ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

THESIS PRESENTED TO
ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

IN PARTIAL FULFILLEMENT OF THE REQUIREMENTS FOR
A MASTER'S DEGREE WITH THESIS
IN INFORMATION TECHNOLOGY ENGINEERING
M. A. Sc.

BY
Maxime TURENNE

# A NEW DOMAIN SPECIFIC LANGUAGE FOR GENERATING AND VALIDATING MIDDLEWARE CONFIGURATIONS FOR HIGHLY AVAILABLE APPLICATIONS

MONTREAL, October 6th, 2015

## ACKNOWLEDGMENTS

I would like to express my sincere gratitude to:

- My supervisors, Dr. Abdelouahed Gherbi not only for giving me the chance to benefit from his large and substantial knowledge, but also for his human approach from which I have learned a lot.
- Dr. Ali Kanso from who I learned almost everything about high availability for his inspiring example as a scientific and colleague.
- The people from the research department of Ericsson Canada for giving me the chance to study among them.
- My girlfriend, Sophie Lamy, for her endless love and support.
- My family for believing in me from the very beginning of my studies.

# UN NOUVEAU LANGAGE SPÉCIFIQUE AU DOMAIN DES INTERGICIELS DE SYSTÈME HAUTEMENT DISPONIBLE POUR GÉNÉRER ET VALIDER LEUR CONFIGURATION

Maxime TURENNE

## RÉSUMÉ

De nos jours, les services hautement disponibles prennent de plus en plus de place dans notre vie quotidienne et leur demande ne cesse d'augmenter. Cependant, mettre en place des systèmes hautement disponibles demeure une tâche très complexe pour la majorité des intégrateurs de systèmes puisqu'ils doivent les construire à partir de composants peu fiables. De plus, ils sont tenus responsables des impacts d'une éventuelle panne de service du système; dans certains cas, il s'agit de très grosses sommes d'argent alors que dans d'autres cas, la vie d'êtres humains peut être menacée.

La haute disponibilité est un terme attribué à un système ou un service qui est disponible au moins 99.999% du temps. Les standards de l'industrie veulent qu'un pareil système soit basé sur un intergiciel spécialisé. Cet intergiciel doit gérer la redondance des composants du système et doit garantir leurs disponibilités. Cependant, la majorité de ces systèmes sont dépendants de leur plateforme et sont rarement de type source ouverte.

Le forum sur la disponibilité des services (SAForum) définit des standards ouverts pour la construction d'un système hautement disponible utilisant leur intergiciel. Néanmoins, la nature de cette tâche reste très complexe et requiert beaucoup de temps sans pour autant réduire les chances de faire des erreurs. Cette situation est engendrée par la configuration complexe de l'intergiciel. Dans ce mémoire, nous présentons une solution pour automatiser la génération des fichiers de description concernant les types d'applications d'un système hautement disponible. Cette solution, basée sur l'approche précédente d'automatisation de la configuration, permet la génération automatique et de manière complète la configuration de l'intergiciel SAForum. Afin d'atteindre cet objectif, nous proposons une approche basée sur un nouveau langage spécifique au domaine de la haute disponibilité basé sur le diagramme de composant UML (component diagram). Cette approche inclut un ensemble de transformations de modèles et afin de vérifier l'approche, notre prototype est présenté au travers d'une étude de cas.

**Mots-clefs :** Haute disponibilité, Ingénierie dirigée par les modèles, langage de modélisation unifié (UML), Diagramme de composants UML, standards SAForum

**A NEW DOMAIN SPECIFIC LANGUAGE FOR GENERATING AND VALIDATING MIDDLEWARE CONFIGURATIONS FOR HIGHLY AVAILABLE APPLICATIONS**

Maxime TURENNE

## ABSTRACT

Nowadays, highly available services are becoming a part of our everyday life and the demand for them tends to always increase as we saw in the recent years. However, building highly available systems remains a challenging task for most system integrators who are expected to build reliable systems from none-reliable components. They have to deal with the constant pressure of the money lost in case of unplanned outages and in other cases; the consequences of such outages can threaten the life of humans.

Highly available is a characteristic given to a system/service that is available 99.999% of the time. The standard in the industry for achieving such availability with a system is to build it on a specialized middleware. Such middleware will manage the redundancy of the components and will ensure their availability. On the other hand, the majority of those systems are platform-dependent and mostly proprietary.

The service availability forum (SAForum) defines open standards for building and maintaining HA systems using the SAForum middleware. Nevertheless, this task remains tedious and error prone due to the complexity of this middleware configuration. In this thesis, we present a solution to automate the generation of description files for HA systems, which enables the automated generation of the middleware configuration of the previous approach. In order to achieve this objective, we propose an approach based on a new domain specific language extending the UML component diagrams, along with a corresponding set of model transformations. We also present our prototype implementation and a case study as a proof of concept verifying the approach.

**Keywords**: High Availability, Model Driven Software Engineering, Unified Modeling Language, UML component diagrams, SAForum standards

# TABLE OF CONTENTS

LIST OF FIGURES

Page

## LIST OF ABREVIATIONS

| | |
|---|---|
| SAForum | Service Availability Forum |
| SAF | Service Availability Forum |
| OpenSAF | Open Service Availability Framework |
| HA | High Availability |
| API | Application Programming Interface |
| IM | Information Model |
| UML | Unified Modeling Language |
| XML | Extensible Markup Language |
| Comp | Component |
| CT | Component Type |
| SU | Service Unit |
| SUT | Service Unit Type |
| SI | Service Instance |
| SvcT | Service Type |
| CSI | Component Service Instance |
| CST | Component Service Type |
| SG | Service Group |
| SGT | Service Group Type |
| App | Application |
| AppT | Application Type |
| NG | Node Group |

# INTRODUCTION

## 1) High Availability

The Information and Communications Technology (ICT) sector has witnessed a significant change over the last decade, where there is an advent of new service delivery models over broadband access. This means that more revenue generating and critical applications are being delivered using ICT systems. The High Availability (HA) of such systems is an essential non-functional requirement in which the service providers are now very interested.

If the service availability is the percentage of time a service is available to the end user in a certain period of time, the high availability is when this service is available 99.999% of this period (Schmidt, 2006). For example, with a period of one year the downtime (i.e. the service outage) shall not exceed 5.26 minutes (SAForum). However, high availability only means that the service is available for the users at any instant but does not guarantee its continuity (i.e. the service can lose its state when a failure occur). An example of this would be in a highly available mobile phone network, the end user could have his call dropped in case of a failure of the network but, shall be able to recall its interlocutor right away. The availability of a system/service can be expressed with a function of the system reliability and the reparability protected by the redundancy of its component(s).

- **Reliability**: This is the measure of the continuous uptime of a system without failure. This is expressed as the mean time between failures (MTBF).
- **Reparability**: This is the measure of the time needed for a failed system/service to be restored. This is expressed as the mean time to repair (MTTR).
- **Redundancy**: Adding redundancy to a given component augments its reparability because in case of failure of the given component, the replica of the component can be brought into service and replaces the failed one. However, it lowers the MTTR only to the needed time to put in service the given component. In addition, there are several form of redundancy configurations (2N, N+M, N-way, etc.) and readiness

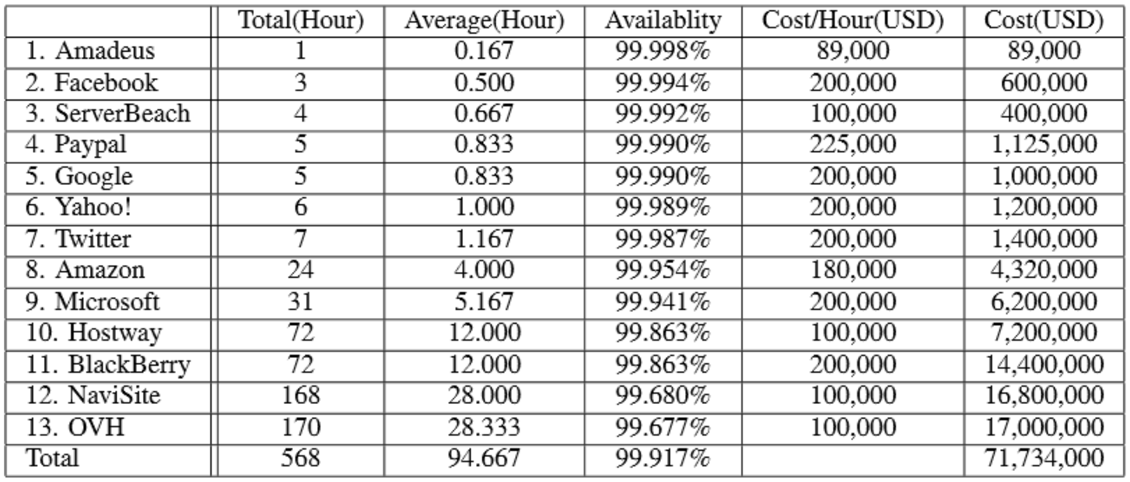
states (cold, warm and hot switchover) in order to appropriately support the level of availability.

By taking those parameters in consideration, the availability of a system/service is expressed by the following equation.

$$Availability = \frac{MTBF}{(MTBF + MTTR)}$$

The industries of today are now acknowledging the importance of the high availability. In 2009, an Information Technology and Intelligence Corp. survey tells us that more than 40% of the companies want at least 99.99% of availability (ITIC). The reason behind this is that they are losing a lot of money when their services are not available. For example, more than half of the fortune 500 companies experience a minimum of 1.6 hours of downtime per week creating an approximate loss of 46 million only in employee's salary say's a Gartner report in 2011 (Gartner). Even if this information is well known by the industry, they still struggle to manage the availability of their systems because a Ponemon Institute study shows that in the years 2012 and 2013, 91% of data centers endured unplanned outages (Ponemon). Furthermore, those outage costs are in average from 90 thousand up to 6 million for the large brokerage businesses and this is by hour of downtime. Table 1 shows the main players in the industry and the money loss the downtimes are generating.

Table 1 Main companies and the impact of the downtimes
taken from (Gagnaire et al., 2012)



| | Total(Hour) | Average(Hour) | Availablity | Cost/Hour(USD) | Cost(USD) |
|---|---|---|---|---|---|
| 1. Amadeus | 1 | 0.167 | 99.998% | 89,000 | 89,000 |
| 2. Facebook | 3 | 0.500 | 99.994% | 200,000 | 600,000 |
| 3. ServerBeach | 4 | 0.667 | 99.992% | 100,000 | 400,000 |
| 4. Paypal | 5 | 0.833 | 99.990% | 225,000 | 1,125,000 |
| 5. Google | 5 | 0.833 | 99.990% | 200,000 | 1,000,000 |
| 6. Yahoo! | 6 | 1.000 | 99.989% | 200,000 | 1,200,000 |
| 7. Twitter | 7 | 1.167 | 99.987% | 200,000 | 1,400,000 |
| 8. Amazon | 24 | 4.000 | 99.954% | 180,000 | 4,320,000 |
| 9. Microsoft | 31 | 5.167 | 99.941% | 200,000 | 6,200,000 |
| 10. Hostway | 72 | 12.000 | 99.863% | 100,000 | 7,200,000 |
| 11. BlackBerry | 72 | 12.000 | 99.863% | 200,000 | 14,400,000 |
| 12. NaviSite | 168 | 28.000 | 99.680% | 100,000 | 16,800,000 |
| 13. OVH | 170 | 28.333 | 99.677% | 100,000 | 17,000,000 |
| Total | 568 | 94.667 | 99.917% | | 71,734,000 |

### 2) SAForum solution

Nonetheless software developers and system integrators still find it challenging to design and implement HA systems. Moreover, the classic HA solutions have suffered from platform dependencies and vendor lock-in. To address this issue, the Service Availability Forum (SAForum) was established by world leading telecom and computing companies in an effort to standardize the way HA systems are built, and enable the portability and interoperability of highly available services across any platform compliant with the standards. In fact, the SAForum establishes a set of specifications defining standard Application Programming Interfaces (APIs), and guidelines to develop and deploy highly available systems. These specifications also define the architecture of a middleware (i.e. the SAForum middleware) capable of maintaining a cluster of servers and the services they host highly available. More specifically, the middleware is a distributed application deployed across the cluster's nodes. At runtime, the middleware will monitor the components providing the services, and in case a failure is detected, the middleware will automatically clean up the faulty components, fail over the services provided by the faulty component to a healthy replica, and attempt to repair the faulty component.

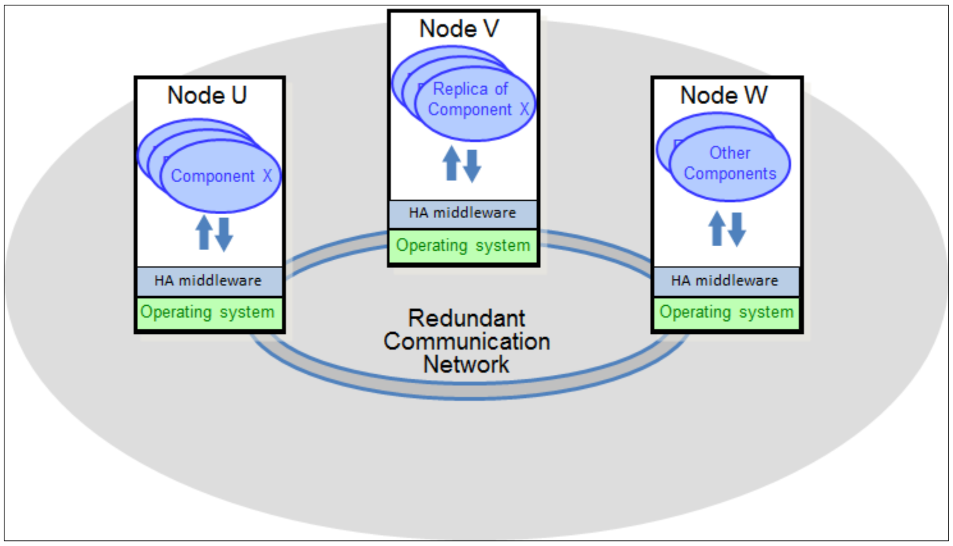


Figure 1.1 Typical clustered HA solution

Figure 1.1 illustrates a typical clustered HA solution where the SAForum middleware can be used. In a solution like this, where the components are abstracting software of hardware that provide services, the Availability Management Framework (AMF) constitutes the core of the SAForum middleware. It is AMF that maintains the HA in the cluster and reacts to failures. AMF's runtime behavior is mainly based on a configuration file (referred to as the AMF configuration) defined by the system integrator(s) designing the HA solution. The AMF configuration specifies the software components that AMF will instantiate at runtime, and defines the redundancy scheme employed (active/active, active/standby etc.), and the default recovery action for a given component, as well as the escalation policy in case the recovery action fails.

## 3) Thesis Motivation and Contributions

In order to define the AMF configuration, the system integrator needs to refer to the types description file. The types description file is expected to be provided by software vendors/providers to describe the software that will be managed by the SAForum middleware. This types description file will describe the type of the software in terms of the service types it can provide and in which capacity, as well as its dependencies and deployment limitations. The format of the types description file is defined by a standardized XML schema, and the content should comply with a set of informally described constraints from two different SAForum specifications (SAI-AIS-AMF; SAI-AIS-SMF). The manual definition of the types description file is a tedious and error prone task that requires deep knowledge of the specification details that many software developers do not necessarily have. In fact, the developers not only need to understand the structure of the elements of this file, but they should also respect the domain constraints that are spread across hundreds of pages in the specifications. This thesis presents an alternative and more intuitive approach for the automatic generation of the types description file. The approach is based on a high-level modeling language that software developers can easily understand, and that also abstracts the domain specific details that our approach automatically generates. More specifically, our approach is based on extending the UML component diagrams to enable expressing the requirements of the domain. In addition, a definition of a sequence of model transformations that eventually yields to the generation of the types description file. Our contributions in this thesis will: (1) enable the system integrators to automatically generate middleware configurations; (2) enable the software vendors to describe their software in a SAForum compliant manner using an approach that facilitates the creation and validation of types description files.

## 4) Thesis Organization

The rest of the thesis is organized in four chapters. In CHAPTER 1, the necessary background information on OpenSAF, a SAForum compliant open source middleware implementation, is provided. More precisely, this chapter elaborates about the needed information for understanding this thesis and the domain. In other words, this chapter explains the important parts of the OpenSAF middleware which are the IMM, AMF and SMF specifications including the ETF file. The CHAPTER 2 explains the existing approach for generating the middleware configuration and reviews the more general related works. In CHAPTER 3, the approach for generating the types description file (i.e. the ETF file) and the HA requirements is presented together with the challenges and issues encountered with their solutions. The prototype tool and its architecture are described in CHAPTER 4 as well as the case studies used to verify the work. Finally, we present the conclusion of this thesis along with a discussion on the possible future directions for this research.

# CHAPTER 1

# BACKGROUND

## 1.1 SAF Middleware

Creating highly available, mission critical system in a clustered solution is a challenging task with heavy responsibilities. Software providers and developers need to distinguish themselves with the value-added functions of the applications they are providing in order to be competitive. Therefore, it became very hard to focus on such quality of service. This is why the SAForum provides specifications for a middleware that manages all the high availability functions and responsibilities. The SAF middleware is a distributed software designed to achieve high availability of the computer-based services provided in a cluster (SAForum). The SAF middleware is composed of two main sets of services; the Application Interface Specification (AIS) services and the Hardware Platform Interface (HPI) services. While the HPI services allow the monitoring and management of the providing hardware, the AIS services enable the management of the high availability at the applications and components level.

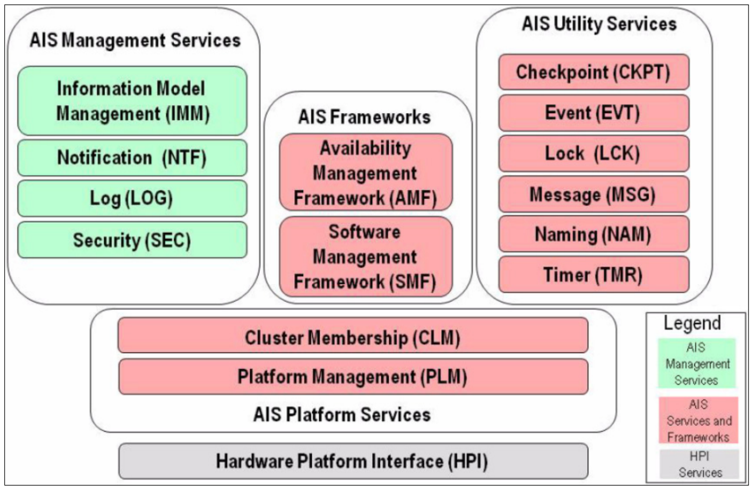


Figure 1.1 The Application Interface Specification services and frameworks (from (SAI-Overview))

As shown in the Figure 1.1, the AIS architecture is composed of 12 services and two frameworks. The four next categories describe briefly the purpose of those services and frameworks and the role they play in enabling the application highly available:

- The **AIS Platform Services** provide a higher level of abstraction of the underling hardware components and of the operating system functions to the other AIS services and application. Those services are grouped into two distinct sets of APIs:
  - <u>Platform Management Service (PLM):</u> It provides a set of APIs that gives a convenient abstraction for monitoring and managing hardware and low level software resources.
  - <u>Cluster Membership Service (CLM):</u> It provides and maintains a consistent view of the healthy node in a cluster membership. It monitors the nodes joining or leaving the cluster and determines if the node is eligible to be a member of the cluster.
- The **AIS Management Services** provides APIs that enable standard management of the highly available cluster to the others AIS services and applications. The following four services groups the concerned APIs:
  - <u>Information Model Management Service (IMM):</u> This service contains APIs that enables definition, manipulation and exposition of both the configuration and the runtime management information. It also enables the invocation of administrative commands on the cluster objects.
  - <u>Notification Service (NTF):</u> Those APIs and data structures answer the needs of notification for alarms, state changes, object life cycle changes, attribute value changes, security alarms and specifics events.
  - <u>Log Service (LOG):</u> It provides APIs that enable logging of alarms, notifications, system messages and application defined logs. This service is made to work at a cluster-wide scale.
  - <u>Security Service (SEC):</u> It provides APIs that enable security management of the access to the AIS and HPI services.

- The **AIS Utility Services** provides the generic APIs that enable functions that highly available distributed systems need to have by definition. Those APIs are organized in the following services:
    - Checkpoint Service (CKPT): It provides an API that allows a process to save any kind of checkpoint data regarding its state. The purpose of this service is to enable a process to retrieve its last saved state after recovering from a failure.
    - Event Service (EVT): It provides an API that allows processes to communicate between them. The service is designed to work as the following; one or many publisher(s) publish an event through an event channel and the subscribers of this channel can read the published event.
    - Lock Service (LCK): It provides an API that is intended to be used in a cluster where processes across different nodes may compete with each other for access to the same shared resource.
    - Message Service (MSG): It provides an API that allows the use of a buffered message-passing system based on the concept of a message queue. The same queue can be acceded everywhere in the cluster and the messages are preserved when they have not been read.
    - Naming Service (NAM): It provides an API that enables the boundary of human readable names to objects of the middleware. Therefore, those objects can be looked up given their names.
    - Timer Service (TMR): It provides an API that enables a mechanism by which the client processes can get notified when a given timer expires. The timer is a logical object dynamically created and it represents an absolute time or duration.
- The **AIS Frameworks** provides important functionalities, APIs and data structures that enable the management of the high availability of the provided services. There is two distinct categories of functionalities, which are grouped in the following frameworks:
    - Availability Management Framework (AMF): This is the core of the high availability management in the SAForum Middleware. The objective of AMF is to achieve the high availability of the services. Therefore, it provides rules,

architectures, constraints and functionalities that enable a cluster to achieve high availability for their services.

- Software Management Framework (SMF): It provides functionalities that are used for managing the applications of the highly available cluster during hardware, operating system, middleware and application upgrades while ensuring their availability.

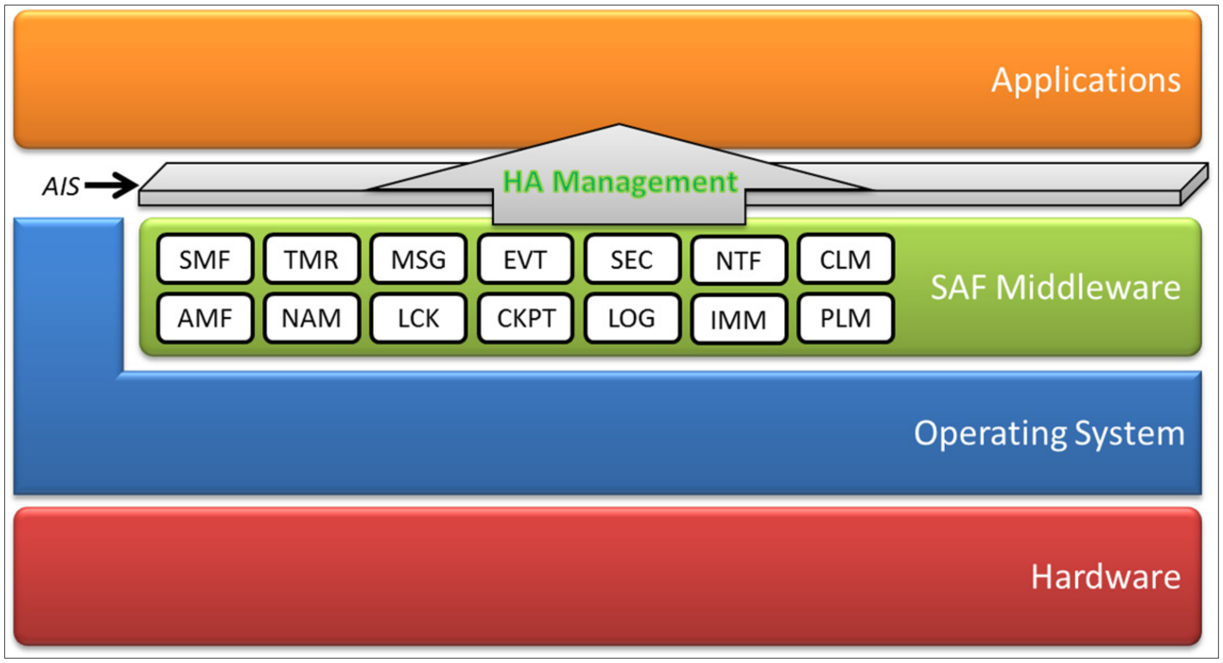


Figure 1.2 SAF Middleware (AIS) high level architecture view

As depicted by the high level architectural view in the Figure 1.2, the SAF middleware operates the HA management of the applications through the AIS services. In fact, this is mainly possible because the APIs provided by those services enable the implementation of callback functions by the applications. In the case of off-the-shelf applications that cannot implement those callback functions, the middleware provide command line supports for using scripts that can replace some of the callback functions. In the next Sub-section, the IMM, AMF and SMF services are described.

## 1.2 Information Model Management (IMM)

In the SAF middleware, all entities defined by the different SAF services have their own logical representation into the SAF information model (IM). These logical representations are the objects and attributes, which abstract and capture the information needed for the various management functions of the different services. Those objects and attributes are expressed in a UML model and its serialization is normalized by an Extensible Markup Language (XML) schema (SAI-AIS-IMM-XSD). For example, any application managed by AMF have its objects and attributes definition in the information model. Another example would be the upgrade campaign from SMF that allows the middleware to migrate from one configuration to a new one. Like the application, the upgrade campaign has its own objects in the information model that captures the information needed in order to be executed.

The entities defined by the different services are not the only objects of the information model. In fact, all the information regarding the state of the cluster is captured by the model. This information is relevant for an appropriate management of a clustered solution based on a HA distributed middleware. The information captured by the objects and attributes of the information model fit into two main categories: (1) the runtime information that mainly characterizes the state of the cluster and its applications at runtime, (2) and the configuration objects that describe the design of the cluster, the managed applications and the desired level of availability.

The IM objects also define administrative operations that can be performed on the represented entities through the SAF interfaces. Since the IM is only providing information, the IMM service provides the actual execution of the operations on the appropriate entities referred to by the IM objects. However, the IMM does not execute the actual operations but, it refers the operations to the appropriate service. Furthermore, it provides the necessary APIs for creating, accessing and managing these objects.

## 1.3 Availability Management Framework (AMF)

The Availability Management Framework is the main software entity that enables the management of the high availability of the applications providing services. It achieves this by coordinating all the entities within a SAF cluster. Furthermore, it defines entities for representing all kind of applications providing a service. Those are called the AMF entities and their life cycle is managed by the Availability Management Framework itself. In order to manage the availability of the applications, the SAForum chose an approach with a distinct separation between the service provided and the application providing the service. Concretely, AMF ensures that the services are always provided and in case of a failure, this is AMF that orchestrate the recovery actions (SAI-AIS-AMF).

Due to the nature of the clustered solution based on an HA middleware like the SAF one, the main strategy to ensure the persistence of the provided services through a failure is to deploy the service providers in a redundant manner. In the case where a failure occurs at the service provider or at the node level, AMF detect the service outage and reassign the service workload to a healthy replica of the service provider. However, AMF is able to perform such operation only when given a proper configuration. This is why the configuration is very important; it allows AMF to know what the resources are and how they can be used. For capturing this information, AMF defines entities and relations between them that capture the necessary information for maintaining the high availability of the applications. The AMF entities are part of the information model as objects, attributes and operation, and together they form a complex UML model. Together with the IMM services APIs, AMF can perform runtime assignation of the service workload in order to keep the service at the desired HA state. In the following Sub-section, the AMF entities are discussed in more details since they are needed to understand the contributions in this thesis.

### 1.3.1 AMF Entities

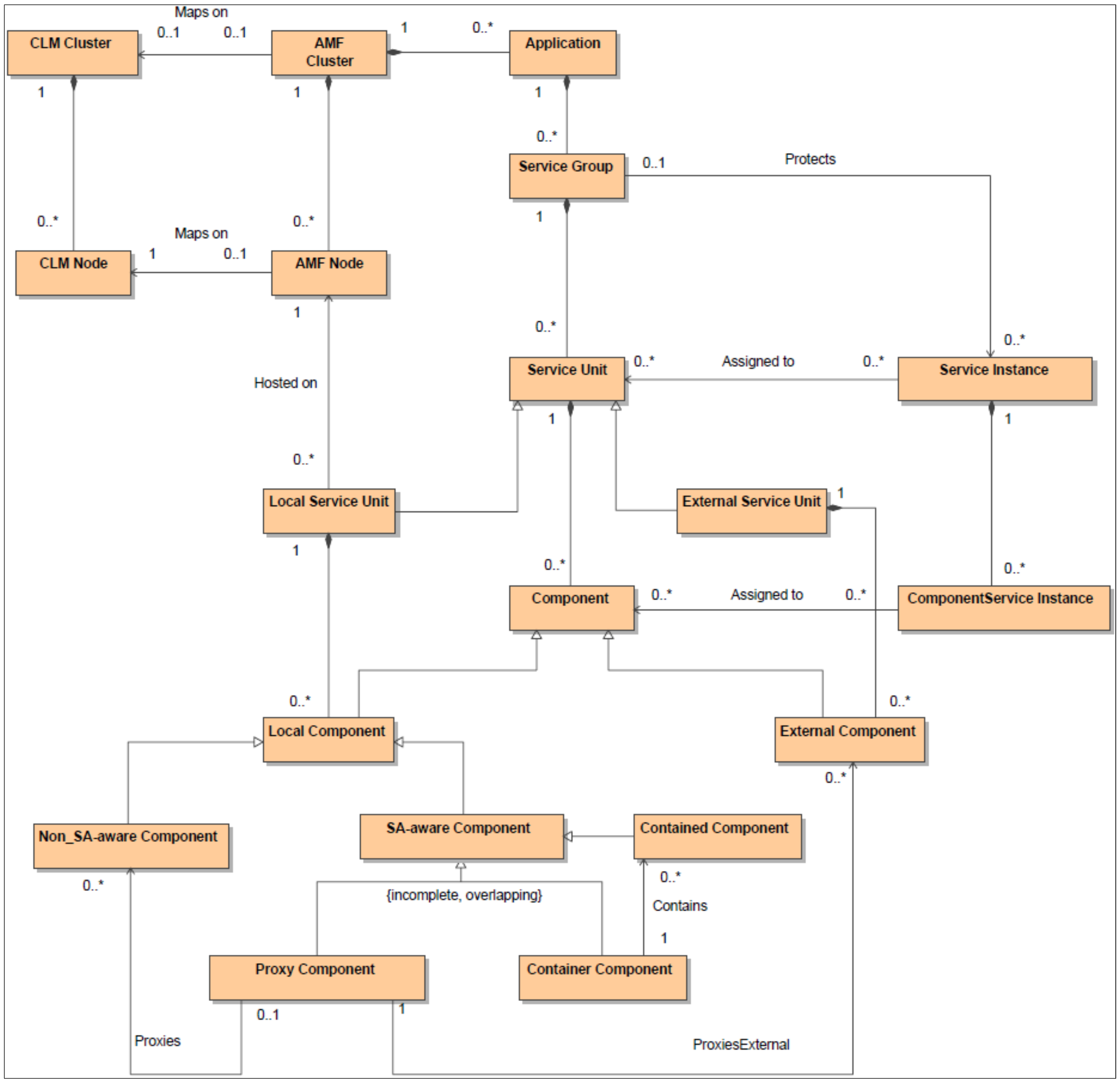


Figure 1.3 AMF logical entities
(from (SAI-AIS-AMF))

In the AMF information model, the AMF logical entities are defined in detail. Those entities are representing the resources under AMF control. All the non-runtime information part of the IM is regarded as the AMF configuration, and it is stored in a repository while the runtime information can be obtained only when the system is operating. The UML class diagram in the Figure 1.3 illustrates the model that abstract the resources under AMF control. The entities of the model are described in the following Sub-sections:

- **CLM Cluster and Node**: The CLM cluster is composed of CLM nodes. This entity abstracts the actual physical cluster composed of physical nodes. In other words, The CLM cluster and its nodes abstract the resources an AMF cluster is going to use for providing services.
- **AMF Cluster and Node**: The AMF cluster groups the AMF nodes. Furthermore, the AMF node is a logical entity that represents all the AMF entities that can be provided by a CLM node.
- **Component**: This is the logical entity that represents the set of resources AMF is going to use for providing services. More precisely, the component entities abstract the specific functionalities that can be provided. These functionalities enable the services and they can be provided by both hardware and/or software resources. This is the smallest entity on which AMF performs the availability management. Depending on where the component is executing, what environment it needs and if it implements the AMF APIs or not, are what distinguish the different specializations of the component. So, in order to distinguish the different behaviors, functionalities and properties, the following categories are defined:
    - **Local Component**: This entity represents any component that is executing within the CLM cluster in which the AMF cluster is running.
        - **SA-aware**: This category means that the software abstracted by the component implements the AMF APIs. When software implements them, the framework has a full control over the life cycle of the component. Consequently, the component must obey the AMF command by implementing the corresponding callback functions. The principal

behavior specific to this type of component is that an instantiated SA-aware component is providing services only when assigned a workload by AMF. This behavior is called pre-instantiable and it is mandatory for all SA-aware components.

- **Container and Contained**: In order to integrate the applications running on a specific controlled execution environment rather than directly on the operating system, AMF defines the concept of Container and Contained components. In such scenario, the container represents any kind of specific execution environment running on top of the operating system (e.g. virtual machines). On the other hand, the application running in the specific controlled environment is represented by the contained component.

▪ **Non-SA-Aware**: Any component that does not implement the AMF APIs is represented as a non-SA-aware component. That means AMF have a limited control over the component life cycle.

- **Non-Proxied, Non-SA-Aware**: In the scenario of a local component without AMF APIs implementation and no application to relay the communication between the framework and the component, AMF is controlling the life cycle only for instantiating or terminating the component.

▪ **Proxy and Proxied**: When a component mediates the communication between another component and AMF, it is represented by a proxy component. All proxy components are SA-aware. However, the component for which the proxy is mediating the communication is called a proxied component, and it is always a non-SA-aware component.

o **External Component**: Any resource running outside the AMF cluster is represented by the external component and it is always non-SA-aware. Also, the usage of a proxy component is mandatory because AMF cannot directly reach components outside the AMF cluster.

- **Component Service Instance (CSI)**: The featured services provided by a component are abstracted and separated from the service provider, namely the components. When AMF assigns a service workload to a component, it means that it assign a component service instance, therefore it represents the workload of providing the service. When the configuration designer defines this entity, it must capture the featured service at a fine granularity level. Having a separation between the service provider and the workload of the service provided allows the middleware to dynamically assign the workload during runtime. Therefore, it enables the middleware to take action in case of a failure and swiftly redirect the workload on a healthy redundant replication of the component.
- **Service Unit (SU)**: The service unit is an entity that aggregates one or many components in order to provide a higher level of service. In other words, service units can be composed of a set of components that needs to combine their functionalities in order to provide a certain high level functionality. There are two categories of service units: the local service unit that aggregates the local components and the external service units that aggregate the external components. Also, it is worth noting that a given component can be in only one SU.
- **Service Instance (SI)**: In the same way the SU aggregates components, AMF defines the aggregation of the CSIs into a logical entity, namely the SI. When AMF assign a SI to a SU, all the CSIs of the SI are assigned to the corresponding components of the SU.
- **Service Group (SG)**: Since the strategy for maintaining high service availability is managing the redundancy of the service provider entities in case of a failure, AMF defines the entity service group that groups the redundant SUs. Moreover, the SUs within the SG are going to be used for protecting the SIs. All the SUs of the SG must be able to take an assignment for any SIs protected by this SG. The SG also defines the notion of redundancy model where the components within the SUs participating in the SG must be able to support. The following is describing the different redundancy models:
    - **The 2N redundancy model**: In a SG with a 2N redundancy model, for all the SIs there is at most one SU with an active assignation and exactly one SU with a standby assignation. Figure 1.4 illustrates a graphical example of this model with

four components grouped on two SUs distributed on two nodes. In the case of a failure of the active SU, AMF is going to switch the standby SU assignment to active.

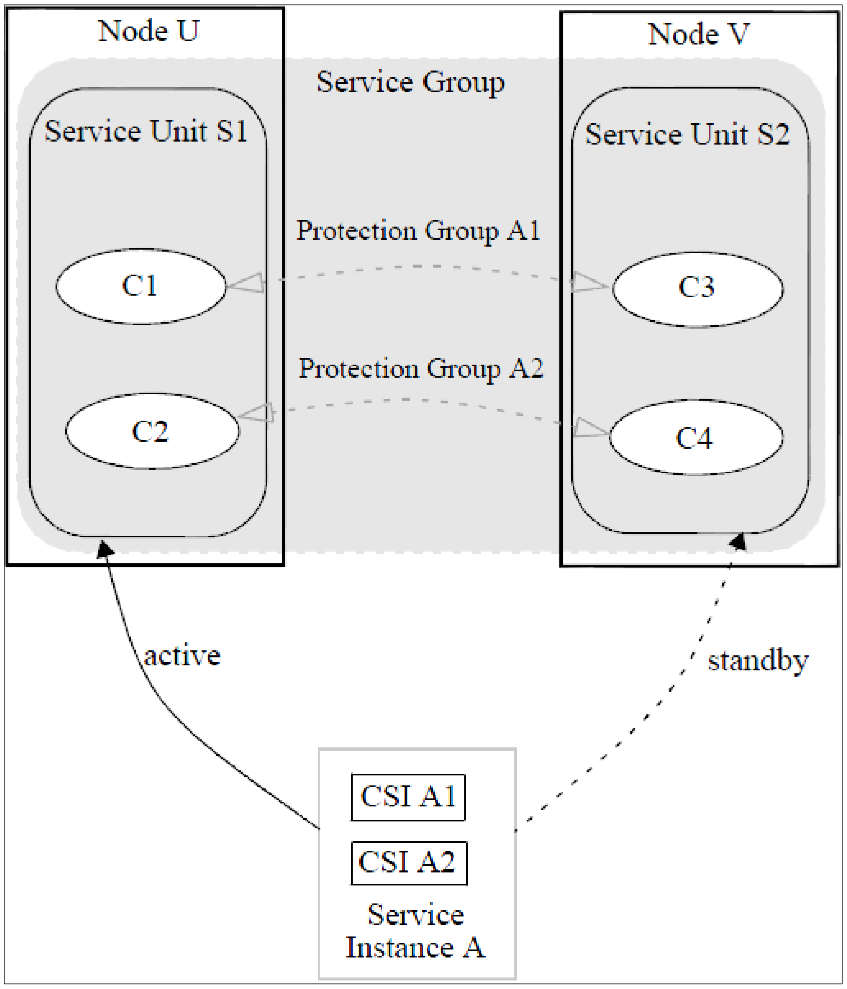


Figure 1.4 Example of the 2N redundancy model
(from (SAI-AIS-AMF))

- **The N+M Redundancy Model**: In a redundancy model of N+M, there is N SUs with active assignments for all the SIs, and M SUs with standby assignments for all the SIs. In this redundancy model, an SI is assigned active for at most one SU and standby also for at most one SU. The Figure 1.5 illustrates an example of the redundancy model with 8 components grouped on four SUs distributed on four nodes providing the services for 3 SIs.

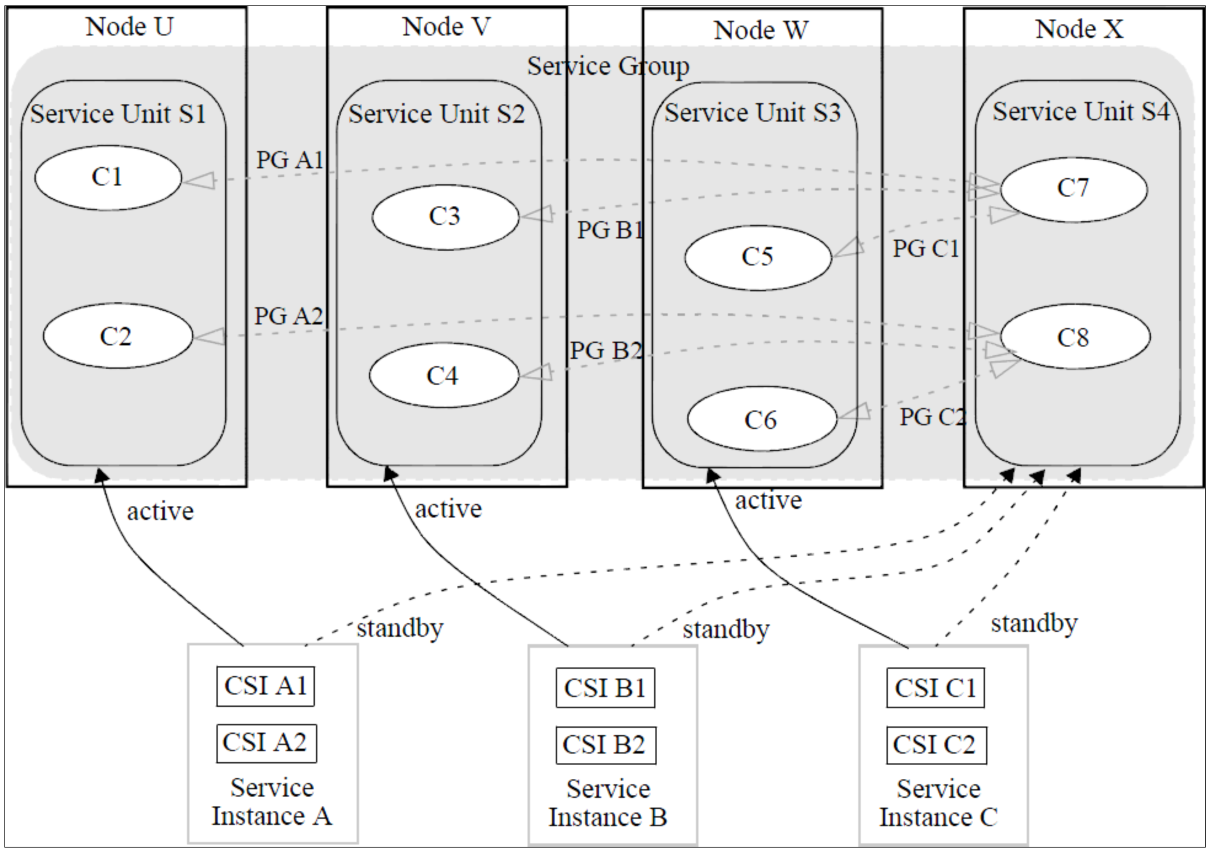


Figure 1.5 Example of the N+M redundancy model
(from (SAI-AIS-AMF))

- **The N-Way redundancy model**: This redundancy model means that there is N SUs protecting the SIs. More importantly, any SUs can simultaneously have active assignment for some SIs while they have some standby assignment for some other SIs. However, the SIs can be assigned active to at most one SU while they can be assigned standby to zero, one or many other SUs. The Figure 1.6 illustrates an example of the model with six components grouped on three SUs distributed on three nodes and the SG protects three SIs. Furthermore, each SIs has one active assignment and two standby assignments.

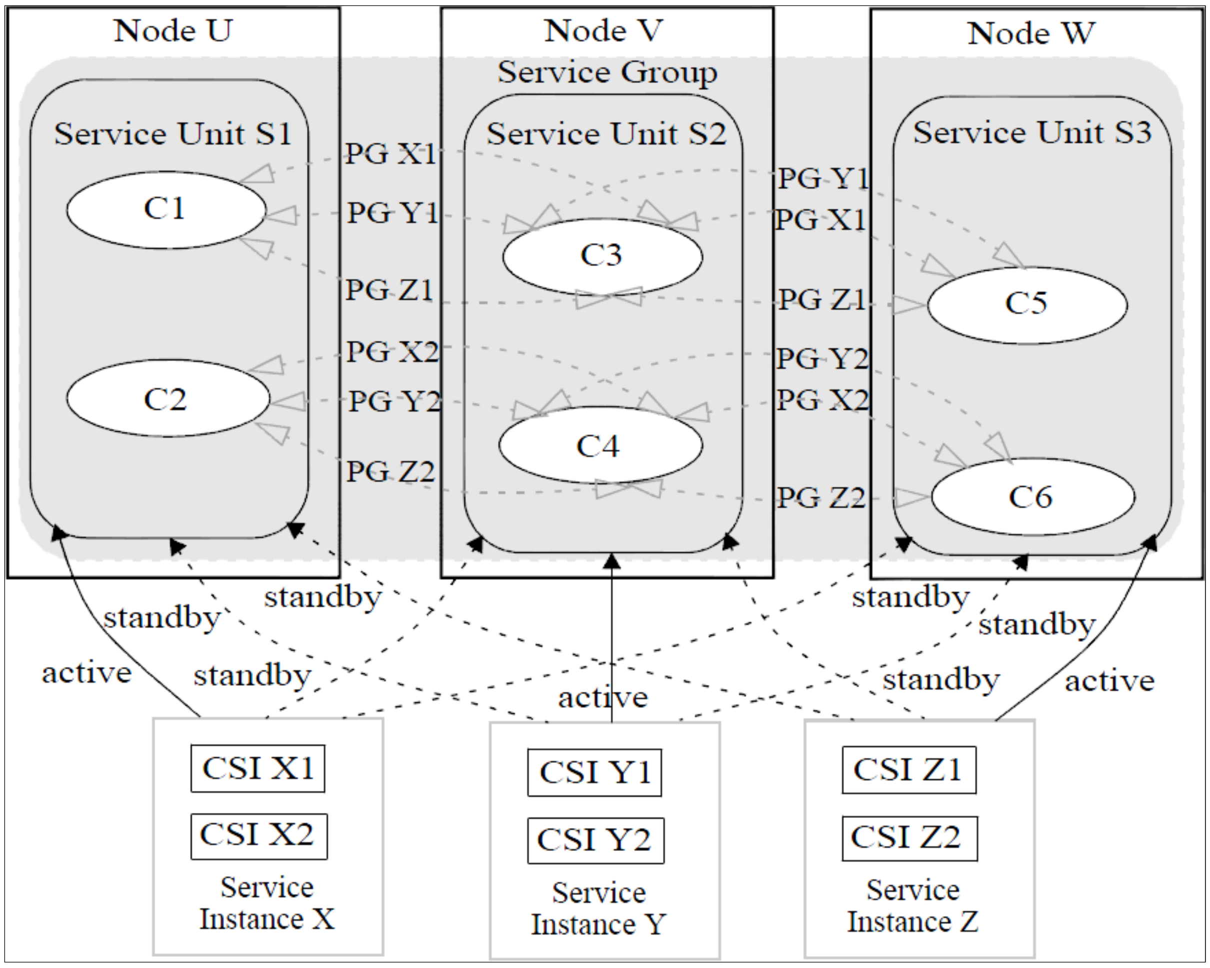


Figure 1.6 Example of the N-Way redundancy model
(from (SAI-AIS-AMF))

- **N-Way Active redundancy model**: The main characteristic of this redundancy model is that it does not support the standby assignation for the SIs. Furthermore, this model allows the SU to have an active assignation for zero, one or many SIs. Likewise, an SI can have an active assignation on zero, one or many SUs. The Figure 1.7 depicts an example of the N-Way active redundancy model with six components grouped on three SUs distributed on three nodes forming an SG that protect three SIs. Additionally, each SI has two active assignations and no standby assignation.

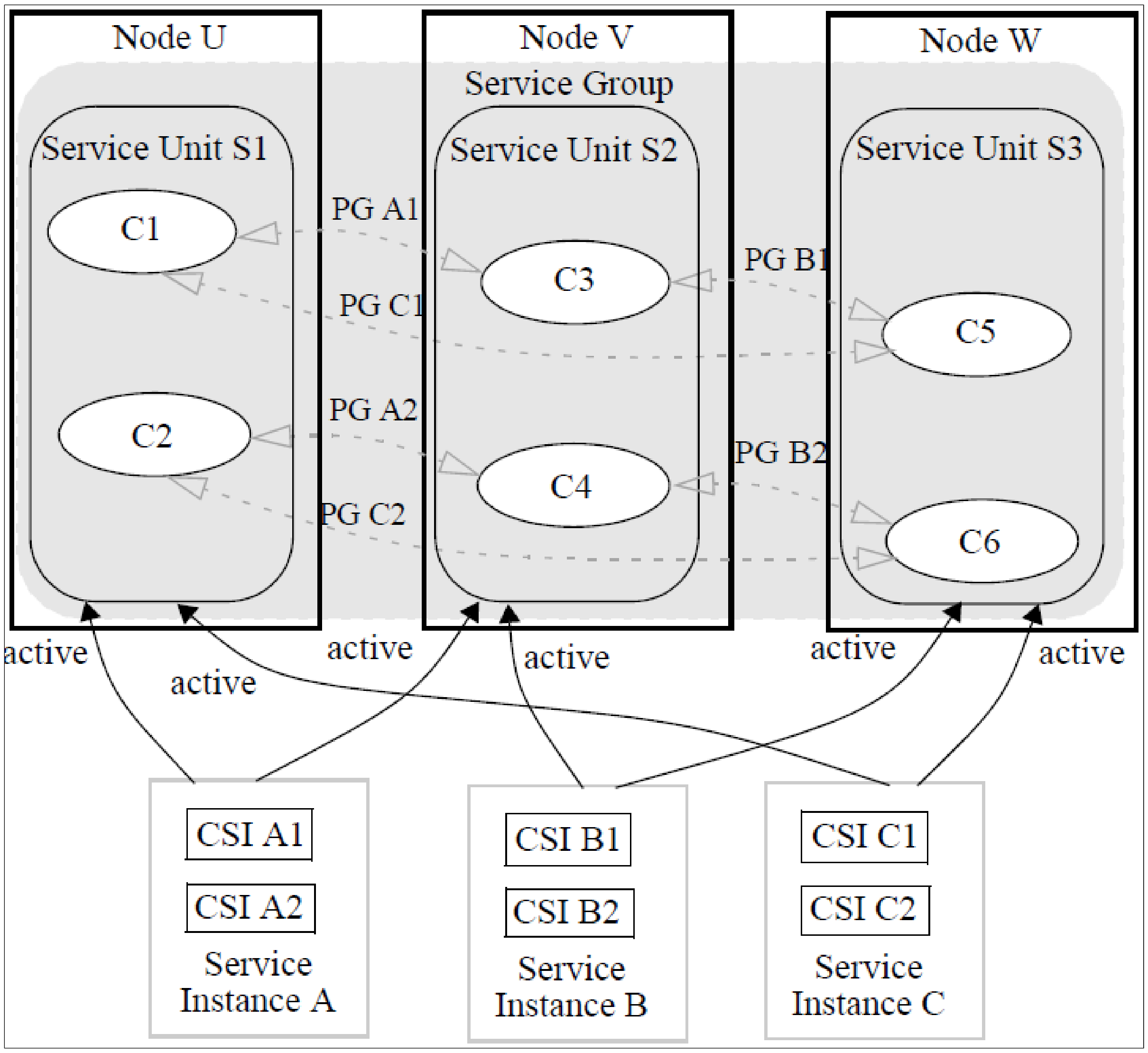


Figure 1.7 Example of the N-Way Active redundancy model
(from (SAI-AIS-AMF))

- **No redundancy model**: This redundancy model typically addresses non-critical components that do not significantly affect the system when they fail. Instead of using standby assignation on some service unit, the SIs can have a minimum level of protection with spare service unit that have no assignation at all. Hence, the SI can have at most one active assignation while the SU can also be active for at most one SI. The Figure 1.8 shows an example of the no redundancy model with six components grouped on three SU distributed on three nodes forming an SG that provides the service for three SIs.

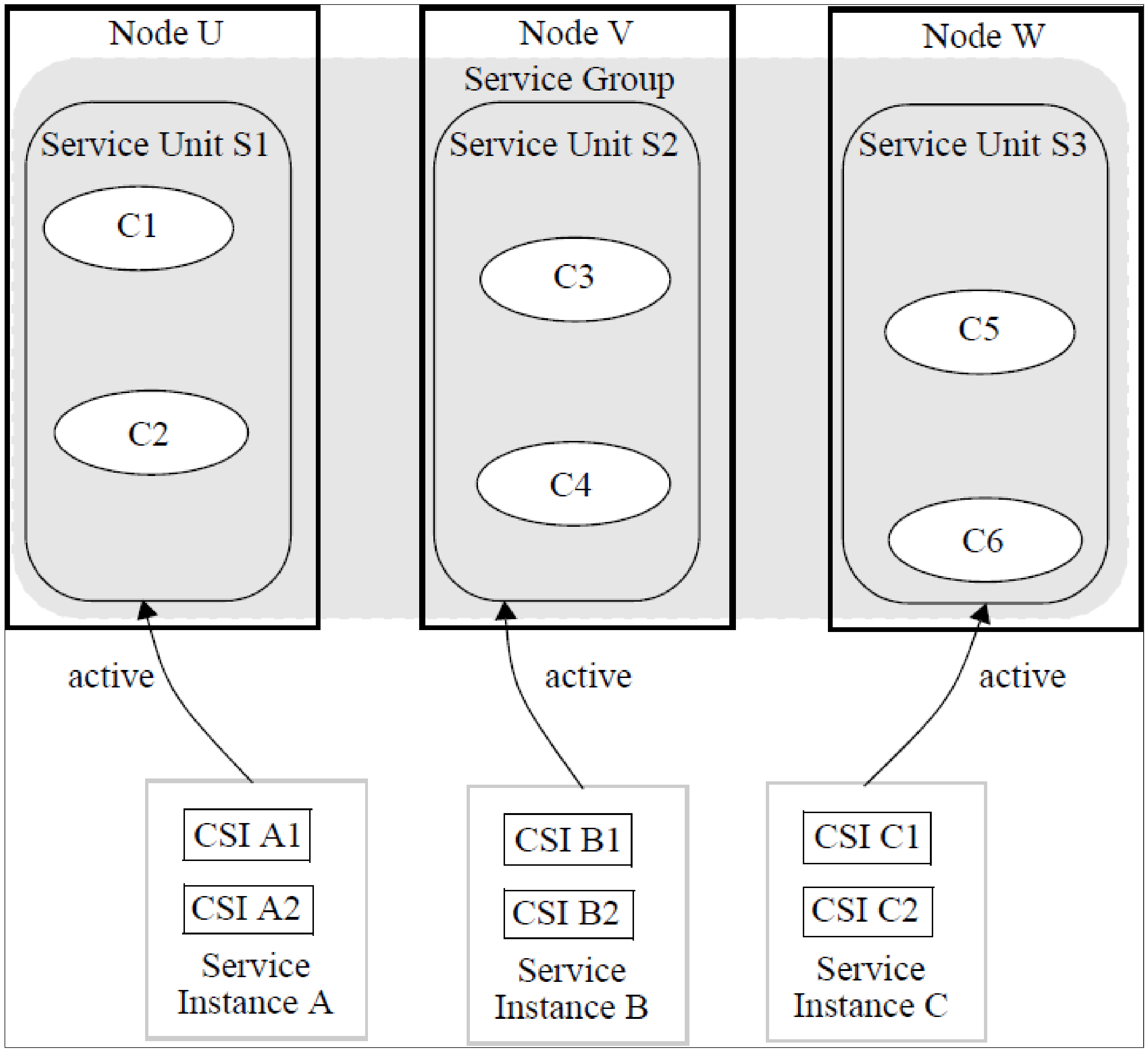


Figure 1.8 Example of the No redundancy model
(from (SAI-AIS-AMF))

- **Application**: The application entity combines the individual functionalities of its SGs. Moreover, this entity represents the highest level of service.
- **Protection Group (PG)**: This is a dynamic entity that informally represents the groups of components to which a CSI has been assigned.

### 1.3.2 AMF Entity Types

In order to facilitate the configuration and the software management, AMF defines types for all of the entities except for the node and the cluster. Moreover, the entity types containing a version are capturing general information, and when entity types differ mainly by the version, they are grouped into an entity base type. Defining the entity types in the configuration is mandatory and allows the specification of important aspects regarding the entities. Thus, an entity type defines important attributes regarding the service that the entity is providing, the limitation of this entity, compatibility, dependency and etc. This information is mainly derived from the Entity Type File discussed in Section 1.5. The following is describing briefly the entity types AMF use in the configuration:

- **Component Type (CT)**: The component type captures information that represent a particular version of the software/hardware implementation used to make the actual component. The main characteristics of a Component Type are the following:
    - The type of service (CST) a component of this type can provide and how it can provide it.
    - The CTs this CT needs in order to provide a certain CST.
    - The component category of the implementation; SA-aware, container, contained, non-proxied non-SA-aware, proxy and proxied.
    - The capability model for each provided CST. In other words, this is the number of CSIs of the CST that can be provided by this type of component. There are seven capability models and the following table shows in which redundancy model a component of the given capability model can participate.

Table 2 Map of the Component Capability model and the Redundancy Model

| **Redundancy Model** / **Component Capability Model** | 2N | N+M | N-Way | N-Way Active | No-redundancy |
|---|---|---|---|---|---|
| x_active_and_y_standby | X | X | X | X | X |
| x_active_or_y_standby | X | X | - | X | X |
| 1_active_or_y_standby | X | X | - | X | X |
| 1_active_or_1_standby | X | X | - | X | X |
| x_active | X | X | - | X | X |
| 1_active | X | X | - | X | X |
| non-pre-instantiable | X | X | - | X | X |

- **Component Service Type (CST)**: All the services that are equivalent and managed in the same manner shall be of the same type and therefore, enabling AMF to see them as equivalent. Meanwhile, the CST is the type of service a component of a given type can provide and therefore, this is the CT that specifies the CST a component can provide.
- **Service Type (SvcT)**: The service type groups a set of CST that a certain SU can provide. Since this type defines the general feature of an SI, it also captures the limits of an SU regarding an SI of a given type. The limits regards the number of CSI of each CST composing the SvcT can compose the SI of that type and therefore, specifying how CSI an SU can support.
- **Service Unit Type (SUT)**: The service unit type is composed of CT and like the SvcT, it can specify the maximum number of components of each CT composing the SUT an SU of that type can contains.
- **Service Group Type (SGT)**: The service group type captures information about what type of availability an SG of this type can offer. Therefore, it specifies the redundancy model and it aggregates the SUT that can participate in this SGT.
- **Application Type (AppT)**: The purpose of this type is to groups the set of SGT that can provide a very high level type of services.

### 1.3.3 OpenSAF implementation

The OpenSAF framework is an open source project that implements the SAForum specifications (OpenSAF). The SAF community recommends the use of this framework as a mature implementation of the standards. In fact, all the services needed for the HA management are implemented and well documented to be usable. Consequently, this middleware is used in this research project as a reference for the purpose of testing and validating the work.

### 1.3.4 AMF Configuration Example

As explained in the section 1.3.2, the types are facilitating the configuration effort and the software management at runtime. Figure 1.9 illustrates the entity types of an AMF configuration example for a web service. In this example, there are two component types, one is representing the version 5.6 of the MySQL implementation and is capable of providing the DB component service type. The second component is representing the version 2.4 of the Apache implementation and is known to be capable of providing an HTTP component service type. Both components are grouped into the SUT Web and this SUT is capable of providing the service type WebSrv. The SUT is grouped by the SGT Web_site that can protect the WebSrv with SU of the type Web. This is the SGT Web_site that composes the application type Web_app. An important detail is expressed in this example; this is the grouping of the component types MySQL_5.6 and Apache_2.4. Because of the nature of a dynamic web site, more than one component is needed in order to provide this higher level service. Basically, it means that the system needs both components types in order to be able to provide a service of the type WebSrv.

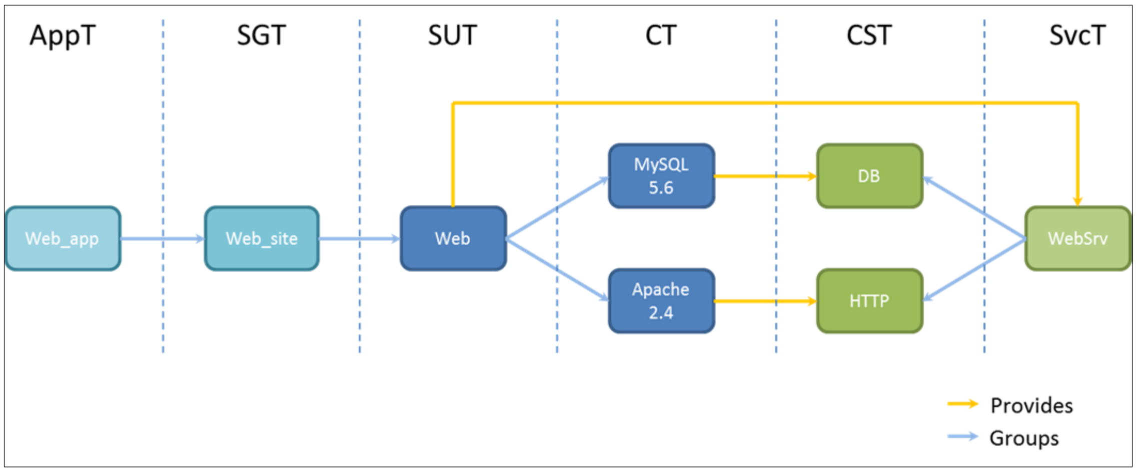


Figure 1.9 Entity Types of an AMF configuration example

Following the same example, Figure 1.10 illustrates a configuration of entities based on the type of the precedent shown in Figure 1.9. Those logical entities represent the running processes that provide the service. In this example, the service is protected in a service group with a 2N redundancy model, meaning that there is one active and one standby SU. Those SU are named *Web SU 1* and *Web SU 2* and they are respectively located on the *Node 1* and *Node 2*. The SUs are grouping two components, namely MySql and Apache. Furthermore, in this configuration the middleware is protecting one SI composed of two CSIs; one abstracting the workload of a database named *DB* and another one abstracting the workload of an HTTP service named *HTTP*. This SI is assigned active on the *Web SU 1* and standby on the *Web SU 2*.

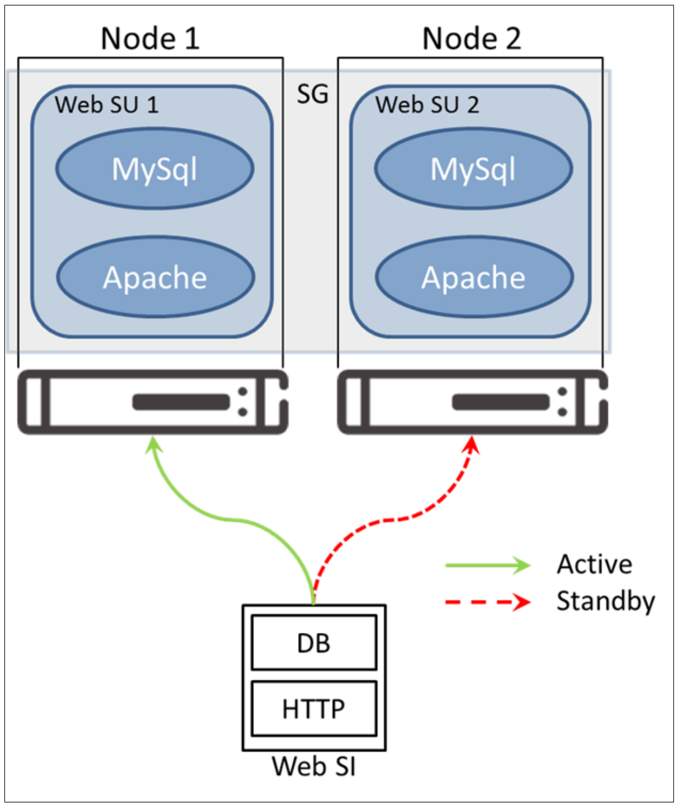
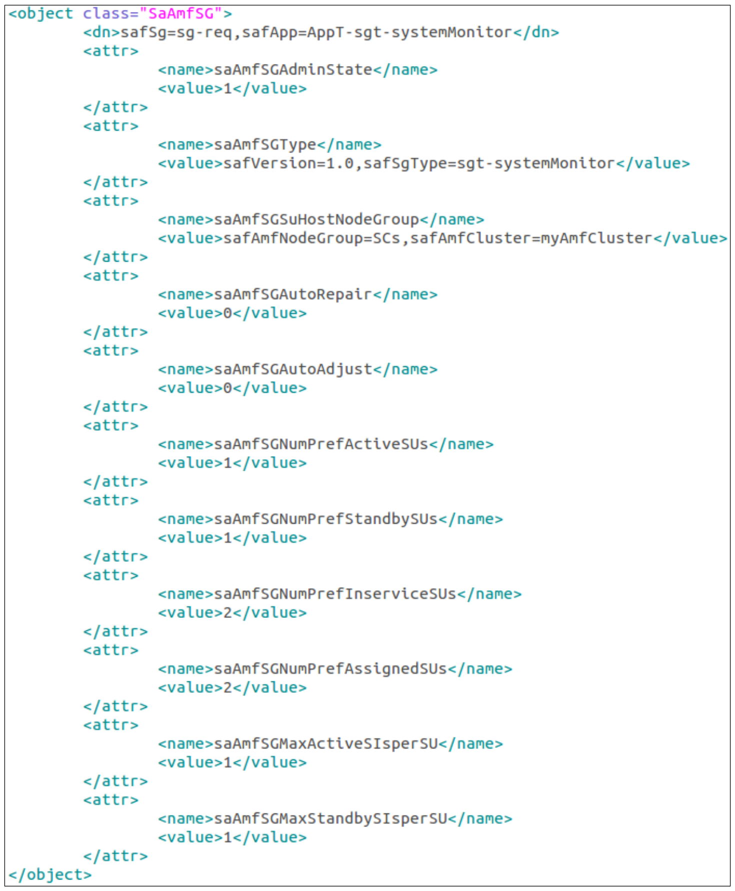


Figure 1.10 Entities of an AMF configuration example

Since the components *MySql* and *Apache* are grouped in the same SU, it means that if one of them is no longer capable of providing its assigned CSI because of a failure, AMF will failover the whole SI including both CSI on the standby *Web SU 2* and therefore, this latter SU will become active for the *Web SI*.

According to the specification, the configuration shall be saved persistently to a file based on a XML schema. However, this XML file is containing all the information of the UML class diagram. For a very simple cluster of two nodes with two applications, the content of the file is already heavy and its management is time consuming and error prone. Figure 1.11 illustrates an overview of the content of such configuration file. In this particular example, the file is reaching more than 13000 lines of XML for a cluster of two nodes and two applications.



```
<object class="SaAmfSG">
        <dn>safSg=sg-req,safApp=AppT-sgt-systemMonitor</dn>
        <attr>
                <name>saAmfSGAdminState</name>
                <value>1</value>
        </attr>
        <attr>
                <name>saAmfSGType</name>
                <value>safVersion=1.0,safSgType=sgt-systemMonitor</value>
        </attr>
        <attr>
                <name>saAmfSGSuHostNodeGroup</name>
                <value>safAmfNodeGroup=SCs,safAmfCluster=myAmfCluster</value>
        </attr>
        <attr>
                <name>saAmfSGAutoRepair</name>
                <value>0</value>
        </attr>
        <attr>
                <name>saAmfSGAutoAdjust</name>
                <value>0</value>
        </attr>
        <attr>
                <name>saAmfSGNumPrefActiveSUs</name>
                <value>1</value>
        </attr>
        <attr>
                <name>saAmfSGNumPrefStandbySUs</name>
                <value>1</value>
        </attr>
        <attr>
                <name>saAmfSGNumPrefInserviceSUs</name>
                <value>2</value>
        </attr>
        <attr>
                <name>saAmfSGNumPrefAssignedSUs</name>
                <value>2</value>
        </attr>
        <attr>
                <name>saAmfSGMaxActiveSIperSU</name>
                <value>1</value>
        </attr>
        <attr>
                <name>saAmfSGMaxStandbySIperSU</name>
                <value>1</value>
        </attr>
</object>
```

Figure 1.11 Content of a SG in the XML configuration file

## 1.4 Software Management Framework (SMF)

In a highly available system based on the SAF middleware, the configuration is loaded from a repository when the system is starting. More precisely, the IMM service is parsing a XML file which is a serialisation of an instance of the XML schema representing the IM and including the AMF configuration model. Using only this service, updating the configuration implies shutting down the whole system, changing the serialized instance of the IM in the XML file and then starting the system again. This procedure implies unnecessary downtime because the whole AMF cluster needs to stop while the XML file is being upgraded. In fact, this procedure is used when a new AMF cluster is being deployed.

In practice, the SAF systems are required to provide their highly available services over a long period of time. During the long term life cycle of the system, the AMF configuration may change in order to follow the evolution of the software being managed. Those changes can imply addition, modification or removal of the AMF entities representing the changing software or hardware resources. By definition, highly available services should not suffer significant loss of service in the case of such changes (SAI-AIS-SMF).

The SMF specification defines the procedure in which the system can swiftly migrate from one configuration to a new one with minimum service loss. In order to achieve this, SMF define entities and in the following, the main entities are briefly described:

- **The upgrade campaign**: This is a configuration object that can be added at runtime. Mainly, it contains the location of the file containing the procedure and the steps for manipulating the entities being updated. It is worth noting that this file is also normalized by a XML schema (SAI-AIS-SMF-XSD).
- **The upgrade procedure**: This entity is runtime only and is created by SMF based on the XML file of the upgrade campaign. This entity specifies the scope of the campaign, meaning the set of entities on which the steps will be executed. It also specifies the upgrade method that defines how the step will be executed, and it aggregates the steps of the campaign.
- **The upgrade steps**: This is also a runtime entity and it represents the action to be taken on the different entities of the campaign scoop.

## 1.5 Entity Type File (ETF)

The types used in the AMF configuration and the upgrade campaign description files are directly derived from the types defined in the ETF file. The ETF file is also normalized by an XML schema (SAI-AIS-SMF-XSD). Basically, the ETF file is the content description of the software that can be delivered to a SAF system. The SMF specification assumes that this file is provided by the software provider.

The content is organized around an entity called the software bundle. While the minimal content that need to be provided in the file is the component type (CT) and the component service type (CST) they can provide, the file can also describe in which SUT, SGT and AppT the components can participate, as well as the SvcT in which the CST can participate. Indeed, the later entity types are optional, and if they are not specified, it means that the child type can participate in any type of parent. For example, if the SUT of a component is not specified, it means that the component is not restricted to any SUT and thus, can participate in any kind of SUT when delivered to the SAF system.

Other than the capacity of the software being delivered to the system, the file shall also specify the commands to install or remove the components from the system. Since the AMF types are derived from this file, it needs to describe the type of implementation (e.g. SA-aware, etc.) of the component and the capacity and limitations. Mostly, the attributes carrying this information are expressed in range of value that can take the derived AMF types. Also, important information captured by this file is the dependency of the component regarding what CST a component need in order to be able to provide another given CST. For example, a web application CT may need the CST of a database in order to provide a web application CST.

# CHAPTER 2

# RELATED WORK

## 2.1 Previous Approach for Generating the AMF Configuration

In previous works (Kanso, Toeroe, Hamou-Lhadj, & Khendek, 2009; Kohzadi, 2009), an approach for the automatic generation of configurations and upgrade campaigns is presented. Figure 2.1 illustrates the global approach for generating the middleware configuration. This approach is based on two inputs; (1) the ETF file expected to be provided by the software provider and (2) the HA requirements that describe the level of service desired by the system designer. Assuming that ETF file is valid and can provide the service level described by the HA requirement, the configuration generator will produce a valid AMF configuration. The system designer can use the generated configuration as input for the middleware only if this is a new system that is being deployed. Otherwise, the upgrade-campaign generator will generate a valid upgrade campaign describing the steps the SAF middleware needs to perform in order to swiftly migrate to a new configuration.

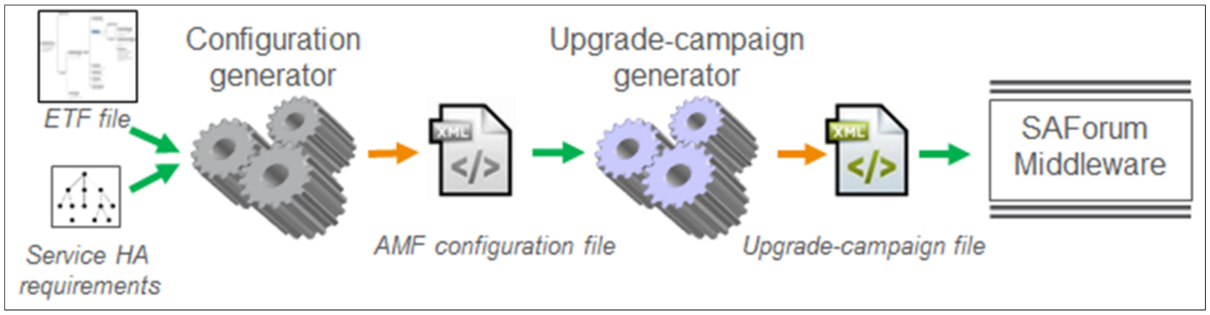


Figure 2.1 Previous configuration generation approach

In practice, it appears that: (1) In large companies using the SAForum middleware, a big portion of the software is developed in-house, and therefore the software vendor or provider and the software user or maintainer are the same company. (2) Even software vendors developing SAForum compliant software prefer to shield their developers from having to manually create and validate ETF files. (3) The use case of using a proxy to allow the SAForum middleware to manage legacy software (non-SAForum compliant) is widely spread, and requires the existence of an ETF file to generate the AMF configuration that includes the legacy applications. For all those reasons, the previous tool chain shown in Figure 2.1 has been extended in order to include one more step for the automatic generation of the ETF file. The ETF file is described in the Sub-section 1.5, AMF configuration in Sub-section 1.3 and the upgrade campaign, part of the SMF, is described in Sub-section 1.4. However, the following Sub-sections describe the other artefacts used for generating the middleware configuration.

### 2.1.1 The HA Requirements

While the ETF file capture only the capacities, limits and functionalities of the components, the HA requirements[1] capture the level of availability desired by the system designer. Concretely, this level of availability for the services is expressed in terms of SIs and CSIs to be provided. It also include how the services shall be protected (i.e. the redundancy model) and also the cluster with the nodes on which the components will be deployed. The author of the AMF configuration generation defines a UML profile including a UML model for capturing the HA requirements. In this UML model, the HA requirements are captured by entities called template[2]. The following describes those template entities:

- **Service Group Template (SGTemp)**: This template captures information about the requirement for the SG that will protect a given set of SIs. This template also specifies the redundancy model needed for protecting the SIs and it also specify the number of SUs the SG will use to protect the SIs. More precisely, this template captures the required number of active, standby and spare SUs. Finally, it groups the SI templates that will represent the SIs to be protected by the resulting SG of this template.

[1] The HA requirements are also referred as the Configuration Requirement (CR) in the authors publications.
[2] They are often called the HA requirement template or simply template for simplicity purpose.

- **Service Instance Template (SITemp)**: This template captures information about SIs to be generated. It specifies the service type (SvcT) of the SIs, the dependency among the SIs, and like the SGTemp, it groups the CSI templates that will represent the CSIs grouped by the generated SIs. Furthermore, it specifies the number of SI to generate and also the number of active and standby assignment each SI of this template will acquire at runtime.
- **Component Service Instance Template (CSITemp)**: This template captures information about the CSIs to be generated. Similarly to the SI template, it specifies the component service type (CST) of the CSIs and how many CSI to generate.
- **Node and Cluster Template**: The Node template and the Cluster template capture the information about the deployment infrastructure. In other words, the requirements for the AMF nodes and the AMF cluster.

### 2.1.2 The Configuration Generator

The configuration generator can only operate once the input is specified. Then, the generation can proceed and if the input was valid, the generator will produce the configuration file in the IMM XML format. The work of the generator is grouped in three main steps; they are described in the following Sub-sections.

1) **Type's selection/creation**: First, the load of SI each SUs is expected to support need to be calculated. Then, starting with the highest defined level of types and templates, the generator will find the types that can support the service requirement from the templates. When a type is missing, it means that there is no restriction on the child types and therefore, the generator will create a type.
2) **Creating the entities and populating their attributes**: Once all the entity types capable of supporting the service are found, the generator will create the needed entities based on those types to support the required level of services. The generator will also create and fill the attributes in regards of the templates.

3) **Distributing the SUs on the Nodes**: Once the entities are created, the SUs will be distributed equally on the nodes assuming the nodes are all identical. Also, if the number of node is sufficient, each of them will contain at maximum one SU of each SG.

Once all of the steps are completed, the generator will produce the XML file based on the IMM XML schema of the specification. Actually, the configuration generator is now known to be capable of generating multiple configurations from the same input (Kanso, 2012; Kanso et al., 2009), and the authors also define an approach for analysing the configuration based on heuristics describing the probability of falling of a component.

### 2.1.3 The Upgrade Campaign Generator

As explained in Sub-section 1.4, the system is expected to evolve without loss of service. So, when the system manager wants to update components, he needs to write an upgrade campaign. Writing an upgrade campaign manually is a complex task and therefore, this is still a time consuming and error prone task. The complexity is relative to the number of entities being updated, the relations between them and also the steps needed to perform the upgrade whit minimum loss of service. Due to the complex nature of the configuration as explained in 1.3, writing the XML upgrade campaign file always imply a high number of entities, relations and steps.

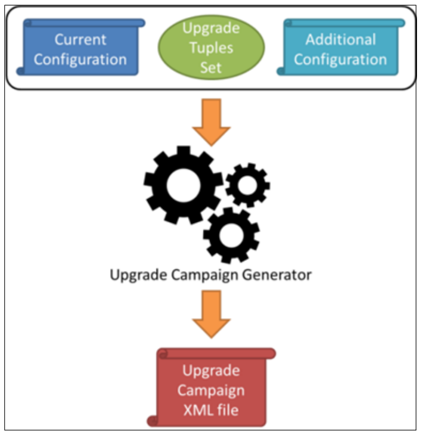


Figure 2.2 Overview of the Upgrade Campaign Generation

Figure 2.2 illustrates an overview of the upgrade campaign. The inputs are the actual configuration of the running cluster, the *Upgrade Tuples Set* and optionally, the configuration containing the entities that are going to be added, if any. After the validation of the input, the generator will proceed and generate the actual upgrade campaign XML file.

The *Upgrade Tuples Set* contains the information about the set of modifications to apply at the current configuration. More precisely, each Upgrade Tuple contains the information about the source entity and target entity, the service group that contains the source and/or will contain the target entities and also, it contains the node where the upgrade will be executed. The *Current Configuration* is the configuration of the running cluster that will be updated; this input is needed because the generator needs to know the configuration in order to manipulate the entities without unnecessary outage of the services. Finally, the *Additional Configuration* includes the entities that will be added to the current configuration. However, this input is optional because not all upgrade scenarios are adding or updating entities, some scenario can simply remove some entities that are no longer needed in the current configuration.

### 2.1.4 ETF UML Domain Models

The work presented in (Salehi, 2012) related to the ETF file discussed in Sub-section 1.5 was a first step towards capturing the domain constraints in a formal way. This results in a UML profile containing a model for the ETF domain. The model is a UML class diagram that captures all possible information about the ETF domain. Plus, the author defined OCL constraints representing the domain constraints initially expressed informally in the specification and in the XML schema. However, those OCL constraints are design for the ETF model and therefore, they can be used for validation only when the ETF file is described whit the ETF model. Given that the work targets mainly the validation and not the automation of the generation, the work did not reduce the complexity of producing an ETF file. This is mainly due to the one to one mapping between the specifications and the model concepts. Considering this work as a first step, the work in this thesis is believed to be its logical evolution leading to the generation of the ETF file allowing the definition of a new approach for generating the middleware configuration. The ETF domain model is used in the new approach presented in the CHAPTER 3.

## 2.2 Other Related Works

In the work from (Szatmári, Kövi, & Reitenspiess, 2008), the authors present a systematic approach for developing application based on the SAF middleware services. This approach is based on a model-driven framework giving the software developer a model for designing an application based on the SAF middleware. The model supports the development of SA-aware application and can generate a generic configuration and a code skeleton based on this type of entity. However, it does not allow the generation of an ETF file, also the generated configuration is not customizable and does not consider the user requirement in term of high availability. Moreover, this approach does not allow the use of commercial off-the-shelf (COTS) which is an important type of application used in clustered solution. In (Kövi & Varró, 2007), another model driven approach (MDA) is used for allowing the software development of SAF middleware compliant applications. However, it does not target the generation of configuration of any type of software, does not consider the user requirement in term of HA and finally, it does not allow the creation of the ETF file.

Several works target the extension of the UML component diagram for different purposes. In (Espindola, Becker, & Zorzo, 2004), the authors extend the component diagram to include the distribution requirements in the early design phase of the software. In (Lu, Halang, & Zhang, 2005), the authors define a component-based UML model to capture the requirements of a real-time system that can later on be transformed into a platform specific model. In (Mahmood & Lai, 2009), the authors present an extension to the UML component diagram allowing the specification and analysis of the stakeholders' requirements. These work relate to our approach from the perspective of extending the UML component diagram to satisfy certain domain requirements, however, none of them target the HA domain, nor the generation of middleware configurations.

# CHAPTER 3

# APPROACH FOR AUTOMATIC CONFIGURATION GENERATION

This chapter addresses the problem of generating the type description file, known as the ETF file in the SAF domain, by proposing a new approach that extends the previous approach for generating the middleware configuration (Gherbi, Kanso, Khendek, Hamou-Lhadj, & Toeroe, 2009; Kanso, 2012; Kanso et al., 2009). The previous approach for generating the configuration (described in Sub-section 2.1) was leaving in the hand of the configuration designer the responsibility of providing an ETF file and therefore, requiring the designer to possess deep domain knowledge. Consequently, there is still a high level of complexity that persists when using the previous approach.

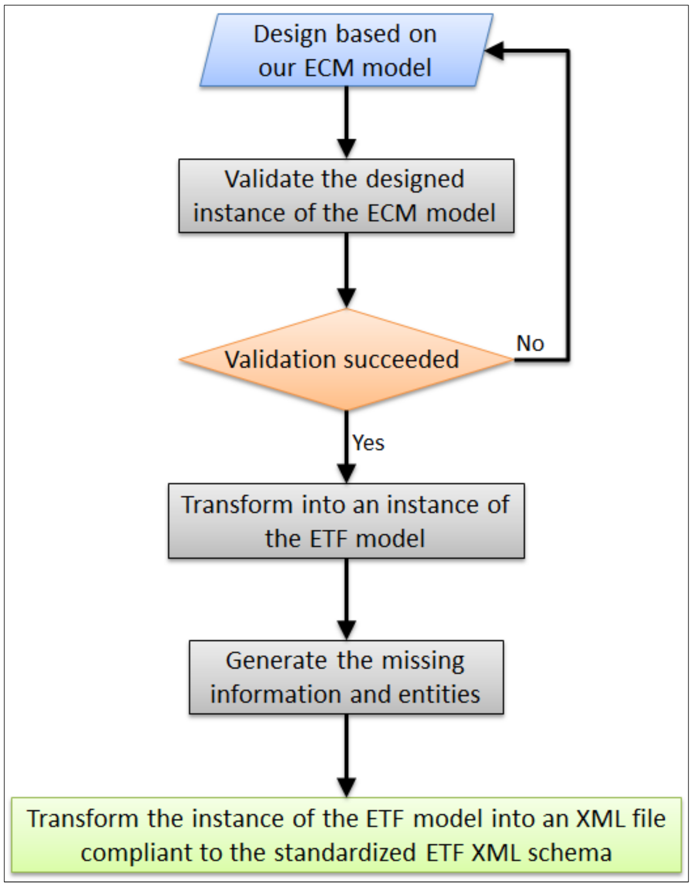

Design based on our ECM model
Validate the designed instance of the ECM model
Validation succeeded
No
Yes
Transform into an instance of the ETF model
Generate the missing information and entities
Transform the instance of the ETF model into an XML file compliant to the standardized ETF XML schema

Figure 3.1 Approach for generating the ETF file

The proposed approach reduces the complexity by capturing the inputs with a graphical modeling language. This language abstracts the complexity of the domain by minimizing the exposure of the user to the concepts and entities of the SAF domain. This abstraction is done by moving to a higher level language that is well known by software developers. Capturing the input with a graphical and domain specific language that abstracts the complexity of the domain enables the automatic generation of the ETF file. Figure 3.1 illustrates an overview of the new approach. In this new approach, the user designs the configuration using a graphical modeling language and he/she validates the design with respect to a set of OCL constraints extracted from the domain. If the validation succeeds, the transformation can proceed and furthermore, the abstracted entities will be inferred from the modeling language semantic. Finally, the process will result in an ETF XML file.

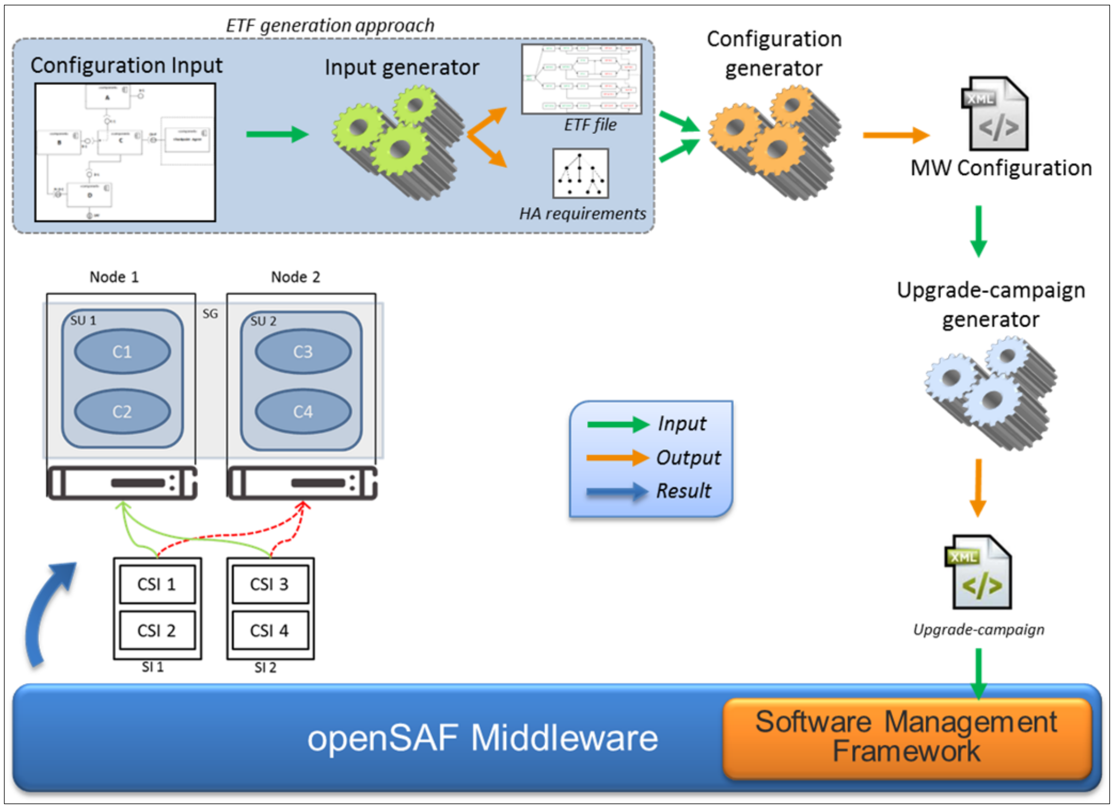


Figure 3.2 New approach for generating the middleware configuration

Figure 3.2 illustrates the new approach for generating the configuration enabled by the automatic generation of the ETF file. The previous approach is extended because the ETF file and the HA requirements are now automatically generated by the input generator that takes its input from the new graphical language. As a consequence, the user is freed from the domain complexity that requires deep domain knowledge. In the next Sub-sections, the different artefacts of the new approach are explained.

## 3.1 Graphical Modeling Language for Expressing Configuration Input

Using a graphical modeling language allows the configuration designer to express the *Configuration Input* at a higher level than the *ETF file* and the *HA requirements*. Furthermore, the main idea is to minimize the exposure of the SAF domain to the configuration designer. However, this new language needs to map the functionality and architecture of the software into the ETF file without exposing to the user the low level details of the ETF domain. While hiding the low level details and complexity, the language must respect the semantic of the ETF file. This semantic is characterised by the association between the two main entities of the ETF file which are the *component types* providing and/or requiring *component service types*. The UML component diagram (UML) is based on the similar semantic where the *Components* provide and/or require *Interfaces*. In the UML component diagram, the *Interfaces* typically represent a functionality provided by a component while the latter one represents the software that may provide and require *interfaces*. Those elements bear similar meaning to the ETF entities. Therefore, the UML *Component* can be mapped to the ETF *Component Type* and the UML *Interfaces* to the *Component Service Type*.

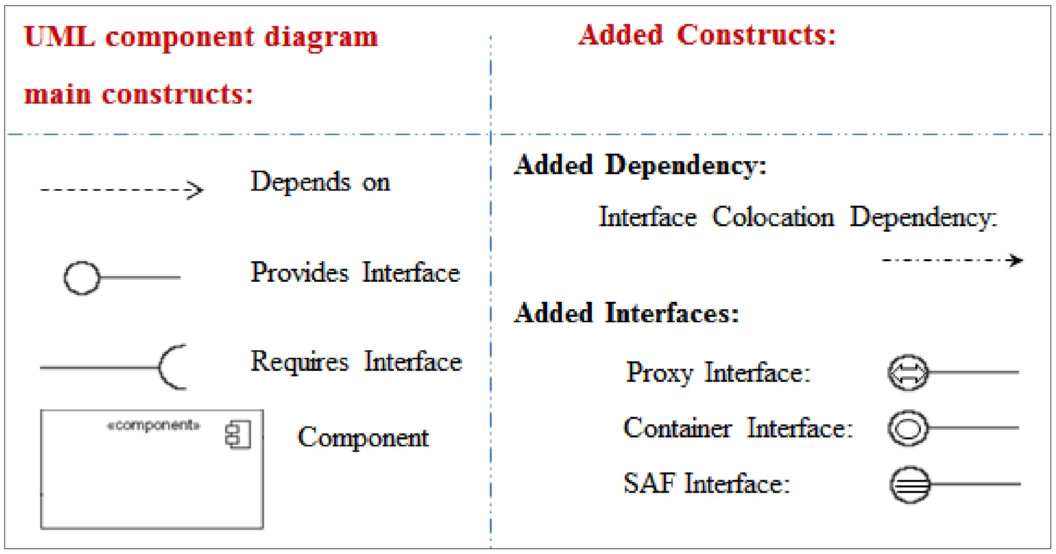


Figure 3.3 Main constructs of the ECM language

The new graphical domain specific language is an extension of the UML component diagram because this is a software friendly language whit similar semantic and elements. However, the UML component diagram expressiveness is limited because the information captured with this diagram only captures the association between basic *components* and *interfaces*. On the other hand, the ETF file also captures information related to the high availability offering of the different component types. Therefore, we extended the UML component diagram in order to include the constructs that capture the minimum information required for generating the ETF file. Since the language will be used in an approach where the ETF file is generated, it is named the ECM language (ETF Component Model). Figure 3.3 illustrates the main constructs of the ECM language. The new constructs are described in the next Sub-sections.

### 3.1.1 Interface Colocation Dependency (ICD)

This association captures the colocation relationship between two Component Types. More precisely, the ICD means that the Component Types related by this association need to work in close relation in order to provide the Component Service Types. For instance, the collocated dependency association in Figure 3.4 can be read as the follows: The CT_1 needs the CT_2 to provide the CST_B in a collocated environment in order to provide the CST_A.

Having CT_1 and CT_2 in the same environment means they should run in the same server. Eventually, the generator will interpret this association as the CT_1 need the CST_B in order to provide the CST_A and in addition, the CT_1 needs to be in the same SUT than the CT_2.

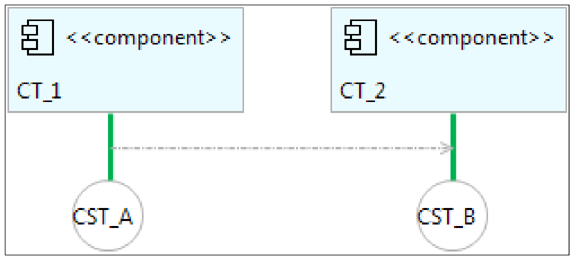


Figure 3.4 Example of the collocation dependency from the ECM language

Assuming than the Component Types are sharing Service Unit Types only when they need to work in close relation, it is possible to remove the notion of Service Unit Type from the language. Therefore, the complexity is greatly reduced because the user can focus on the dependency of the software without considering the complex constraints of the SUT concept.

### 3.1.2 Interfaces

There are four categories of interfaces in the ECM language. They represent different kinds of CSTs and capture specific information. The following explain them in detail, but the attributes of those entities are discussed in Sub-section 3.2:

- **Regular Interface**: This interface captures the type of service a component type can provide to the user or to another component type. Typically, this interface represents a service non-related to the SAF middleware and it can be provided by all component types of the language. For example, with a CT representing an Apache HTTP server type, the interface representing the HTTP service type will be a *normal interface*. Furthermore, the generator will transform this interface into a CST representing the HTTP service type.

- **Proxy Interface**: This interface captures information about the CST representing the workload of a *proxy CT* when proxying a given *proxied CT*. In other word, a CT providing a Proxy Interface is automatically considered as a *Proxy Component Type* and the requiring CT of this interface is also automatically considered as a *Proxied Component Type*.
- **Container Interface**: This interface captures the information about the CST of the containing service type a given container CT gives to its contained CT. Like the proxy interface, the component type providing the *Container Interface* is automatically considered as a *Container Component Type* and the requiring component type is also automatically considered as a *Contained Component Type*.
- **SAF Interface**: This interface does not capture information about a CST but, denotes that a given CT is implementing the SAF AMF APIs. In other words, when a given CT is providing this interface, it implies that it is an SA-aware CT.

With those interfaces, some rules are added to the basic semantic of the UML component diagram in order to comply with the domain constraints. For example, a Component Type cannot require a Container Interface while providing a Proxy Interface meaning that a Component Type cannot be both Contained and Proxy. Another important rule is that in order to provide a Proxy or/and a Container interface, the Component Type must be SA-aware and therefore, must provide the SAF interface. Those rules are further discussed in Sub-section 3.3 because they are implemented as OCL constraints.

### 3.1.3 Component Type

This entity is named Component in the ECM language in order to avoid confusion since this is how it is named in the UML component diagram. However, in this thesis this entity is always referred to as the Component Type because of its equivalent entity in the ETF file. The *component type* represents a type of software that can be deployed in a cluster and it contains the attributes characterizing the implementation. It can provide and/or require interfaces and based on those associations, the type of the component will be determined.

### 3.1.4 Provide and Require Associations

The provide association between a component and an interface in the UML component diagram does not capture any information about how the interface is provided by the component is terms of capacity, limitation and dependency. However, this information is mandatory in the ETF file and it cannot be simply inferred. This information is known exclusively by the software developer and/or the configuration designer. In order to capture this information, the association itself is extended by having some attributes essential for the generator when it will create the proper association in the ETF file. The basic semantic of the associations provide and require remains the same as the one in the UML component diagram. In other words, in the ECM language, the components are providing and/or requiring interfaces. The require association however, have no need to capture more information than the UML component diagram one and therefore, is simply reused.

### 3.1.5 HA Requirements and Deployment Information

The HA requirements entities capture information about the level of availability of the service to be provided. The concept is very similar to the original one explained in Sub-section 2.1.1. However, it has been adapted to better fit the semantic of the language. The *CSITemp* is represented by the *interface template*. Also, the user specifies only the association between the *interface* and the *interface template* because the concepts of the SvcT and SUT are not part of the language for simplification purpose. On the other hand, the user still needs to specify the entities from the concept of the *service instance template* and the *service group template* because they capture important information that cannot be inferred. Those entities do not have an association with their corresponding types because they do not exist in the ECM language since they have been abstracted. The template entities will be transformed into an instance of the original template UML model after the generation of the ETF file. Therefore, the missing association will be filled with the corresponding types created for the ETF file.

### 3.1.6 An Example to Illustrate the ECM Language

The added constructs have been carefully designed in order to comply with the main semantic of both the domain and the UML component diagram while being friendly to the user. Based on the semantic, the generator can infer the appropriate missing entities and thus, build a complete ETF file and HA requirement template.

Figure 3.5 illustrates a complete example of a HA system described by the ECM language. In this example there are four different *components* and each of them is providing an *interface* representing a specific type of service. The *component* named HTTP-server provides to the client a HTTP service and requires the APP-server to handle the dynamic content. The APP-server *component* needs to access the database (DB) for managing the persistent data and it needs a collocated access for performance purpose and also because the APP-server is not meant to provide its service without the database. Furthermore, the DB *component* is providing HA functions but, it does not implement the SAF APIs. Therefore, if the user wants to make those functions available to the middleware, he need to use a proxy for mediating the communication between AMF and the DB *component*. This is why the DB *component* is proxied and thus, requiring a proxy *interface* provided by the DB-proxy *component*. However, each *interface* (HTTP-i, APP-i, DB-i and P-i) is respectively representing the feature of their associated *component*. The APP-server and DB-proxy *components* are implementing the SAF AMF API making them SA-aware *component* and therefore, they provide the *SAF interface*. The four *components* are grouped in the *software bundle* named SWB and they participate in a *service group type* that will protect the provided services with a redundancy model of 2N.

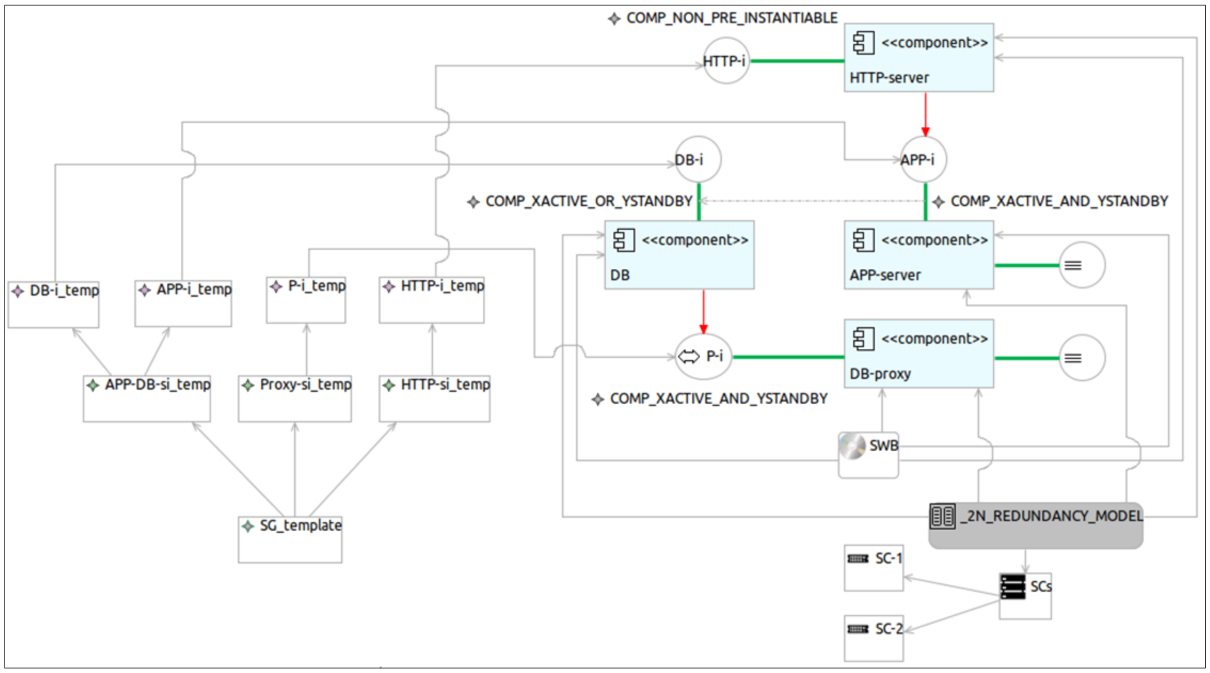


Figure 3.5 Usage example of the ECM language

The *nodes* SC-1 and SC-2 on which the *service group type* can be deployed are grouped into the *node group* named SCs. Those entities are not the same as the one in the template UML model because they capture the information about the actual deployment of the software instead of information about what kind of node is needed. This way of capturing the deployment information is used because the user is expected to know on which node the software can be deployed. Furthermore, the user can specify the level of availability of the services with the templates which is used by the configuration generator for generating the AMF configuration. In the example illustrated by Figure 3.5, the user wants to generate the AMF configuration and this is why the desired level of service is captured by the templates at the left of the figure. As explained in Sub-section 3.1.5, the user only needs to specify the association between the *interface template* and the *interface* in order to specify the type of service is described by the template.

## 3.2 The ECM Metamodel Definition

Behind the ECM language explained in Sub-section 3.1, there is a metamodel capturing the relevant information about the system description for generating the middleware configuration. This *ETF Component Metamodel* (ECM) is composed of five packages, the *Components*, the *Interfaces*, the *RequirementTemplates*, the *Cluster* and the *DataTypes* as shown in Figure 3.6. The next Sub-sections are describing the different packages and their class.

### 3.2.1 Components Package

The *Components* package groups the *Component* class and the closely related classes. There are several specialisations of the *Component* class for representing the different types of *Component* that can be configured in the system. However, the entities like the *Service Group Type* or the *Software Bundle* are part of this package because they are strongly related to the Components. Furthermore, the component hierarchy is a simplification of the ETF metamodel because it is possible to infer certain categorizations based on the interface provided by the component, e.g. there is no need for a *ProxyComponent* or *ContainerComponent* because they do not have unique attribute and it is possible to infer them based on the interface provided by an *IndependentComponent*. The main classes of this package are described in the following:

- **Component**: This class specifies the attributes that any AMF managed component must have regardless of its category. Those attributes captures information about the default recovery action the middleware must take for this component in case of failure. They also capture the command and their arguments used by the middleware for managing the component life cycle. Plus, it captures the timeouts regarding those commands after which the middleware will assume the component is faulty. Additionally, the maximum and minimum number of components of this type that can be grouped in a specific SU is captured by the attributes. This entity is having the very important provide and require associations to the interface representing the featured services of a component.

Furthermore, the component is specialized by the following classes that represent the different types of component:

- **NonProxiedNonSaAwareComponent**: This class represents the *Non-Proxied, Non-SA-Aware* entity type. When managing this particular type of entities, the middleware needs to know the command for instantiating, terminating and cleaning the component in term of life cycle. Hence, this class specifies attributes regarding those commands and the arguments they may need.
- **ProxiedComponent**: This class represents the *Proxied* entity type. Since this type of entity is not directly managed by the middleware, but rather with a proxy mediating the communication, the middleware only needs to know the command for cleaning the instance in case the proxy fails to mediate the terminating command. Plus, there is an attribute for the quiescing timeout and also for the type of instantiation the component supports.
- **SaAwareComponent**: This class represents the *SaAware* component types. Furthermore, components of this type are associated with a SAFInterface and this class is specialized by two other classes characterizing more precisely components of this type.
  - **ContainedComponent**: This class represents the *ContainedComponent* which is also a *SaAwareComponent*. However, there is an attribute for the quiescing timeout.
  - **IndependentComponent**: This class represents an independent *SaAwareComponent* meaning that a component of this type does not require any *proxy* or *container* interface. This class specifies the attributes about the commands and arguments regarding the instantiation and the cleaning of a component of this type. And plus, there is an attribute for the quiescing timeout.

- **ServiceGroupType**: This class is very similar to the *ServiceGroupType* of the ETF metamodel; the difference is that it includes *components* instead of grouping *SUT* and it is not grouped into any *application type* since this entity does not exist in this metamodel. During the generation of the missing ETF entities, the association to the component of this class will be adapted to the corresponding created SUT. Furthermore, this class specifies attributes regarding how the availability of the participating components and SUT will be protected with the specified redundancy model.

- **SoftwareBundle**: This class is a one-to-one mapping to the ETF SoftwareBundle. They are both composing components that can be deployed into a HA system. This class specifies attributes about commands for managing the deployment of the grouped components.

### 3.2.2 Interfaces Package

This package contains the *Interface* class at the top of the hierarchy. The *Interface* class is specialized by several classes that mainly characterize different types of abstracted service. While the ETF entity corresponding to the *ServiceInterface* class is the *Component Service Type*, the semantic of a *component* providing an *Interface* is also used to capture certain characteristic regarding the type of a component, i.e. when a component is providing the *SAFInterface*; it means that the component is *SaAware* even if the *SAFInterface* itself does not represent a CST. The main classes of this package are described in the following:

- **Interface**: This class only represents the concept of the interface and does not abstract any workload service. Nonetheless, this class has specializations that either represents a workload or a characteristic of the component implementation.
    - **ServiceInterface**: This class specifies attributes that represent a general workload at runtime. Furthermore, the content of this class will be transformed in a CST. There are three specializations to this class:
        - **RegularInterface**: This class represents the runtime services that are not related to the middleware. For example, a database has a SQL runtime service and therefore, the component representing this database should provide a *RegularInterface* describing the SQL runtime service.
        - **ProxyInterface**: This class represents the SAF middleware specific runtime service of a proxy proxying another component. When providing this interface, the *IndependentComponent* will be interpreted as a

*ProxyComponent* by the generator and it will be expected to mediate the communication between the middleware and the components requiring its *ProxyInterface*. Furthermore, the component requiring this interface needs to be a *ProxiedComponent*.

    - **ContainerInterface**: This class represents the SAF middleware specific runtime service of a container containing another component. When providing this interface, the *IndependentComponent* will be considered as a *ContainerComponent* by the generator and it will be expected to provide the execution environment used by the components requiring its *ContainerInterface* (this is regarded as containing because the *ContainedComponent* can only be executed inside the container specific execution environment). However, the component requiring this interface needs to be a *ContainedComponent*.

- **SAFInterface**: This class represents the SaAware features. In other words, providing this interface means that the component implements the SAF AMF API and therefore, providing the runtime service of supporting all the AMF life cycle features.

### 3.2.3 RequirementTemplates Package

This package encloses the classes that feature the attributes used to capture the level of availability desired by the user. The classes of this package are very similar to the one found in the original model (Salehi, 2012). However, because of the semantic of the component diagram, the associations of those classes were adapted to better fit the ECM metamodel. The *ServiceGroupTemp* class specifies attributes about the number of *SUs* that will be used and the redundancy model used to protect the service. In other words, they describe the requirement for a *ServiceGroup* and the number of *SU* to be deployed. Furthermore, it is composed of *ServiceTemp* class that will describe the type of services the *ServiceGroup* will protect. More precisely, the *ServiceTemp* class specifies attributes about the level of service the *SUs* will be expected to provide. And finally, the *InterfaceTemp* class composing the

*ServiceTemp* class specifies attributes about the level of service the *components* of the *SUs* will be expected to provide. All of those template classes contain names that will be used by the generator for creating the entities of the configuration like the SGs, SIs, SUs, CSIs and components.

### 3.2.4 Cluster Package

This package encloses two classes specifying attributes about the nodes on which the software is actually deployed. The attributes of the *Node* class describe the capacity of the node plus how the middleware should react in case of failure at the node level. However, the *Node* classes are grouped by the *NodeGroup* class, which is associated with the *ServiceGroupType* class of the *Components* package. This association specifies on which group of *nodes* (*NodeGroup*) the *components* of a *ServiceGroupType* can be deployed.

### 3.2.5 DataTypes Package

The classes of this package help the user to better specify the information he wish to specify for the different class. Besides this, they are specific to the domain and mainly reflect the data types specified in the SAF specifications. An example of this is the enumeration that captures the redundancy models we can have in this domain.

### 3.2.6 ComponentService Association Class

This is an association class between the *Component* and the *Interface*. This class is also involved in the reflective association that captures the collocation dependency. However, the main responsibility of this class is to capture information about how the *Component* is capable of providing the given *Interface* with the *CompCapabilityModel* attribute. Also, the collocation dependency is captured by the reflective association of this class.

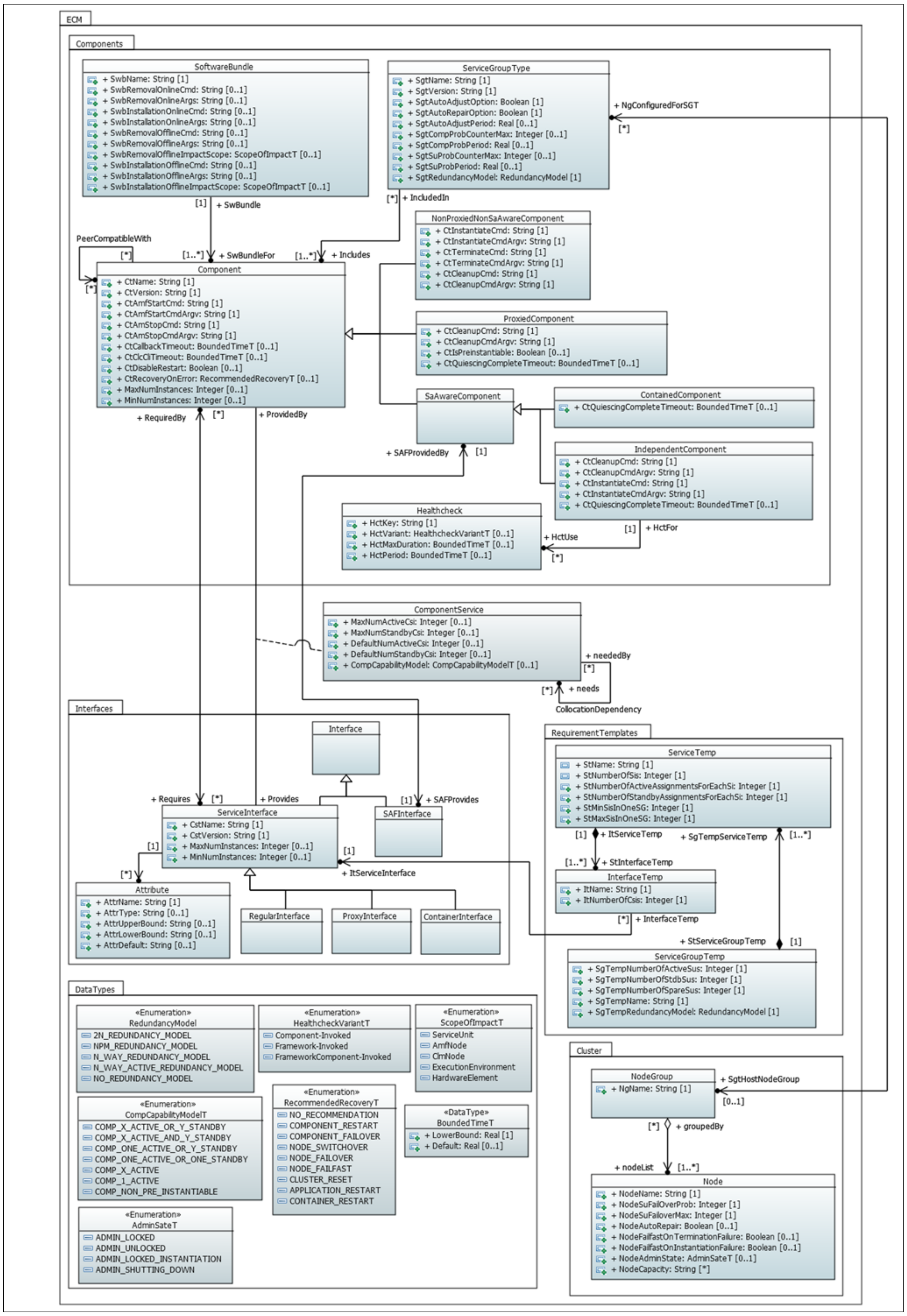
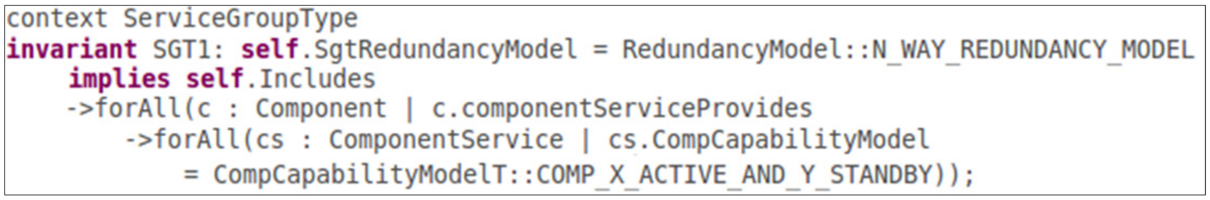

ECM
Components
SoftwareBundle
+ SwbName: String [1]
+ SwbRemovalOnlineCmd: String [0..1]
+ SwbRemovalOnlineArgs: String [0..1]
+ SwbInstallationOnlineCmd: String [0..1]
+ SwbInstallationOnlineArgs: String [0..1]
+ SwbRemovalOfflineCmd: String [0..1]
+ SwbRemovalOfflineArgs: String [0..1]
+ SwbRemovalOfflineImpactScope: ScopeOfImpactT [0..1]
+ SwbInstallationOfflineCmd: String [0..1]
+ SwbInstallationOfflineArgs: String [0..1]
+ SwbInstallationOfflineImpactScope: ScopeOfImpactT [0..1]
ServiceGroupType
+ SgtName: String [1]
+ SgtVersion: String [1]
+ SgtAutoAdjustOption: Boolean [1]
+ SgtAutoRepairOption: Boolean [1]
+ SgtAutoAdjustPeriod: Real [0..1]
+ SgtCompProbCounterMax: Integer [0..1]
+ SgtCompProbPeriod: Real [0..1]
+ SgtSuProbCounterMax: Integer [0..1]
+ SgtSuProbPeriod: Real [0..1]
+ SgtRedundancyModel: RedundancyModel [1]
+ NgConfiguredForSGT
[*]
[1] + SwBundle
[*] + IncludedIn
PeerCompatibleWith
[*]
[1..*] + SwBundleFor
[1..*] + Includes
[*]
Component
+ CtName: String [1]
+ CtVersion: String [1]
+ CtAmfStartCmd: String [1]
+ CtAmfStartCmdArgv: String [1]
+ CtAmStopCmd: String [1]
+ CtAmStopCmdArgv: String [1]
+ CtCallbackTimeout: BoundedTimeT [0..1]
+ CtClcCliTimeout: BoundedTimeT [0..1]
+ CtDisableRestart: Boolean [0..1]
+ CtRecoveryOnError: RecommendedRecoveryT [0..1]
+ MaxNumInstances: Integer [0..1]
+ MinNumInstances: Integer [0..1]
NonProxiedNonSaAwareComponent
+ CtInstantiateCmd: String [1]
+ CtInstantiateCmdArgv: String [1]
+ CtTerminateCmd: String [1]
+ CtTerminateCmdArgv: String [1]
+ CtCleanupCmd: String [1]
+ CtCleanupCmdArgv: String [1]
ProxiedComponent
+ CtCleanupCmd: String [1]
+ CtCleanupCmdArgv: String [1]
+ CtIsPreinstantiable: Boolean [0..1]
+ CtQuiescingCompleteTimeout: BoundedTimeT [0..1]
SaAwareComponent
ContainedComponent
+ CtQuiescingCompleteTimeout: BoundedTimeT [0..1]
IndependentComponent
+ CtCleanupCmd: String [1]
+ CtCleanupCmdArgv: String [1]
+ CtInstantiateCmd: String [1]
+ CtInstantiateCmdArgv: String [1]
+ CtQuiescingCompleteTimeout: BoundedTimeT [0..1]
+ RequiredBy
[*]
+ ProvidedBy
+ SAFProvidedBy
[1]
Healthcheck
+ HctKey: String [1]
+ HctVariant: HealthcheckVariantT [0..1]
+ HctMaxDuration: BoundedTimeT [0..1]
+ HctPeriod: BoundedTimeT [0..1]
+ HctUse
[*]
[1] + HctFor
ComponentService
+ MaxNumActiveCsi: Integer [0..1]
+ MaxNumStandbyCsi: Integer [0..1]
+ DefaultNumActiveCsi: Integer [0..1]
+ DefaultNumStandbyCsi: Integer [0..1]
+ CompCapabilityModel: CompCapabilityModelT [0..1]
+ neededBy
[*]
[*] + needs
CollocationDependency
Interfaces
Interface
+ Requires
[*]
+ Provides
[1] + SAFProvides
ServiceInterface
+ CstName: String [1]
+ CstVersion: String [1]
+ MaxNumInstances: Integer [0..1]
+ MinNumInstances: Integer [0..1]
SAFInterface
[1]
[*]
[1]
+ ItServiceInterface
Attribute
+ AttrName: String [1]
+ AttrType: String [0..1]
+ AttrUpperBound: String [0..1]
+ AttrLowerBound: String [0..1]
+ AttrDefault: String [0..1]
RegularInterface
ProxyInterface
ContainerInterface
RequirementTemplates
ServiceTemp
+ StName: String [1]
+ StNumberOfSis: Integer [1]
+ StNumberOfActiveAssignmentsForEachSi: Integer [1]
+ StNumberOfStandbyAssignmentsForEachSi: Integer [1]
+ StMinSisInOneSG: Integer [1]
+ StMaxSisInOneSG: Integer [1]
[1] + ItServiceTemp
+ SgTempServiceTemp
[1..*]
[1..*] + StInterfaceTemp
InterfaceTemp
+ ItName: String [1]
+ ItNumberOfCsis: Integer [1]
[*] + InterfaceTemp
+ StServiceGroupTemp
[1]
ServiceGroupTemp
+ SgTempNumberOfActiveSus: Integer [1]
+ SgTempNumberOfStdbSus: Integer [1]
+ SgTempNumberOfSpareSus: Integer [1]
+ SgTempName: String [1]
+ SgTempRedundancyModel: RedundancyModel [1]
DataTypes
«Enumeration»
RedundancyModel
2N_REDUNDANCY_MODEL
NPM_REDUNDANCY_MODEL
N_WAY_REDUNDANCY_MODEL
N_WAY_ACTIVE_REDUNDANCY_MODEL
NO_REDUNDANCY_MODEL
«Enumeration»
HealthcheckVariantT
Component-Invoked
Framework-Invoked
FrameworkComponent-Invoked
«Enumeration»
ScopeOfImpactT
ServiceUnit
AmfNode
ClmNode
ExecutionEnvironment
HardwareElement
«Enumeration»
CompCapabilityModelT
COMP_X_ACTIVE_OR_Y_STANDBY
COMP_X_ACTIVE_AND_Y_STANDBY
COMP_ONE_ACTIVE_OR_Y_STANDBY
COMP_ONE_ACTIVE_OR_ONE_STANDBY
COMP_X_ACTIVE
COMP_1_ACTIVE
COMP_NON_PRE_INSTANTIABLE
«Enumeration»
RecommendedRecoveryT
NO_RECOMMENDATION
COMPONENT_RESTART
COMPONENT_FAILOVER
NODE_SWITCHOVER
NODE_FAILOVER
NODE_FAILFAST
CLUSTER_RESET
APPLICATION_RESTART
CONTAINER_RESTART
«DataType»
BoundedTimeT
+ LowerBound: Real [1]
+ Default: Real [0..1]
«Enumeration»
AdminSateT
ADMIN_LOCKED
ADMIN_UNLOCKED
ADMIN_LOCKED_INSTANTIATION
ADMIN_SHUTTING_DOWN
Cluster
NodeGroup
+ NgName: String [1]
+ SgtHostNodeGroup
[0..1]
[*] + groupedBy
+ nodeList
[1..*]
Node
+ NodeName: String [1]
+ NodeSuFailOverProb: Integer [1]
+ NodeSuFailoverMax: Integer [1]
+ NodeAutoRepair: Boolean [0..1]
+ NodeFailfastOnTerminationFailure: Boolean [0..1]
+ NodeFailfastOnInstantiationFailure: Boolean [0..1]
+ NodeAdminState: AdminSateT [0..1]
+ NodeCapacity: String [*]

Figure 3.6 ECM Metamodel

## 3.3 The OCL Constraints from the SAF Domain

The language semantic captured by the ECM metamodel covers the basis of the SAF domain where the *Component* entity is expected to provide and/or require some *Component Service Instance* entity. However, they do not capture all the constraints of the domain expressed informally in the very large SAF specifications (SAForum). In fact, one of the objectives of this work is the validation of the user input and therefore, the metamodel instances represent an input validated against the domain constraints. This is why the domain constraints hiding in the SAF specifications where deeply analyzed, extracted and expressed in the OCL constraint language. However, as explained in Sub-section 2.1.4, the authors of the ETF model have already designed OCL constraints for validation purpose. The OCL constraints defined for the ETF model (Salehi, 2012) were not directly reusable on our metamodel, hence we had to re-adapt them in this project to fit the ECM metamodel. By annotating the metamodel with such constraints, any of its instances can be validated against the domain constraints. Due to the lack of space, we will illustrate in details some important OCL constraints and leave out dozens of other constraints:

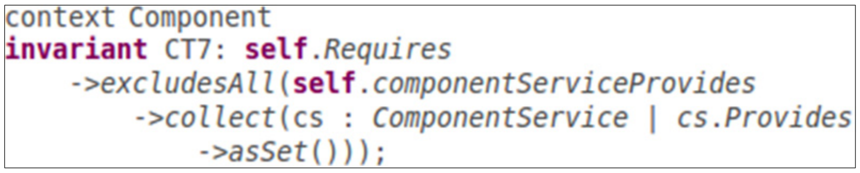
In the SAF domain, the *Component Types* participating in a given *Service Group Types* with the N-Way redundancy model must support the *X_active_and_Y_standby* capability model as seen in Table 1. One way of expressing this constraint in OCL is by looking at all the *Components* associated to the *ServiceGroupType* and check in the association class *ComponentService* if the capability model is the correct one. The Figure 3.7 illustrates the OCL resulting from this constraint.



```
context ServiceGroupType
invariant SGT1: self.SgtRedundancyModel = RedundancyModel::N_WAY_REDUNDANCY_MODEL
    implies self.Includes
    ->forAll(c : Component | c.componentServiceProvides
        ->forAll(cs : ComponentService | cs.CompCapabilityModel
            = CompCapabilityModelT::COMP_X_ACTIVE_AND_Y_STANDBY));
```

Figure 3.7 OCL about the redundancy model

Since the validation is expected to cover all aspects of the domain, the OCL must also cover more intuitive domain constraints. One of those constraints is that a given *Component* must not *provide* and *require* the same *Interface*. The resulting OCL constraint is illustrated in Figure 3.8.

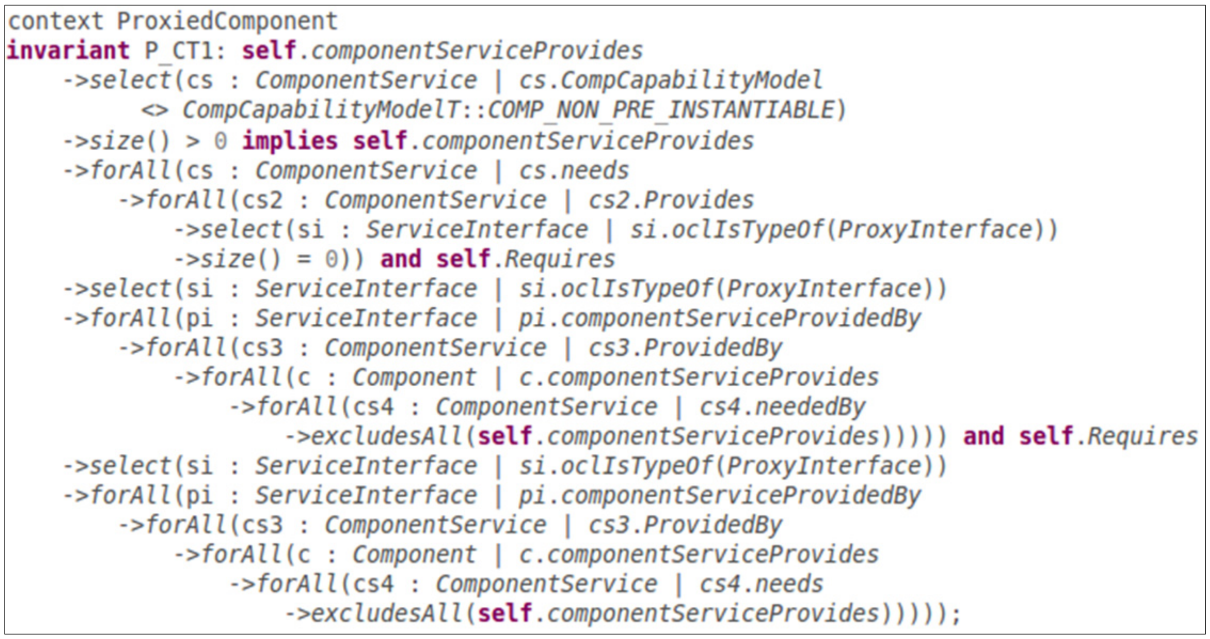


```
context Component
invariant CT7: self.Requires
    ->excludesAll(self.componentServiceProvides
        ->collect(cs : ComponentService | cs.Provides
            ->asSet()));
```

Figure 3.8 OCL about the Component provide/require association

The different specialisations of the *Component* class also have their own constraints. The user may know the basis of them, but being aware of such constraints requires deep knowledge about the SAF domain specifications. Those constraints can be considered as small details by the specification readers, but if they are not respected, the configuration will not be valid and consequently, the system will not provide the expected level of service.

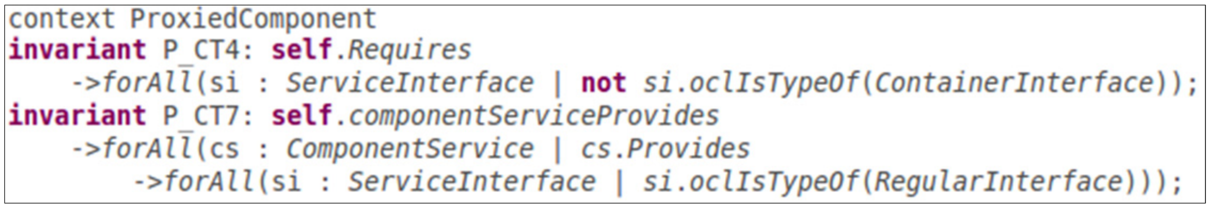
An example of a constraint that can easily be missed is the one illustrated in the Figure 3.9. It is checking that a *pre-instantiable ProxiedComponent* is not sharing the same SUT than its *ProxyComponent*. However, this constraint is complicated to check in the ECM metamodel because this is how the *Interfaces* are provided that determines if the *Component* is pre-instantiable or not. More precisely, this information is featured by the *ComponentService* association class (see Sub-section 3.2.6). Another detail that make the constraint hard to check is that the ECM metamodel does not have SUT for abstraction purpose. The SUTs will be generated based on the *collocation dependency* and therefore, the OCL constraint must first check if the *ProxiedComponent* is pre-instantiable and then, check if its *ProxyComponent* is among the collocated *Components*, if any.



```
context ProxiedComponent
invariant P_CT1: self.componentServiceProvides
    ->select(cs : ComponentService | cs.CompCapabilityModel
          <> CompCapabilityModelT::COMP_NON_PRE_INSTANTIABLE)
    ->size() > 0 implies self.componentServiceProvides
    ->forAll(cs : ComponentService | cs.needs
        ->forAll(cs2 : ComponentService | cs2.Provides
            ->select(si : ServiceInterface | si.oclIsTypeOf(ProxyInterface))
            ->size() = 0)) and self.Requires
    ->select(si : ServiceInterface | si.oclIsTypeOf(ProxyInterface))
    ->forAll(pi : ServiceInterface | pi.componentServiceProvidedBy
        ->forAll(cs3 : ComponentService | cs3.ProvidedBy
            ->forAll(c : Component | c.componentServiceProvides
                ->forAll(cs4 : ComponentService | cs4.neededBy
                    ->excludesAll(self.componentServiceProvides))))) and self.Requires
    ->select(si : ServiceInterface | si.oclIsTypeOf(ProxyInterface))
    ->forAll(pi : ServiceInterface | pi.componentServiceProvidedBy
        ->forAll(cs3 : ComponentService | cs3.ProvidedBy
            ->forAll(c : Component | c.componentServiceProvides
                ->forAll(cs4 : ComponentService | cs4.needs
                    ->excludesAll(self.componentServiceProvides)))));
```

Figure 3.9 OCL about the ProxiedComponent pre-instantiable attribute

In the specification it is explained that the *Component* cannot be both proxied and contained at the same time. Concretely, in the ECM metamodel it means that a *ProxiedComponent* cannot require and/or provide a *ContainerInterface*. Figure 3.10 illustrates this constrains with two different OCLs.



```
context ProxiedComponent
invariant P_CT4: self.Requires
    ->forAll(si : ServiceInterface | not si.oclIsTypeOf(ContainerInterface));
invariant P_CT7: self.componentServiceProvides
    ->forAll(cs : ComponentService | cs.Provides
        ->forAll(si : ServiceInterface | si.oclIsTypeOf(RegularInterface)));
```

Figure 3.10 OCL about the ProxiedComponent interfaces

## 3.4 ETF Generation Based on Model Transformation

The ECM language explained in Sub-section 3.1 abstracts the complexity of the domain when defining the middleware configuration because the entities that can be inferred are explicitly expressed using the language. They can be inferred based on the dependency between the entities of the ECM model (i.e. an instance of the ECM metamodel). Furthermore, the missing entities are generated based on three model-to-model transformations (Sendall & Kozaczynski, 2003). Since the target was to extend the previous approach for generating the middleware configuration, the outputs of this generation process is the ETF file expressed in XML and the HA requirements (i.e. the templates).

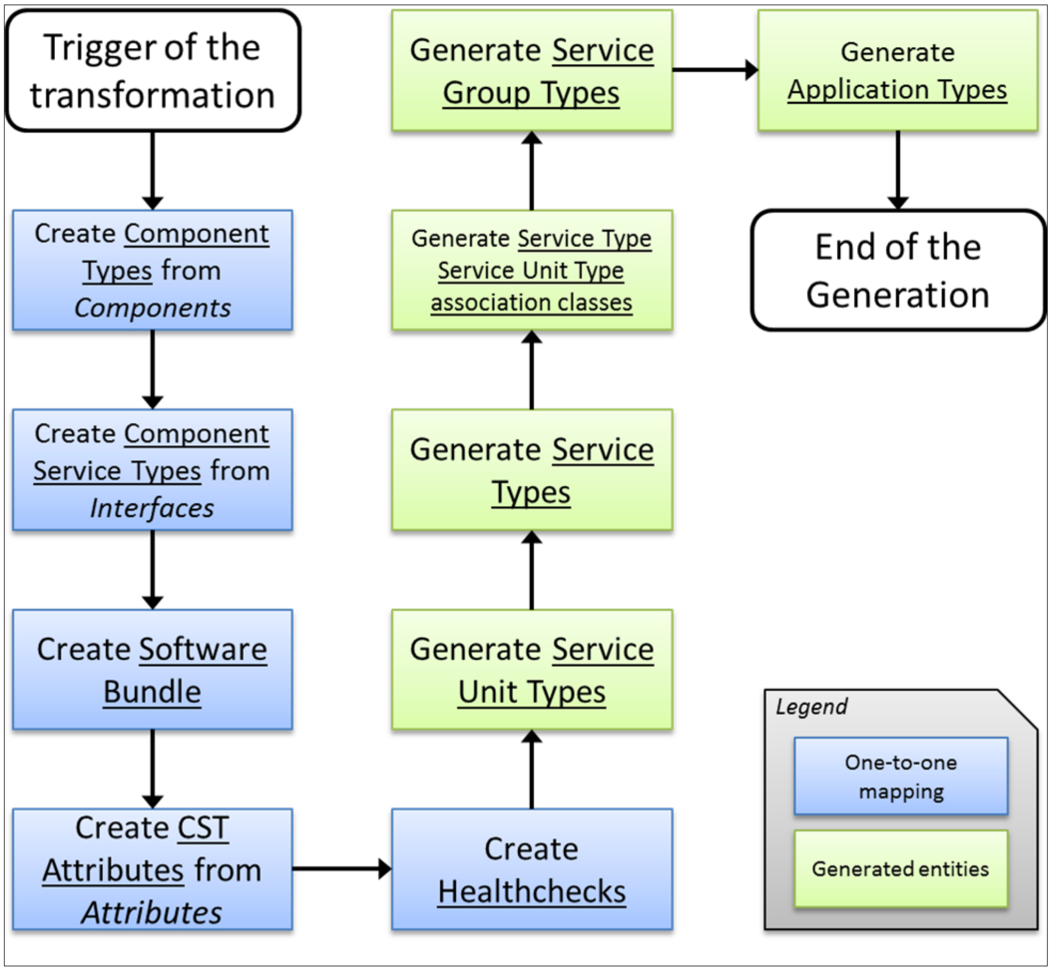

Trigger of the transformation
Create Component Types from Components
Create Component Service Types from Interfaces
Create Software Bundle
Create CST Attributes from Attributes
Create Healthchecks
Generate Service Unit Types
Generate Service Types
Generate Service Type Service Unit Type association classes
Generate Service Group Types
Generate Application Types
End of the Generation
Legend
One-to-one mapping
Generated entities

Figure 3.11 Transformation steps of the algorithm

In the first transformation, an instance of the ETF domain model is generated. As illustrated by Figure 3.11, from a validated ECM model the first steps are the generation of the entities that have a one-to-one mapping: the *Components*, *Interfaces*, *Software bundles*, *Attributes* and *Healthchecks*. They will be respectively transformed into *Component types*, *Component* *service types*, *Software bundles*, *CST Attributes* and *Healthcheck* of the ETF domain model. However, at this stage it is not possible to fill all the association because some entities are not yet created. Therefore, a mapping between the ECM entity and the created ETF entity type is saved for further completion of the associations in a further step.

The second step is to generate the types that are not included in the ECM metamodel. This is where the missing entities are inferred based on the dependency in the instance of the ECM metamodel (i.e. the ECM model). The Colocation Dependency in the ECM metamodel is not only mapped into the *Component type* dependency that is specified in ETF domain model, but it also indicates that the sponsor and the dependent must be grouped in the same *Service unit type*. Therefore, this is based on this information that the *Service unit type* is generated. Figure 3.12 illustrates the algorithm used for creating the SUT.

```
FOR each Component of the ECM model
        IF Component is not grouped by any SUT THEN
                INIT Components
                ADD Component to Components
                CALL findAllCollocatedComp with Components, Component
                INIT SUT
                ADD Components to SUT.grouping
                ADD SUT to ETF model
        ENDIF
END FOR
```

Figure 3.12 Service Unit Type creation algorithm

Actually, the *findAllCollocatedComp* recursive function will build a list of *Components* related together by navigating through all the *Components* of the Collocation Dependency association. The *Components* found by the function, along with the starting *Component*, will be grouped in the newly created *SUT*.

Once the *SUT* is created, the *SvcT* is generated based on the same information. However, the *SvcT* is not directly associated with *CTs* but rather with its *CSTs*. Figure 3.13 illustrates the algorithm used for creating the *SvcT* and finding the corresponding *CSTs* based on the *Components* grouped by the *SUT*.

```
FOR each SUT of the ETF model
    INIT SvcT
    FOR each CompType of the SUT
        FOR each CST of the CompType
            ADD the CST to the SvcT
        END FOR
    END FOR
    ADD SvcT to the ETF model
    SAVE the SvcT and the SUT in a map
END FOR
```

Figure 3.13 Service Type creation algorithm

With both the *SUTs* and *SvcTs* created, the association class between them is created. The corresponding *SUT* of a given *SvcT* was saved into a map as seen in Figure 3.13 in order to enable the creation of the association class in a separate step. This separation is needed because all *SUTs* and *SvcTs* need to be created before setting the *require* association of the association class. They need to be created first because if the association class was created at the same time of the *SUT* or the *SvcT*, it would be possible that a require association points to an instance that has not been created yet. This association represents a dependency that is mapped to the normal require association of the ECM metamodel and the middleware needs this association to be specified in order to manage the dependency between *SvcTs*. Figure 3.14 illustrates the algorithm for the creation of the *SvcTSut* association class.

```
FOR each SUT of the ETF model
    INIT SvcTSut
    FOR each CT grouped by the SUT
        FOR each ServiceInterface corresponding with the CT  require association
            GET the CST corresponding to the ServiceInterface
            INIT requiredSvcts
            FOR each SvcT grouping the CST
                ADD the SvcT to the requiredSvcts
                IF the requiredSvcts contains the SvcT corresponding to the SUT
                    SET requiredSvcts  to empty
                    BREAK
                END IF
            END FOR
            FOR each SvcT of the requiredSvcts
                SET the require association of the SvcTSut with the SvcT
            END FOR
        END FOR
    END FOR
    ADD the SvcTSut to the ETF model
END FOR
```

Figure 3.14 Service Type Service Unit Type association class creation algorithm

In the last step, the *Service group type* is created based on the *Service group type* of the ECM model. This is almost a one-to-one mapping but, in the ECM metamodel the *SGT* is grouping *CTs* and in the ETF model it is grouping *SUTs*. Therefore, based on the map pairing the *Component* of the ECM model and its corresponding *CT* of the ETF model, the algorithm is going to find the *SUT* in which the corresponding *CT* was grouped. After the creation of the *SGTs*, for each of them an *Application Type* will be created with default value since this entity does not capture essential information. Then, the association class between the proxy and its proxied component is created and finally, the association class between the contained and its container component is also created.

After the transformation of the ECM model into an instance of the ETF model, the generator will perform the second transformation where the ETF model will be transformed in an instance of the ETF schema (SAI-AIS-SMF-XSD) which is the ETF file expressed in XML.

This transformation is straightforward because the ETF model captures the same information than the ETF schema. Nevertheless, there are some associations that are expressed differently and therefore, this is not a one-to-one mapping. Moreover, the generation of this file is always a valid instance of the ETF schema and is also always valid against the constraints of the domain because the input is validated before the generation with the OCL as explained in the Sub-section 3.3.

The last transformation is where the Template model instance is created. In this transformation there are three entities concerned; the *ServiceTemp*, *InterfaceTemp* and *ServiceGroupTemp* and they are transformed in this order. This is a one-to-one mapping except for one association, the one between the *ServiceTemp* and the *ScvT*. This association is not present in the ECM metamodel and therefore, the generator algorithm must look in which *SvcT* are grouped the *CSTs* associated with the grouped *InterfaceTemp*. After this last step, the generation of the ETF file and the HA requirements (i.e. the templates) is done and they can be used by the current approach for generating the complete middleware configuration.

## 3.5 Assumptions and Discussion

The process of generating the configuration targets the generation of an ETF file and the high level HA requirements. However, only the main scenario of generating a new file has been considered because the stakeholders (i.e. the software providers or the configuration designers), due to the nature of the ETF file, are likely to generate a new file each time they need one. The software providers may need to update a given ETF file in the case of a new version of existing software but, mainly they will create a new one. Furthermore, using this approach, they can persistently save the ECM model representing given software for future modifications. However, this is why in the ECM metamodel there is some information that has not been captured intentionally. As a result, if an instance of the ETF model were transformed in an instance of the ECM metamodel, information would be lost. Nevertheless, loss of information would be minimal and would not affect the overall level of availability because this information regards the *Application Type* and the *SGTs* it groups. Indeed, the only attributes of this entity is its name, its version and the *SGTs* it is grouping. Therefore, this information was considered as not relevant for the main purpose of this language that is, moving to a higher level of abstraction for simplifying the domain complexity.

The fact that the stakeholders no longer need to be expert in the domain does not mean that there is not a minimum knowledge required by the approach. More precisely, the user needs to be aware of the basic concepts of the domain. For example, the user needs to know that there is a separation about the service and the service provider and it enables the management of the high availability for the middleware. Also, the middleware support the notion of proxy and thus, allowing the use of a larger range of software resources. However, if the user is aware of those elementary details, he can easily use this new approach for generating the type description file and furthermore, generating the middleware configuration without having deep knowledge in the HA and SAF domain.

# CHAPTER 4

# THE ET-AM TOOL CHAIN

In this chapter, we discuss the details of our prototype implementation. Our prototype fully implements the ECM language, the ECM metamodel and all its annotated OCL constraints. It can perform a full transformation resulting in the ETF file and the HA requirements. Furthermore, it also implements the previous approach in order to better illustrate the completeness of the combination of both the new and the previous approach. Figure 4.1 illustrates the workflow of the prototype.

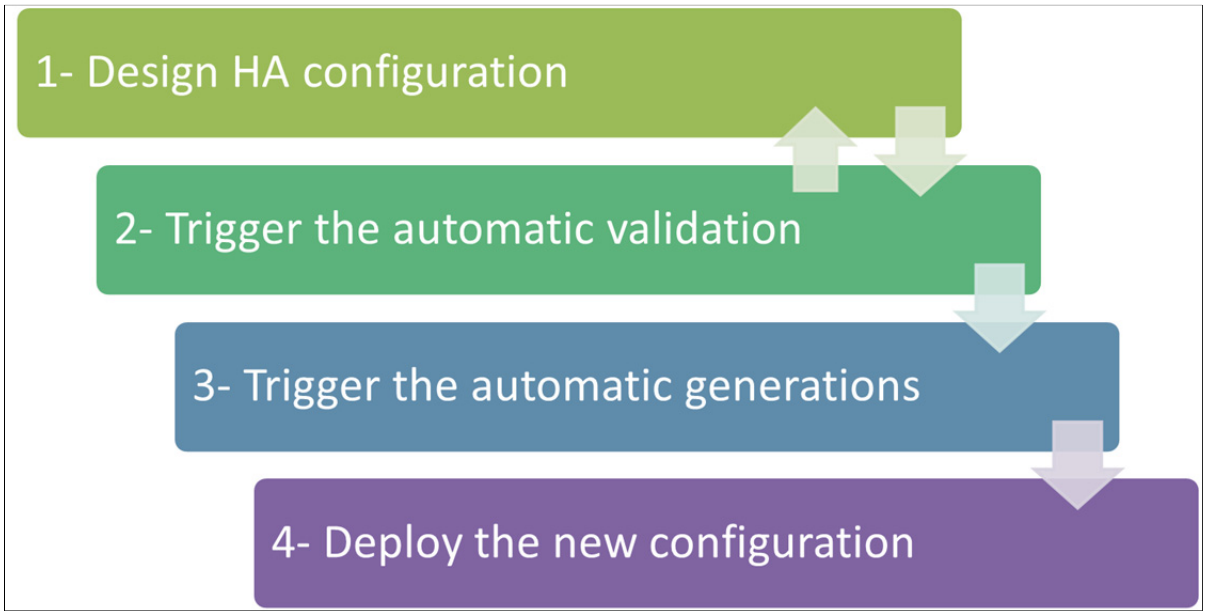

1- Design HA configuration
2- Trigger the automatic validation
3- Trigger the automatic generations
4- Deploy the new configuration

Figure 4.1 Workflow of the prototype tool

The workflow strongly reflects the approach as seen in Figure 3.2. Indeed, all the steps of the approach are implemented from the user input to the deployment of the configuration with the upgrade campaign into the middleware. More precisely, the step one is where the user designs the middleware configuration by specifying the systems components with the ECM language. In the second step, the user triggers the validation using the implementation of the OCL designed for the ECM metamodel. It is worth noting that designing the HA configuration in the first step will result in an instance of the ECM metamodel. If the validation fails, the user must update the incorrect design of the system following the instruction received after the validation. Then, having a valid design of the system, the third step is to trigger the automatic generations. In this step, there are five generators concerned; (1) the generator that will transform the instance of the ECM metamodel into an instance of the ETF model, (2) the generator that will transform the instance of the ETF model into an XML file (i.e. the ETF file), (3) the generator that will create an instance of the Template model from the ECM model, (4) the generator that will create the actual middleware configuration by taking as input an ETF file and an instance of the Template (i.e. the HA requirements) model and finally, (5) the generator that will create an upgrade campaign based on the generated middleware configuration. In the fourth step of the workflow, the user can deploy the generated configuration by using the generated upgrade campaign. Furthermore, the tool features an upgrade campaign monitor that informs the user of the state of the upgrade campaign and also implements its main commands (e.g. start, stop, commit and rollback) as seen in Sub-section 4.3.3. This way, the user can deploy the configuration without using the administration tools of the middleware that are accessible only by command line.

## 4.1 Architecture and Technologies

In the architecture of the ET-AM tool chain, the Graphical Definition module implements the concrete syntax of the ECM language. It is based on the Eclipse's Graphical Modeling Framework (GMF). Since the ECM language is an extension of the UML component diagram as explained in Sub-section 3.1, its look and feel is very similar. Moreover, this module uses the modeling infrastructure which is the implementation of the ECM metamodel and the OCL constraints. Any design action of the user in this module will update the ECM model and at this step, only the semantic based on the entities associations are automatically validated. However, this is the *Input/output Module* that ensures the serialization of the input into an ECM model.

The validation against the OCL constraints is the responsibility of the *Model Validator module*. The OCL constraints have been implemented with the OCLinEcore editor (OCLinEcore) from the OCL Eclipse project in the Modeling Infrastructure. The module validates the instance of the ECM metamodel with the OCL constraints and outputs the validation errors, if any, to the user by message dialogs with a code. If the user wishes more detail about validation errors, he can get detailed information in the documentation based on the error code.

The *Input Generator module* implements the three *model-to-model transformers* needed for generating the Configuration requirements (i.e. HA requirements) and the ETF XML file. They use the model implementations in the Modeling Infrastructure and since those implementations are in Java, the transformers are also implemented in Java. The Input Module de-serializes the ECM model and makes the content available to the ECM-to-CR and ECM-to-ETF Transformer. Furthermore, the Output Module features the serialization of the HA requirements and the ETF XML content in respectively the Configuration requirements and the ETF file.

The Modeling Infrastructure is implemented in Java but, the models of the prototype have been designed using the Eclipse Modeling Framework (Steinberg, Budinsky, Merks, & Paternostro, 2008). Therefore, the models are expressed in an Ecore file. Those files, using EMF, are used for generating the Java implementation of the models of this domain. The OCLinEcore editor allows the definition of OCL constraints directly into the Ecore file and furthermore, the OCL will be implemented with the model when it will be generated in Java. Figure 4.2 illustrates an architectural view of the prototype and it is worth nothing that the tool prototype is an Eclipse plugin in order to let the user manipulate freely the editors of the tool for the Ecore instance file, the XML files and the ECM model file. However, the prototype integrates the feature issued from the prototype developed for the previous approach and presented in this paper (Gherbi et al., 2009). Overall, the prototype can perform all the featured functionality of the new approach presented in CHAPTER 3 plus the feature of the previous approach presented in Sub-section 2.1. The four main resulting features of the prototype are explained in Sub-section 4.2.

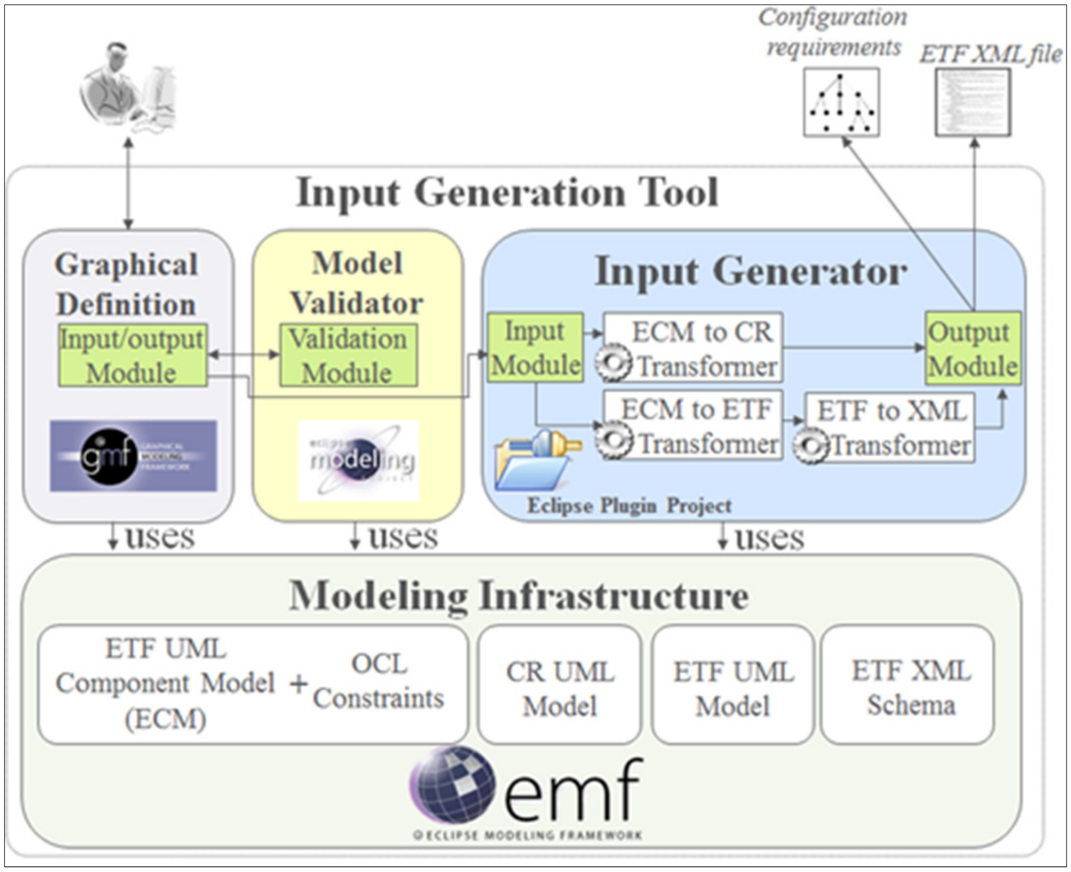


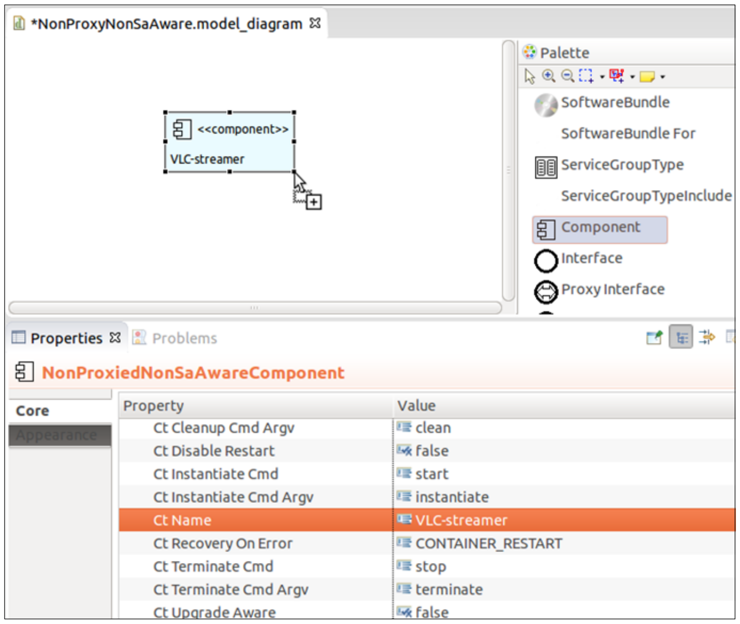
Figure 4.2 Tool architectural view

## 4.2 Features of the Tool

In this Sub-section, the features of the tools are shown in details with different use cases showing how to perform the generations of the ETF file, the HA requirements, the AMF configuration and the Upgrade Campaign.

### 4.2.1 Generating the ETF File and the High Availability Requirements

Using the ET-AM tool prototype, it is possible to generate only the ETF file. In order to generated this file, the generator need to take as input a complete type description of the application. More precisely, the user only needs to design the application graphically, then trigger the validation and once the design is valid, trigger the generation.

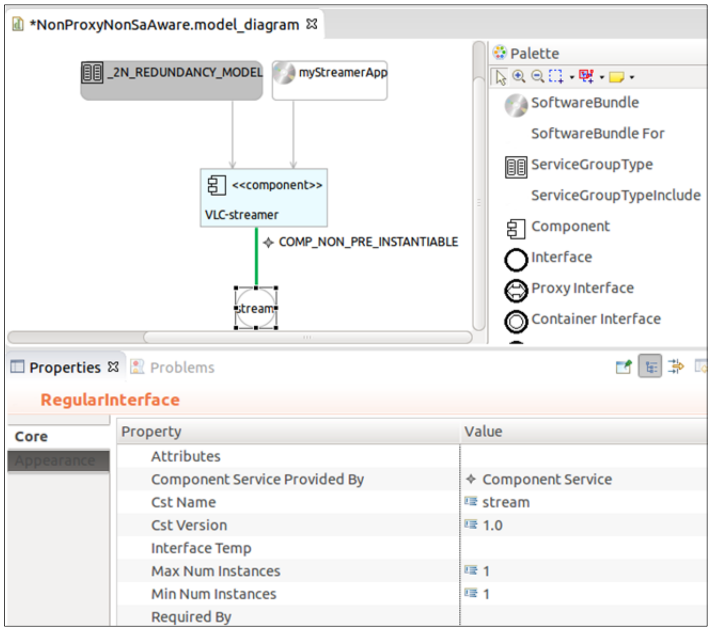


NonProxiedNonSaAwareComponent

| Property | Value |
|---|---|
| Ct Cleanup Cmd Argv | clean |
| Ct Disable Restart | false |
| Ct Instantiate Cmd | start |
| Ct Instantiate Cmd Argv | instantiate |
| Ct Name | VLC-streamer |
| Ct Recovery On Error | CONTAINER_RESTART |
| Ct Terminate Cmd | stop |
| Ct Terminate Cmd Argv | terminate |
| Ct Upgrade Aware | false |

Figure 4.3 Adding a Component with the ECM language

Figure 4.3 is a snapshot of the ET-AM tool illustrating how the user can create a component by selecting the *Component* icon from the right hand side palette, and then drawing the component in the editor area. The palette includes the elements defined in the ECM model. The user can enter the attribute values of the component using the properties panel. This is a contextual panel and always shows the attributes of the selected element of the editor area. Therefore, in the context of this current selection the properties of a *Non-Proxied, Non-SA-Aware component* are shown. However, for generating a complete ETF file, the transformer needs a complete input in order to be able to generate the missing entities. More precisely, the generator needs to know what type of service the *Component* is providing and furthermore, in what kind of redundancy the component can operate. Figure 4.4 is a snapshot where the design has been completed by adding the minimum needed entities for generating the ETF file. Those entities are the regular interface, the service group type and the software bundle. The contextual properties panel is showing the attributes of the selected regular interface.

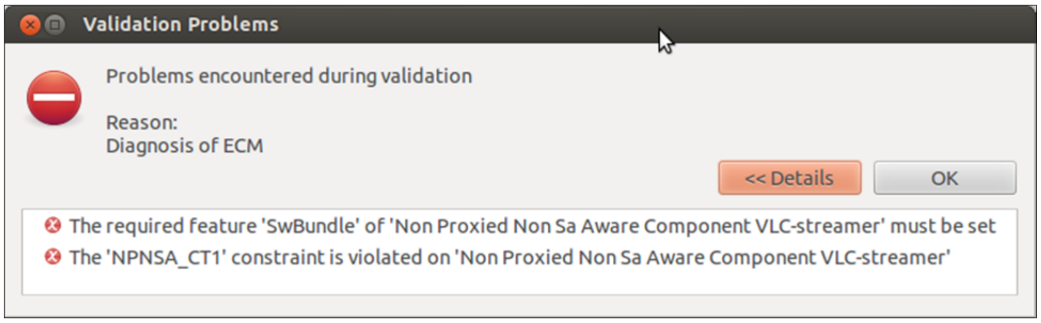


RegularInterface

| Property | Value |
|---|---|
| Attributes | |
| Component Service Provided By | Component Service |
| Cst Name | stream |
| Cst Version | 1.0 |
| Interface Temp | |
| Max Num Instances | 1 |
| Min Num Instances | 1 |
| Required By | |

Figure 4.4 Complete design of an application with the ECM language

If the user did a mistake during the design of the application with the ECM language, the validation part of the prototype can always point out the errors as illustrated by Figure 4.5. In this particular case, the validation was triggered before the component was associated to a software bundle. Plus, the NPNSA_CT1 constraint is violated. This constraint checks that a Non-Proxied, Non-SA-Aware component (the one shown in Figure 4.3 and Figure 4.4) is only providing interfaces with the capability model COMP_NON_PRE_INSTANTIABLE.



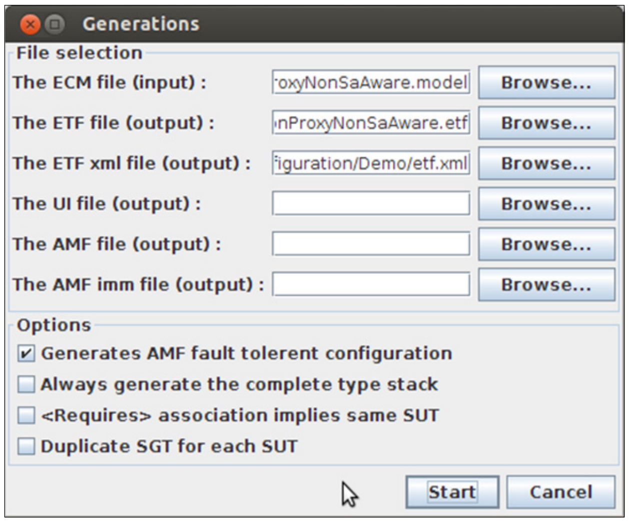
Problems encountered during validation

Reason:
Diagnosis of ECM

The required feature 'SwBundle' of 'Non Proxied Non Sa Aware Component VLC-streamer' must be set
The 'NPNSA_CT1' constraint is violated on 'Non Proxied Non Sa Aware Component VLC-streamer'

Figure 4.5 Validation of the input design

Once the design is validated, the user can trigger the generation of the ETF file. As illustrated by Figure 4.6, the user must specify the path to the ECM model (i.e. the serialized file), the path for saving the instance of the ETF model and also the path for the actual ETF XML file. When those options are set, the user can click the start button and by this, triggering the generation.



File selection

The ECM file (input) : oxyNonSaAware.model
The ETF file (output) : nProxyNonSaAware.etf
The ETF xml file (output) : iguration/Demo/etf.xml
The UI file (output) :
The AMF file (output) :
The AMF imm file (output) :

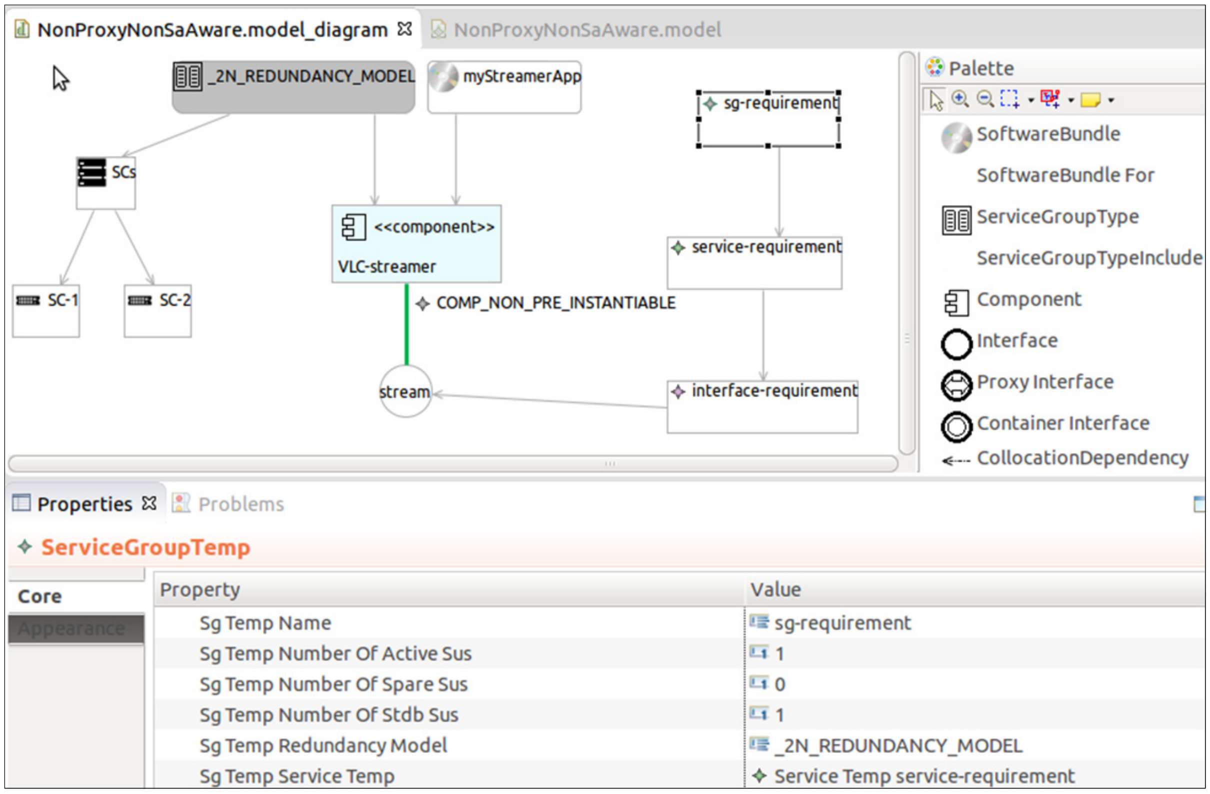
Options

☑ Generates AMF fault tolerent configuration
☐ Always generate the complete type stack
☐ <Requires> association implies same SUT
☐ Duplicate SGT for each SUT

Figure 4.6 Generations window

When the generation is over, the user space of the eclipse project will be updated with both the ETF file and also the serialized instance of the ETF model.

The feature of generating only the ETF file exists because the tool can be used by the software vendor and therefore, shall be able to produce only the description file. However, another scenario is taken in consideration. This is the scenario of producing both the ETF file and the HA requirements. This enables the combined usage of this prototype and the tool from the previous approach. In order to generate the HA requirements, the user must defines the needs in term of availability with the templates of the ECM language. Figure 4.7 illustrates the design of the application including the template information plus the deployment information. The templates entities are at the right of the edition area. The selected entity is the SG-requirement and the properties panel is showing its attributes. Furthermore, this entity is associated with the service-requirement entity and this later one is associated with the interface-requirement. Those three entities are capturing information about the level of availability the application should reach as explained in Sub-section 2.1.1 and 3.1.5.



| Property | Value |
|---|---|
| Sg Temp Name | sg-requirement |
| Sg Temp Number Of Active Sus | 1 |
| Sg Temp Number Of Spare Sus | 0 |
| Sg Temp Number Of Stdb Sus | 1 |
| Sg Temp Redundancy Model | _2N_REDUNDANCY_MODEL |
| Sg Temp Service Temp | Service Temp service-requirement |

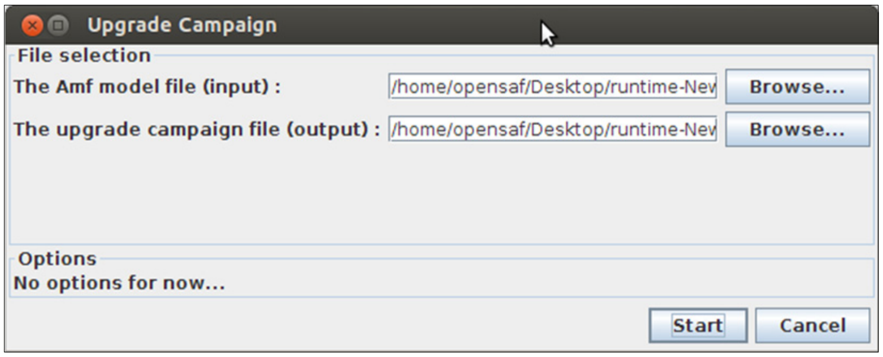
Figure 4.7 Complete design with HA requirements and deployments entities

### 4.2.2 Generating the AMF Configuration

The ET-AM tool also generates the AMF configuration if the output is specified. This feature is provided by the previous tool (Gherbi et al., 2009) and since the implementation was available, it has been integrated into the prototype. Hence, the user can generate both the ETF file and the AMF configuration with the same tool. The user triggers this generation with the window illustrated by Figure 4.6 by filling the corresponding field with a valid path. More precisely, if the user specified the needed entities for generating the ETF file and the HA requirements, the tool will also generate the AMF configuration if an output file is specified.

### 4.2.3 Generating the Upgrade Campaign

In this work (Kohzadi, 2009), the authors are presenting a tool for generating the upgrade campaign. The source code of this project was available and therefore, it was added into the prototype under a plugging that work independently of the main plugging. Figure 4.8 illustrates the window where the user can trigger the generation.



The Amf model file (input) : /home/opensaf/Desktop/runtime-Nev
The upgrade campaign file (output) : /home/opensaf/Desktop/runtime-Nev
No options for now...

Figure 4.8 Upgrade Campaign generation window

This module takes as input the serialized instance of the AMF model and generates as output the XML upgrade campaign file. This XML file contains the instructions the middleware needs in order to migrate from a given configuration to a new one with minimum service outages as explained in Sub-section 1.4 and 2.1.3. It is possible to include paths to scripts that can install the software automatically in the Software Bundle (see myStreamerApp in Figure 4.4). However, if there is no scripts for the installation of the software, the user needs to manually deploy the software into the cluster before deploying the configuration.

## 4.3 Verification of the Produced Artefacts

In this Sub-section, the methods for verifying the output files are discussed. More precisely, the different outputs of the prototype were verified in order to make sure they were valid. However, tools have been used in some cases in order to verify that the file was valid.

### 4.3.1 Verification of the ETF File

The ETF XML file generated by the prototype is assumed to be a valid ETF file as discussed in CHAPTER 3. In order to verify that it was the case, two strategies have been used. (1) A mature XML editor (XMLSpy) was used in order to check the structure of the ETF file against its XML Schema. It is worth nothing that the XML Schema is very complete and also includes its own sets of constraints. (2) The tool chain from the previous approach was used with an ETF file generated from the ET-AM tools and the generated middleware configuration was successfully used in a real cluster.

### 4.3.2 Deploying a New AMF Configuration on a Cluster

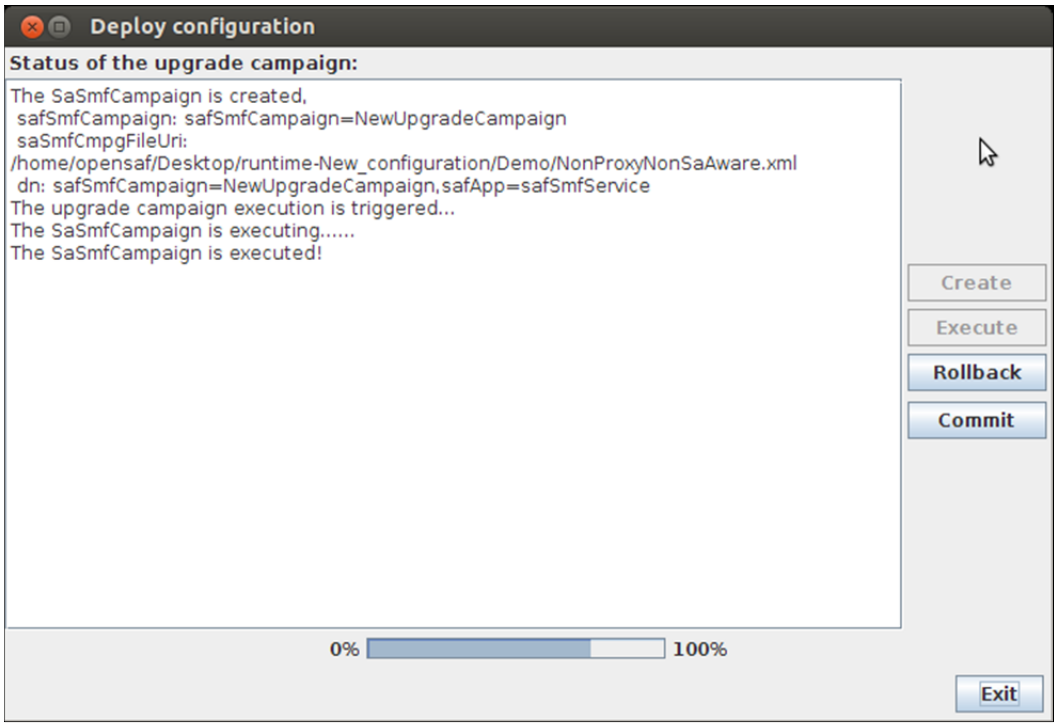
For testing the middleware configuration generated by the ET-AM prototype, the scenario of adding a new application into a cluster of two nodes was used. Without using the upgrade campaign, it is possible to merge the current configuration with another one in the case of adding a new application. The OpenSAF implementation of the middleware features a tool for merging such configuration and thus, it was possible to verify that the middleware accept the generation middleware configuration. This method requires shutting down the entire cluster before merging the two configurations and therefore, in order to avoid such outage the upgrade campaign must be used (see Sub-section 1.4 and 2.1.3). However, like the ETF file the AMF configuration also have a XML schema and using the XMLSpy XML editor, the configuration was validated against its complete and mature schema.

The OpenSAF implementation also offer a tool for verifying a given configuration file. This tool features helping functionalities telling the user what is wrong with its file. For example, if a name reference contains an error, the tool will point which entity is concerned and what is the error (e.g. the referred entity is impossible to find or the format of a given attribute is not valid). For this reason, this tool was used in order to know in detail what was the error during the development of the prototype. However, now the prototype is producing a valid middleware configuration file.

### 4.3.3 Live Integration of New AMF Configuration Using the Upgrade Campaign

The same kind of environment used for testing the AMF configuration was reused in the verification of the upgrade campaign generator. In fact, an upgrade campaign was generated for adding a new application in a cluster of two nodes with two applications already running. The two applications were protected with a redundancy model of 2N (i.e. one active and one standby). The generated upgrade campaign executes successfully in the middleware and the added application will be highly available as soon as the campaign is executed.

In order to monitor the state of the campaign, a feature was added into the prototype. This feature allows the user to trigger an upgrade campaign from the generated file and monitor the execution of the campaign. Figure 4.9 illustrates the window of the upgrade campaign monitor. The user can perform the main operations with the window like the *Create*, *Execute*, *Rollback* and *Commit*.



Status of the upgrade campaign:

```
The SaSmfCampaign is created,
 safSmfCampaign: safSmfCampaign=NewUpgradeCampaign
 saSmfCmpgFileUri:
/home/opensaf/Desktop/runtime-New_configuration/Demo/NonProxyNonSaAware.xml
 dn: safSmfCampaign=NewUpgradeCampaign,safApp=safSmfService
The upgrade campaign execution is triggered...
The SaSmfCampaign is executing......
The SaSmfCampaign is executed!
```

Figure 4.9 Upgrade Campaign monitor

# CONCLUSION

## 1) Research Contributions

In this thesis, we presented an approach for automating the design and generation of the types description files (i.e. the ETF file in the OpenSAF domain) and the HA requirements from a graphical input. This approach naturally completes the previous approach and by using it, both the software provides and the configurations designer can generate the types description file and/or the middleware configuration from a graphical input. This input is expressed using a graphical language based on the well-known UML Component Diagram and it is named ECM (ETF Component Model). Because of its inherited semantic and look and feel from the UML Component Diagram, this ECM language is easy to understand and manipulate for software developers. In order to capture the information expressed by the graphical language in a formal manner, a metamodel as been created and it takes its name from the ECM language; the ECM metamodel. This metamodel is enriched by OCL based on constraints carefully extracted from the domain specifications and thus, making any input expressed with the ECM language a valid input against the domain constraints. The problem solved by this approach lies as much in the amount of constraints as in the number of entities and especially since someone needs to go through the thousands of specification pages in order to master them. This learning process is not affordable for most software and service providers of the domain and therefore, the complexity needed to be abstracted by an automated approach.

The approach is based on three model-to-model transformations; (1) the first model-to-model transformer will take as input an instance of the ECM metamodel, and then it will infer the abstracted entities based on the semantic and dependency and then it will create an instance of the ETF model as an output. (2) The second transformer will take as input an instance of the ETF model and will create an instance of the ETF Schema which is the ETF XML file. (3) The last transformer will take as input an instance of the ECM metamodel and will extract from it the HA requirements, if any.

In order to verify the work done in this research we have developed an Eclipse-based prototype tool that can be used by configuration designers. This prototype implements all artefacts of the new approach. In other words, it implements the graphical language, the ECM metamodel, the transformers and also the validation based on the OCL constraints. In addition, the prototype includes the features developed in the prototype resulting from the previous approach and therefore, can generate and integrate the middleware configuration from the only input of the ECM graphical language.

## 2) Future Work

One of the targets of this research is to create an approach that can be used in more than one HA domain. Our approach targets the generation of the type description file however, in other system like Linux Pacemaker (Pacemaker), such file may not be needed. Therefore, the approach shall be adapted in order to target the needed artefacts for such middleware. The adaptation of the approach should not differ dramatically from our approach since the concepts used in this domain are mainly the same than the one used in the SAForum domain (i.e. the redundancy management, the escalation policies, etc…). Furthermore, we believe that the configuration of any similar middleware could be automatically generated from an approach derived from the one presented in this thesis (see CHAPTER 3).

## 3) Closing Remarks

The need for highly available systems is constantly on the rise, especially with the high costs of unplanned outages. Therefore the use of HA middleware to manage the system availability is in high demand. Using the SAForum middleware to manage the HA of the system can be challenging especially since the previous approaches for automatically generating the middleware configuration did not provide the means to automatically generate the ETF XML file. Writing this XML file manually can be a tedious and error prone task. Consequently, the contribution of this research is freeing the user from the very complicated technical aspect of the types description files and allows him/her to focus on the availability of the system without being a domain expert.

## BIBLIOGRAPHY

Espindola, A. P., Becker, K., & Zorzo, A. (2004). *An extension to UML components to consider distribution issues in early phases of application development.* Paper presented at the System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on.

Gagnaire, M., Diaz, F., Coti, C., Cerin, C., Shiozaki, K., Xu, Y., . . . Lubiarz, S. (2012). Downtime statistics of current cloud solutions. *International Working Group on Cloud Computing Resiliency, Tech. Rep., June*, 176-189.

Gartner. Retrieved from http://www.gartner.com

Gherbi, A., Kanso, A., Khendek, F., Hamou-Lhadj, A., & Toeroe, M. (2009). *A Tool Suite for the Generation and Validation of Configurations for Software Availability.* Paper presented at the Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering.

GMF. Graphical Modeling Framework. Retrieved from http://eclipse.org/gmf-tooling/

ITIC. Information Technology Industry Council. Retrieved from http://www.itic.org/

Kanso, A. (2012). *Automated Configuration Design and Analysis for Service High-Availability.* Concordia University.

Kanso, A., Toeroe, M., Hamou-Lhadj, A., & Khendek, F. (2009). *Generating AMF configurations from software vendor constraints and user requirements.* Paper presented at the Availability, Reliability and Security, 2009. ARES'09. International Conference on.

Kohzadi, S. (2009). *Automatic generation of upgrade campaign specifications Setareh Kohzadi.* Concordia University.

Kövi, A., & Varró, D. (2007). An eclipse-based framework for ais service configurations *Service Availability* (pp. 110-126): Springer.

Lu, S., Halang, W., & Zhang, L. (2005). *A component-based UML profile to model embedded real-time systems designed by the MDA approach.* Paper presented at the Embedded and Real-Time Computing Systems and Applications, 2005. Proceedings. 11th IEEE International Conference on.

Mahmood, S., & Lai, R. (2009). *RE-UML: An Extension to UML for Specifying Component-Based Software System.* Paper presented at the Software Engineering Conference, 2009. ASWEC'09. Australian.

OCLinEcore. Retrieved from https://wiki.eclipse.org/OCL/OCLinEcore

OpenSAF. Open Service Availability Forum. Retrieved from http://www.opensaf.org/

Pacemaker. Retrieved from http://clusterlabs.org/wiki/Pacemaker

Ponemon. Ponemon Institute. Retrieved from http://www.ponemon.org/

SAForum. Service Availability Forum. Retrieved from http://www.saforum.org/

SAI-AIS-AMF. SAI-AIS-AMF-B.04.01: SAForum.

SAI-AIS-IMM-XSD. SAI-AIS-IMM-XSD-A.01.02: SAForum.

SAI-AIS-SMF-XSD. SAI-AIS-SMF-XSD-A.01.02: SAForum.

SAI-AIS-SMF. SAI-AIS-SMF-A.01.02: SAForum.

SAI-Overview. SAI-Overview-B.05.03: SAForum.

Salehi, P. (2012). *A Model Based Framework for Service Availability Management.* Concordia University.

Schmidt, K. (2006). *High availability and disaster recovery: concepts, design, implementation* (Vol. 22): Springer Science & Business Media.

Sendall, S., & Kozaczynski, W. (2003). *Model transformation the heart and soul of model-driven software development*. Retrieved from

Steinberg, D., Budinsky, F., Merks, E., & Paternostro, M. (2008). *EMF: eclipse modeling framework*: Pearson Education.

Szatmári, Z., Kövi, A., & Reitenspiess, M. (2008). *Applying MDA approach for the SA forum platform.* Paper presented at the Proceedings of the 2nd workshop on Middleware-application interaction: affiliated with the DisCoTec federated conferences 2008.

UML. Unified Modeling Language Specification. Retrieved from http://www.omg.org/spec/UML/2.4.1/

XMLSpy. Altova XMLSpy XML Editor. Retrieved from http://www.altova.com/