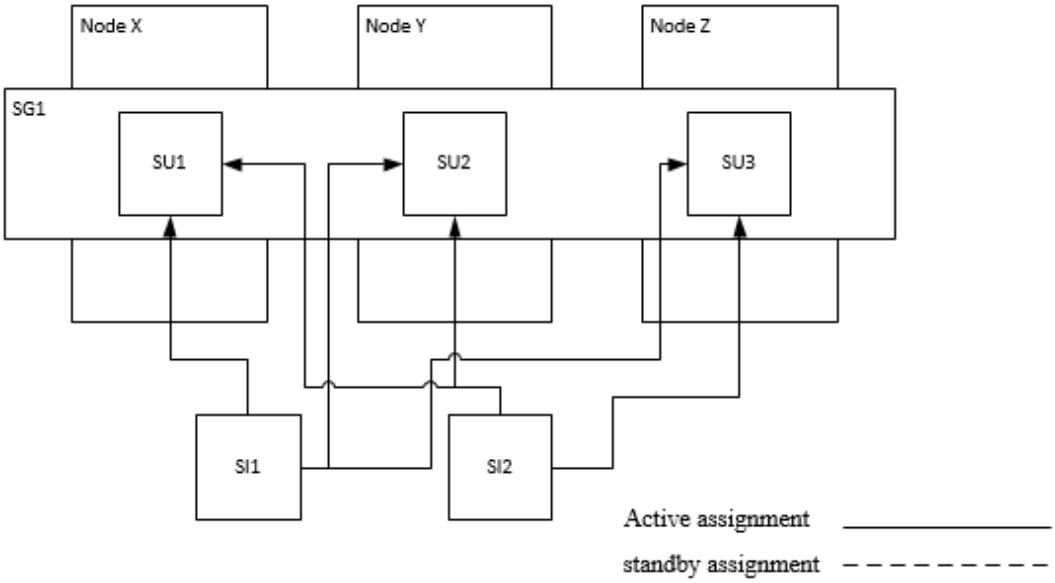


Figure 2-4: example of N-Way Active redundancy model

- No-Redundancy redundancy model (Figure 2-5): this redundancy model allows each SI to have at most one active assignment and no standby assignment. It also allows every SU to be assigned at most one SI in the active role, and no SU can be stand by at any given time.

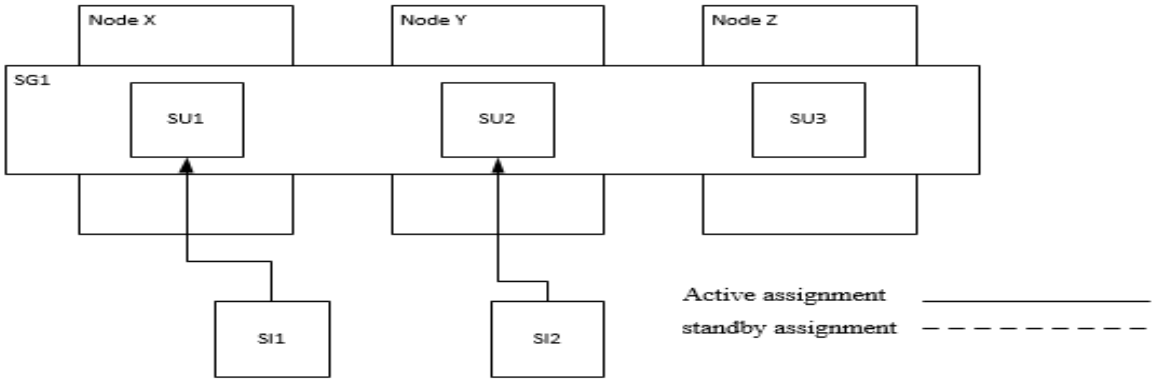


Figure 2-5: example of No-Redundancy redundancy model

2.2.1.6. Application and Application type

An AMF application is composed of a set of SIs and the SGs protecting them. And as applications are relatively independent from one another, they form fault zones on which AMF repair actions can take place in some cases. Naming-wise, an Application is the parent of all the SGs and SIs that compose it and does not have a parent.

As the types always hold the common attributes of entities, in the case of applications, applications types specify the list of SG types that can compose applications of that type.

2.2.1.7. Node, Node Group, and Cluster

For the sake of simplicity, we will restrain the definition of Node as given in [2, 3] and we will consider a Node as a physical or a virtual host. Note that a Node is also considered as a fault zone, and AMF has some repair actions that are taken at Node level.

Nodes can be grouped into Node Groups. A Node can be part of zero or many Node Groups. Node Groups are usually used for:

- Specifying a set of Nodes on which SUs of the same SG will be deployed.
- Avoiding collocation between multiple SUs or SIs. This is done by configuring the subjects of non-collocation on disjoint Node Groups.

The set of all Nodes configured in the AMF configuration compose the Cluster. The Cluster is considered as the biggest fault zone as the AMF repair actions that are taken at Cluster level interrupt all the services provided by the system. Also note that when we talk about one AMF configuration we are necessarily talking about one and only one Cluster. From a naming perspective, a Cluster is the parent entity for all the Nodes that compose it and the Node Groups grouping a subset of these Nodes.

2.2.2. State models

AMF uses multiple state models in order to manage the availability of the components through its configured logical entities. These state models are used to manage the lifecycle of the components, detect failures, assign services to the right resources, etc. In this

thesis we were interested in four state models in particular, namely, the presence state of AMF components, the HA state of a component for a given CSI (resp. of a SU for a given SI), the administrative state which is the same for SUs, SGs, Nodes, and SIs, and finally the assignment state of SIs.

The presence state of AMF components is used to manage the lifecycle of these components. The transition from state to state is triggered through the previously mentioned lifecycle management operations. The state set is composed of the following states:

- ✓ Uninstantiated: is the state where a component is when the software instance it represents is not running. It can either be when the component is first deployed or when it successfully transitions from the terminating state.
- ✓ Instantiating: the component transitions to this state from the uninstantiated state when the instantiate lifecycle management operation is called on it. That means that the instantiation is triggered or being retried if an instantiation attempt failed within the allowed number of instantiation attempts.
- ✓ Instantiated: is the state to which a component transitions from the instantiating state if the instantiation call was successful, or restarting if the restart was successful
- ✓ Terminating: is the state to which a component transitions from the instantiated state when the terminate lifecycle management operation is called on it. Terminate can consist either of a simple terminate operation, or a terminate operation followed by a cleanup when the terminate operation fails to terminate the component.
- ✓ Restarting: a component transitions to this state from the instantiated state when the restart administrative operation is called on it or in case of error recovery. Restart operation can be carried on in one of the three following ways:
  - o Terminate + Instantiate.
  - o Terminate + Cleanup + Instantiate.
  - o Cleanup + Instantiate.
- ✓ Instantiation-failed: a component transitions to this state from the instantiating state if all allowed attempts to instantiate it have failed. Note that in AMF we distinguish between instantiation with delay and instantiation without delay. For each type of instantiation we have a configured number of maximum allowed attempts for each component. In addition, in the case of the instantiation with delay, the delay between attempts of instantiation is also configured at component level.
- ✓ Termination-failed: a component transitions to this state from the terminating state or the restarting state when the cleanup operation fails.

Assigning a CSI (resp. SI) to a component (resp.SU) means that AMF gives that component (resp. SU) a HA state on behalf of that CSI (resp. SU). The state set of the HA state model is composed of the following states:

- ✓ Active: means that the component (resp. SU) is currently handling the CSI (resp. SI).
- ✓ Standby: means that the component (resp. SU) acts as a standby for the CSI (resp. SI).
- ✓ Quiescing: means that the assignment of the CSI (resp. SI) is being gracefully removed from the component (resp. SU). This implies that new requests for that service will have to be redirected to another component (resp. SU).
- ✓ Quiesced: means that this component (resp. SU) has no role for that CSI (resp. SI) and that an HA state can be given to another component (resp. SU) on behalf of that CSI (resp. SI).

The administrative state of an entity reflects the eligibility of this entity to provide a service. An SU's administrative state for example reflects if this SU is eligible to be assigned a service or not. Similarly, if an SG (resp. Node) is not eligible to provide a service that means that all the SUs grouped (resp. hosted) in this SG (resp. Node) are not eligible to provide a service. In the case of the SI, it only means whether an SI can be assigned or not. Transition from an administrative state to another is done via the administrative operations. AMF translates every call to an administrative operation on a given target into a set of AMF component life cycle and management operations that on components that belong to that target. The state set of the administrative state model is composed of the following states:

- ✓ Locked: an entity transitions to the locked state when it is previously on the unlocked state and the lock administrative operation was called on it. It can also transition to this state from the locked-instantiation state when the unlock-instantiation administrative operation is called on it. A locked entity is not eligible to provide the service. Similarly a locked service cannot be provided.
- ✓ Unlocked: an entity transitions to this state when the unlock administrative operation is called on it while being in the locked administrative state. It reflects that the entity is eligible to provide the service. Similarly an unlocked service is a service that is allowed to be provided.
- ✓ Locked-instantiation: means the entity is terminated (note that Sis do not have this administrative state). An entity transitions to this state when the lock-instantiation operation is called on it.
- ✓ Shutting-down: means the entity is not allowed to take new assignments and is trying to gracefully remove existing ones.

The last state model we are interested in is the SI's assignment state. It reflects the level of service provisioning of an SI and whether it has the minimum resources required to provide the required availability. The transition of a SI from state to state is mainly driven by the administrative state of the SUs that compose the SG protecting this SI as well as the assignment state of its sponsor SIs if any. The state set of this state model is composed of the following states:

- ✓ Fully assigned: means that the SI has the minimum resource to meet the required availability and that all the SIs on which this SI depends (Sponsors) have at least one assignment.
- ✓ Partially assigned: means that the SI has at least one assignment, all of its sponsors have at least one assignment, but the SI does not have enough resources to meet the required availability.
- ✓ Unassigned: means the SI is not assigned to any SU or one of its sponsors was unassigned for longer than the configured tolerance time.

2.3. Software Management Framework

The Software Management Framework (SMF) [4] is the service defined in the SAF specifications and which is responsible for the software management in a SAF compliant system. This software management is defined within two aspects:

- ✓ Software delivery: which defines how a SAF compliant software bundle should be delivered, and how it is described in the software repository.
- ✓ Software deployment: which defines the software deployment process including how to specify a software deployment or a configuration change in general on one hand, and how the specified instructions should be carried on and executed on the other hand.

2.3.1. Software delivery

A SAF compliant software is delivered as a software bundle, which is defined as set of interdependent files (including binary files and scripts), that can be used to install, validate, repair or remove the software at any time. Every software bundle is accompanied by an Entity Types File (ETF), a software vendor provided file that describes the software bundle as well as the entity types it delivers. This description includes the following:

- ✓ Software bundle description: which consists of the name of the software bundle, as well as various installation and removal scripts. In addition to the scope of impact of the installation/removal and which can be amongst the previously listed AMF fault zones (component, SU, container, or node), the ETF also specifies the paths to scripts of different types of installation/removals which include:
  - Online installation/removal: meaning that the installation/removal takes place while the service is being provided.
  - Offline installation/removal: meaning that the installation/removal is done while the service is not provided by the service provider under upgrade
- ✓ The entity types the software bundle delivers: including the component types and the component service types they can handle. Sometimes the vendor can also specify SU types and SG types.

2.3.2. Software deployment

The specification of software deployment in the SMF standard is done through the following aspects:

- ✓ Description of an upgrade campaign specification: an upgrade campaign specification is a road map that SMF follows to perform the changes specified in it. It should be conform to an XSD schema that is given in the standard [4].
- ✓ Description of the execution of an upgrade campaign: the behaviors that SMF should implement once it is given an upgrade campaign specification as input.

The upgrade campaign specification schema defines the different concepts that should be used by the campaign designer to specify the changes to be performed. The three main concepts of an upgrade campaign specification are:

- ✓ The upgrade campaign: as one upgrade campaign specification can only specify the changes to be done within one upgrade campaign. The upgrade campaign groups the set of procedures that perform the changes required to move the system to the target configuration.
- ✓ The upgrade procedures: used to specify the body of the upgrade campaign. Each upgrade procedure is composed of the set of upgrade steps performing the same changes on similar sets of entities.
- ✓ The upgrade steps: an upgrade step is a set of related actions to be taken on a set of entities. Upgrade steps are mainly resolved by the SMF service at the runtime, but the upgrade campaign specification still has to specify the common attributes of the upgrade steps of every upgrade procedure.

2.3.2.1. Upgrade campaign

The upgrade campaign is the root element of the upgrade campaign specification xml file, it comes with one attribute which is the name DN of the upgrade campaign. The behavior that SMF implements for an upgrade campaign is described in [4] using a state machine, and the specification of an upgrade campaign is composed of the following sections:

- ✓ Campaign info: provides some information about the time the upgrade campaign is supposed to take as well as the version of the configuration on which it should operate. When the current deployed configuration is of a different version than the one specified in this section, the upgrade campaign will not be started by SMF.
- ✓ Campaign initialization: a set of pre-campaign actions that need to be done before the execution of the body of the upgrade campaign. These actions may include IMM related actions (additions of AMF entity types and entities representing software bundles), command line calls, callback calls, or administrative operations on AMF entities. SMF can only execute the body of the campaign after all the actions specified in the initialization succeed.
- ✓ Campaign body: a set of partially ordered upgrade procedures each specifies a set of changes to be done during the campaign. The upgrade campaign can either be

executed in the forward mode (performing the upgrade), or the rolling back mode (undoing the upgrade).

- ✓ Campaign wrap-up: a set of post-campaign actions that need to be taken either to complete the changes to be done, or to validate the campaign. These actions include removals from IMM, command line calls, callback calls, or administrative operations on AMF entities. The campaign can be committed only after the successful execution of its wrap-up.

2.3.2.2. Upgrade procedure

The specification of the body of an upgrade campaign is done using upgrade procedures. An upgrade procedure is the set of steps to be performed on identical entities. For each upgrade procedure we specify the name of the upgrade procedure and its execution level. The execution level is the attribute that specifies the partial ordering in which the upgrade procedures should be executed. Upgrade procedures are executed in an increasing order according to their execution level. SMF standard [4] does not specify how upgrade procedures with similar execution levels should be executed. However, it states that an SMF implementation should be able to execute these procedures sequentially. In the rest of this document we refer to the upgrade campaigns that are executed in an SMF implementation that only supports sequential execution of upgrade procedures as fully ordered. Similarly, we refer to the upgrade campaigns executed in a SMF implementation that supports parallel execution of upgrade procedures with similar execution levels as partially ordered.

In addition to these attributes, the upgrade procedure also specify an initialization and wrap-up sections that can perform more actions than the upgrade campaign initialization and wrap-up. The additional actions include additions, removals, and modifications of AMF entities and not only AMF entity types. The specification of the body of the procedure, on the other hand, covers the following aspects:

- ✓ Common attributes for upgrade steps: will be explained later in this document.
- ✓ Upgrade scope: which is the set of entities that will be impacted during the execution of this upgrade procedure. An upgrade scope is composed of a set of Nodes (Node group), set of SUs or set of components (identified based on their parent SG and type).
- ✓ Upgrade method: based on which the SMF implementation decides how an upgrade procedure will be decomposed. SMF supports two different upgrade methods:
    - o Single step upgrade procedure: which is an upgrade procedure that performs all the changes in one step. Therefore, all the entities within the scope are impacted and taken out of service, if necessary, at the same time. This kind of upgrade procedures is usually used for additions and removals of software and new entities.
    - o Rolling upgrade procedure: which is used to upgrade software as the upgrade procedure is decomposed into steps per entity in the scope (step per

SU, step per component, or step per Node). These steps are executed sequentially thus allowing to maintain the availability of the service while performing the upgrade. Note that a rolling upgrade procedure can have a roll base which is the number of entities within the scope to be upgraded at the same time. An upgrade procedure that is targeting SUs, for instance, and that has a roll base of two, will be decomposed into steps that upgrade can be executed two at a time. This feature can have impact on the number of SUs that will be available to provide the service during the upgrade, but it helps perform the upgrade faster.

- ✓ The actions this upgrade procedure should perform. Mainly configuration modifications, or software installations/removals. These actions can also include calls to administrative operations, or command line calls in order to prepare for a given a change. These actions can either be performed in one step or several steps depending on the upgrade method.

The behavior that SMF implements for each upgrade procedure is described in [4] using a finite state machine. The execution or rollback of an upgrade procedure is triggered by the upgrade campaign, and as a result it triggers the execution of the upgrade steps that compose this upgrade procedure. In addition, this behavior also covers the failure cases, and proper messages to send to the upgrade campaign in order to take appropriate measures to stop the propagation of the fault and correct it.

2.3.2.3. Upgrade step

Upgrade steps are only specified through the common attributes of the steps that compose a given upgrade procedure. These attributes are:

- ✓ Max retry: which is the maximum number of times a step is allowed to be retried before it triggers a campaign suspension or failure [4].
- ✓ Restart option: when the scope of a procedure is composed of components, its steps can avoid taking the assignments away from these components by just restarting the components. This can only take place when this attribute is set.

Among the entities in the scope of the upgrade procedure, each step will take some of them out of service, and put another subset in service. The set of entities a step takes out of service is called a deactivation unit (DU), while the set of entities the step puts in service is called activation unit (AU). When the activation and the deactivation units are the same, we call that set a symmetric activation unit (SAU).

During its execution, an upgrade step has standard actions that it takes. These actions go along the following lines:

- ✓ Online installation of new software.
- ✓ Lock deactivation unit.
- ✓ Terminate deactivation unit.

- ✓ Offline uninstallation of old software.
- ✓ Modify information model.
- ✓ Offline installation of new software.
- ✓ Instantiate activation unit.
- ✓ Unlock activation unit.
- ✓ Online uninstallation of old software.

The steps for which the restart option attribute is set are called reduced steps and they take the following actions:

- ✓ Online installation of new software.
- ✓ Modification of information model.
- ✓ Restart symmetric activation unit.
- ✓ Online removal of old software.

These actions are taken during the execution of the step. When an action fails, SMF undoes all the actions that were taken by this step before the failure. Once these actions are successfully undone, SMF reattempts this step if the retry count has not yet exceeded the specified max retry. Otherwise, this can only lead either to the failure of the upgrade campaign, or a suspension of the upgrade campaign.

Note that the administrative operation taken on the deactivation/activation unit are the main cause of service outage. In addition, the bigger the entity on which the administrative operation is taken, the more time consuming and exposed to failure this latter is.

2.4. Model Driven Engineering

Model Driven Engineering (MDE) is a new trend in software engineering. It focuses on models to make them more of assets than overheads, in contrary to traditional software engineering methodologies that use models only for documentation. This transition can be made by using appropriate tools that can help perform various validation, evolution and extraction of software engineering artifacts on/from models. Thus, providing an environment that enables full or partial automation of most of software engineering activities as well as reuse the models as they are defined at a high level of abstraction. There is a wide range of tools that, if combined, can enable an MDE process. Computer Aided Software Engineering (CASE) tools were the first to be introduced, usually used for model editing, visualization, and automatic code generation. This category includes tools such as MagicDraw [40], RSA [41], StarUML [42], ArgoUML [43], EMF [45], Papyrus [44], etc. The other MDE related domain that interested software engineering tools vendors was model management, including transformation, validation, merging, comparison, and every other activity that might relate or operate on models. In this category, we find tools such as ATL [28], EPSILON [21], Kermeta [47], QVTO [46], etc. The Object Management Group (OMG) [48], one of the most influential consortiums and communities in the software engineering domain, published a standard called Model Driven Architecture

(MDA) [49]. This standard can serve as a reference for MDE tools developers and promote the interoperability between the tools. It was based on existing OMG standards such as the Unified Modeling Language (UML) [50], the Object Constraints Language (OCL) [51], the Query View Transformation (QVT) [52], the XML Metadata Interchange (XMI) [53], and the Meta-Object Facility (MOF) [54]. Several MDE tool vendors already adopted this standard at least partially.

2.5. EPSILON

Epsilon is a self-contained model management environment that was created specifically to overcome some limitations of existing tools, such as:

- ✓ No support for model modification capabilities.
- ✓ No support for multiple models navigation or inter-model constraints expression.
- ✓ No independence of the modeling technology. Once the technology used in input or output models changes, all model transformations will have to be changed as well.

Epsilon is built on top of the general purpose language Epsilon Object Language (EOL) [24] as a family of task specific languages for model management. EOL solves all the aforementioned limitations of other languages by offering:

- ✓ A family of task specific languages that we will define further in this document.
- ✓ Capabilities for multiple models navigation, multiple input models and multiple output models.
- ✓ Capabilities for inter-model constraints checking.
- ✓ Capabilities and ease of extending this family of languages with a new language.
- ✓ Support for java code injection in the code of all the languages.

Some of the task specific languages that were built on top of EOL include:

- ✓ Epsilon Validation Language, EVL: a language for constraints checking and inconsistencies repairing.
- ✓ Epsilon Transformation Language, ETL [22]: a language for model to model transformation, it can take an arbitrary number of input models and generate an arbitrary number of output models. This type of transformation is called mapping transformation because usually the input models and output models do not use the same modeling language.
- ✓ Epsilon Wizard Language, EWL: a language for in-place model modifications, also called update transformations.
- ✓ Epsilon Generated Language, EGL [23]: a language for model to text transformation, the ease of use of EGL is basically due to the fact that it is template based.
- ✓ Epsilon Comparison Language, ECL: a language for rule based models comparison.
- ✓ Epsilon Merging Language, EML: a language for rule based model merging.

- ✓ Epsilon Flock for Model Migration: a language for updating a model in response of meta-model changes, or to migrate the model from one technology to another.
- ✓ Epsilon Pattern Language, EPL: a language for pattern matching in models.

2.6. DEVS formalism

Discrete Even System Specification, DEVS, is a general purpose system modeling formalism invented by Bernard P. Zeigler in 1976 [31]. Even if DEVS is mainly based on discrete events, its applicability is not restricted to discrete events based system. In fact, it has various extensions and specializations used for a wide range of systems including parallel systems, real time systems, cellular systems, and dynamic systems where the coupling structure changes dynamically.

DEVS enables system modeling using elements called DEVS models. A DEVS model has the ability to change its state independently of its environment, and it has ports through which it receives input events and fires output events. The behavior of DEVS is defined through a set of functions, and its structure is defined either through ports only, or ports and other DEVS models depending on the nature of the DEVS model.

DEVS formalism introduces two types of DEVS models:

- ✓ DEVS atomic models: are models that, structurally speaking, do not contain any other DEVS models. And they are defined using the tuple: <X, Y, S, Ta, $Delt_{In}$, $Delt_{Ext}$, Lambda>
  - X: is the set of input events.
  - Y: is the set of output events.
  - S: the set of states.
  - Ta: time advance function, which defines the time spent in each state (lifespan of a state).
  - $Delt_{In}$: the function of internal transitions, in this kind of transitions the decision is made just based on the system state without referring to external events.
  - $Delt_{Ext}$: the function of external transitions, in this kind of transitions the decision is made based on the current state of the system and the recent external events.
  - Lambda: the output function, this function specifies the events that the atomic model should fire after every state transition.
- ✓ DEVS coupled models: are complex models that contain other coupled or atomic models within them. And they are defined using the tuple: <X, Y, D, $\{M_i\}$, C_{xx}, C_{yy}, C_{yx}, Select>
  - X: is the set of input events.
  - Y: is the set of output events.
  - D: the set of subcomponents' names.

- $\{M_i\}$: the set of subcomponents.
- C_{xx} : is the set of external input couplings. From coupled model input to one or some of its subcomponents inputs.
- C_{yy} : is the set of external output couplings. From one or some of the subcomponents outputs to one or some of the coupled model outputs.
- C_{yx} : is the set of internal couplings. Couplings between subcomponents.
- Select: is the function which defines how to select the event from the set of simultaneous events.

Many simulators use DEVS as a modeling formalism including MS4 system, DEVS-Suite [29], PyDEVS, and many others.

2.7. Related work

Component dependencies and dynamic reconfiguration of component based systems have been thoroughly investigated in the literature. Chen [8] proposes an approach for dynamic dependency management for dynamic reconfiguration of component based systems. He considers the "static" (design time) known functional dependencies among components, but defines the concept of dynamic dependency that holds temporarily when a client component calls a method in a server component. The idea is that the dependencies defined at design time do not hold all the time during execution but only when a component is using another component. The proposal is to monitor the interactions of each component using a virtual stub that registers ongoing interactions, block interactions when needed for the reconfiguration, and resume blocked interactions after reconfiguration. The proposed approach is not applicable in the context of service high-availability as blocking interactions between components while they are still providing service will certainly lead to service outage.

Matevska et al. [9] tackle the problem for the same kind of dependencies as in [8], with the goal of minimizing service outage. They define the concept of dynamic de-pendency graph to keep track of which component is currently using which other component. Components can be in different states, free, passive and active. Components are only changed when they are in free and passive states; before the changes are performed they are blocked and incoming invocations are queued. To avoid service outage, the idea is to find the optimal point in time during the evolution of this dependency graph, and change a component when there is no runtime dependency to it. The authors are concerned with high-availability and service outage, but there is no guarantee there will be an optimal point in time and there is no guarantee about the duration of the changes while incoming invocations/requests are blocked. A similar approach, where software modules are only upgraded in safe states and future incoming requests are buffered is described in [10].

Dependencies relevant for upgrades in the context of AMF have been studied in [11]. Two kinds of dependencies are considered: functional dependencies (directed dependencies in this paper) and upgrade dependencies. Upgrade dependencies are dependencies between two upgraded components that did not exist between the original components.

A directed graph is created from these dependencies and taken into account for the design of the upgrade campaign. However, not all relevant dependencies are considered (e.g. collocation dependencies) and the type of applications that are considered is limited.

Other works [12], [13] consider component dependencies during dynamic recon-figuration of component based applications from the perspective of application consistency, not removing a component while it is being used by another one, avoiding dangling references, etc., but with no consideration to the service outage and high-availability. Dependencies are determined at runtime and taken into account before removing or updating a component. Other approaches rely on specific operating systems [14], container environments [15] or component models [16] or use low-level approaches with wrapper-like functions [16] or modify source code [18]. An overview of techniques used for dynamic reconfiguration can be found in [19].

Upgrade execution time estimation is one of the main concerns of systems administrators. This execution time is an aggregation of execution times of the actions taken during the campaign's execution. These actions can be either software installation or removals or some component management action. The upgrade time of database and software has been investigated in the recent years by both research and industry such as the work presented in [33]. For some database based software and for some specific types of upgrades, bounded by some constraints on the type of changes to be done on the schema and the data, Michal et al. in [33] propose an evolutionary algorithm that uses data from previous upgrades to estimate the time required to upgrade every atomic entity in the system (in this case business objects), and then estimate the time required for the next upgrade based on the business objects that will be upgraded and the previously estimated respective upgrade durations.

Two of the major challenges of availability are the unplanned downtime due to software glitches and the planned downtime caused by system upgrades. That makes the outage induced by an upgrade campaign one of the main factors to consider while estimating the availability of a system on the long term. The work in [34] presents an assessment of reliability and availability of a Storage Area Network (SAN) after two types of extensions: SONET based SAN extension, or an IP based SAN extension. In the analytical models used to compare the solutions, the authors took into consideration system upgrades by setting a fixed number of upgrades per year, their durations and the downtime they induce. Using these hypothesis they were able to estimate the availability and reliability of the SAN in the long term. In our work, we are not targeting the system configuration that will make the system most reliable and available, but we suppose the administrator already has this configuration and the upgrade campaign specification that can take the system to that configuration, and we try to estimate the cost of this migration in terms of execution time and service outage.

Unlike the work in [34], Kanso et al. in [35, 36] did not consider the effect of upgrade campaigns on the downtime in the long term. The authors proposed an approach to estimate the availability of a service provided by a SAF system based on its current configuration. They used Deterministic and Stochastic Petri Nets (DSPNs) [39] to model the configuration, and they explored components' failure modes and recovery actions mapped to every failure mode. They extended the SAF models in a similar way as we did in order to provide the missing time and failure data. Unlike our work, the work in [35] was using a different formalism to model a SAF system, in addition in our work we estimate the downtime during an upgrade campaign which is the planned downtime, while [35] was focused mainly on the unplanned downtime.

Determining execution time of hard real-time system has been thoroughly investigated during the past two decades [37, 38]. In order to guarantee the satisfaction of the deadlines of such systems, designer need to analyze/measure the execution time of the different tasks of the system. This can be done using static code analysis or measurement during execution prior to deployment. Worst Case Execution Time (WCET) analysis determines the upper bounds of the execution time of the different tasks [38]. This is usually done by characterizing the different paths in the program and the constituent instructions (machine level) and the execution time of each instruction. This analysis is generally hardware dependent. The analysis of the execution time of upgrade campaign specification is different in a sense that it is at a higher level and takes into account failures and retrials. Moreover, we also determine the service outage in our simulation.

Chapter 3 - Upgrade campaign specification generation

This chapter presents the challenging issues of upgrade campaign specification design and the need to automate it. It also introduces our model driven approach for automatic upgrade campaign specification generation. This chapter also includes a presentation of the modeling framework used to express various artifacts involved in the generation process.

3.1. Challenges

An upgrade campaign is the process of migrating the system from one configuration to another. In SAF compliant systems, SMF is the engine responsible for the orchestration of such a migration. To perform the changes, SMF takes an upgrade campaign specification file as input. This file contains different actions required to perform system changes. When designing an upgrade campaign specification, the campaign designer can face several challenging issues. These issues arise from the complexity and the size of the system as well as the amount of actions required to move the system from one configuration to another.

When upgrading a highly-available system, an administrator has two main concerns:

- ✓ Correctness of the upgrade: correctness means that the instructions passed to the upgrade engine are performing the required changes.
- ✓ The outage and service disruption the campaign can induce: in HA systems outage can be very costly. While system upgrades are necessary for maintenance purposes, they are considered one of the main causes of outage. Several system aspects can be managed in order to reduce the outage such as:
    - Dependencies: Systems' building blocks can depend on each other, and upgrading one component may impact another. This effect should be considered by the campaign designer while ordering the changes.
    - Upgrade methods: SAF specifications define two upgrade methods, and using the wrong upgrade method to perform the changes might induce some unnecessary outage.
    - Upgrade scope: Choosing the right activation/deactivation unit of an upgrade procedure can reduce significantly the service outage and disruption induced by an upgrade campaign.

Considering all these aspects, designing an upgrade campaign is a significantly complex and error prone task. This task becomes more difficult with more complex systems as we see it in the cloud context. Model driven techniques have been widely used to increase the level of abstraction and ease transformations. In the following sections, we will show

how we addressed these issues to devise a model driven approach for the automatic generation of upgrade campaign specifications.

3.2. Modeling framework

The upgrade campaign specification generation process includes several activities. Each activity has a set of inputs and a set of outputs according to its role. We designed a modeling framework that enables us to express various artifacts generated/exchanged during the generation process. This framework includes three meta-models which represent the three domains of interest for our work. The change meta-model is used to express the changes a system can undergo. The dependencies meta-model is used to capture the relationships between AMF entities and the dependencies between the components they represent. Finally, the upgrade campaign specification meta-model is used to express SAF compliant upgrade campaign specifications. These meta-models are described in the following subsections.

3.2.1. Change meta-model

The change meta-model (Figure 3-1) is used to describe the changes to be performed to take the system from the source configuration to the target configuration. These changes, can be either IMM related changes (ModifyImm, AddToImm, RemoveFromImm) or software related changes (SoftwareChange).

The software related changes (SoftwareChange) are the installation (SwInstallation) or removals (SwRemoval) of software bundles from a set of nodes (UCGNode).

The IMM related changes can be either addition to (AddToImm), removal from (RemoveFromImm) or modification made to (ModifyImm) the existing content of IMM.

A removal from IMM can be done just based on the DN of the object to be deleted, an addition, on the other hand, needs all the attributes of the object to be added. This is why we have every AMF object mapped to a set of attributes characterized by a name, type and value. For modifications we need the source object and target object to be able to define which attributes were modified.

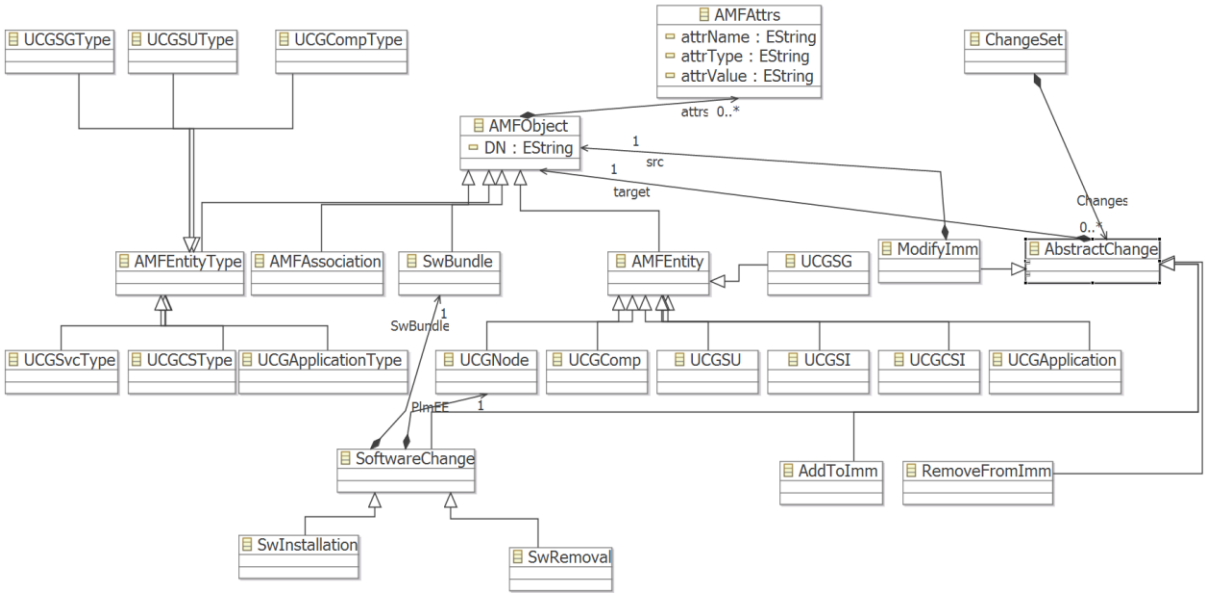


Figure 3-1: Change meta-model

3.2.2. Dependencies meta-model

The dependencies meta-model (shown in Figure 3-2) captures the different dependencies that may exist between entities in an AMF configuration and their relations. These dependencies can be used for different purposes depending on their category:

- ✓ Directed dependencies are used to order the changes. The ordering of the changes will be described in more details later on this document.
- ✓ Symmetrical dependencies are used to handle compatibility and service protection issues. This gives significant insight on what upgrade method to choose (rolling upgrade, or single step upgrade). These dependencies are also used to order the upgrade procedures not only based on the relationships between their targets but also the nature of the change to be performed on each target (addition, upgrade, or removal) [55].
- ✓ Collocation dependencies are used both to improve the choice of the upgrade method and to determine which procedures can be merged with each other.

Further explanations and details regarding these dependencies will be given later in this document.

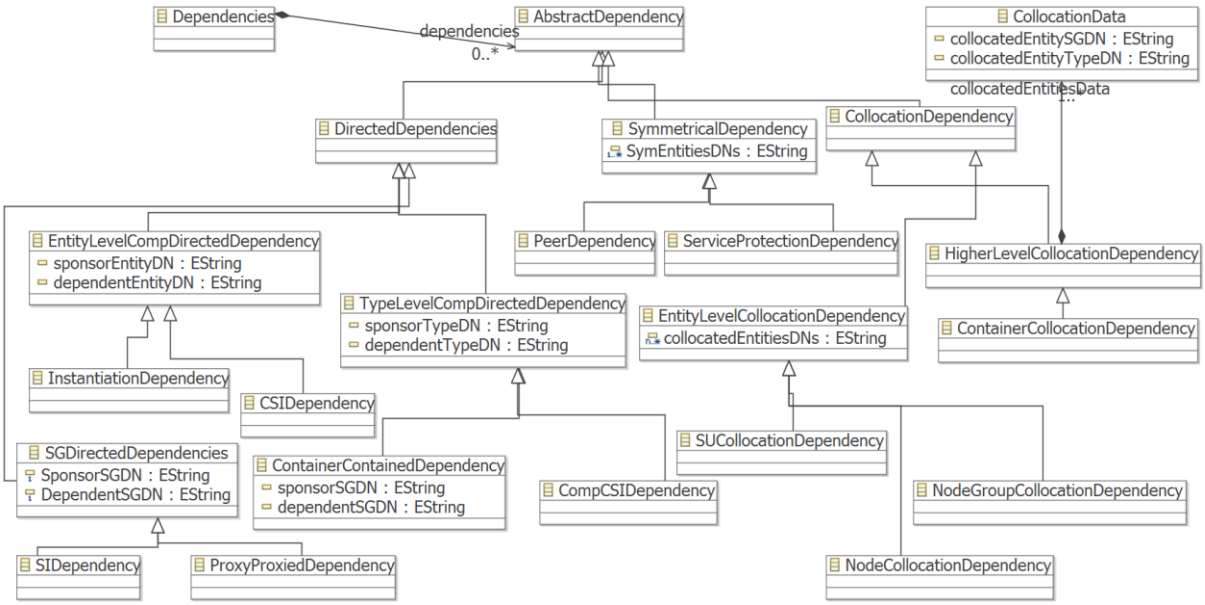


Figure 3-2: Dependencies meta-model

3.2.3. Upgrade campaign specification meta-model

The upgrade campaign specification meta-model (Figure 3-3) captures the concepts needed to generate a SAF compliant upgrade campaign specification. It is described using upgrade objects (UCG_UpgradeObject). Every upgrade object is defined through the definition of its initialization, body, and wrap up sections.

The main upgrade objects (UCG_UpgradeObject) used to specify an upgrade campaign are:

- ✓ The upgrade campaign (UCG_UpgradeCampaign): It is the root element of the upgrade campaign specification.
- ✓ The upgrade procedure (UCG_UpgradeProcedure): The upgrade objects composing the upgrade campaign's body specifying how a set of changes should be deployed.

The upgrade campaign body is composed of upgrade procedures, while the upgrade procedure body is composed of the upgrade step description, which is an ordered list of upgrade actions (UCG_Action), and the target entities. This set of upgrade actions may be repeated for different subsets of the target entities.

The initialization and the wrap up sections are also composed of ordered upgrade actions.

An upgrade action can be:

- ✓ IMM related operation (UCG_ImmOp): like addition, removal or modification of an object in IMM.

- ✓ Software related action (UCG_SwOp): like installation and removals, usually called based on the software bundle DN and the node on which the installation should be done (PlmEE).
- ✓ Administrative operation (UCG_AdminOp): administrative operations defined in the AMF standard (Lock, Unlock, Lock-Instantiation, and Unlock-Instantiation), these operations are called on objects based on their DNs, and for every administrative operation we should define how it should be done in the direct execution and in rollback (doing and undoing of the administrative operation).
- ✓ Callback (UCG_Callback): called on entities deployed in the system, based on their DNs.
- ✓ CLI Command (UCG_Cli): commands called using the CLI (command line interface), they are specified using the path of the command, its arguments. For each command we specify how it is called in the execution and in the rollback paths (doing and undoing of the command).

While software related actions can only be part of a procedure body, the other four can be used in the initialization and wrap-up sections as well.

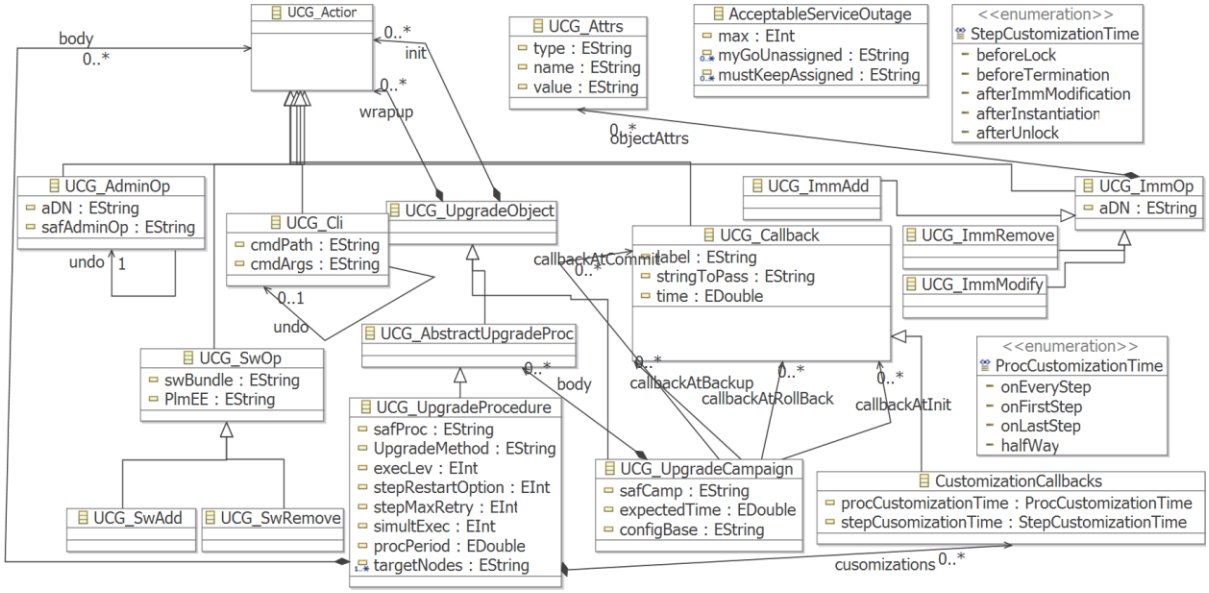


Figure 3-3: Upgrade campaign specification meta-model

3.3. Overall approach

The main inputs of our approach for the upgrade campaign specification generation are:

- ✓ Source configuration: The configuration describing the system in its current state.
- ✓ Target configuration: The configuration describing the state where we want to take the system after executing the upgrade campaign.

- ✓ ETFs: The Entity Types Files (ETF) of the software available in the software repository describing the software bundles deployed and to be deployed in the system and the entity types they provide.

The source configuration and the target configuration go through a first transformation, as shown in Figure 3-4, that creates a model called a change model. The change model is an instance of the change meta-model and describes the changes that need to be performed on the source configuration to get to the target configuration. From this change model, and using another transformation, we generate an elementary upgrade campaign specification model, instance of the upgrade campaign specification meta-model, that contains an upgrade campaign element for each change of the change model. This upgrade campaign specification model goes then through a first refinement that matches upgrade campaign elements that can be performed in the same upgrade procedure. For each match, all the upgrade campaign elements are merged into an upgrade procedure and the elements not required anymore are deleted from the upgrade campaign specification model. The three inputs should also go through a transformation where the different dependencies between the system's entities are extracted into a dependencies model, instance of the dependencies meta-model, from each of the configurations and the provided ETFs. Finally, and using the dependencies in the dependencies model, the upgrade campaign specification model is refined for the second time using a transformation that takes into consideration the dependencies between the system components to create a partial order and to determine the optimal scope for each set of matched changes.

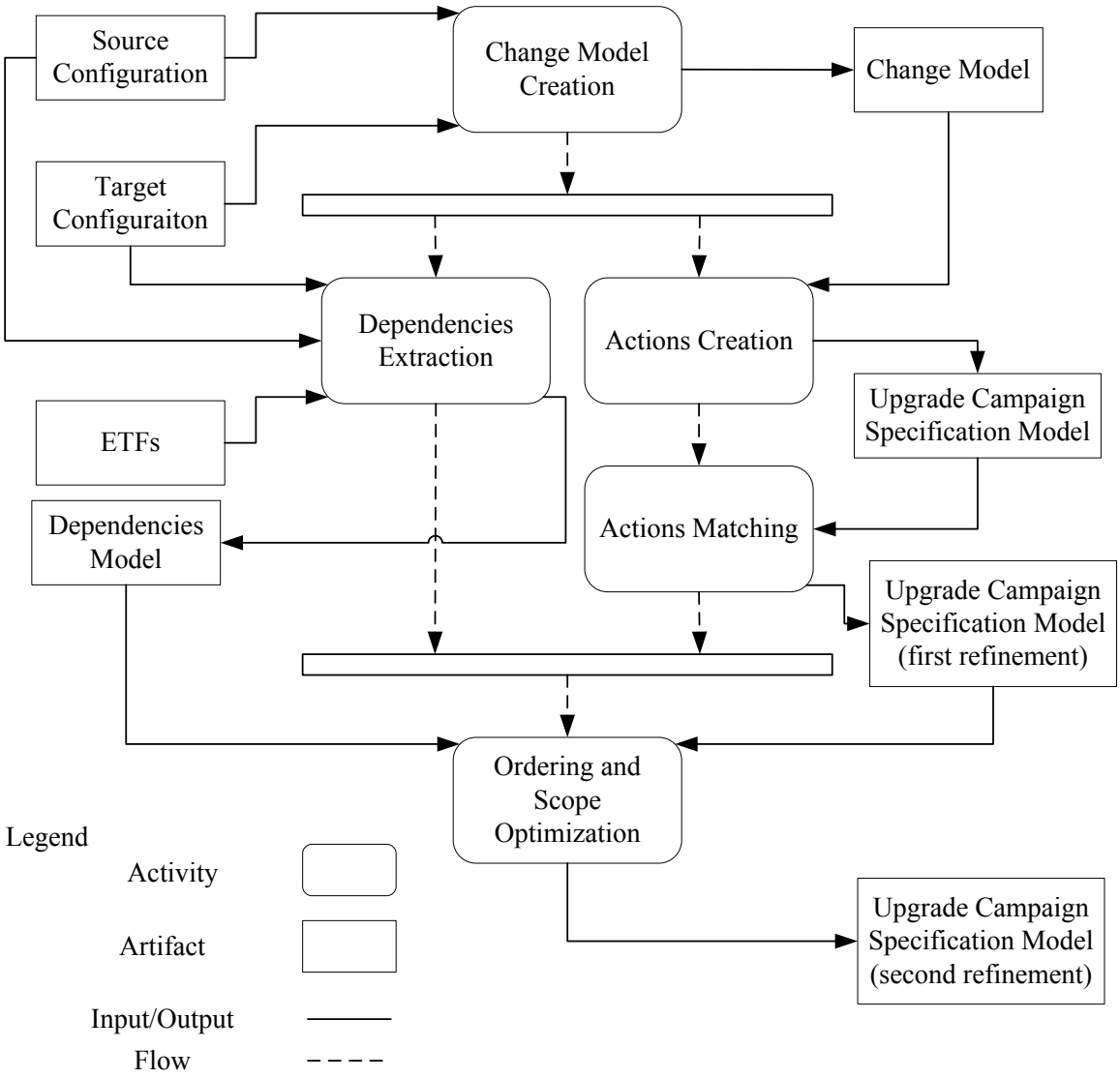


Figure 3-4: Overall approach

3.4. Transformations

3.4.1. Change model creation

The creation of the change model consists of comparing the source and target configurations and determining the changes between them, which should imply the actions required to take the system from the state describe by the source configuration to the desired state of the target configuration. This comparison is not straight forward, and was handled in the work in [6]. This section will describe briefly the challenging issues of this task and how the work in [6] proposed to solve them.

The source and the target configurations may not use the same naming for the entities. The assumption though is that at least the service instances, which are provided before,

after and throughout the upgrade have the same name in the two configurations as name changes of entities in the runtime system is not permitted.

Starting with this assumption the work in [6] proposes to identify the configuration objects representing the provider entities of each service instance in each of the configurations and map them (i.e. their names) to each other in each of the configurations to have a consistent naming of configuration objects. Next one should compare the mapped configuration objects, their attributes and their associations. If the configuration of the represented entities, their types and associations are identical in the two configurations then they are not targeted by the upgrade and they are not considered any further. The work in [6] did not follow a model driven approach, and it does not output a model. The change meta-model that we designed and described earlier allows us to put the results of this difference calculation into a model for further uses.

Mapped configuration objects representing entities whose attributes, types or associations change between the two configurations are the AMF entities targeted for modification in the upgrade and based on the differences between the configurations we add to the change model the appropriate AMFEntity objects and their ModifyImm changes. If there is a type change we check if we need to also add a new AMFEntityType objects associated with the AddToImm and SwInstallation changes, an old AMFEntityType objects associated with the RemoveFromImm and SwRemoval changes, or a AMFEntityType objects associated with ModifyImm change for any modified type.

For configuration objects (AMF entities, AMF types and objects of association classes) present only in the source configuration and not handled yet we add the appropriate AMFEntity and/or AMFType objects and the RemoveFromImm and SwRemoval changes to the change model. Similarly for configuration objects present only in the target configuration we add to the change model the appropriate AMFEntity and/or AMFType objects and the AddToImm and SwInstallation changes as necessary.

3.4.2. Actions creation

This transformation takes the change model as input and outputs an elementary upgrade campaign specification model which contains an upgrade campaign element for each change contained in the change model.

These upgrade campaign elements are created according to the standard upgrade campaign schema and they respect the logic described in the different SAF specifications. This implies of putting the right upgrade actions in the right sections of the upgrade campaign specification some parts of which require explicit specification of actions and their targets (e.g. initialization, wrap-up), while other parts have standard actions and require only their parametrization (e.g. upgrade step).

Table 3-1: Mapping changes to corresponding SAF compliant actions and their positions

	Campaign initialization	Procedure initialization	Procedure body	Procedure wrap up	Campaign wrap up
Software installation	Add software bundle to IMM		Software installation		
Software removal			Software removal		Remove software bundle from IMM
Add type	Add type to IMM				
Remove type					Remove type from IMM
Add SI		Add SI in locked state		Unlock SI	
Remove SI		Lock SI		Remove SI	
Modify SI		Lock SI, modify SI		Unlock SI	
Add SU, Node, or Comp			Add entity in AU		
Remove SU, Node, or Comp			Remove entity in DU		
Modify SU, Node, or Comp			Modify entity in symmetric AU		
Add/modify other entities		Additions or modification			
Remove other entities				Removal	

The elementary upgrade campaign specification model may not yet be fully compliant to the model of the standard upgrade campaign specification schema as at this stage some schema elements are only partially defined. E.g. an upgrade procedure may only have an initialization and/or a wrap-up section, but not a body.

In the change model, we have distinguished whether the target of an IMM operation is related to the AMF entity type or an AMF entity. This was to be able to put these operations in the right section of the upgrade campaign specification as indicated in Table 3-1. Another aspect of respecting the logic of SAF specifications stipulates the need of preparing for an upgrade action implementing a change before performing it, for example before removing an SI it should be first locked.

This set of rules is summed up in Table 3-1. It indicates for each possible change of the change model (rows) the corresponding upgrade actions (cells) and the section of the upgrade campaign specification (columns) they need to be placed

The modification of types and AMF associations is not a straight forward task, and requires a lot of precaution to decide how to perform and where to put such a modification to keep the configuration consistent. A problem often faced with type modifications is the case when during the upgrade we have peer (aka collaborating redundant) entities that operate under different types (configurations). To be able to handle this kind of changes we classified the relevant attributes of the types and associations into three categories, and each category is treated differently as shown in Table 3-2.For the attributes with the "set Max before" strategy, we start by setting the changed attribute to the maximum of old and new values, and set it to the new value later after upgrading the entities. For the attributes using the "set Min before" we set the value of the changed attribute to the minimum of new and old value. Attributes of the third category are data collections and require that we extend them before their upgrade to make the entities able to operate in both old and new configuration, and later on we remove what has to be removed and keep only the desired configuration.

With respect to the placement of the changes the modifications of types can be of two kinds:

- ✓ Modification of the type of the entity being upgraded: In this case we place the changes to prepare for the upgrade (the set Max before for example) in the initialization of the procedure, and to perform the rest we put them in the wrap-up section of the same procedure.
- ✓ Modification of the type of the sponsor or dependent entity: Modifications are put in the initialization and the wrap-up of an independent procedure specifically ordered with respect to the procedure operating on the entity being upgraded based on compatibility

Table 3-2: Types and AMF Associations Attributes Handling Strategies

Class	Attribute	Set Max before	Set Min before	Add first, remove later
SaAmfSGType	saAmfSgtValidSuTypes			X
	saAmfSgtDefCompRestartProb	X		
	saAmfSgtDefCompMaxRestart	X		
	saAmfSgtDefSuRestartProb	X		
	saAmfSgtDefSuMaxRestart	X		
SaAmfSUType	saAmfSutProvidesSvcTypes			X
SaAmfSutCompType	saAmfSutMaxNumComponents	X		
	saAmfSutMinNumComponents		X	
SaAmfSvcType	saAmfSvcDefActiveWeight	X		
	saAmfSvcDefStandbyWeight	X		
SaAmfSIDependency	saAmfToleranceTime	X		
SaAmfSvcTypeCSTypes	saAmfSvctMaxNumCSIs	X		
SaAmfCompType	saAmfCtDefClcCliTimeout	X		
	saAmfCtDefCallbackTimeout	X		
	saAmfCtDefInstantiationLevel	X		
SaAmfCtCSType	saAmfCtDefNumMaxActiveCSIs		X	
	saAmfCtDefNumMaxStandbyCSIs		X	
SaAmfCompCsType	saAmfCompNumMaxActiveCSIs		X	
	saAmfCompNumMaxStandbyCSIs		X	

3.4.3. Actions matching

The actions matching transformation refines the elementary upgrade campaign specification model generated by the previous transformation by matching the upgrade campaign elements that should be done within the same procedure. After this matching transformation the upgrade campaign specification model should be fully compliant to the model of the standard upgrade campaign specification schema.

To perform the matching we propose the following rules:

- ✓ Rule #1: actions on peers match. Meaning that upgrade campaign elements on components or SUs, which are redundant of each other, should be done in the same procedure. E.g. upgrade of SUs of the same SG match and the upgrade campaign elements are merged into a rolling upgrade procedure
- ✓ Rule #2: actions on entities match actions on their children. That means that the upgrade campaign elements with upgrade actions on a parent and on its child need to be done within the same procedure. E.g. the upgrade of an SU and its component match their upgrade campaign elements are merged together into the same upgrade procedure.
- ✓ Rule #3: actions on entities match actions on software bundles providing their types or the types of entities they match. That means that the upgrade campaign elements with the modifications of entities and with the installation/removals of software associated with these modifications should be done in the same procedure. E.g. the upgrade of a component and the installation of the software bundle delivering its type match and merged into the same upgrade step/procedure.
- ✓ Rule #4: actions on services match actions on entities protecting them. Meaning that any modification of a service entity matches the modifications on their service provider entities. For example, a modification of an SI matches the modifications on the SUs protecting the SI.
- ✓ Rule #5: for the matching of actions on AMF associations we distinguish four cases:
    - Type to Type associations: do not need to be matched since they will be done in the campaign initialization or wrap-up section.
    - Service to Service associations:
        - for addition: they are matched to last added and put in the wrap-up of the procedure,
        - For removal: they are matched to first removed and put in the initialization of the procedure.

- Type to Entity associations: matched to the actions performed on the entity, they are added in the wrap-up of the procedure and removed in the initialization.
- Entity to Entity associations:
  - For addition: they are matched to the first added and put in the wrap-up,
  - For removal: they are matched to the last removed and they are put in the initialization.

These rules if applied in arbitrary order might result in the wrong matching up of actions. For instance, let assume that we have two SGs with SUs of the same type protecting two service instances of the same service type. In this case, if these rules are applied in the wrong order that might result in the matching of actions on SUs of the first SG with actions on SUs on the second SG, i.e. if Rule #3 is applied first. In order to avoid such mistaken matching, we recommend that Rules #1, #2, and #3 to be applied in this order, while the implementer may decide of any ordering for the rest of the rules.

3.4.4. Dependencies extraction

In this transformation we extract from the ETFs, the source and the target configurations the different dependencies between the system's components based on their attributes and the way they are deployed. The different dependencies that we can extract are as follows:

- ✓ SI dependency: between two SGs one of them protecting a sponsor SI (Sponsor SG) and the other is protecting a Dependent SI (Dependent SG).
- ✓ Proxy-Proxied dependency: between two SGs one of them protecting a Proxy SI (Sponsor SG), and the other is protecting its Proxied SI (Dependent SG).
- ✓ Instantiation dependency: between a pre-instantiable component of a SU (Sponsor), and another pre-instantiable component of the same SU with a lower Instantiation Level (Dependent). Lower instantiation Level means that the value of the attribute saAmfCompInstantiationLevel is higher.
- ✓ CSI dependency: between CSIs of the same SI. Implying that the sponsor CSI should be assigned before the dependent CSI.
- ✓ Container-Contained dependency: we characterize the sponsor and dependent in this kind of dependency based on their SG and type, meaning that components belonging to the SG SponsorSG and of type SponsorType, are containers of components of the SG dependent SG and of type DependentType.
- ✓ CompCSIDependency: this kind of dependency is the instantiation dependency version for non-pre-instantiable components. It exists between components of the

same SU, and means that components of the SU of type DependentType depend on components of the same SU that are of type SponsorType. This dependency imposes only a partial order, since we say that this dependency exists between two types if and only if none of the CSIs handled by the SponsorType depend on any of the CSIs handled by the DependentType and all the CSIs handled by the DependentType depend on at least one of the CSIs handled by the SponsorType. In this case we can deduce an ordering for the changes on the components of that SU otherwise we upgrade them all at once taking as AU/DU at least to SU level (no restartability option).

- ✓ SU collocation dependency: exists between components sharing the same SU.
- ✓ Node collocation dependency: exists between SUs configured for the same Node, or SGs having a Node in common.
- ✓ NodeGroup collocation dependency: exists between SGs sharing the same Node-Group.
- ✓ Container collocation dependency: derived from container-contained dependency, and identifies the pairs (dependent type, dependent SG) that share the same pair (sponsor type, sponsor SG).

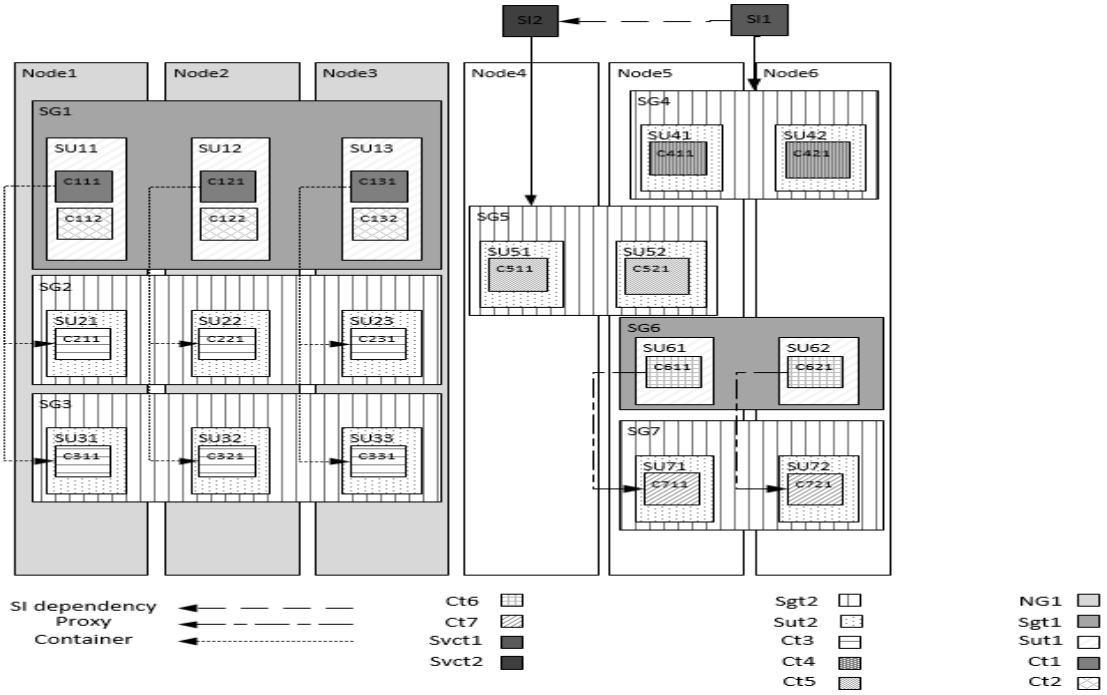


Figure 3-5: Example of a deployment configuration

The deployment configuration shown in Figure 3-5 has one Node Group (NG1) on which three SGs (SG1, SG2, and SG3) are deployed. Each one of these SGs has three SUs, and

the components of the SUs of SGs 2 and 3 are contained by the first components of the SUs of SG1. This part of the configuration shows the following kinds of dependencies:

- ✓ Node Group collocation dependency between SGs 1, 2, and 3.
- ✓ SU collocation dependency between components of the same SU (mainly at the SG1's SUs level).
- ✓ Service protection dependency between SUs of the same SG.
- ✓ Container-Contained dependency components of SUs of SG1 and components of SUs of SGs 2 and 3.
- ✓ Container collocation dependency between components of SUs of SGs 2 and 3.
- ✓ Peer dependency between components in the same SG and of the same type (in our figure same filling).

The second part of this deployment configuration shows SGs 4, 5, 6, and 7 for which the SUs are configured per Node. SG4 protects SI1, SI1 depends on SI2 that is protected by SG5; and SG6 contains components that proxy other components within SG7. So, in addition to the already mentioned dependencies (Service protection and Peer), this part of the configuration shows other types of dependencies:

- ✓ Node collocation dependency between SUs configured for the same Node (SU41, SU52, SU61, and SU71 for example)
- ✓ SI dependency between the Sis protected by SG4 (dependent) and SG5 (sponsor).
- ✓ Proxy-Proxied dependency between components within SG6 (proxy a.k.a sponsor) and components within SG7 (proxied a.k.a dependent).

Figure 3-6 shows a dependencies model (instance of the dependencies meta-model) that describes the dependencies in the already described configuration, attributes of some selected dependencies instances were emphasized in order to show the attributes we use to describe each type of dependency based on different participants.

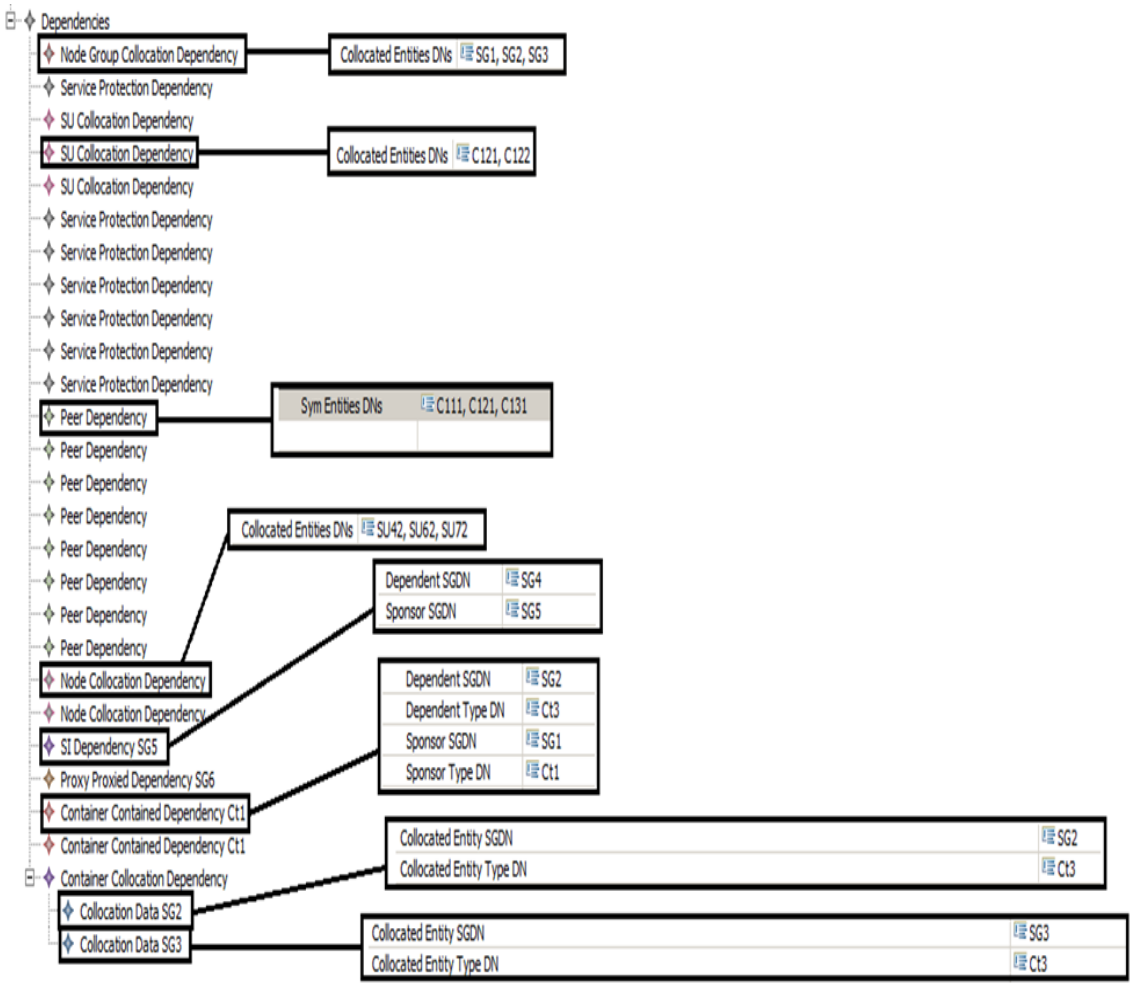


Figure 3-6: Figure 3-5 deployment corresponding dependencies model

3.4.5. Ordering and scope optimization

The last transformation is takes into consideration the dependencies extracted from the source, the target configurations and ETFs to determine the appropriate ordering of the execution of upgrade procedures. As mentioned before two categories of dependencies have to be considered in the ordering of upgrade procedures: symmetrical and directed dependencies.

The changes of entities related by a symmetrical dependency are to be ordered as described in [55]:

- ✓ Addition can happen whenever possible with respect to any directed dependencies the entities are involved.
- ✓ Upgrade of an entity cannot happen before the addition of all the entities which need to be added, and which are related to this entity through a symmetrical dependency.

- ✓ The removal of an entity cannot happen before the upgrade of all entities which need to be upgraded, and which are related to this entity through a symmetrical dependency.

The directed dependencies are mainly driven by the compatibility between the sponsor and the dependent entities, and the rationale that the dependent entity cannot exist without the sponsor. The ordering as given in [57] is summarized in Table 3-3.

Table 3-3: Ordering rules for directed dependencies

Change on sponsor	Change on dependent	Order
Addition	Addition	Sponsor first
Removal	Removal	Dependent first
Addition	Upgrade	Sponsor first
Upgrade	Addition	Sponsor first
Upgrade	Upgrade	Depends on compatibility
Removal	Upgrade	Dependent first
Upgrade	Removal	Dependent first

The rules described in Table 3-3 impose an order that allows for performing the changes without violating the different dependencies. However, these rules do not cover other factors that impact the outage the upgrade campaign may induce, such as:

- ✓ The choice of AU/DU.
- ✓ The choice of the upgrade method.
- ✓ If there is a chance for the upgrade campaign triggering a rollback it is preferred if this happens as early as possible in the execution. Rollback is triggered by a failure and implies that all the procedures executed successfully before the failure are rolled back. Hence earlier this happens less impact it has on the system.

We propose some heuristics to improve the quality of an upgrade campaign specification. This quality improvement can reduce the outage the upgrade campaign may induce and the time it may take. In the following, we work under the assumption that the Node has the biggest scope of impact, followed by the Container then the SU, while the Component has the smallest scope of impact:

- ✓ Heuristic #1: keep the AU/DU to the minimal scope of impact. In other words, the AU/DU will be at most of the scope of impact of the software bundle installation/removal. If the upgrade consists only of IMM modifications then the AU/DU will be modification scope bounded. For instance, if an SU is to be modified, there is no need to lock the Node.
- ✓ Heuristic #2: put as many changes as possible into the procedures having a bigger scope of impact. For instance, if the upgrade of a contained entity has the scope of impact of the Container we can upgrade with it all collocated contained entities.
- ✓ Heuristic #3: procedures with bigger scope of impact should be executed as early as possible. The rationale is that more actions a step may take more likely it will fail. Since a step with a Node as its AU/DU, for instance, can take actions on any and all of the hosted entities it is more likely to fail than a step that has an SU as its AU/DU.
- ✓ Heuristic #4: an execution level should contain procedures of the same scope of impact. In order to force Heuristic #3 to apply also in a fully ordered upgrade campaign.

3.5. Summary

The design of an upgrade campaign specification is not a straight forward task. This design implies insuring the validity of the upgrade campaign specification as well as the proper ordering of the changes it should perform. In order to achieve this, one should handle several aspects of the system including the dependencies between its components. Automation is a viable solution to minimize human intervention in this process and make it more efficient. In this thesis we propose a model driven approach to automate upgrade campaign specification generation. Our approach goes through different steps which either generate new artifacts or refine already generated ones. These artifacts are models, and we propose a modeling framework that can express them. This modeling framework includes:

- ✓ Change meta-model: used to express the set of changes required to move the system from the source configuration to the target configuration.
- ✓ Dependencies meta-model: used to model various dependencies between system components.
- ✓ Upgrade campaign specification meta-model: used to model a SAF compliant upgrade campaign specification.

Our approach takes as input the source configuration, the target configuration and the ETFs. From the source and target configurations we derive the change set in a first step which outputs a change model (conforms to the change meta-model). From the change model we generate the first upgrade campaign specification model (conforms to the upgrade campaign specification meta-model). This upgrade campaign specification model

undergoes a refinement to become SAF compliant. We use the aforementioned three inputs to extract the dependencies between system components into a dependencies model (conforms to the dependencies meta-model). We use these dependencies to make the previously generated upgrade campaign specification model undergo yet another refinement. This refinement applies a set of heuristics that can help reduce the outage the campaign may induce and the time it may take.

The work in this thesis enables the generation of upgrade campaign specifications that can be used to perform a wide spectrum of types of changes. These types of changes include the modification, addition and removal of AMF entities from the AMF configuration as well as software installations/removals. Yet, there are some types of changes that this approach (as is) cannot handle properly. An example of these changes is the change of redundancy models of SGs. During the change of the redundancy model of an SG, this latter will be composed of SUs that are meant to run under different configurations. A simple rolling upgrade as the ones that an implementation of [4] supports is not a safe choice to perform such a change. Moreover the use of single step upgrades will induce an outage which might not be tolerated. Another such case is when upgrading components to a version of the software with no backward compatibility. As peer components (components playing similar roles in SUs redundant of one another) need to communicate and synchronize states, the incompatibility between the old and the new version of the software might not allow that. One might consider extending this approach to support such types of changes as a potential extension.

Finally, it is worth noting that the work that was presented in this section was published in [57] in the context of this thesis.

Chapter 4 - Upgrade campaign specification evaluation

In order to upgrade a system, the administrator may have multiple candidate upgrade campaign specifications of which he has to choose the one to execute. Choosing one upgrade campaign specification over another requires insight on:

- ✓ The outage each upgrade campaign specification may induce.
- ✓ The time each upgrade campaign specification may take.

Having this kind of information will not only help the administrator compare upgrade campaign specifications, but also check for their applicability. The applicability check in this thesis is based on the following:

- ✓ The outage should not exceed the allowed outage.
- ✓ The execution time should fit within the maintenance window.

In order to perform such an evaluation, one should consider several parameters. These parameters include system components, their behaviors, and dependencies, as well as the specified behaviors that SMF and AMF implement during an upgrade campaign. Therefore, upgrade campaign evaluation is a tedious and error prone task if done manually, thus the need for automation. In the following subsections we will discuss a previously proposed approach for simulation based upgrade campaign evaluation. We will expose its limitations as well as the various extensions we proposed to overcome these limitations. We will also describe the method we propose in this thesis for upgrade campaign specification elimination/selection based on the results of their applicability checks.

4.1. Previous work: simulation based approach for upgrade campaign evaluation

The previous work in [7] proposed an approach based on the DEVS formalism for upgrade campaign specification evaluation.

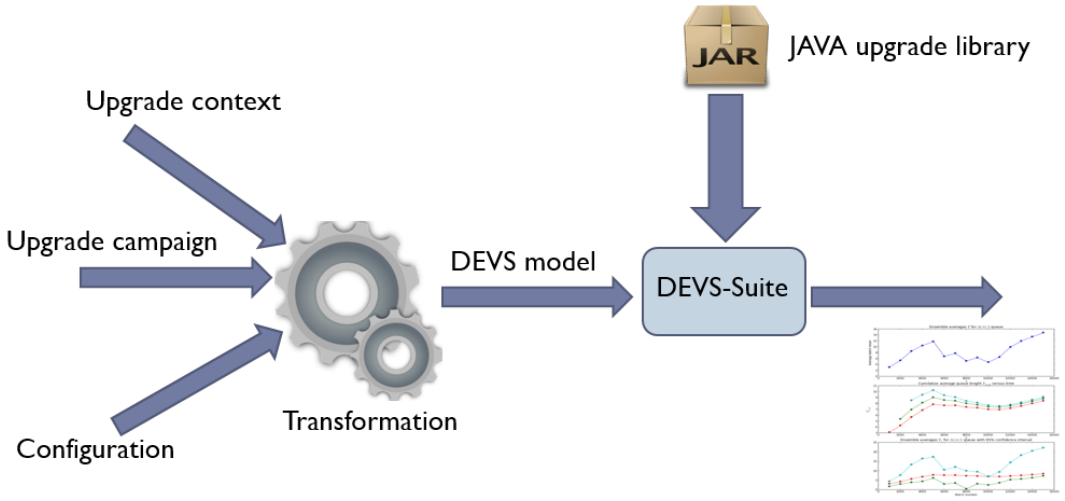


Figure 4-1: approach of the previous work on the evaluation of upgrade campaign specifications [7]

The proposed approach (Figure 4-1) takes as input an upgrade campaign specification, system's current configuration and an upgrade context. These three input go through a model transformation that instantiates for each entity in the upgrade campaign specification and the AMF configuration an associated atomic DEVS model defined in the Java upgrade library. The generated coupled DEVS model as well as the Java upgrade library are loaded into the DEVS-Suite simulator in order to run the simulation. During the simulation we track the assignment state of SIs to trace the outage during the upgrade campaign. For each upgrade procedure and the upgrade campaign we trace the time they spend in the executing state to get the execution time of each procedure and of the whole campaign as well.

The upgrade context in [7] is an input that will provide the failure models of the system's components, and the time attributes for software operations (installation/removal) and administrative operations. However, it did not specify how the upgrade context can be used in a simulation. In addition, no behavior was associated with the DEVS atomic models that represent the AMF components. In other words, AMF components were represented as DEVS atomic models that remain idle during the simulation. Without a behavior associated with these models, one cannot simulate a time-constrained administrative operation. In fact, and administrative operation call usually translates into a set of AMF component lifecycle and management operation calls. These calls are the source of the time constraints that apply to administrative operations.

4.2. Extended simulation approach and limitations of random simulation

In this work we extended the previously described approach in order to:

- ✓ Introduce the attributes related to the upgrade context.

- ✓ Extend and modify the atomic DEVS models to include the components and thus put the upgrade context in action during the simulation.
- ✓ Expose some limitations of the simulation for the evaluation and comparison of upgrade campaign specifications and overcome those limitations using a solution that we will describe further in this document.

4.2.1. Extended approach for upgrade campaign specification simulation

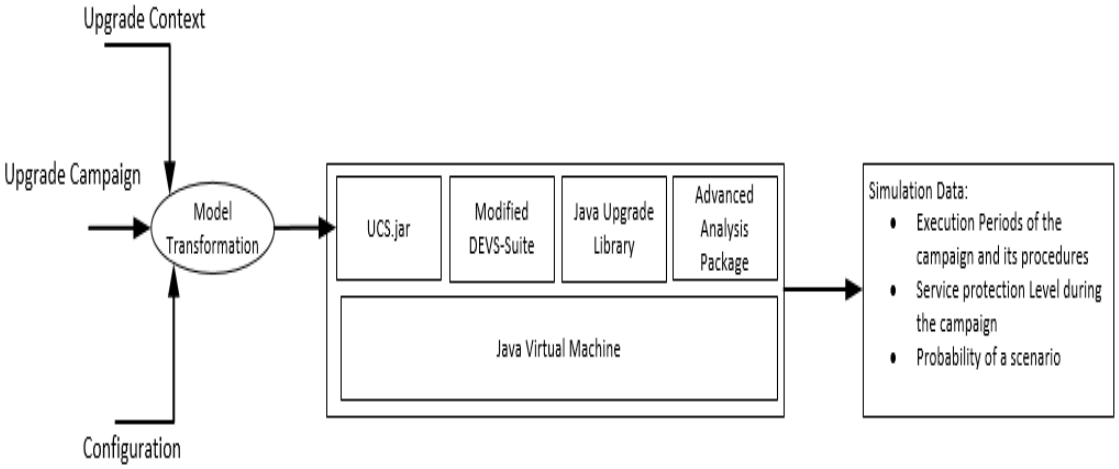


Figure 4-2: overall approach and simulation environment

In our new approach (Figure 4-2) we kept the same set of inputs, however, in addition to the java upgrade library, we introduced two new building blocks of the simulation environment:

- ✓ A modified DEVS-Suite: a modified version of DEVS-Suite simulator that eases the simulation of orchestrated collaborations. The extension to both the formalism and the simulator will be explained further in this document.
- ✓ Advanced analysis package: which help implement some controllable scenarios (Best case and Worst case scenarios), and use them to guide the simulation in order to overcome the limitations of the simulation.

4.2.1.1. Upgrade context

The upgrade context provides additional attributes required for the simulation. It consists of probability and time related data that describe the real behavior of the deployed components.

For the components, we extended ETF model and added further attributes: As shown in Figure 4-3, the ETF model represents the information as described by the ETF xsd of the SMF standard. For the upgrade context we added the bounded times (upper bounds and lower bounds) for component lifecycle and management operations and their respective failure rates as well as the switchover duration.

We added similar extensions for the software bundle, i.e. probability attributes for the success of the different software operations (online installation, online removal, offline installation, and offline removal) and their bounded times.

In addition, the upgrade context includes data at node level: the startup and shutdown durations to be used when a node level recovery action takes place.

The timeout attributes are provided as bounded times. Accordingly, we can use in the simulation as necessary a randomly generated value between the upper and lower bound following a given distribution (commonly exponential distribution), the upper or the lower bound values to simulate the time a given operation takes.

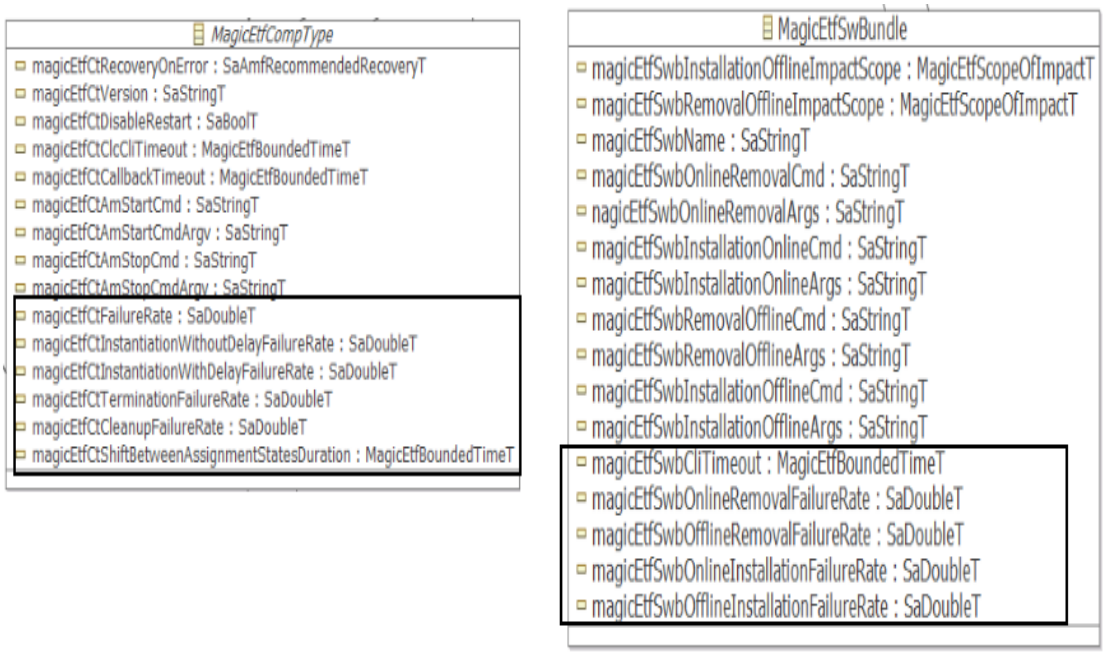


Figure 4-3: Extensions to ETF

4.2.1.2. Extensions to DEVS formalism and DEVS-Suite

The DEVS formalism, as it is, can only model choreographically performed collaborations, meaning that the logic of the collaboration is distributed among collaborating entities. In our case, some of the collaborations are orchestrated, meaning that there is a central entity orchestrating the collaboration between a set of collaborating entities. If we take AMF, for example, at any point in time it may have different services to switchover to different entities with different time offsets for the next event. To make the orchestration of these events easier, the atomic DEVS model representing AMF should have a state-independent time-awareness. To capture this kind of collaborations we extended the DEVS time function in a way that not only the states have life spans but also the events. Therefore, and unlike a usual DEVS model that is considered imminent (or ready for a transition) only when it receives an incoming event or the life span of its current state

expires, a DEVS model simulated under this extension will also be considered imminent when the lifespan of one of its events expires. The DEVS model responds to this transition trigger by firing that event to its destination and going back to wait for the next transition trigger.

We extended the DEVS-Suite simulator APIs to handle output events with lifespan as argument. Keeping in mind backward compatibility, we defined zero (0) as the default lifespan value for an event that should be issued right after its creation. We also extended the simulation mechanisms (simulators and coordinators) to include the event lifespan in the calculation of the next iteration time.

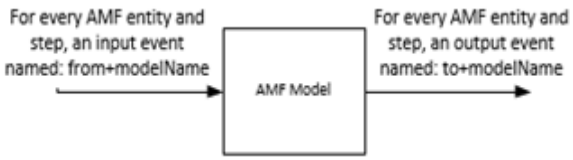
4.2.1.3. New DEVS atomic models in the JAVA upgrade library

The model transformation shown in Figure 4-2 generates a DEVS coupled model from the given configuration, upgrade context and upgrade campaign specification files. This DEVS coupled model is composed of DEVS atomic models obtained by mapping the instances of configuration and upgrade objects to instances of their corresponding DEVS atomic models. These DEVS atomic models are defined in the Java Upgrade Library and their mapping to configuration/upgrade objects is summarized in Table 4-1. This mapping is established according to the communication patterns between the different objects involved in the execution of an upgrade campaign of a given system.

The upgrade campaign object, for instance, needs to communicate with the administrator and its procedures, which explains the I/O ports associated with its DEVS model. Similarly, an upgrade procedure communicates with the campaign and the associated steps. Upgrade steps exchange events with the procedure and with the AMF to perform actions on the logical entities in the AMF configuration. Thus, there is a need for an atomic DEVS model representing AMF. It was designed with an input event from and an output event to every upgrade step and to every configuration object in the final DEVS model.

The main responsibility of the atomic DEVS model representing AMF is the interpretation of the administrative operations issued by upgrade steps on configuration objects. This interpretation is basically a decomposition into associated time constrained AMF component management operations as defined in [3]. The time constraints for these AMF components management operations are given in the system configuration.

Table 4-1: Objects involved in the simulation and their associated DEVS models.

	DEVS Models
Upgrade campaign	
Upgrade procedure	
Upgrade step	
AMF entity	
AMF	

4.2.2. Limitations of random simulation

Our goal is to use the simulation to perform upgrade campaign simulation and evaluation. When comparing two upgrade campaign specifications, and due to the randomness of the simulated behavior (as the times the simulated actions take and failures are randomly decided based on distributions), one upgrade campaign specification simulation might take a better execution path than the other. Thus, making the simulation results unreliable for comparison. To overcome this challenge we have chosen to inject controllable scenarios to make all the upgrade campaign specifications that we want to compare take the same execution path (execution time wise and failure wise). However, using any scenario for the comparison might not be relevant for the evaluation of upgrade campaign specifications. That is why we propose the use of best case and worst case scenarios as they will enable:

- ✓ Fair comparison of upgrade campaign specifications.

- ✓ The estimation of the best and worst execution a time an upgrade campaign might take and the best and the worst service outage it might induce.

In the following subsection we will describe these two controllable scenarios as we defined them.

4.3. Controllable scenarios

The best and worst case scenarios are used to control the execution path that the simulation will take. They are defined in a way to take the simulation to the edge cases, thus taking into consideration the following aspects of the simulated runtime environment:

- ✓ Software operations: The execution of a software operation may succeed or fail and does not always take the same duration. The upgrade context previously described captures this fact by associating with every software operation a failure rate and a bounded duration attribute specifying a lower bound and an upper bound. However, the upgrade campaign specification may constrain these operations by a default timeout, which effectively replaces the upper bound value as SMF engages the appropriate upgrade repair mechanism if the timer expires.
- ✓ Upgrade repair mechanisms: Upgrade actions are subject to failure, and depending on the failure stage SMF can engage different chains of actions involving both software and component management operations (when the failed step goes to the undoing states, and should undo all the previously performed actions before the failure). Moreover, SMF implementation takes also into consideration the specified max retries value of each step before reporting the failure. For different stages of failures and different failures' count during the execution of every step in the campaign we can have different scenarios that induce different levels of service disruption and take different durations. We use edge combinations of the two factors in the definition of the best and worst case.
- ✓ AMF configuration object behavior: As mentioned previously, administrative operations are decomposed into AMF component management operations. Since each applicable management operations may succeed or fail taking different amounts of time the upgrade context includes a failure rate and a bounded time attribute for each of these management operations. In addition, a given AMF configuration also constrains all these operations by timeouts, meaning that if an operation is taking more time than the configured timeout, AMF assumes the operation has failed and engages in a recovery and repair actions, which map into AMF management operations sometimes at component in other cases at node level. Some AMF component management operations such as the instantiation, can be reattempted several times before AMF reports them as failed. All these aspects are used to define the best execution and worst execution of an operation on a given component.

In addition to the advantage of evaluating an upgrade campaign specification in edge cases, the use of controllable scenarios also offers the possibility of performing an evaluation without the use of the simulation. The administrative operations specified in AMF, and used in the upgrade process are: lock, lock-instantiation, unlock-instantiation, unlock, and shutdown. These operations can be called on targets of different sizes (Node, SU hosting a container component, or just a simple SU). The time the administrative operation can take varies depending on the type of the target, its size, and other dependencies factors (collocation within the target). So the time estimation for administrative operations is strictly based on what is provided in the configuration and the ETFs, and takes into consideration the following aspects:

- ✓ The decomposition of the administrative operations into AMF component management operations.
- ✓ The timeouts of these operations as provided in the AMF configuration, and their bounded times as provided in the ETFs.
- ✓ The ordering of these actions as imposed by the instantiation level and CSI dependencies within a SU.
- ✓ Lifecycle dependencies between SUs of the same Node (container-contained).
- ✓ Components category (pre-instantiable Vs non-pre-instantiable).
- ✓ How AMF reacts to failure of administrative and component management operations: termination escalates to cleanup and instantiation is retried with and without delay.
- ✓ The time a SMF implementation can wait for a callback is usually specified in the upgrade campaign specification and there is no need to estimate it.

Based on this, in the following subsections we will define the best case and worst case scenarios as well as the expressions that one can use to evaluate an upgrade campaign specifications in these scenario without the use of the simulation. The details of these formula can be found in appendix 1.

4.3.1. Best case

The best case is defined as follow:

- ✓ Time wise: in this perspective we take into consideration the execution time of every action.
  - o Every action takes as much time as the lower bound of the related ETF attribute, and succeeds the first time it is called (instantiation, termination, lock, unlock, installation, removal, node restart).
  - o Every upgrade step succeeds at the first time of its execution.

- ✓ Outage wise: in this perspective we take into consideration the interference between steps of procedures of the same execution level.
    - For fully ordered execution there are no consideration as we execute one procedure at a time, and thereby one step at a time.
    - For a partially ordered execution, as the procedures of the same execution level are executed in parallel, interference between their respective steps might take place. So, in the best case, the steps of procedures of the same execution level are arranged in a way that impacts the system the least. In other words, the steps that are executed in parallel are the ones that induce the minimum outage when executed concurrently.

4.3.2. Worst case

The worst case is defined as follows:

- ✓ Time wise: in this perspective we take into consideration the execution time of every action.
    - Every upgrade step succeeds only on the last permitted attempt of its execution (max retry).
    - The failure of the upgrade step execution takes place at specific stages based on the configuration and the entities in the activation and deactivation unit of the step. So different actions might cause this failure, the choice is made based on the damage and recovery time for each stage. These stages are ranked as shown in Table 4-2.
    - All AMF component management operations take as much time as their configured timeout and succeed only the last time they are allowed to be executed.
    - Every recovery from a failure of an AMF component management operation should escalate to node level.
- ✓ Outage wise: similarly to the best case, as there is no interference between upgrade procedures in the fully ordered execution, we only consider the partially ordered execution.
    - In a partially ordered execution, the upgrade steps of the upgrade procedures of the same execution level are arranged in a way to induce the maximum service outage.

Table 4-2: ranked list of upgrade step points of failure

rank	Condition	Stage of failure
1	The activation unit contains a non-pre-instantiable component that will take the assignment after the unlocking, and saAmfNodeFailfastOnInstantiationFailure is set for the node hosting it	Unlock Activation Unit
2	The activation unit contains a pre-instantiable component, and saAmfNodeFailfastOnInstantiationFailure is set for the node hosting it	Instantiation of Activation Unit
3	The deactivation unit contains a pre-instantiable component, and saAmfNodeFailfastOnTerminationFailure is set for the node hosting it	Termination of Deactivation Unit
4	The deactivation unit contains a non-pre-instantiable component that has the assignment, and saAmfNodeFailfastOnTerminationFailure is set for the node hosting it	Lock Deactivation Unit
5	Default	Online uninstallation of old software

4.4. Selection/elimination of upgrade campaign specifications

Given a set of upgrade campaign specifications that can take a system from its current configuration to the same target configuration, we can use our simulation approach to evaluate them and decide which ones are applicable considering some targeted acceptable outage and maintenance window. The applicability of the scenarios discussed in the previous section depends on the capability of the SMF engine. Accordingly, if the SMF engine is not capable of parallel execution the scenarios of the partially ordered execution do not apply. All SMF engines must be capable of fully ordered upgrade campaign execution. Since our goal is to meet some acceptable outage and maintenance window we evaluate the upgrade campaigns from the perspective whether these goals can be achieved

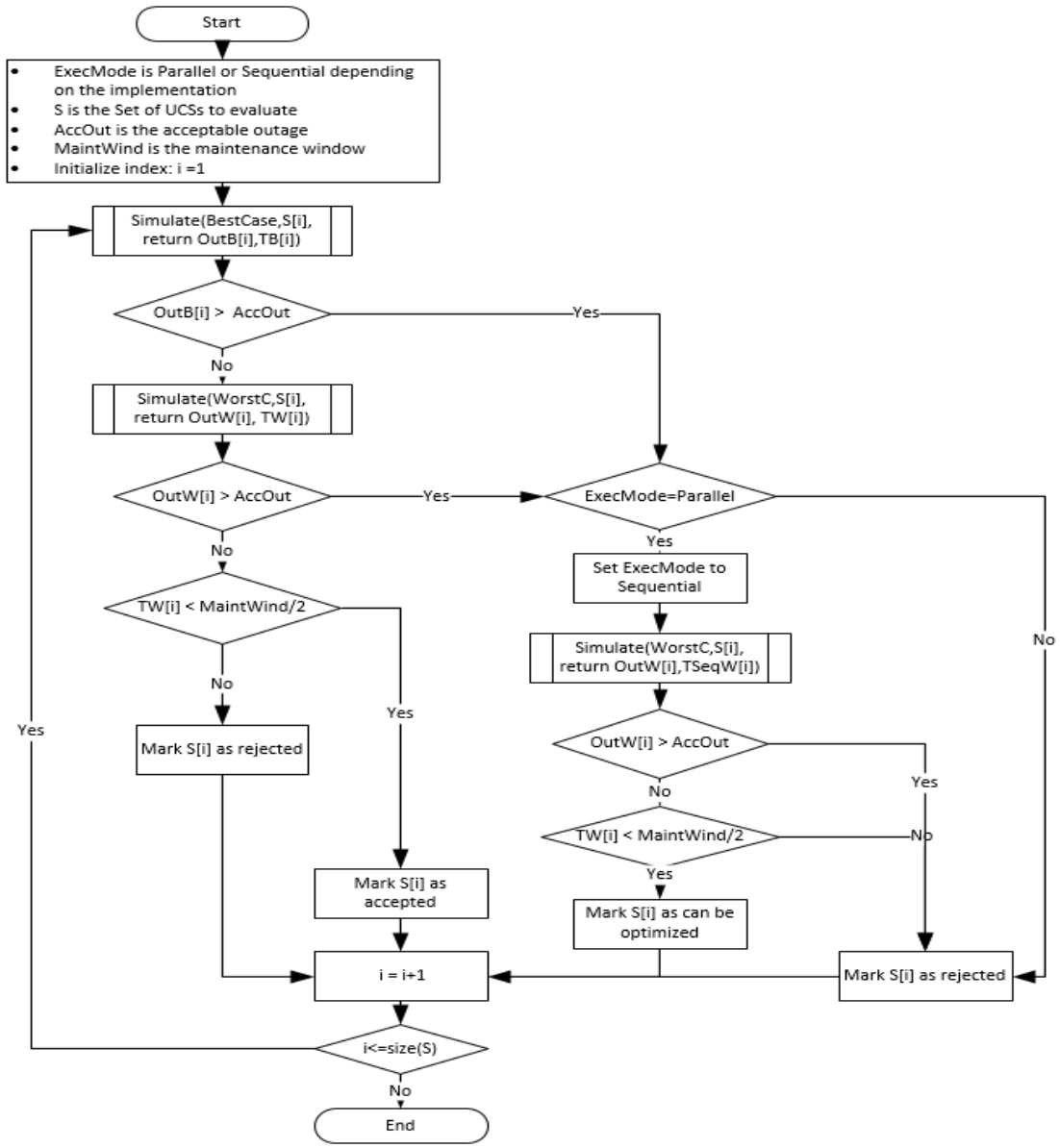


Figure 4-4: upgrade campaign specification selection/elimination process

rather than selecting “the best upgrade campaign”. This is because it is not straightforward how service outage trades for execution time. Our evaluation (summarized in Figure 4-4) goes along the following lines:

- ✓ First all the upgrade campaign specifications are evaluated for the execution mode applicable to the SMF engine and both the best and the worst case scenarios are evaluated for their execution time and induced outage.
- ✓ If the execution mode of the SMF engine is sequential all the upgrade campaign specifications that induce an unacceptable outage for either the best or the worst case scenario can be safely eliminated as there is no guarantee they can meet the outage constraint.
- ✓ In the case of parallel execution mode, the upgrade campaign specifications violating the outage constraint are further evaluated for the sequential execution as it typically induces less outage. Those upgrade campaign specifications which still result in an unacceptable outage for either the best or the worst case are eliminated. The remaining upgrade campaign specifications are marked for potential serialization.
- ✓ Next the execution times of all the pre-selected upgrade campaign specifications are evaluated with respect to the maintenance window. The main consideration is that we would like to complete the upgrade campaign within half of the available maintenance window. This allows for a graceful rollback of the system to its original configuration should anything go wrong unexpectedly during campaign execution. Accordingly, we eliminate all the upgrade campaigns that result in an execution time greater than the half of the targeted maintenance window. This criterion may be relaxed if a partial or full restoration of the system from a backup is an acceptable recovery and therefore can be used to shorten the rollback time.

The selected upgrade campaign specifications are acceptable albeit some with the need for serialization. They can be compared and further analyzed from the perspective of their induced outages and execution times to pick the one that is the most suitable for the given system and constraints. A system administrator may choose the campaign that takes the least time in order to make better use of the maintenance window, while another one may choose the one that takes the longer time because, for example, it specifies more upgrade steps retries and thereby is more reliable. The choice can also be based on the probabilities associated with the best case and the worst case scenarios and select the one which has the highest probability for the best case scenario. This is also where the tradeoff execution time for service outage and vice versa becomes important. Our selection/elimination process ends with a set of applicable upgrade campaign specifications.

4.5. Summary

Induced outage and execution time are two important quality factors for an upgrade campaign. Estimating these factors can help: analyze the impact of the upgrade campaign on the system, decide if the upgrade campaign is applicable, and compare various valid ways

to upgrade the system. Manually performing this estimation is a tedious and error prone task because of the complexity of the systems. The previous work in [7] proposed a simulation based approach for upgrade campaign evaluation. This approach uses DEVS [31] formalism to enable the simulation. The simulation is run using the DEVS-Suite simulator [29] which takes a DEVS model as input. This DEVS model is expressed as a Java class that is automatically generated from an upgrade campaign specification, an AMF configuration, and an upgrade context. The work in [7] described the upgrade context as a source from which one can retrieve the failure models of system components but it did not propose a solution for using it in the simulation. In this thesis we extended the work in [7] by introducing an upgrade context as an extension of the vendor provided ETFs and enabling the use of the failure models in the simulation. We have also exposed the limitations of the random simulation. In fact, using a random simulation to compare upgrade campaign specifications can be misleading as one upgrade campaign specification simulation can follow a better execution path than the other. To overcome this limitation we propose the use of best case and worst case scenarios simulation to guide the simulation. Guiding the simulation in these scenarios will not only help make sure that all the simulations follow the same execution path, it will also help evaluate the impact the various campaigns can have on the system in edge cases. We also show how the use of these scenarios can enable for an upgrade campaign evaluation that is not based on the simulation. Finally, we proposed a method that one can follow to eliminate/select applicable upgrade campaign specifications. This method uses the results of best case and worst case scenarios' evaluations. It checks whether the estimated outage is within the allowed outage as well as whether the execution time can allow for a graceful rollback within the maintenance window. At the end, each upgrade campaign specification can be marked as accepted, rejected, or can be optimized.

In this thesis we proposed the best case and the worst case scenarios to evaluate upgrade campaign specifications. These two scenarios can give an insight of what to expect from an upgrade campaign, but one cannot disregard the fact that these scenarios are very unlikely to happen (very small probability). An approach that might target accuracy would suggest the use of more scenarios and more simulations for evaluation. One potential extension of this work can be the design of a method that can give a minimum set of scenarios (and their definitions) that can be expressive enough (with a tolerance error margin) of the expected result of running a given upgrade campaign specification on a given configuration. Accordingly, one can also extend the upgrade campaign specification selection/elimination process to narrow down the number of selected upgrade campaign specifications based on further criteria.

Chapter 5 - Prototypes

This chapter describes the prototypes implemented in this thesis. It covers the challenges faced during the implementation (if any) as well as the tools used in the implementations. We will first start by describing the upgrade campaign specification generation prototype. We will introduce the tools that were used, and illustrate using a running example. Similarly we will describe the simulation based evaluation prototype, and give an illustrative example for this prototype as well.

5.1. Upgrade campaign specification generation prototype

The upgrade campaign specification generation consists of many activities. Each activity is implemented as at least one transformation using the appropriate language from the Epsilon family of languages:

- ✓ The change model creation consists of a comparison of two models (source configuration and target configuration) that is why we chose to implement it using Epsilon Comparison Language ECL.
- ✓ The actions creation takes an input model and generates a totally different output model. The most appropriate language of the Epsilon family was the Epsilon Transformation Language ETL. The same applied to the dependencies extraction activity.
- ✓ The actions matching activity is a refinement of the input model, and running it once on the input model might not get the desired result. In this case, we benefited from Epsilon Pattern Language EPL feature to rerun the transformation as long as there is a specified pattern that can be detected.
- ✓ Similarly for the optimization, the application of the heuristics and ordering was implemented using EPL.
- ✓ The generation of the XML file from the EMF model was implemented using the Epsilon Generation Language EGL.
- ✓ The workflow orchestration was done using ant [56].

The change model creation was a reimplementation of the work in [6] using ECL. The

```
rule MatchSIs
  match s : Source!t_object
  with v : Target!t_object {
  guard : ((s.a_class == "SaAmfSI") and (v.a_class == "SaAmfSI"))
  compare : (s.c_dn.text.first().substring(0,s.c_dn.text.first().indexOf(",saf")) == v.c_dn.text.first().substring(0,s.c_dn.text.first().indexOf(",saf")))
  do{
```

Figure 5-1: Matching SIs

work in [6] was based on some assumptions and elaborated a set of rules to overcome the

challenge of the difference of namespaces between two configurations representing the same system. Some of these rules were straightforward based on comparison of attributes (such as the RDNs). Others were more complex and were based on the previously established matches. Figure 5-1 gives an example of rule that is straightforward and based only on RDNs comparison. Other rules, such as the one in Figure 5-2, are a bit tricky to implement and rely on the already established matches.

```
rule MatchSGs
    match s : Source!t_object
    with t : Target!t_object{
    guard : s.a_class=="SaAmfSG" and t.a_class=="SaAmfSG"
    compare {
                return s.matchSG(t) and
                    Source!t_object.all.select(o|o.c_dn.text.first()==s.c_dn.text.first().getParent()).first().matches(
                        Target!t_object.all.select(o|o.c_dn.text.first()==t.c_dn.text.first().getParent()).first())
                and
                    Source!t_object.all.select(
                    o|o.a_class=="SaAmfSI" and o.getAttr("saAmfSIProtectedbySG")==s.c_dn.text.first()).exists(
                    ssi|Target!t_object.all.select(o|o.a_class=="SaAmfSI"
                and o.getAttr("saAmfSIProtectedbySG")==t.c_dn.text.first()).exists(tsi|ssi.matches(tsi)));
    }
    do{
```

Figure 5-2: Matching SGs

Now we will try to show the different output models our chain of transformations generates in the process. We will start with a configuration that is running an in house developed component called the Http component. The SG in this configuration has two SUs,

```
<?xml version="1.0" encoding="ASCII"?>
<ChangeMM:ChangeSet xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xmlns:ChangeMM="http://ChangeMM.ecore">
  <Changes xsi:type="ChangeMM:AddToImm">
    <target xsi:type="ChangeMM:UCGSU"
    DN="safSu=SU_2_NWayActiveHTTP,safSg=SGNWayActiveHTTP,safApp=AppNWayActiveHTTP">
      <attrs attrName="saAmfSUType" attrValue="safVersion=4.0.0,safSuType=SUBaseTypeForNWayActiveHTTP"/>
      <attrs attrName="saAmfSUHostNodeOrNodeGroup" attrValue="safAmfNode=SC-2,safAmfCluster=myAmfCluster"/>
      <attrs attrName="saAmfSURank" attrValue="1"/>
    </target>
  </Changes>
  <Changes xsi:type="ChangeMM:AddToImm">
    <target xsi:type="ChangeMM:UCGComp"
    DN="safComp=comp_1_NWayActiveHTTP,safSu=SU_2_NWayActiveHTTP,safSg=SGNWayActiveHTTP,safApp=AppNWayActiveHTTP">
      <attrs attrName="saAmfCompType" attrValue="safVersion=4.0.0,safCompType=CompBaseTypeForNWayActiveHTTP"/>
      <attrs attrName="saAmfCompInstantiationLevel" attrValue="1"/>
      <attrs attrName="saAmfCompInstantiateCmdArgv" attrValue="8080"/>
      <attrs attrName="saAmfCompNumMaxInstantiateWithDelay" attrValue="2"/>
      <attrs attrName="saAmfCompNumMaxInstantiateWithoutDelay" attrValue="2"/>
      <attrs attrName="saAmfCompDelayBetweenInstantiateAttempts" attrValue="100000000"/>
      <attrs attrName="saAmfCompTerminateTimeout" attrValue="2000000000"/>
      <attrs attrName="saAmfCompInstantiateTimeout" attrValue="2000000000"/>
      <attrs attrName="saAmfCompCleanupTimeout" attrValue="2000000000"/>
    </target>
  </Changes>
</ChangeMM:ChangeSet>
```

Figure 5-3: Change Model

each one of them has one Http component. We manually added a third SU (with its component) to a copy of this configuration within this SG. Now, the output of the change

model creation activity is shown in Figure 5-3. As you can notice, the transformation detected the added SU and component, and it instantiated appropriate model elements (two addToImm instances, one for the SU and the other for the component) conforming to the Change meta-model. We then make this change model go through the actions creation transformation. As shown in Figure 5-4, this transformation created two upgrade procedures with the actions representing the addition to IMM in the body of the procedure

```
<?xml version="1.0" encoding="ASCII"?>
<UCS_MM:UCG_UpgradeCampaign xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:UCS_MM="http://UCS_MM.ecore" safCamp="campaign">
  <body xsi:type="UCS_MM:UCG_UpgradeProcedure"
        safProc="addingsafSu=SU_2_NWayActiveHTTP,safSg=SGNWayActiveHTTP,safApp=AppNWayActiveHTTP"
        UpgradeMethod="singleStepUpgrade" execLev="1">
    <body xsi:type="UCS_MM:UCG_ImmAdd"
            aDN="safSu=SU_2_NWayActiveHTTP,safSg=SGNWayActiveHTTP,safApp=AppNWayActiveHTTP">
      <objectAttrs name="saAmfSUType" value="safVersion=4.0.0,safSuType=SUBaseTypeForNWayActiveHTTP"/>
      <objectAttrs name="saAmfSUHostNodeOrNodeGroup" value="safAmfNode=SC-2,safAmfCluster=myAmfCluster"/>
      <objectAttrs name="saAmfSURank" value="1"/>
    </body>
  </body>
  <body xsi:type="UCS_MM:UCG_UpgradeProcedure"
        safProc="addingsafComp=comp_1_NWayActiveHTTP,safSu=SU_2_NWayActiveHTTP,safSg=SGNWayActiveHTTP,safApp=AppNWayActiveHTTP"
        UpgradeMethod="singleStepUpgrade" execLev="1">
    <body xsi:type="UCS_MM:UCG_ImmAdd"
            aDN="safComp=comp_1_NWayActiveHTTP,safSu=SU_2_NWayActiveHTTP,safSg=SGNWayActiveHTTP,safApp=AppNWayActiveHTTP">
      <objectAttrs name="saAmfCompType" value="safVersion=4.0.0,safCompType=CompBaseTypeForNWayActiveHTTP"/>
      <objectAttrs name="saAmfCompInstantiationLevel" value="1"/>
      <objectAttrs name="saAmfCompInstantiateCmdArgv" value="8080"/>
      <objectAttrs name="saAmfCompNumMaxInstantiateWithDelay" value="2"/>
      <objectAttrs name="saAmfCompNumMaxInstantiateWithoutDelay" value="2"/>
      <objectAttrs name="saAmfCompDelayBetweenInstantiateAttempts" value="100000000"/>
      <objectAttrs name="saAmfCompTerminateTimeout" value="2000000000"/>
      <objectAttrs name="saAmfCompInstantiateTimeout" value="2000000000"/>
      <objectAttrs name="saAmfCompCleanupTimeout" value="2000000000"/>
    </body>
  </body>
</UCS_MM:UCG_UpgradeCampaign>
```

Figure 5-4: The first upgrade campaign specification model

conforming to the rules we described in section 3.4.2. We make this output go through the first refinement, as it shows in Figure 5-5 the second rule of the matching was applied (matching children to parents). There was no software installation operation because we

```
<?xml version="1.0" encoding="ASCII"?>
<UCS_MM:UCG_UpgradeCampaign xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:UCS_MM="http://UCS_MM.ecore" safCamp="campaign">
  <body xsi:type="UCS_MM:UCG_UpgradeProcedure"
        safProc="addingsafSu=SU_2_NWayActiveHTTP,safSg=SGNWayActiveHTTP,safApp=AppNWayActiveHTTP"
        UpgradeMethod="singleStepUpgrade" execLev="1">
    <body xsi:type="UCS_MM:UCG_ImmAdd"
            aDN="safSu=SU_2_NWayActiveHTTP,safSg=SGNWayActiveHTTP,safApp=AppNWayActiveHTTP">
      <objectAttrs name="saAmfSUType" value="safVersion=4.0.0,safSuType=SUBaseTypeForNWayActiveHTTP"/>
      <objectAttrs name="saAmfSUHostNodeOrNodeGroup" value="safAmfNode=SC-2,safAmfCluster=myAmfCluster"/>
      <objectAttrs name="saAmfSURank" value="1"/>
    </body>
    <body xsi:type="UCS_MM:UCG_ImmAdd"
            aDN="safComp=comp_1_NWayActiveHTTP,safSu=SU_2_NWayActiveHTTP,safSg=SGNWayActiveHTTP,safApp=AppNWayActiveHTTP">
      <objectAttrs name="saAmfCompType" value="safVersion=4.0.0,safCompType=CompBaseTypeForNWayActiveHTTP"/>
      <objectAttrs name="saAmfCompInstantiationLevel" value="1"/>
      <objectAttrs name="saAmfCompInstantiateCmdArgv" value="8080"/>
      <objectAttrs name="saAmfCompNumMaxInstantiateWithDelay" value="2"/>
      <objectAttrs name="saAmfCompNumMaxInstantiateWithoutDelay" value="2"/>
      <objectAttrs name="saAmfCompDelayBetweenInstantiateAttempts" value="100000000"/>
      <objectAttrs name="saAmfCompTerminateTimeout" value="2000000000"/>
      <objectAttrs name="saAmfCompInstantiateTimeout" value="2000000000"/>
      <objectAttrs name="saAmfCompCleanupTimeout" value="2000000000"/>
    </body>
  </body>
</UCS_MM:UCG_UpgradeCampaign>
```

Figure 5-5: The upgrade campaign specification model after the first refinement

are working with the assumption that if an SG is deployed on a node, even if that node does not have configured SU, it should have the software required for that SU installed on it. Because unless the SUs are configured per node, there is no way to know whether AMF will instantiate an SU on that node the first time the SG is instantiated or not. As we have only one upgrade procedure, and no special pattern applies there, the second refinement will not really have any effect. We then generate the xml file using the EGL template as shown in Figure 5-6.

```
<upgradeCampaign safSmfCampaign="safSmfCampaign=campaign">
<upgradeProcedure safSmfProcedure="addingsafSu=SU_2_NWayActiveHTTP,safSg=SGNWayActiveHTTP,safApp=AppNWayActiveHTTP" saSmfExecLeve="1">
    <outageInfo>
        <acceptableServiceOutage>
            <none/>
        </acceptableServiceOutage>
        <procedurePeriod saSmfProcPeriod="0.0"/>
    </outageInfo>
    <upgradeMethod>
        <singleStepUpgrade>
            <upgradeScope>
                <forAddRemove>
                    <activationUnit>
                        <actedOn>
                                <byName objectDN="safSu=SU_2_NWayActiveHTTP,safSg=SGNWayActiveHTTP,safApp=AppNWayActiveHTTP"/>
                        </actedOn>
                        <added ObjectClassname="SaAmfSU" parentObjectDN="safSg=SGNWayActiveHTTP,safApp=AppNWayActiveHTTP">
                            <attribute name="saAmfSUType" type="">
                                <value>safVersion=4.0.0,safSuType=SUBaseTypeForNWayActiveHTTP</value>
                            </attribute>
                            <attribute name="saAmfSUHostNodeOrNodeGroup" type="">
                                <value>safAmfNode=SC-2,safAmfCluster=myAmfCluster</value>
                            </attribute>
                            <attribute name="saAmfSURank" type="">
                                <value>1</value>
                            </attribute>
                        </added>
                        <added ObjectClassname="SaAmfComp" parentObjectDN="safSu=SU_2_NWayActiveHTTP,safSg=SGNWayActiveHTTP,safApp=AppNWayActiveHTTP">
                            <attribute name="saAmfCompType" type="">
                                <value>safVersion=4.0.0,safCompType=CompBaseTypeForNWayActiveHTTP</value>
                            </attribute>
                            <attribute name="saAmfCompInstantiationLevel" type="">
                                <value>1</value>
                            </attribute>
                            <attribute name="saAmfCompInstantiateCmdArgv" type="">
                                <value>8080</value>
                            </attribute>
                            <attribute name="saAmfCompNumMaxInstantiateWithDelay" type="">
                                <value>2</value>
```

Figure 5-6: The xml file generated from the last upgrade campaign specification model

5.2. Upgrade campaign evaluation prototype

Implementing a model transformation requires a prior knowledge of the number of models/files that will be given to the transformation as input. A model transformation, for instance, cannot take three input models in an execution if it was programed to take only two. Moreover a transformation cannot be implemented for an unknown number of inputs as each one of the inputs is known through an alias which is hard coded in the transformation. This was one of the main challenges of the implementation of this prototype. In fact, we only know the number of ETF files that will be given as input to the transformation at the time we run the transformation. So far, no model transformation language or model management environment has a solution for the case when the number of input models is not known at the time of the implementation of the transformation. So, we

ended up implementing a solution that makes use of Epsilon's capability to embed Java code in the body of a model transformation.

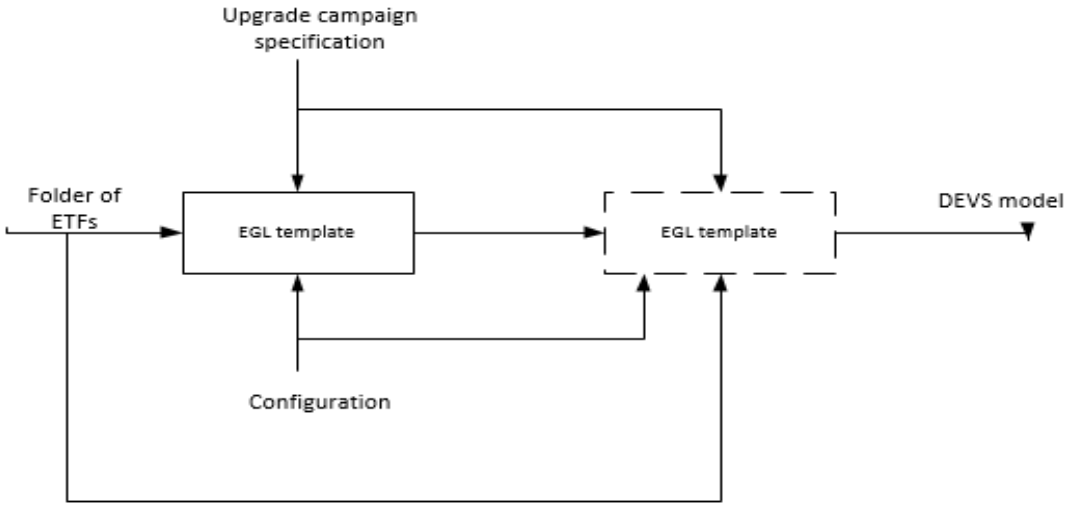


Figure 5-7: Enabling unknown number of input models

As Figure 5-7 shows, we created an EGL template that takes the upgrade campaign specification, the configuration and the folder containing the ETFs as arguments. This template generates another EGL template that is implemented specifically for a number of inputs equals to the number of ETF files in the folder plus two (a configuration and an upgrade campaign specification). Then we give the ETFs, the upgrade campaign specification, and the configuration as input to this generated EGL template in order to generate UCS.java. In fact, another file (the ant [56] build workflow orchestration file) is also generated using this same template (the first template), we omitted it from the Figure to keep the flow simple to follow. It is at the level of this first EGL template execution when the upgrade context is gathered (attributes in the extension of SAF model). The GUI shown in Figure 5-8, is launched be the first EGL template and used to collect the different probabilities and required durations. The user can still use just the default values (stored in a properties file) by checking "Use default for all" checkbox, or a "Use default" checkbox for a specific node, component type or software bundle. The values inputted through this GUI will be stored in a properties file right after the users clicks "Ok".

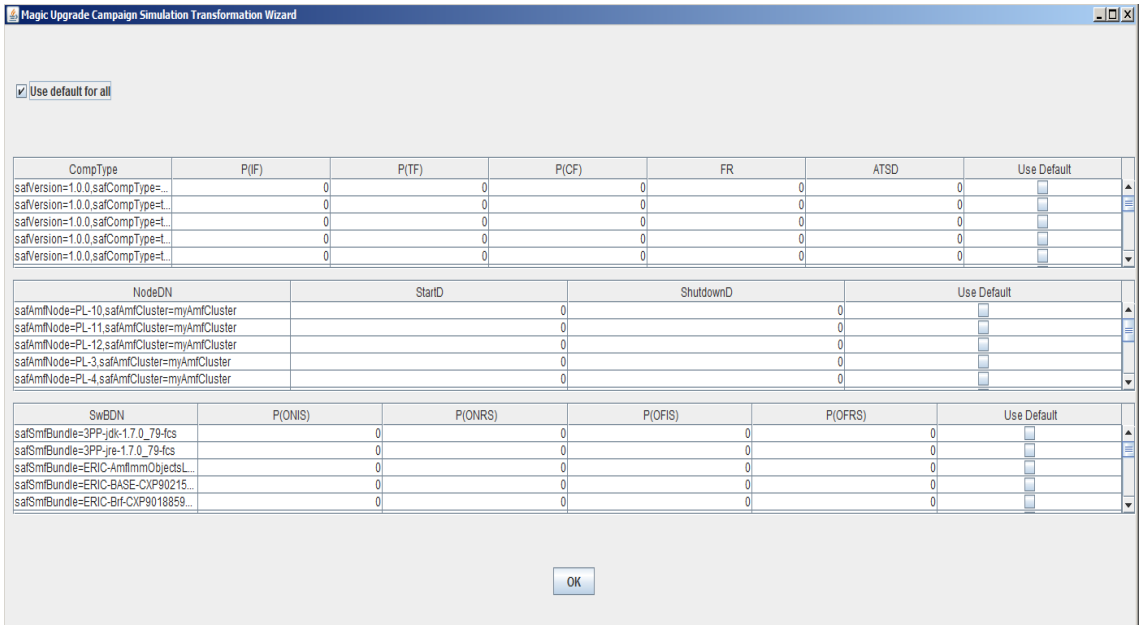


Figure 5-8: The GUI used to input the upgrade context

The user can then run the generated EGL template using the generated workflow orchestration ant file to generate the DEVS model (UCS.java) which he can then load into the simulator. Once the model loaded, if the user chooses to use the graphical component of the simulator, this latter will render the DEVS model associated with the configuration and the upgrade campaign specification as shown in Figure 5-9.

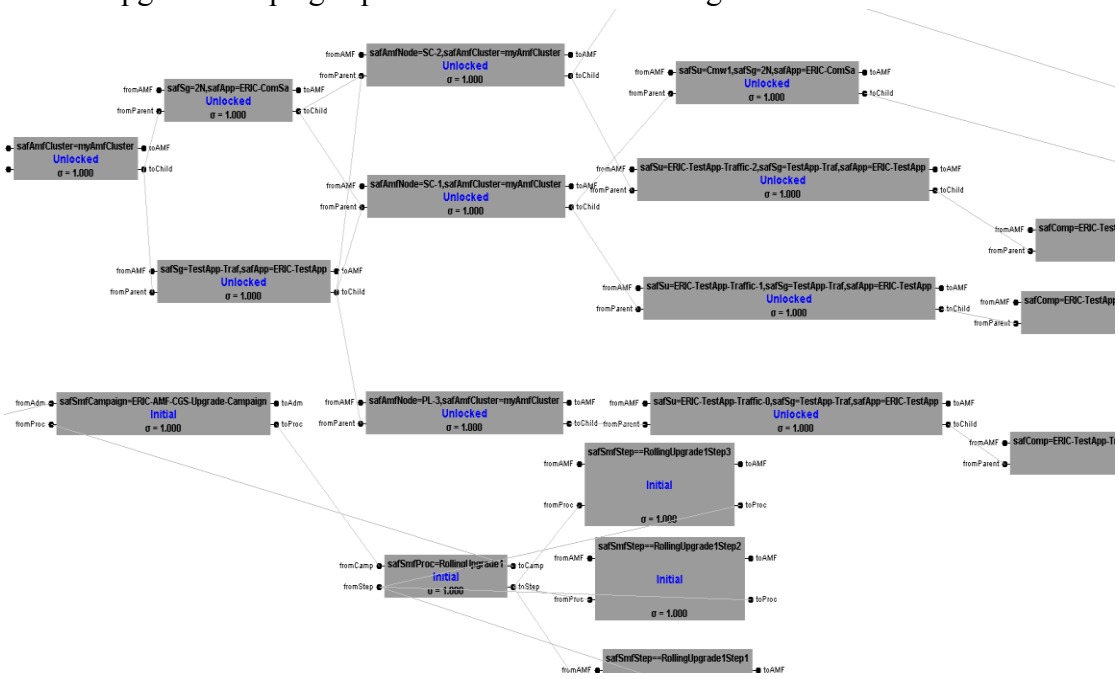


Figure 5-9: Upgrade campaign specification and the configuration rendered in the graphical component of the DEVS-Suite simulator

Initially, all the SIs are fully assigned, in the case of this simulation, we are tracking only one SI as shown in Figure 5-10. Similarly the upgrade campaign is in its initial state

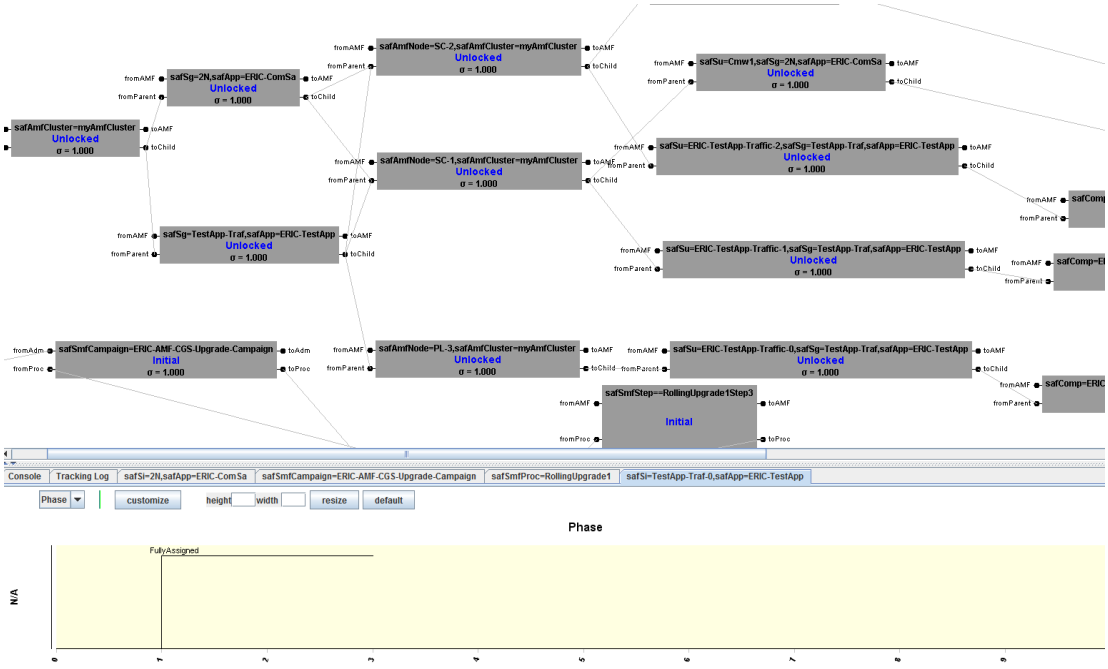


Figure 5-10: SI initially fully assigned

before triggering its execution as shown in Figure 5-11. Once the execution is triggered,

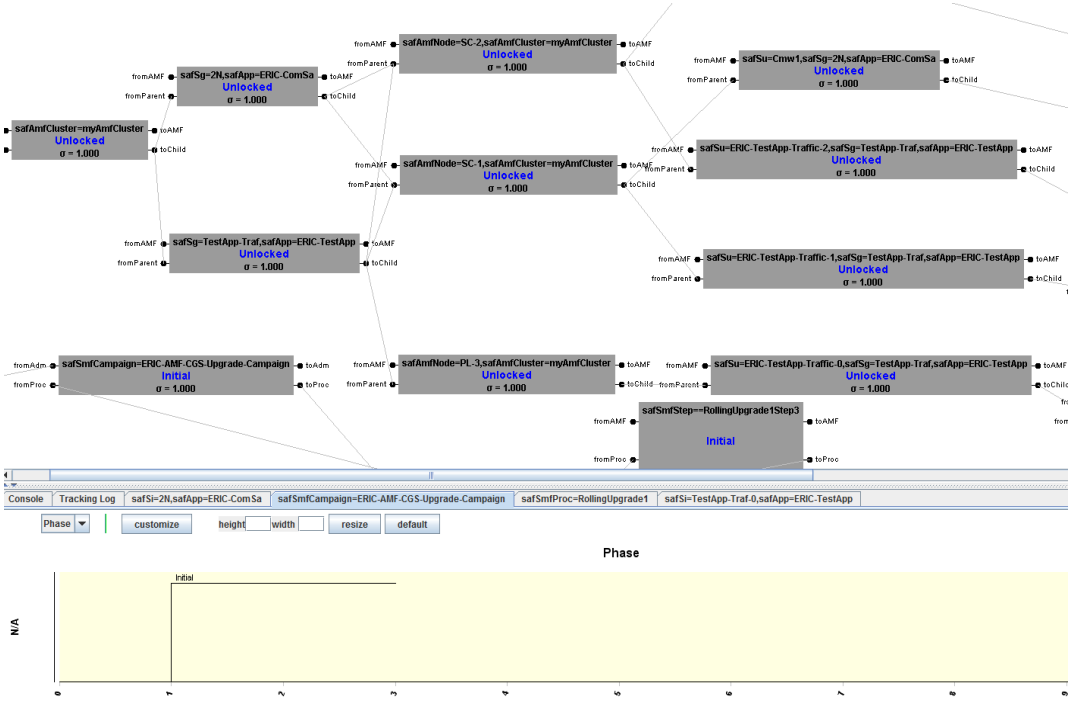


Figure 5-11: Upgrade campaign initially in the initial state

the campaign transitions to the Executing state, and triggers the execution of its first upgrade procedure as shown in Figures 5-12 and 5-13 respectively. Once the procedure

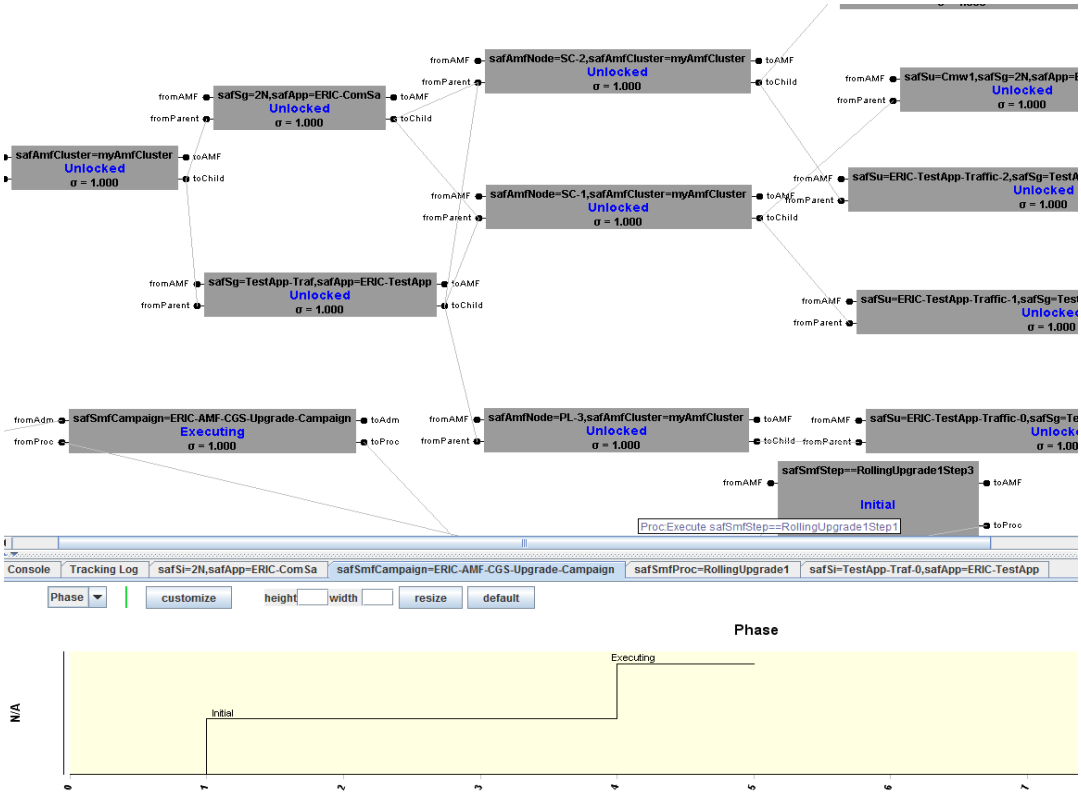


Figure 5-12: Upgrade campaign transitions to Executing state

starts its steps, we start noticing the impact on the assignment state of the SI.

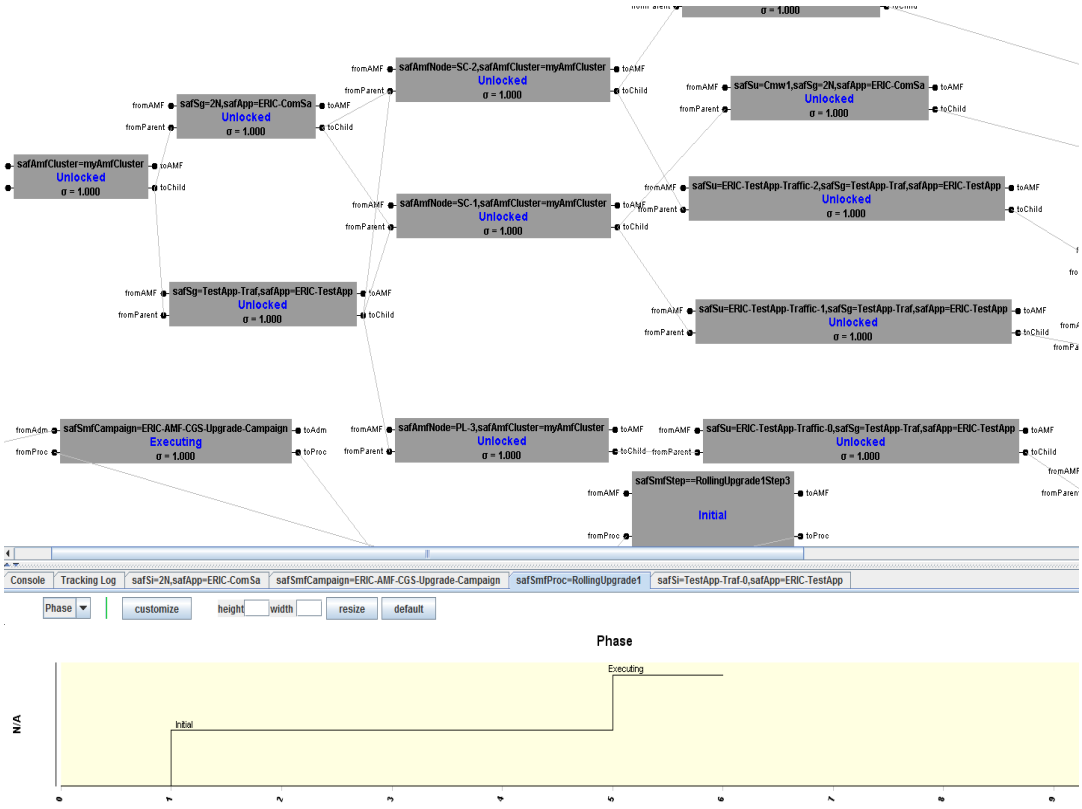


Figure 5-13: Upgrade procedure transitions to Executing state one time unit after the upgrade campaign

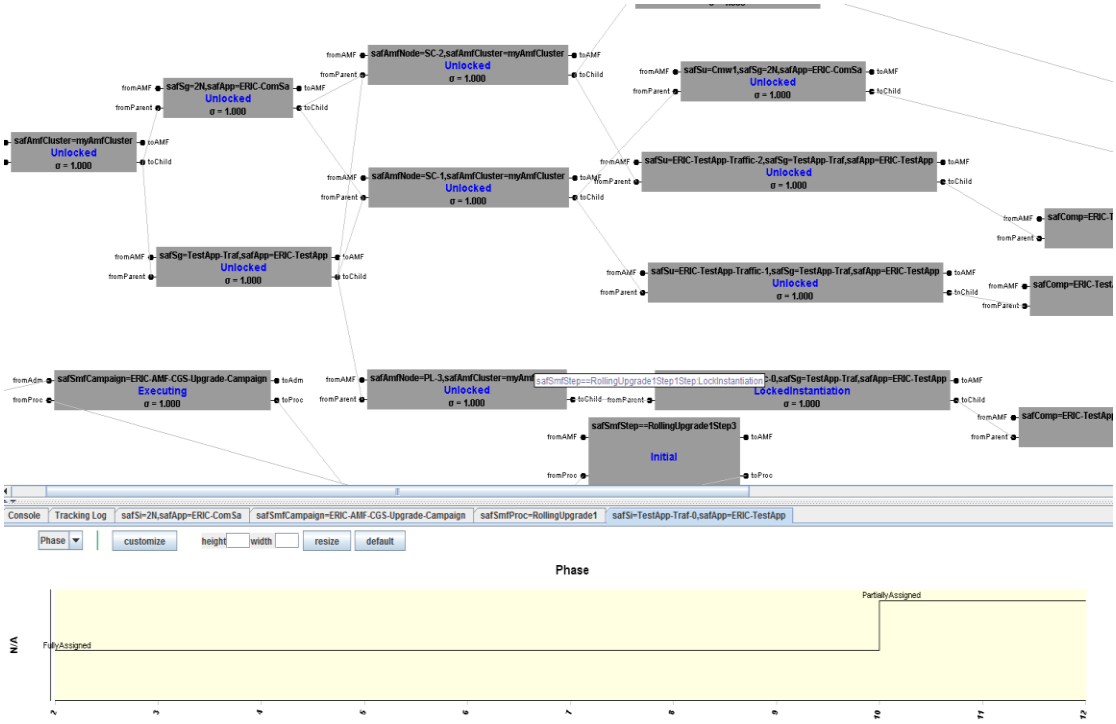


Figure 5-14: SI going to the partially assigned state

Figures 5-14 and 5-15 show respectively the transitions done by the SI at the beginning of the procedure as well as towards the end of that same procedure. At the beginning of

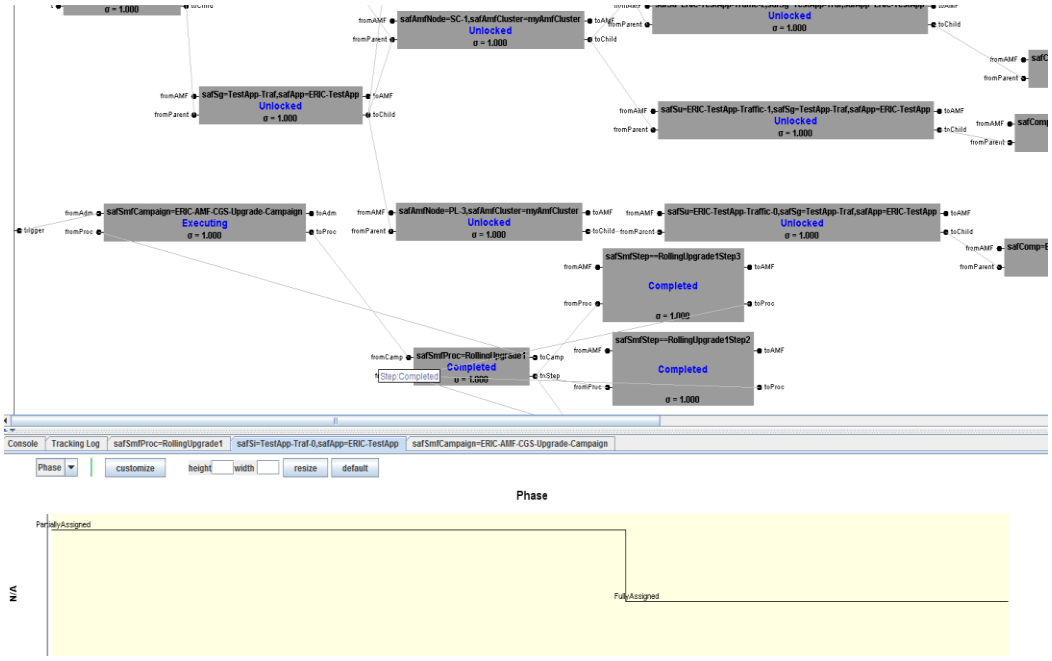
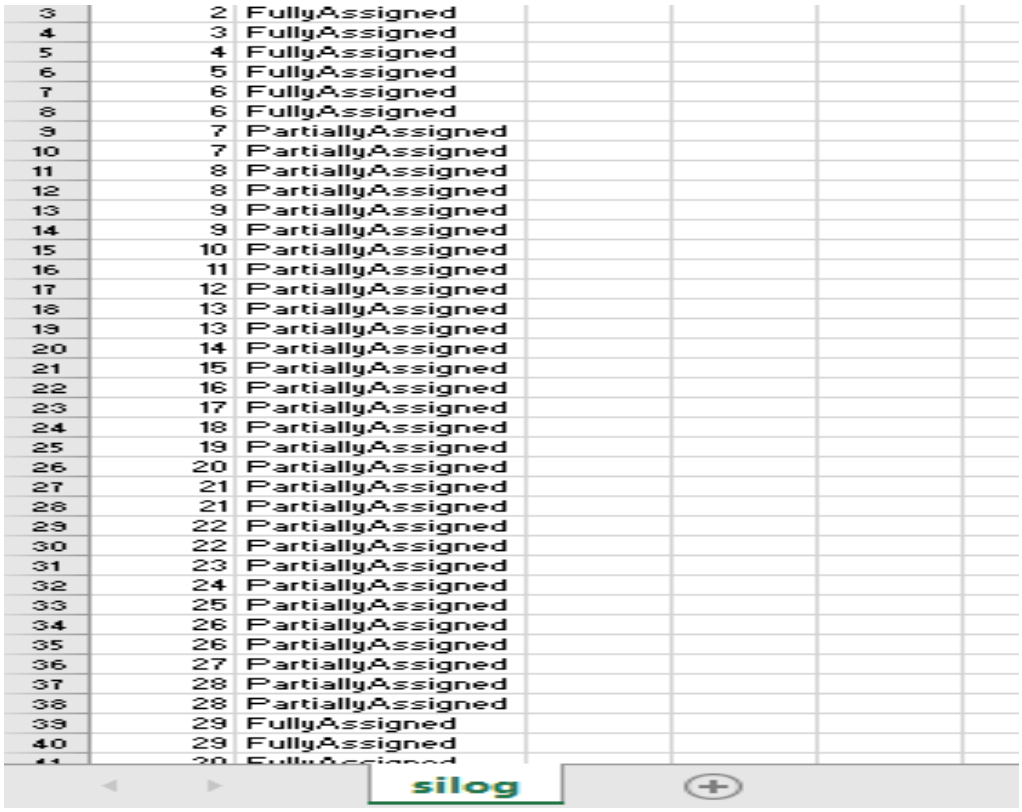


Figure 5-15: SI going back fully assigned

the procedure, when the first step locks the DU the SI goes to the partially assigned state.

Similarly, when the last step of the upgrade procedure unlocks the AU the SI goes back to the fully assigned state. At the end of the simulation one can export the tracking logs as excel files, which look exactly as illustrated in Figure 5-16.



3	2	FullyAssigned
4	3	FullyAssigned
5	4	FullyAssigned
6	5	FullyAssigned
7	6	FullyAssigned
8	6	FullyAssigned
9	7	PartiallyAssigned
10	7	PartiallyAssigned
11	8	PartiallyAssigned
12	8	PartiallyAssigned
13	9	PartiallyAssigned
14	9	PartiallyAssigned
15	10	PartiallyAssigned
16	11	PartiallyAssigned
17	12	PartiallyAssigned
18	13	PartiallyAssigned
19	13	PartiallyAssigned
20	14	PartiallyAssigned
21	15	PartiallyAssigned
22	16	PartiallyAssigned
23	17	PartiallyAssigned
24	18	PartiallyAssigned
25	19	PartiallyAssigned
26	20	PartiallyAssigned
27	21	PartiallyAssigned
28	21	PartiallyAssigned
29	22	PartiallyAssigned
30	22	PartiallyAssigned
31	23	PartiallyAssigned
32	24	PartiallyAssigned
33	25	PartiallyAssigned
34	26	PartiallyAssigned
35	26	PartiallyAssigned
36	27	PartiallyAssigned
37	28	PartiallyAssigned
38	28	PartiallyAssigned
39	29	FullyAssigned
40	29	FullyAssigned

Figure 5-16: Track log for the SI's assignment state

5.3. Summary

In this chapter we have shown the prototypes we implemented for the approaches described in chapter 3 and chapter 4. The use of Epsilon environment was beneficial. On one hand the implementation of the upgrade campaign specification generation required a transformation engine that is able to rerun the transformation as long as some conditions are met. EPL was the only technological solution that we found and which could offer such a feature, as it can be configured to only stop the transformation when no more specified patterns are detected. On the other hand, the ease of querying raw xml files using Epsilon (compared to other transformation languages) enabled us to use input files in the format that complies with the standard without the need to reverse engineer them into EMF models.

Chapter 6 - Conclusion

In this thesis we presented a model driven approach for automated generation of upgrade campaign specifications. We took into considerations the dependencies between the entities composing the system in order to establish a proper ordering for the upgrade procedures and avoid unnecessary outage. We have also proposed a set of heuristics to improve the quality of the generated upgrade campaign specification by reducing the outage and the execution time.

We have also extended a previous work that targeted a simulation based upgrade campaign evaluation for SAF systems, and enriched it with the best case and worst case scenarios to make the simulation results more reliable to use in upgrade campaign comparison and more relevant for evaluation purposes. We have also proposed a way to perform the same evaluation without the use of the simulation. In addition, we proposed a method that makes use of these evaluation results to check the applicability of upgrade campaign specifications within a maintenance window and a given acceptable outage. This method takes as input a set of upgrade campaign specifications and marks them either as rejected, accepted, or need further optimization according to the applicability check criteria.

We discussed the prototypes we implemented for these contributions, and which made use of model management tools (Epsilon), and simulation environments (DEVS-Suite).

This work can be extended in many ways. One can investigate the ways we can manage the tradeoff between time and outage, as at the end of the selection/elimination process, we end up with multiple applicable upgrade campaign specifications but we have to pick one. Another potential track that can be followed on this area is to check the applicability of this approach for upgrade of other kinds of systems (such as clouds), and taking this work to a higher level of abstractions to make it independent of the SAF standard and the SAF compliant domains.

References

1. Service Availability Forum, www.saforum.org
2. M. Toeroe, F. Tam. “Service Availability: Principles and Practice”. Wiley (May 29 2012)
3. Availability Management Framework specification: SAI-AIS-AMF-B.04.01.AL.
4. Software Management Framework specification: SAI-AIS-SMF-A.01.02.AL.
5. Information Model Management specification: SAI-AIS-IMM-A.03.01.AL.
6. A. Mishra, “Automated AMF Configuration Difference Generation”, Master Thesis, Electrical and Computer Engineering, Concordia University, 2011.
7. O. Jebbar, “Modeling and Simulation of Upgrade Campaign Specifications”, Rapport de Projet de Fin d’Etudes, Mathematiques Reseaux et Informatiques, INPT, Rabat, Maroc, 2014.
8. C. Xuejun, “Dependence management for dynamic reconfiguration of component-based distributed systems”, in proceedings 17th IEEE International Conference on Automated Software Engineering, IEEE Computer Society, 2002, pp. 279–284.
9. J. Matevska and W. Hasselbring, “A Scenario-based Approach to Increasing Service Availability at Runtime Reconfiguration of Component-based Systems”, in proceedings of 33rd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO 2007). IEEE, Aug. 2007, pp. 137–148.
10. L. Yu, G. Shoja, H. Muller, and A. Srinivasan, “A framework for live software upgrade”, in proceedings of 13th ISSRE, IEEE Computer Society, 2002, pp. 149–158.
11. A. Wolski and K. Laiho, “Rolling Upgrades for Continuous Services”, in ISAS, LNCS Vol. 3335, M. Malek, M. Reitenspieß, and J. Kaiser (Eds.) Springer, May 2005, pp. 175–189.
12. F. Kon and R. Campbell, “Dependence management in component-based distributed systems”, IEEE Concurrency, Vol. 8, No. 1, pp. 26–36, 2000.
13. B. Morin, G. Nain, O. Barais, and J. M. Jézéquel, “Leveraging Models From Design-time to Runtime. A Live Demo”, in proceedings of 4th workshop of Models@runtime, MODELS 2009.
14. C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, “Safe and automatic live update for operating systems”, ACM SIGARCH Computer Architecture News, Vol. 41, No. 1, May 2013, pp. 279-292.
15. M. Milazzo, G. Pappalardo, E. Tramontana, and G. Ursino, “Handling run-time updates in distributed applications”, in Proceedings of the ACM SAC ’2005, New York, USA, pp. 1375-1380.
16. J. Almeida, M. Wegdam, M. van Sinderen, and L. Nieuwenhuis, “Transparent dynamic reconfiguration for CORBA”, in proceedings 3rd International Symposium on Distributed Objects and Applications. IEEE Computer Society, 2001, pp. 197-207.

17. S. Ajmani, B. Liskov, and L. Shrira, “Scheduling and simulation: how to upgrade distributed systems”, in proceedings of the 9th conference on Hot Topics in Operating Systems. USENIX, May 2003, pp. 8-8.
18. H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew, “POLUS: A POwerful Live Updating System”, in proceedings of ICSE’2007. IEEE, May 2007, pp. 271–281.
19. E. Miedes and F. D. Munoz-Escoi, “A Survey about Dynamic Software Updating”, Instituto Universitario Mixto Tecnologico de Informatica, Universitat Politecnica de Valencia, Technical Report, 2012.
20. D. S. Kolovos, D. Di Ruscio, A. Pierantonio, and R. F. Paige, “Different models for model matching: An analysis of approaches to support model differencing”, in proceedings of the ICSE Workshop on Comparison and Versioning of Software Models (CVSM '2009). IEEE Computer Society, Washington, DC, USA, 2009, pp. 1-6.
21. Epsilon, http://www.eclipse.org/epsilon.
22. D. S. Kolovos, R. F. Paige, F. A. C. Polack, “The Epsilon Transformation Language”, A. Vallecillo, J. Gray, A. Pierantonio (Eds.), ICMT 2008, LNCS 5063, pp. 46–60, 2008.
23. L. M. Rose, R. F. Paige, D. S. Kolovos, Fiona A. C. Polack, “The Epsilon Generation Language”, I. Schieferdecker and A. Hartman (Eds.), ECMDA-FA, LNCS 5095, pp. 1–16, 2008.
24. D. S. Kolovos, R. F. Paige, F. A. C. Polack, “The Epsilon Object Language (EOL)”, in proceedings of the 2nd European conference on Model Driven Architecture: foundations and Applications (ECMDA-FA'06) LNCS Springer, 2006 Vol. 4066, pp. 128-142.
25. M. Francis, D. S. Kolovos, N. Matragkas, R. F. Paige, “Adding Spreadsheets to the MDE Toolkit”, in proceedings of MODELS’2013, LNCS Vol. 8107, pp 35-51.
26. J. Woodcock, J. Davies, “Using Z: Specification, Refinement, and Proof”, Prentice Hall, March 1996.
27. Community Z Tools, http://czt.sourceforge.net
28. ATL, https://eclipse.org/atl/
29. DEVS-Suite, http://acims.asu.edu/software/devs-suite/
30. S. Kim, H. S. Sarjoughian, V. Elamvazhuthi.” DEVS-suite: a simulator supporting visual experimentation design and behavior monitoring”. SpringSim '09 Proceedings of the 2009 Spring Simulation Multiconference.
31. B. P. Zeigler, H. Praehofer, T. G. Kim. ”Theory of Modeling and Simulation”, Second Edition. Academic Press; 2 edition. 2000.
32. Cisco, https://supportforums.cisco.com/discussion/12483781/time-estimate-pcd-upgrade-cucm
33. M. Fornadel ; Fac. Inf. & Inf. Tech., Slovak Univ. of Technol., Bratislava, Slovakia ; P. Lacko ; A. Danko. “Estimation of Legacy Application Upgrade Time using Evolutionary Approach”. Computational Intelligence and Informatics (CINTI), 2013 IEEE 14th International Symposium on, pp. 493-498.

34. X. Qiu ; Nortel Networks, Ottawa, Ont., Canada ; R. Telikepalli ; T. Drwiega ; J. Yan. “Reliability and Availability Assessment of Storage Area Network Extension Solutions”. IEEE Communications Magazine (Volume:43 , Issue: 3), 2005, pp. 80-85.
35. A. Kanso, M. Toeroe, F. Khendek. “Configuration-Based Service Availability Analysis for Middleware Managed Applications”. System Analysis and Modeling: Theory and Practice. Volume 7744 of the series Lecture Notes in Computer Science pp 229-248
36. A. Kanso, M. Toeroe, F. Khendek. “Automating Service Availability Analysis: An Application to a Highly Available Media-Streaming Service”. Software Security and Reliability-Companion (SERE-C), 2013 IEEE 7th International Conference on, pp.94-101
37. A. C. Shaw. “Real-time Systems and Software”. Wiley, 2001
38. P., P., and A. Burns. Guest editorial: ”A review of worst-case execution-time analysis”. Real-Time Systems 18.2 (2000): 115-128.
39. A. Marsan, M., Chiola, G. “On Petri Nets with Deterministic and Exponentially Distributed Firing Times”. In: Rozenberg, G. (ed.) APN 1987. LNCS, vol. 266, pp. 132–145. Springer, Heidelberg (1987)
40. MagicDraw, http://www.nomagic.com/products/magicdraw.html
41. RSA, http://www-03.ibm.com/software/products/en/ratsadesigner
42. StarUML, http://staruml.io/
43. ArgoUML, http://argouml.tigris.org/
44. Papyrus, https://eclipse.org/papyrus/
45. EMF, https://www.eclipse.org/modeling/emf/
46. QVTO, https://projects.eclipse.org/projects/modeling.mmt.qvt-oml
47. Kermeta, http://diverse-project.github.io/k3/
48. OMG, http://www.omg.org/
49. MDA, http://www.omg.org/mda/
50. UML, http://www.omg.org/spec/UML/
51. OCL, http://www.omg.org/spec/OCL/
52. QVT, http://www.omg.org/spec/QVT/
53. XMI, http://www.omg.org/spec/XMI/
54. MOF, http://www.omg.org/mof/
55. A. Davoudian, F. Khendek, M. Toeroe, “Ordering Upgrade Changes for Highly Available Component Based Systems”, in proceedings of IEEE HASE 2014, Florida, January 2014, pp. 259-260.
56. Apache ant, http://ant.apache.org/
57. O. Jebbar, M. Sackmann, F. Khendek, M. Toeroe. “Model Driven Upgrade Campaign Generation for Highly Available Systems”. System Analysis and Modeling 2016, LNCS Vol. 9959, pp. 148-163.