Automated AMF Configuration Difference Generation

Anik Mishra

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science (Electrical Engineering and Computer Science) at
Concordia University
Montreal, Quebec, Canada

December 2010

CONCORDIA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Anik Mishra

Entitled: "Automated AMF Configuration Difference Generation"

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

Master of Applied Science

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

Signed by the final examining committee:

____ Chair
Dr. D. Qiu

____ Examiner, External
Dr. J. Rilling, CSE To the Program

____ Examiner
Dr. A. W. Hamou-Lhadj

____ Supervisor
Dr. F. Khendek

____ Supervisor
Dr. M. Toeroe

Approved by: ____
Dr. W. E. Lynch, Chair
Department of Electrical and Computer Engineering

____________20_____ __________________________________
Dr. Robin A. L. Drew
Dean, Faculty of Engineering and
Computer Science

Abstract

Automated AMF Configuration Difference Generation

Anik Mishra

Many domains require computer clusters to meet clients' service level expectations. As cluster size increases, component failure becomes more likely. Rapid failure recovery is required to maintain high availability. The SA Forum has created specifications enabling management of multi-vendor solutions. These long-lived systems require tailored upgrade campaigns to keep their configuration up-to-date.

Passed works on SA Forum's Availability Management Framework (AMF) have created an automated AMF configuration generator and upgrade campaign generator. However, to generate an upgrade campaign for an already configured cluster based on a new target configuration, a configuration difference generator is needed. Furthermore, while Distinguished Names (DNs) usually uniquely identify object in a configuration, configuration generators do not guarantee that DNs in the new configuration will match. DN modification is not possible in a system without object replacement, causing service loss. Non-DN based inter-configuration object association is needed to restore old DNs.

Our objective is to devise a technique to perform difference generation with limited knowledge of DNs and to find what data is needed to achieve this. To accomplish this, we analyse the AMF configuration model, then propose assumptions in a top down approach based on maintaining service availability during upgrade.

We propose a two phase technique that first associates objects in-between configurations and then outputs the differences. We have implemented a tool that implements this technique on the Eclipse Platform and integrates with MAGIC's Automated Upgrade Campaign Generator. We then present a case study based on the PHASE example.

Acknowledgments

I would like to express my gratitude to:

- My family for their support.
- My supervisor, Dr. Khendek Ferhat for giving me the opportunity to peruse my thesis under his supervision.
- Dr. Maria Toeroe (Ericsson Canada Inc.) for providing guidance throughout my thesis.
- My colleagues and all those who supported me to achieve this work.
- Concordia University and Ericsson Canada for offering their facilities and resources.

This work has been partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Ericsson Software Research as part of the MAGIC Project.

Table of Contents

List of Figures

List of Tables

List of Acronyms

AIS:	Application Interface Specification
AMF:	Availability Management Framework
CKPT:	Checkpoint Service
CLM:	Cluster Membership
COTS:	Commercial off the Shelf
CSI:	Component Service Instance
CSType:	Component Service Type
DBMS:	Database Management System
DN:	Distinguished Name
DP:	Data Processor
DS:	Data Send
EE:	Execution Environment
EMF:	Eclipse Modeling Framework
EVT:	Event Service
GUI:	Graphical User Interface
HA:	High Availability

HPI:	Hardware Platform Interface
IMM:	Information Management Model
JVM:	Java Virtual Machine
LCK:	Lock Service
LOG:	Log Service
MSG:	Message Service
MTBF:	Mean Time Between Failures
MTTR:	Mean Time To Repair
NAM:	Naming Service
NTF:	Notification Service
OAM:	Operation, Administration, and Maintenance
PLM:	Platform Management
RDN:	Relative Distinguished Name
RO:	Read Only
SA Aware:	Service Availability Aware
SA Forum:	Service Availability Forum
SC:	Service Continuity

SDC:	Sensor Data Collector
SEC:	Security Service
SG:	Service Group
SI:	Service Instance
SMF:	Software Management Framework
SNMP MIB:	Simple Network Management Protocol - Management Information Base
SU:	Service Unit
SUT:	Service Unit Type
SW:	Software
SWT:	Eclipse Standard Widget Toolkit
TMR:	Timer Service
UCGen:	Upgrade Campaign Generation
XML:	Extensible Markup Language

Chapter 1

Introduction

In this chapter we will introduce High Availability, Service Continuity and introduce what the Service Availability Forum has done to enable those. We will then provide our motivations, contributions and the organisation of this thesis.

1.1 High Availability and Service Continuity

A key feature for computer systems; be they for telecommunications, banking, e-commerce, etc; is that they always be ready and able to serve a user request. For this and other reasons, there are reliability ratings for most computer components and, often, for full systems. This metric is habitually presented as the Mean Time Between Failures (MTBF) [HAF01]. This is a metric given in hours that, when dividing its time unit, gives a probability of failure of that component during that time when operated within the designed lifetime and environmental conditions. When combining components into a system, the resultant MTBF decreases as the number of components increases. Furthermore, as systems get increasingly complex and are expected to have longer service lives, failures within a system become likely and even inevitable.

Knowing how long a system will take to be repaired once a failure has occurred becomes important. The metric for this is Mean Time To Repair (MTTR) [HAF01]. As the name implies, this is the mean time taken to repair a specified fault, component or system. Since repairs can often require manual intervention, for instance hardware components may require physical repair or replacement; such values may not simply be dependent on individual components.

However, neither the MTBF nor the MTTR indicate whether, when a user requests a service, that it will be provided. This is termed Availability and is computed as follows:

$$Availability = \frac{MTBF}{MTBF + MTTR}$$ [HAF01]

As this is necessarily a number between 0 and 1, it is usually given as a percentage. Furthermore, since having near perfect availability is usually the goal for mission critical systems, the number of consecutive leading 9's is an often quoted figure. High Availability is defined as five 9's of availability or better. That is to say 99.999% availability, or approximately 5 minutes of failure per year [TS02].

Since the larger and more complex the system, the more likely there will be a failure, two options to achieve High Availability are available. The first is to increasing MTBF in all components. This becomes an increasingly costly proposition with little gains. Instead the second option offers more promise: decreasing MTTR. Mathematically, if MTTR is decreased to 0, the Availability becomes 1 independently of the reliability of the components.

For instance, while having a technician coming in to repair a fault in a system is a time-intensive operation, maintaining spare hardware online and available means a replacement is immediately available. Furthermore, if fault detection and repair are automated, these faults can be fixed in minutes, seconds, or even milliseconds. These quick turnarounds can allow for the use of otherwise unreliable components to be used in highly available systems.

If service takes a non-negligible amount of time to complete an additional challenge exists. For instance if a task requires two hours of service to complete, and the service provider fails every 5 minutes, but takes only 1ms to repair, the task will never complete, despite availability being high. The solution to this is the notion of Service Continuity: when a component fails, not only is the service repaired or replaced, but session state is also maintained across this procedure [TS02]. As long as the repair does not make the service exceed any timeliness constraints, this could possibly be seen, from a user perspective, as having eliminated the fault.

1.2 Service Availability Forum

The Service Availability Forum is a consortium whose purpose is the development, education and promotion of open specifications for carrier-grade and mission-critical systems [SAF10]. Their purpose is to enable the use of commercial off the shelf components, be they software or hardware, in high availability systems. For this to happen, monitoring and management systems must be able to interact with these systems.

In the current market, every vendor has their own proprietary management implementation with limited compatibility and portability. This makes it very difficult to mix and match components from different vendors without having to manage them separately. Thus, there is a need for standardisation.

With this in mind, the SAForum has created several standards. Specifically, their Hardware Platform Interface (HPI) [SAFH09] standardises interfaces for requesting information about and managing hardware. Their Application Interface Specification (AIS) [SAFO08] specifies the interface in between management middleware and applications; as well as specifying interfaces to certain common clustering services.

This thesis will mainly deal with AIS concepts and, more specifically, Availability Management Framework (AMF) [AMF08] and the Software Management Framework (SMF) [SMF08]. These frameworks are respectively responsible for component management and software lifecycle management.

1.3 Thesis Motivation and Contributions

While the SAForum has made standards with regards to how components are managed, the choice of how to assemble these components was intentionally left open. To tell the management system how the components depend on each other, how and in what order they need to be started, etc, a configuration file must be supplied. These configuration files are quite long and complex. Configuration generators have been created in previous works [KA08, KA08B] to alleviate this problem.

Cluster configurations need to change over time due to updated requirements, software and hardware. To migrate a highly available system from one configuration to another, one cannot usually just stop service, load the new configuration and then restart service with the new configuration. A structured approach based on applying changes to arrive at the new configuration is needed. To help with that, a configuration difference generator is needed to be able to find the difference in between the current and target configurations. This is the goal of this thesis.

However, one additional and key constraint exists in our objective. AIS identifies elements based on unique identifiers called Distinguished Names [SAFO08]. Configuration generation is done without knowledge of an existing configuration. There is no attempt to reuse or recreate these names as they may be in a previous configuration. Furthermore, a current or target configuration may be handmade, thus they may not be related by any naming convention.

For this reason we have created an approach to properly find these differences with limited use of those unique identifiers. To do this we have made assumptions based on maintaining service availability. Furthermore, we have implemented this algorithm as a plug-in for the Eclipse platform and have integrated it with the Upgrade Campaign Generator developed by Setareh Kohzadi [KS09].

1.4 Thesis Organisation

The remainder of this thesis is divided into 4 chapters. Chapter 2 presents the background information needed for understanding this thesis and the related work. Specifically, this background will focus on the configuration and availability aspects of AMF and prerequisite topics. Upgrade campaigns and the Eclipse framework will also be introduced. Chapter 3 presents our approach, including the rational for our methodology and assumptions. Chapter 4 presents our prototype implementation, giving its structure and a walkthrough; then presents a case study. Chapter 5 presents our conclusion and future research possibilities.

Chapter 2

Background on High Availability Standards and the Tooling Framework

In this chapter, we will introduce the SAForum specifications. We will then consider in depth selected topics from the Information Management Model, the Availability Management Framework and the Software Management Framework. Next, we will introduce the software foundations of our prototype tool. We will do this by explaining the Eclipse platform and Modeling Framework as well as the AMF models created by the MAGIC Project. Finally, we will examine related works.

2.1 SA Forum Standards

To be able to fulfil their goal of enabling interoperability of Highly Available software and systems, enabling full systems built from Commercial off the Shelf (COTS) components, the SA Forum has positionned their specifications as a midleware, ilustrated in Figure 1. These specifications are split into two groups: the Hardware Platform Interface (HPI) [SAFH09] Middleware and the Application Interface Specification (AIS) [SAFO08] Middleware. These specifications abstract functionality allowing the COTS components to see a familiar interface, despite being in a mixed environment.

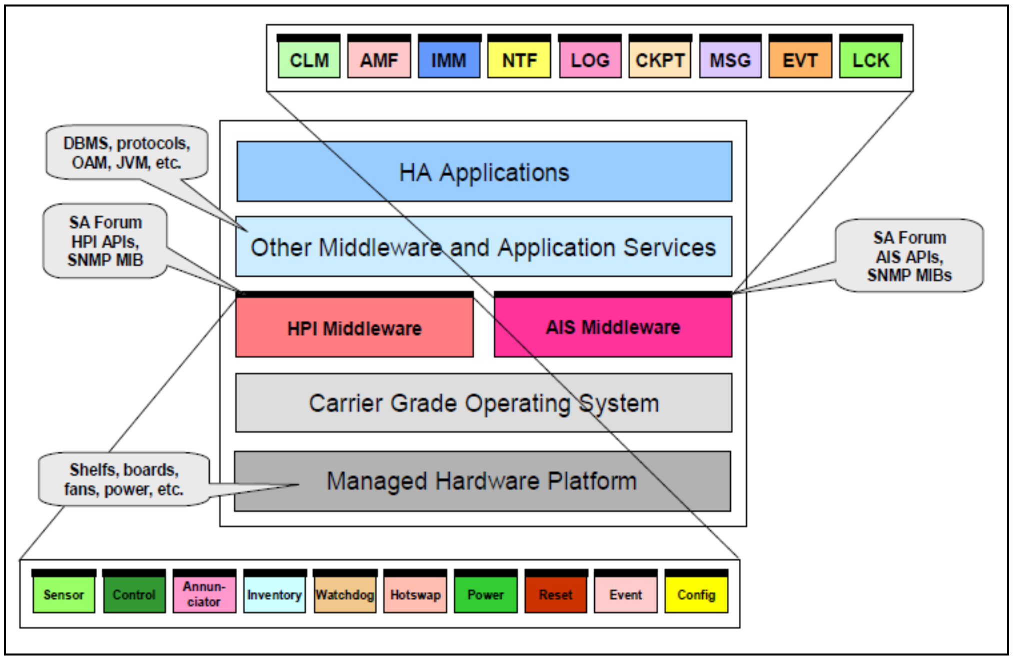

CLM
AMF
IMM
NTF
LOG
CKPT
MSG
EVT
LCK
DBMS, protocols, OAM, JVM, etc.
HA Applications
Other Middleware and Application Services
SA Forum HPI APIs, SNMP MIB
SA Forum AIS APIs, SNMP MIBs
HPI Middleware
AIS Middleware
Carrier Grade Operating System
Shelfs, boards, fans, power, etc.
Managed Hardware Platform
Sensor
Control
Annun-ciator
Inventory
Watchdog
Hotswap
Power
Reset
Event
Config

Figure 1: SA Forum Middleware Architecture, from SA Forum AIS Overview Document [SAFO08]

2.1.1 Hardware Platform Interface

The HPI Middleware abstracts the hardware aspect of High Availability (HA) systems. This middleware lies on top of the operating system and provides a software interface to be able to discover and manage hardware features such as fans, disks, hardware boards, etc. These tasks can all be performed using a standard software interface, without detailed knowledge about the workings of each of the underlying hardware components.

2.1.2 Application Interface Specification

As counterpart to HPI focusing on the hardware, AIS focuses on the software. Its specifications can be divided into sub-groups based on their objective, as illustrated Figure 2.

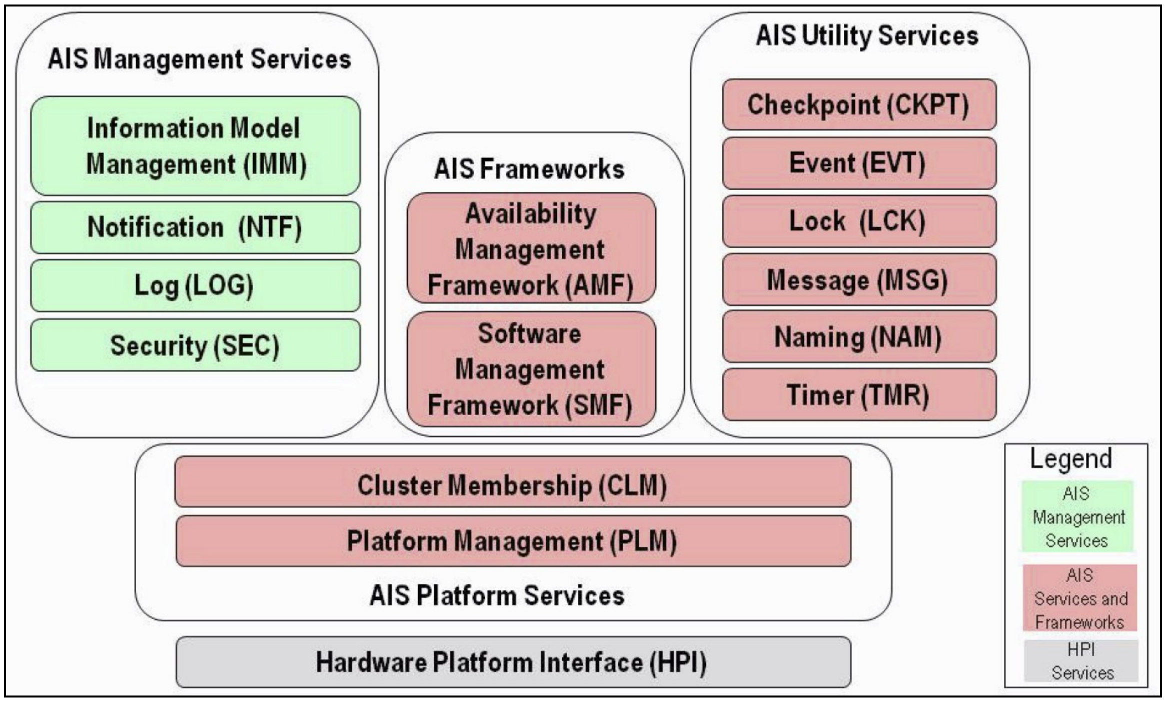


Figure 2: Application Interface Specification Standards, from SA Forum AIS Overview Document [SAFO08]

The management services group represents services that enable the AIS specifications to work in collaboration with each other. They can also be used by any software both for the same purposes or, independently, for any appropriate task. Of particular interest is the Information Model Management (IMM) service which maintains the system information model and mediates the administrative operations performed on the objects of the model. It will be further detailed in section 2.1.3.

In the platform services group, the Cluster Membership (CLM) [SAFC08] and Platform Management (PLM) [SAFP08] specifications provide further abstraction of the hardware for management use. We will go over them in more detail in section 2.1.6.

The frameworks section contains specifications that are meant to manage the cluster as a whole. The Availability Management Framework (AMF) covers the service execution lifecycle, with the objective of being able to promptly initiate recovery actions in the event of failure. More details on this framework will be given in section 2.1.4. The Software Management Framework (SMF) [SMF08] aims to manage the software lifecycle, i.e. software installation, upgrade and removal; while permitting low service impact. More details on this framework will be given in section 2.1.5.

Finally, in the utility services group, various helper services are defined. These represent common distributed tasks that an HA application would traditionally need to implement its-self to be able to achieve Service Continuity.

2.1.3 Information Management Model

All AIS specifications list certain data that must be made available both for reading and modification, to allow service implementers and external applications to query, interact with and reconfigure the management systems. All services store their public data in objects with specific definitions, called classes. Both these objects and their class definitions are stored and, optionally, made persistent by the IMM service.

2.1.3.1 Object Names

Within the IMM service all objects are stored hierarchically based on their Distinguished Name (DN). This DN is unique to each object, may not be changed and allows these objects to be accessed and referenced. The DN its-self is a string, composed of an object's Relative Distinguished Name (RDN), followed by a coma, and the DN of the parent object. In the case of top level objects, the DN and RDN are equivalent.

RDNs themselves can also arbitrary strings and character escaping semantics exist to make sure special characters in RDNs are not interpreted as hierarchical markers. The specifics of these semantics are not required to understand this work.

2.1.3.2 Object Properties

IMM defines objects as being an unordered list of named properties with typed values. These types are limited to a specific set of simple types. Generalising, they can be strings, times and numbers. Default values can be supplied in class definitions.

The properties also have attributes. The attributes are as follows: CONFIG/RUNTIME, INITIALIZED, PERSISTANT, CASHED, MULTI_VALUE, WRITABLE and RDN.

- CONFIG means that the property is a configuration property. Its value is persisted across system shutdowns.
- INITIALIZED is an attribute only relevant to CONFIG properties. It indicates that the value must be set at object creation.
- RUNTIME means that this property is not a configuration property. Their values can be changed by object implementers and are used to indicate the state of an object.

- PERSISTANT and CASHED are attributes only relevant to RUNTIME properties. They respectively indicate that the value should be kept across restarts and that the IMM service can keep a copy of the value instead of querying the object implementer each time the value is requested.
- MULTI_VALUED indicates that several values can be specified. This means that the value is in fact a list.
- WRITEABLE means that the value of a CONFIG property may be changed after object creation. Throughout this thesis, we will use the negation of this property that we will term as “read-only”. This nomenclature matches better with the AMF specification.
- RDN indicates that the property contains the object’s RDN.

2.1.3.3 IMM Interchange Format: IMMXML

AIS specifies an import/export file format for IMM’s data: IMMXML. It is an XML file [W3C10] with a specific schema [SAFI08] that contains all the class and object definitions in a non-hierarchical and unordered fashion. Object definitions are paired with their respective DNs so the knowledge of this hierarchy is not lost.

2.1.4 Availability Management Framework

AMF has for objective providing a basis for managing a cluster of computers as a whole, including the services that are meant to run on it. It does this by both specifying responsibilities for a compliant management system, and interfaces for software to properly interact with it, so that they may properly perform their tasks.

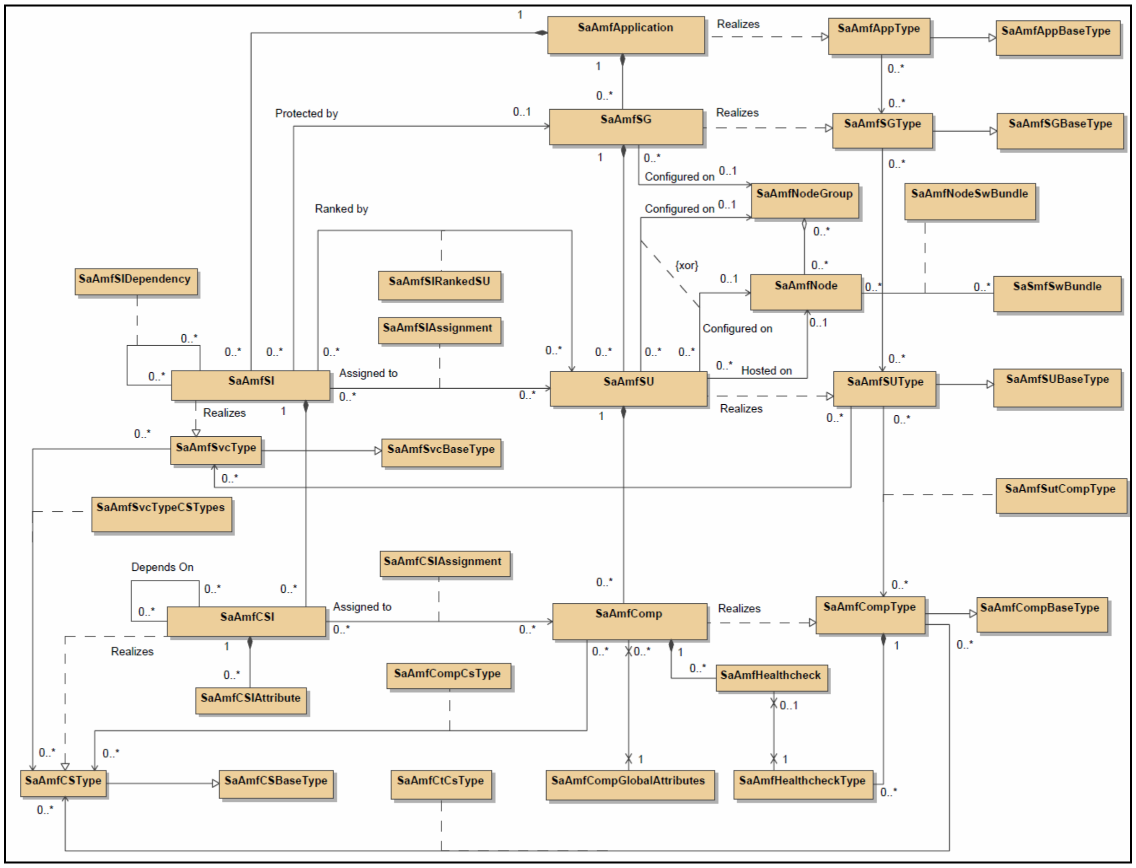


Figure 3: AMF Class overview, from SAI-AIS-AMF-B.04.01 [SAFA08]

In this section, we will go over AMF's data model and the management concepts behind it.

Overall, the data model contains basic machine entities; templates; instantiations of those templates representing specific functionality; and administrative groupings of these to define redundancy groups or failure boundaries. Figure 3 shows all the classes within the AMF data model. The specifics about the contents of each class are listed in the specification [SAFA08]. However, there have been some proposed changes to the writeability of certain configuration properties, as well as prerequisites regarding the modification of values at the Grenoble December 2008 SAForum meeting [SAFD08]. Specifically, this presentation relaxes certain constrains the AMF specification currently has. We will be considering AMF with those modifications. To summarise the property attribute changes in that document, class definitions will be presented throughout this explanation. RUNTIME properties have been omitted since our difference generation algorithm must work offline. A legend is shown in Table 1.

Read Only	Class name (DN format)
X = Yes	RDN property name
	property (unspecified type)
	Reference property ⇒ [multiplicity] referenced class

Multiplicity: 1 – one; + – one or more; ? – zero or one; * – zero or more

Table 1: Legend for Class Diagrams

2.1.4.1 Application

Applications represent the administrative collection of a logical group of services, their redundancy information and the Components they are to be executed on. This collection can be used as a top level administrative control. As example of this, an administrator can tell the management system to *lock* an Application and all its services will be stopped.

Furthermore, it is also used by the management system as a failure zone. When more granular recovery actions have been found to be ineffective, either because of repeated failures or because the configuration or trouble report indicates it is the proper recovery action, then the application as a whole is restarted. The only more global recovery action is a full cluster restart.

Because Application's main objective is this grouping, its data object contains very few properties, shown in Table 2. Instead the IMM hierarchy is used. Objects are put under an Application's control by creating them as children of that Application.

RO	SaAmfApplication ("safApp=…")
X	safApp
	saAmfAppType ⇒ [1] saAmfAppType

RO	SaAmfAppType ("safVersion=…,safAppType=…")
X	safVersion
	saAmfApptSGTypes ⇒ [+] saAmfSGType

Table 2: Application and AppType Classes

2.1.4.2 AppType

Throughout its specification, AMF uses types classes to specify data that is not specific to one instance object. The only property an Application has is a reference to its AppType. The AppType too is pretty barren, shown in Table 2: it only contains a list of acceptable Service Group Types that contained Service Groups may have. We will go over Service Groups in section 2.1.4.4.8.

2.1.4.3 The Concept of Services

When a user thinks of a service, for instance a website they wish to communicate with, it is typically regarded as a single entity they interact with. However there are often many discrete software systems that make the website run. For instance, the web server may also require an e-mail server on the same system to be able to send out e-mail confirmations. This division translates into AMF's service concept.

It should be noted that AMF distinguishes the concept of administratively requesting a service to be provided and the configuration of system components that actually can provide the requested functionality.

2.1.4.3.1 Component Service Instance and CSType

The concept of requesting one specific component of a service to be provided is done by creating a Component Service Instance (CSI), shown in Table 3. These objects are fairly simple: they contain an optional dependency list, to indicate that other CSIs should be started first; and they contain a reference to a Component Service Type (CSType), shown in Table 4. CSTypes are the templates for CSIs. They describe the service the CSI needs performed for the CSI to be regarded as being provided. However, since this CSType does not actually indicate how to perform the task, they are quite simple. In fact, they only contain a list of named parameters that a component that can provide the given component service must be willing to accept. We will see in section 2.1.4.4 that Components perform the requested tasks. These Components indicate that a given task is supported by also referring to this CSType.

RO	SaAmfCSI ("safCsi=...,safSi=...,safApp=...")	
X	safCsi	
	saAmfCSType ⇒ [1] saAmfCSType	CSType
	saAmfCSIDependencies	Dependencies

Table 3: CSI Class

RO	SaAmfCSType ("safVersion=...,safCSType=...")	
X	safVersion	
	saAmfCSAttrName	CSI Attribute names

Table 4: CSType Class

2.1.4.3.2 CSIAttribute

CSIAttribute objects are children of CSIs that allow these parameters to be set. Each of these objects, shown in Table 5, can specify one property and give any number of string values to configure that property. This is how each CSI can be specified to perform a different task.

RO	SaAmfCSIAttribute ("safCsiAttr=...,safCsi=...,safSi=...,safApp=...")
X	safCsiAttr
	saAmfCSIAttriValue

Table 5: CSIAttribute Class

2.1.4.3.3 Service Instance

Service Instances (SIs), shown in Table 6, are administrative groupings of these CSIs. Several of these CSIs are often required to work with a tight coupling to be able to provide a service. For instance both a program and a local data source may be required to operate together for a given service to be provided. The SI provides this grouping.

To realize this goal SIs perform several functions. The first is to guarantee that each component working for a CSI within the same SI will be allocated in the same Service Unit. We will see what exactly that means in section 2.1.4.4.6. However, for the moment we can limit ourselves to the knowledge that if there is a failure requiring moving a CSI to another system, all CSIs in the SI will be moved to the new location. The remaining functions SIs perform are property based. They indicate what redundancy group they will be protected by. We will see more about this when we describe Service Groups in section 2.1.4.4.8. They also have several parameters that allow setting the service rank, that indicates what SIs to prioritize if there is a shortage of resources; the resources consumed on Nodes when both in active and standby modes; and the preferred numbers of such assignments.

Many Service Instances may be required to provide the desired functionality. For instance a web based service may need both a web server to execute its code and a separate database server to store its information. Furthermore, multiple service instances may be needed, either to account for load, or simply to provide customized services to different customers. This is why multiple SIs can be aggregated into Applications.

RO	SaAmfSI ("safSi=…,safApp=…")	
X	safSi	
	saAmfSvcType ⇒ [1] saAmfSvcType	Service Type
	saAmfSIProtectedbySG ⇒ [?] saAmfSG	Protection group
	saAmfSIRank	
	saAmfSIActiveWeight	
	saAmfSIStandbyWeight	Lifecycle properties
	saAmfSIPrefActiveAssignments	
	saAmfSIPrefStandbyAssignments	

Table 6: SI Class

2.1.4.3.4 SvcType, SvcTypeCSType and SIDependency

As with CSIs, SIs also have templates: SvcTypes, shown in Table 7. They provide default values for active and standby resource usages. They also indicate what CSTypes may be contained within them through an association class: SvcTypeCSType, shown in Table 8. This class lists a maximum number of CSIs of the given type that may be contained within the SI. The idea behind this is that a vendor can supply the definitions for both the SvcTypes and CSTypes and indicate some limits on how many separate component services can be grouped together.

RO	SaAmfSvcType ("safVersion=…,safSvcType=…")	
X	safVersion	
	saAmfSvcDefActiveWeight	Defaults for resource usage
	saAmfSvcDefStandbyWeight	

Table 7: SvcType Class

RO	SaAmfSvcTypeCSTypes ("safMemberCSType=…,safVersion=…,safSvcType=…")
X	safMemberCSType ⇒ [1] saAmfCSType
X	saAmfSvctMaxNumCSIs

Table 8: SvcTypeCSType Class

SIs can also have dependencies with each other. This is specified in an association class SIDependecy, shown in Table 9. An additional lifecycle property indicates how long the SI can survive without the SI it depends on.

RO	SaAmfSIDependency ("safDepend=…,safSi=…,safApp=…")	
X	safDepend ⇒ [1] saAmfSi	
	saAmfToleranceTime	Lifecycle property

Table 9: SIDependency Class

2.1.4.4 Providing Services

2.1.4.4.1 Component

In an AMF system Components, shown in Table 10, are the workhorses. AMF assigns CSIs to them, and then they perform their given tasks. The vast majority of its properties serve only to control the lifecycle of the software that provides the requested functionality: the class contains commands and arguments to start, stop and cleanup the component; timeouts for error conditions; and recovery procedures.

In the case of proxied or contained components, when components lifecycle is to be controlled by another Component either through the assignment of another separate proxy CSI or a container CSI, this CSI is also noted in the configuration. The proxy situation usually results from when a specialised program, hardware component or system is adapted to fit into AMF's management model. The proxy bridges the gap in between the management systems. It should be noted that proxy and proxied components are considered separate and that a failure in one does not affect the other. Container Components on the other hand have a much stronger relationship with their contained Components. If a container fails, all its contained components have failed. Examples of this are components that provide a structured environment for others to run in, such as the Java Virtual Machine. While one can have many programs running inside a same virtual machine, if the container dies, they all die.

RO	SaAmfComp ("safComp=…,safSu=…,safSg=…,safApp=…")	
X	safComp	
	saAmfCompType ⇒ [1] saAmfCompType	Type
X	saAmfCompCmdEnv	
	saAmfCompInstantiateCmdArgv	
	saAmfCompInstantiateTimeout	
	saAmfCompInstantiationLevel	
	saAmfCompNumMaxInstantiateWithoutDelay	
	saAmfCompNumMaxInstantiateWithDelay	
	saAmfCompDelayBetweenInstantiateAttempts	
	saAmfCompTerminateCmdArgv	
	saAmfCompTerminateTimeout	
	saAmfCompCleanupCmdArgv	
	saAmfCompCleanupTimeout	Lifecycle properties
	saAmfCompAmStartCmdArgv	
	saAmfCompAmStartTimeout	
	saAmfCompNumMaxAmStartAttempts	
	saAmfCompAmStopCmdArgv	
	saAmfCompAmStopTimeout	
	saAmfCompNumMaxAmStopAttempts	
	saAmfCompCSISetCallbackTimeout	
	saAmfCompCSIRmvCallbackTimeout	
	saAmfCompQuiescingCompleteTimeout	
	saAmfCompRecoveryOnError	
	saAmfCompDisableRestart	
	saAmfCompProxyCsi ⇒ [?] saAmfCSI	Proxy CSI
	saAmfCompContainerCsi ⇒ [?] saAmfCSI	Container CSI

Table 10: Component Class

2.1.4.4.2 CompType

Components also have a Type, shown in Table 11. Like previous Types, this one is used to provide defaults to many of the values. However it also contains two key sets of data. First it references the Software Bundle associated with the Component and lists relative command paths, based on the installation location. The second is the Component category. This property indicates as a bit field whether the Component is any of Proxy/Proxied/Proxied-non-pre-instantiable, Container/Contained, Local and SA Aware. Being SA Aware means that the Component has fully implemented the AMF management interface. This means that the act of starting it up (instantiation) can be disassociated from providing it with a task to service. This ability can also be termed as being pre-instantiable. Local is a flag only notable when it is absent. It indicates that the actual implementer is not directly managed by AMF, and thus is only relevant to proxied components.

RO	SaAmfCompType ("safVersion=…,safCompType=…")	
X	safVersion	
X	saAmfCtCompCategory	Component category
X	saAmfCtSwBundle ⇒ [?] saSmfSwBundle	Related SWBundle
X	saAmfCtDefCmdEnv	
	saAmfCtDefClcCliTimeout	
	saAmfCtDefCallbackTimeout	
X	saAmfCtRelPathInstantiateCmd	
	saAmfCtDefInstantiateCmdArgv	
	saAmfCtDefInstantiationLevel	
X	saAmfCtRelPathTerminateCmd	
	saAmfCtDefTerminateCmdArgv	
X	saAmfCtRelPathCleanupCmd	Lifecycle defaults
	saAmfCtDefCleanupCmdArgv	
X	saAmfCtRelPathAmStartCmd	
	saAmfCtDefAmStartCmdArgv	
X	saAmfCtRelPathAmStopCmd	
	saAmfCtDefAmStopCmdArgv	
	saAmfCtDefQuiescingCompleteTimeout	
	saAmfCtDefRecoveryOnError	
	saAmfCtDefDisableRestart	

Table 11: CompType Class

2.1.4.4.3 CtCSType and CompCsType

CompTypes also have an association class with CSType. Instances of CtCSType, shown in Table 12, indicate what CSType any given Component associated with that CompType can support. This is used by AMF to perform CSI assignments. Furthermore, it provides a crucial piece of information: the component capability model. This indicates how many CSIs Components of the given Type can support in active and standby mode, and whether it can support CSIs in both active and standby mode simultaneously. Furthermore, certain capability models limit what redundancy model a component can participate in.

The specific number of assignments supported, if larger than one is not specified in the capability model itself. Two properties within CtCsType provide default values for those. A child object of Component, CompCsType, shown in Table 13, contains the properties to override those values. An instance of this object must exist for each Component to CSType association even if no override is needed.

RO	SaAmfCtCsType ("safSupportedCsType=…,safVersion=…,safCompType=…")	
X	safSupportedCsType	
X	saAmfCtCompCapability	Capability
X	saAmfCtDefNumMaxActiveCSIs	Default Max CSIs
X	saAmfCtDefNumMaxStandbyCSIs	

Table 12: CtCsType Class

RO	SaAmfCompCsType ("safSupportedCsType=…,safComp=…,safSu=…,safSg=…,safApp=…")	
X	safSupportedCsType	
	saAmfCompNumMaxActiveCSIs	Max CSIs
	saAmfCompNumMaxStandbyCSIs	

Table 13: CompCsType Class

2.1.4.4.4 CompGlobalAttributes

Components also get lifecycle properties from a configuration object of the CompGlobalAttributes class. There is only ever one instance of it for the entire cluster. As it is unique, it is trivially recognisable in a configuration. We will therefore omit its details.

2.1.4.4.5 Healthcheck and HealthcheckType

While Components can self report errors, certain checks, like liveliness checks, usually require external stimuli. AMF provides a monitoring facility for this. Possible checks are defined using the HealthcheckType class, shown in Table 15, and actual checks are defined using the Healthcheck class, shown in Table 14. For both of these, the RDN represents a magic value that specifies to the component what exactly the check does. The Component itself is responsible for performing the check and acknowledging it within the time limit specified in the configuration object.

RO	SaAmfHealthcheck ("safHealthcheckKey=…,safComp=…,safSu=…,safSg=…,safApp=…")
X	safHealthcheckKey
	saAmfHealthcheckPeriod
	saAmfHealthcheckMaxDuration

Table 14: Healthcheck Class

RO	SaAmfHealthcheckType ("safHealthcheckKey=…,safVersion=…,safCompType=…")
X	safHealthcheckKey
	saAmfHctDefPeriod
	saAmfHctMaxDefDuration

Table 15: HealthcheckType Class

2.1.4.4.6 Service Unit

Components are administratively grouped into Service Units (SUs), shown in Table 16. The only rule AMF itself places on this is a minimum requirement that all SIs that can be assigned to the SU must be able to be supported by the Components within the SU. This check is performed based on supported CSTypes.

SU objects also serve two other purposes. First, they optionally provide a location where their Components will be hosted. An optional property saAmfSUHostNodeOrNodeGroup specifies on what Node or NodeGroup the Components exist. The details of those objects will be presented in section 2.1.4.5.1. If a NodeGroup is referenced, it indicates that every Node within the NodeGroup can support the grouped Components. It is then up to AMF to decide on which single Node to reserve all Components of the SU. If this property is not specified, a NodeGroup can also be specified in the parent Service Group class. If not specified there, then any Node within the cluster can support the SU.

The second purpose served is as a failure zone. When a fault needs escalation above the level of a Component, the SU is the next in line for recovery actions to be taken. This adds an intermediate level in between Component and Node, potentially limiting downtime for unrelated services.

SUs further have two properties to configure priority for SI assignments and for failure recovery policies.

RO	SaAmfSU ("safSu=…,safSg=…,safApp=…")	
X	safSu	
	saAmfSUType ⇒ [1] saAmfSUType	Type
	saAmfSUHostNodeOrNodeGroup ⇒ [?] saAmfNode|saAmfNodeGroup	Node location
	saAmfSURank	Lifecycle properties
	saAmfSUFailover	
	saAmfSUMaintenanceCampaign	Upgrade indicator

Table 16: SU Class

2.1.4.4.7 SUType and SutCompType

SUs also have a type classe. This SUType, shown in Table 17, contains a default for the above recovery policy and list what SvcTypes are supported by SUs of this type. It also has one important property: saAmfSutIsExternal. As with Components indicates whether the SU is directly managed by AMF. In fact external components may only be contained within external SUs. An association class exists in between SUType and CompType: SutCompType, shown in Table 18. This allows the SUType to limit what Components may be included within its SUs. To further this, SutCompType has properties to set a minimum and maximum number of each Component that must be present within its SUs.

RO	SaAmfSUType ("safVersion=…,safSuType=…")	
X	safVersion	
X	saAmfSutIsExternal	External SUT?
	saAmfSutDefSUFailover	Lifecycle default
	saAmfSutProvidesSvcTypes ⇒ [+] saAmfSvcType	Provided SvcTypes

Table 17: SUType Class

RO	SaAmfSutCompType("safMemberCompType=…,safVersion=…,safSuType=…")	
X	safMemberCompType	
X	saAmfSutMaxNumComponents	Component
X	saAmfSutMinNumComponents	Min and Max

Table 18: SutCompType

2.1.4.4.8 SG, SGType and Redundancy Models

SUs are aggregated into Service Groups (SGs), shown in Table 19. While the SG class contains several lifecycle and fault escalation properties, the overall purpose of SGs are to provide redundancy. When one SU becomes unable to provide for an SI, another SU within the SG is assigned the task, if there are resources available.

RO	SaAmfSG ("safSg=…,safApp=…")	
X	safSg	
	saAmfSGType ⇒ [1] saAmfSGType	SG Type
	saAmfSGSuHostNodeGroup ⇒ [?] saAmfNodeGroup	Node location
	saAmfSGAutoRepair	
	saAmfSGAutoAdjust	
	saAmfSGNumPrefActiveSUs	
	saAmfSGNumPrefStandbySUs	
	saAmfSGNumPrefInserviceSUs	
	saAmfSGNumPrefAssignedSUs	
	saAmfSGMaxActiveSIsperSU	Lifecycle properties
	saAmfSGMaxStandbySIsperSU	
	saAmfSGAutoAdjustProb	
	saAmfSGCompRestartProb	
	saAmfSGCompRestartMax	
	saAmfSGSuRestartProb	
	saAmfSGSuRestartMax	

Table 19: SG Class

RO	SaAmfSGType ("safVersion=…,safSgType=…")	
X	safVersion	
X	saAmfSgtRedundancyModel	Redundancy Model
	saAmfSgtValidSuTypes ⇒ [+] saAmfSUType	Valid SUTypes
	saAmfSgtDefAutoRepair	
	saAmfSgtDefAutoAdjust	
	saAmfSgtDefAutoAdjustProb	
	saAmfSgtDefCompRestartProb	Lifecycle Defaults
	saAmfSgtDefCompRestartMax	
	saAmfSgtDefSuRestartProb	
	saAmfSgtDefSuRestartMax	

Table 20: SGType Class

The precise redundancy model used is given through the SG's Type, shown in Table 20. As with previous classes, the SGType defines default values for several lifecycle properties and also indicates what SUTypes SUs in its SGs may have. However, its key property is saAmfSgtRedundancyModel that defines the specific redundancy model used. This can be one of five values that set the overall behaviour:

- No Redundancy: A single active SU is assigned for each SI, with no standbys. Only one SI may be assigned per SU in the SG. In case of SU failure, another SU is assigned. This actually quite useful for services with no internal state.
- 2N: All SIs of the SG are assigned active on one SU and standby on one other SU. Other SUs may be present however they will act as spares: they will not receive any assignments unless an SU with an assignment ceases to be able to service its SIs.
- N+M: This is similar to 2N in that all SIs get one active and one standby assignment (resources permitting). Also, SUs still cannot mix active and standby assignments. However, SIs do not have the restriction that they all need to be assigned to the same SUs; or even that SIs with identical active assignments have matching Standby assignments.

- N-way: A single active SU and N-1 standby SUs are assigned for each SI. Active and standby SIs can be mixed on the same SU. Only components with the capability model x_active_and_y_standby can be used.
- N-way active: N active SUs are assigned with no standbys. Multiple SIs can be assigned to the same SU.

In general, ignoring capacity issues, when an active SU fails, the standby is given the active assignment and another standby is allocated. When the redundancy model does not call for a standby, another SU is assigned active without the intermediary standby assignment.

2.1.4.5 System Entities

While Applications aggregate services and the components they run on, there is an orthogonal concept that underlies this. All Components must execute on something. This is the concept of Node. While SUs are restricted to only exist within an Application, Nodes may service multiple SUs in different Applications. The low level aspects of their implementation are captured by the PLM and CLM services that we will detail in section 2.1.6. However, the management framework also needs to retain specific information about them.

2.1.4.5.1 AMF Node

Nodes, shown in Table 21, represent a single environment where software can be executed. AMF adds two main concepts to the node not inherited from their lower level implementation in CLM, further explained in section 2.1.6. The first is about how to treat various categories of failure and the second is about capacity information of the Node. It should be noted that this last piece of information is structured as a list of strings containing name and unit-less numerical value pairs. AMF does not know what each name actually represents. This system allows for both ensuring that there is not an over allocation of resources on the node, and for binding specific program instances to a given resource.

RO	SaAmfNode ("safAmfNode=...,safAmfCluster=...")	
X	safAmfNode	
	saAmfNodeClmNode ⇒ [?] saClmNode	CLM reference
	saAmfNodeCapacity	Capacity
	saAmfNodeSuFailOverProb	
	saAmfNodeSuFailoverMax	
	saAmfNodeAutoRepair	Failure handling
	saAmfNodeFailfastOnTerminationFailure	
	saAmfNodeFailfastOnInstantiationFailure	

Table 21: Node Class

2.1.4.5.2 NodeGroup

Nodes can be grouped into NodeGroups, shown in Table 22. These can later be used in various circumstances to refer to a set of Nodes. NodeGroups serve no other purpose than this and thus their data class only contains a list of member nodes. It should also be noted that a Node can belong to any number of NodeGroups, including none at all.

RO	SaAmfNodeGroup ("safAmfNodeGroup=...,safAmfCluster=...")	
X	safAmfNodeGroup	
	saAmfNGNodeList ⇒ [+] saAmfNode	Node List

Table 22: NodeGroup Class

2.1.4.5.3 Cluster

Both Nodes and NodeGroups all belong to a single and unique Cluster, shown in Table 23. Its purpose is to link with the CLM Cluster object and to provide timeout time for cluster start-up.

RO	SaAmfCluster ("safAmfCluster=...")	
X	safAmfCluster	
	saAmfClusterClmCluster ⇒ [?] saClmCluster	CLM Reference
	saAmfClusterStartupTimeout	Timeout

Table 23: Cluster Class

2.1.4.5.4 SoftwareBundle and NodeSwBundle

SaSmfSoftwareBundles, or Software Bundles, are, as their name implies, bundles of software. These contain pieces of software that are installed as a whole on a Node. While each bundle can supply software to support multiple CSTypes to the Nodes they are installed on, the installation itself cannot be further sub-divided. This object class is actually part of SMF, as the “SaSmf” prefix suggests, however AMF uses it directly as it has no need to extend its definition.

An association class, NodeSwBundle, shown in Table 24, serves to indicate that a given Software Bundle is installed on a given Node. With this objective, it also stores the location on the Node that the Bundle was installed to.

RO	SaAmfNodeSwBundle ("safInstalledSwBundle=...,safAmfNode=...,safAmfCluster=...")
X	safInstalledSwBundle ⇒ [1] saSmfSwBundle
X	saAmfNodeSwBundlePathPrefix

Table 24: NodeSwBundle Class

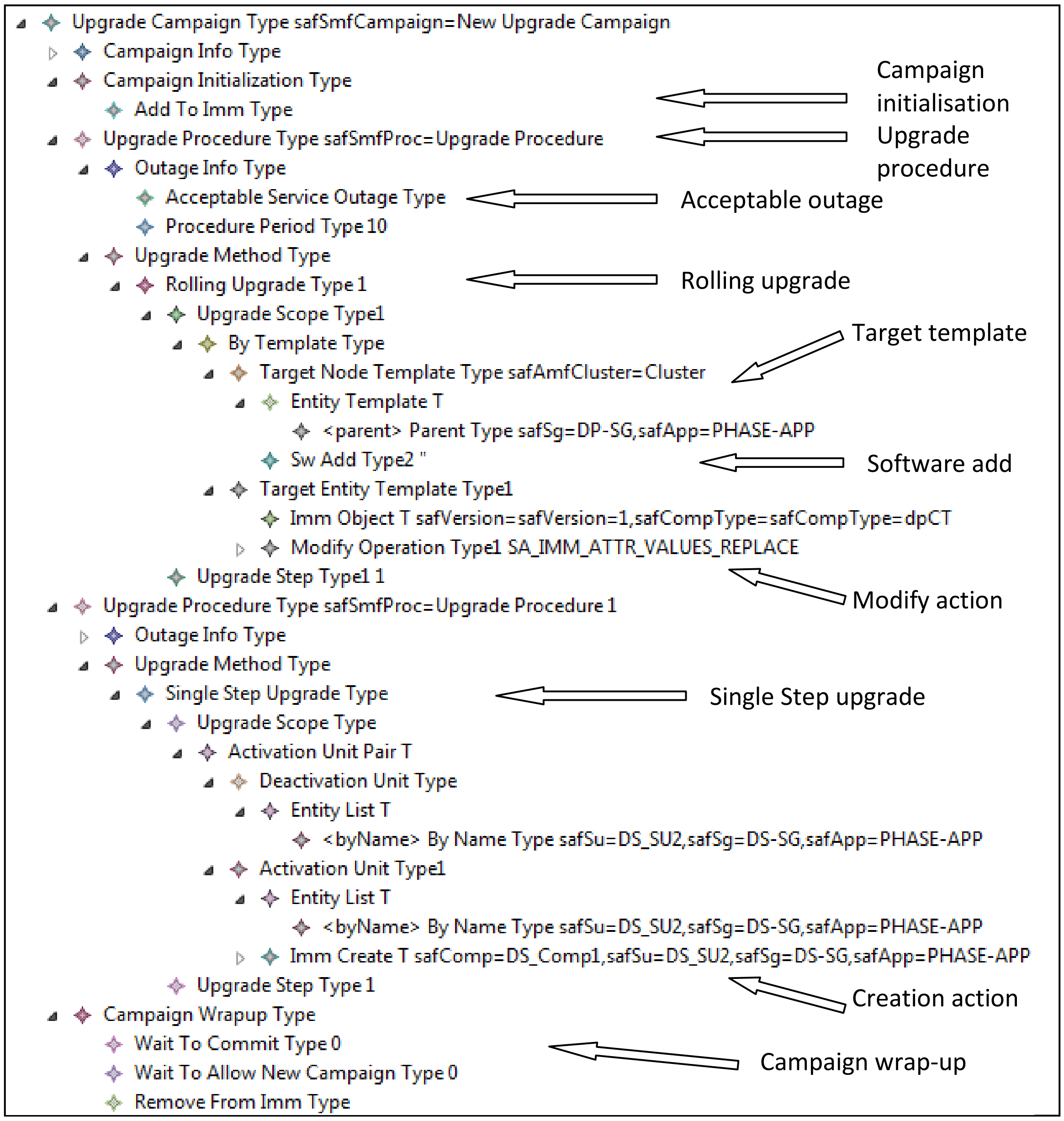
2.1.4.6 AMF BaseTypes

In Figure 3 one can find six BaseTypes. In AMF their purpose is simply to provide a base name to a set of Types. Name wise, each of these Types only have a version to distinguish them. There is no other purpose to these BaseTypes, and thus they have no other content than their name.

2.1.5 Software Management Framework and the Upgrade Process

HA systems are expected to run for long periods of time, measured in years. During this period it is likely that new versions of installed software will be released. It is also quite likely that service requirements will change along this period. For instance, increased service demands may require more nodes to provide the same service level; or the cluster may need to add new services to meet evolving customer expectations. Service cannot stop while these reconfigurations are made. Therefore, there is a need for cluster wide software management.

SMF codifies a framework for performing these upgrades. It does so by specifying how a campaign should be structured; defining limiting criteria to halt an upgrade procedure if it would unduly impact service availability; and providing a structure for rolling back an upgrade that is discovered to be faulty.



```
Upgrade Campaign Type safSmfCampaign=New Upgrade Campaign
  Campaign Info Type
  Campaign Initialization Type
    Add To Imm Type
  Upgrade Procedure Type safSmfProc=Upgrade Procedure
    Outage Info Type
      Acceptable Service Outage Type
      Procedure Period Type 10
    Upgrade Method Type
      Rolling Upgrade Type 1
        Upgrade Scope Type1
          By Template Type
            Target Node Template Type safAmfCluster=Cluster
              Entity Template T
                <parent> Parent Type safSg=DP-SG,safApp=PHASE-APP
              Sw Add Type2 "
            Target Entity Template Type1
              Imm Object T safVersion=safVersion=1,safCompType=safCompType=dpCT
              Modify Operation Type1 SA_IMM_ATTR_VALUES_REPLACE
        Upgrade Step Type1 1
  Upgrade Procedure Type safSmfProc=Upgrade Procedure 1
    Outage Info Type
    Upgrade Method Type
      Single Step Upgrade Type
        Upgrade Scope Type
          Activation Unit Pair T
            Deactivation Unit Type
              Entity List T
                <byName> By Name Type safSu=DS_SU2,safSg=DS-SG,safApp=PHASE-APP
            Activation Unit Type1
              Entity List T
                <byName> By Name Type safSu=DS_SU2,safSg=DS-SG,safApp=PHASE-APP
              Imm Create T safComp=DS_Comp1,safSu=DS_SU2,safSg=DS-SG,safApp=PHASE-APP
        Upgrade Step Type 1
  Campaign Wrapup Type
    Wait To Commit Type 0
    Wait To Allow New Campaign Type 0
    Remove From Imm Type
```

Figure 4: Model of an Upgrade Campaign File

An example of an upgrade campaign file is shown in Figure 4. SMF's upgrade campaigns are structured as follows. First, there is an initialisation section. While any administrative operation or command can be specified, one of particular interest is the addition and removal of IMM objects. For instance, if there is a new SWBundle, providing new CompTypes, these can be added in this step. Next, one or more UpgradeProcedures are listed. Each of these indicates what SIs may go unassigned during their execution. Before the campaign starts, if SMF detects that any procedure would produce unacceptable downtime, the campaign is aborted.

Just like the campaign itself, each procedure also has an optional initialisation section. It is followed by the selection of an upgradeMethod. This may be one of two types: singleStepUpgrade and rollingUpgrade. As its name suggests, singleStepUpgrade is meant to operate in a single step. By contrast rollingUpgrades are meant to operate on several entities one by one to perform an update. The advantage of this is that, if the execution of the upgrade implies downtime, in a rolling upgrade only one entity is taken out at a time, whereas in single step, all entities are handled simultaneously. In the situation where only added and removed entities have their availability impacted, this general outage is not a problem: either, in the case of removal, services will not be able to remove them after the step, or, in the case of addition, they were not able to use them before. So no additional availability constraints are being put on the system. Thus, the use of singleStepUpgrade is recommended for addition and removal of entities, whereas rollingUpgrade is recommended for modifications of existing entities. Following this, singleStepUpgrade has sections specifically for adding and removing entities, whereas rollingUpgrade only has an entity modification section. However, it does have Software add and remove sections.

After the upgradeMethod, some cleanup can be performed in an optional procedure wrap-up section. Similarly the campaign has a wrap-up section. However, other than a campaign completion subsection, the execution of this section is delayed for a campaign specific amount of time. This allows a probation period such that delayed failures can still be attributed to the upgrade campaign and rollback can more easily be initiated.

2.1.6 Platform Management and Cluster Membership Services

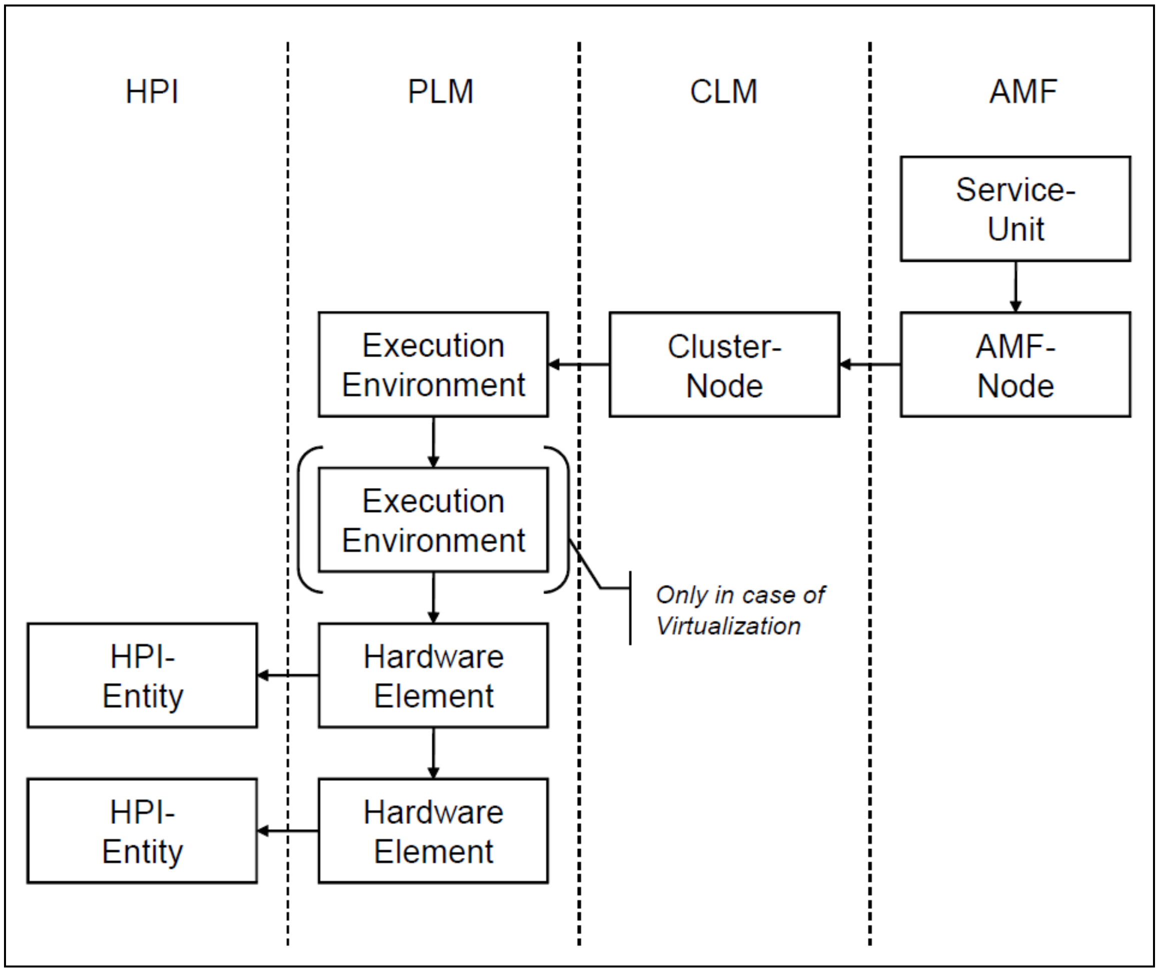


Figure 5: PLM and CLM, bringing hardware concepts to AMF, from SAI-AIS-PLM-A.01.02 [SAFP08]

The purpose of the Platform Management Service is to provide AIS entities to manage the platform. This is illustrated in Figure 5. Its major contributions are the Hardware Elements which represent HPI entities; i.e. hardware devices. When appropriate, an Execution Environment (EE) object is also created, in conjunction with each Hardware Element. While the true definition of EE does not limit its-self to the following, conceptually each EE instance can be thought of as one running operating system or hypervisor be it in a virtual environment or on actual hardware. For this reason, EEs can be children of another EE.

The purpose of the Cluster Membership Service is to take these PLM Execution Environments give them an identity with the configured cluster. It is at this stage that the concept of Node comes into play. After being further extended by AMF, it will represent one unit on which Components.

2.2 Eclipse Framework

We will explain the Eclipse framework our prototype was built on by first describing the Eclipse platform and its plug-in model. Then we will give a brief explanation of the Eclipse Modeling Framework, and we will end by showing MAGIC's StdAMF Model and listing some of its quirks.

2.2.1 Eclipse Platform

The Eclipse platform [EP10], made by the Eclipse Foundation, is a software development platform that codifies various extension methods and contribution points to enable "plug-ins", external code bundles, to extend its functionality. The platform, when invoked, will follow a standard set of procedures for finding and loading these bundles. Each of these bundles can then register features that either they can provide or that other bundles can enhance. This approach has enabled the Eclipse platform to be adapted for a wide range of tasks. Our implementation packages itself as one such plug-in and adds a menu and toolbar item to allow the user to invoke it.

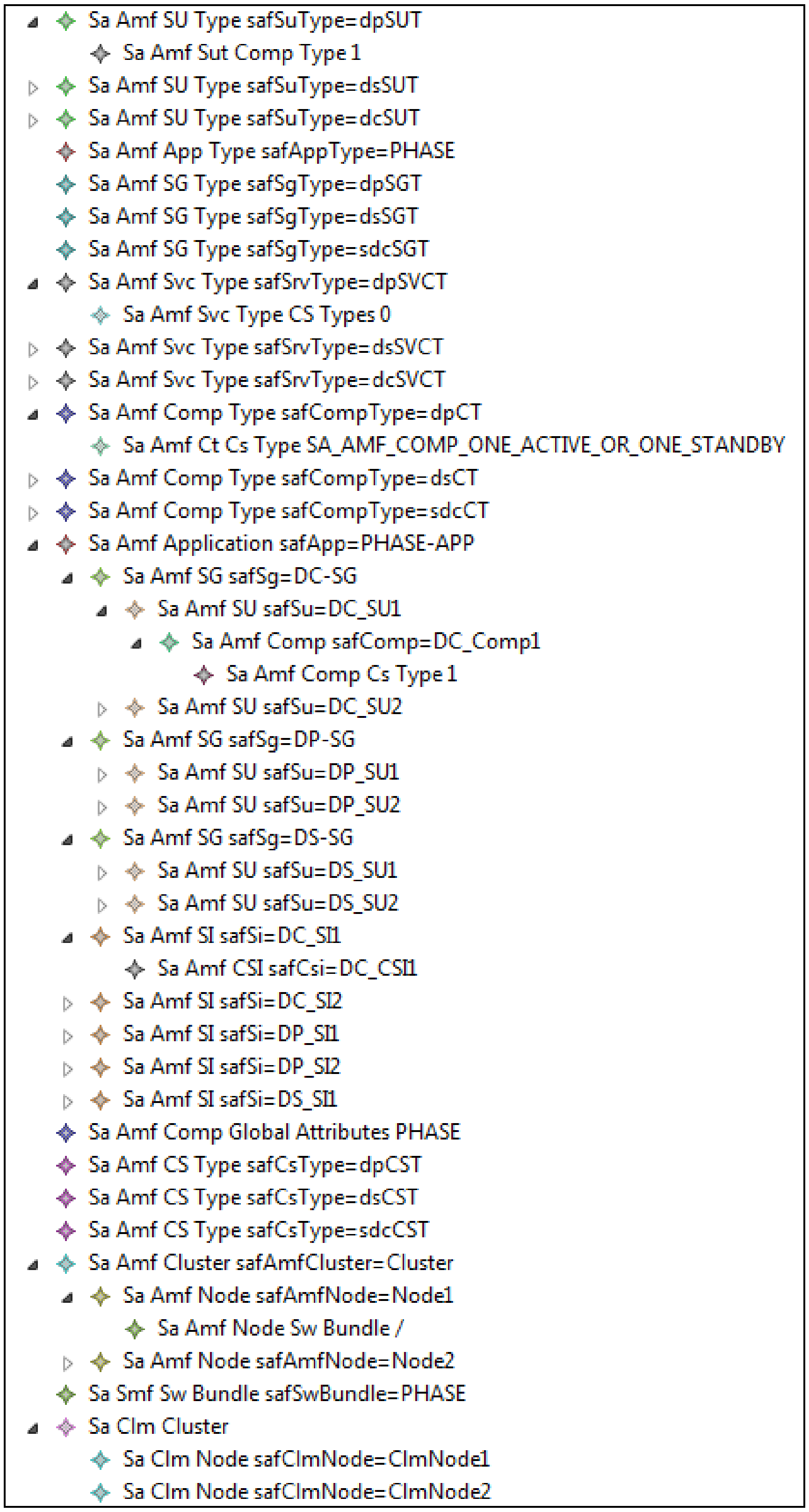
The base platform by itself only provides basic functionality; however, it does include the Standard Widget Toolkit (SWT) [ES10]. This toolkit allows for the creation of user interfaces whose look integrates with the native OS. Furthermore it provides, partially through the use of other Java user interface toolkits, simplified construction methods for common window elements. We will be using it for our interaction with the user.

2.2.2 Eclipse Modeling Framework

Eclipse's plug-in framework has also allowed other developers to create frameworks to enable and simplify certain tasks. These frameworks are often made freely available. One such project is the Eclipse Modeling Framework (EMF). It is a framework to enable development of structured model based programs. Amongst other features, it combines a basic meta-model (ECore), with code generation and model import/export abilities to greatly simplify the coding aspects of including a model in a tool. These features are notable because they were used to create the AMF models that we used for our implementation. We will cover them in the next section.

2.2.3 AMF Ecore Model

As part of previous works within the MAGIC project, a standard AMF model has been implemented on ECore. The implementation model itself bears little overall visual interest as it simply looks like a flat list of classes and their properties. Instead Figure 6 shows the model that we will be using for our case study in section 4.3. While the view shown is not comprehensive, instances of all of AMFs major classes can be seen. The IMM hierarchy is also clearly visible through the nesting of the objects.



```
Sa Amf SU Type safSuType=dpSUT
    Sa Amf Sut Comp Type 1
Sa Amf SU Type safSuType=dsSUT
Sa Amf SU Type safSuType=dcSUT
Sa Amf App Type safAppType=PHASE
Sa Amf SG Type safSgType=dpSGT
Sa Amf SG Type safSgType=dsSGT
Sa Amf SG Type safSgType=sdcSGT
Sa Amf Svc Type safSrvType=dpSVCT
    Sa Amf Svc Type CS Types 0
Sa Amf Svc Type safSrvType=dsSVCT
Sa Amf Svc Type safSrvType=dcSVCT
Sa Amf Comp Type safCompType=dpCT
    Sa Amf Ct Cs Type SA_AMF_COMP_ONE_ACTIVE_OR_ONE_STANDBY
Sa Amf Comp Type safCompType=dsCT
Sa Amf Comp Type safCompType=sdcCT
Sa Amf Application safApp=PHASE-APP
    Sa Amf SG safSg=DC-SG
        Sa Amf SU safSu=DC_SU1
            Sa Amf Comp safComp=DC_Comp1
                Sa Amf Comp Cs Type 1
        Sa Amf SU safSu=DC_SU2
    Sa Amf SG safSg=DP-SG
        Sa Amf SU safSu=DP_SU1
        Sa Amf SU safSu=DP_SU2
    Sa Amf SG safSg=DS-SG
        Sa Amf SU safSu=DS_SU1
        Sa Amf SU safSu=DS_SU2
    Sa Amf SI safSi=DC_SI1
        Sa Amf CSI safCsi=DC_CSI1
    Sa Amf SI safSi=DC_SI2
    Sa Amf SI safSi=DP_SI1
    Sa Amf SI safSi=DP_SI2
    Sa Amf SI safSi=DS_SI1
Sa Amf Comp Global Attributes PHASE
Sa Amf CS Type safCsType=dpCST
Sa Amf CS Type safCsType=dsCST
Sa Amf CS Type safCsType=sdcCST
Sa Amf Cluster safAmfCluster=Cluster
    Sa Amf Node safAmfNode=Node1
        Sa Amf Node Sw Bundle /
    Sa Amf Node safAmfNode=Node2
Sa Smf Sw Bundle safSwBundle=PHASE
Sa Clm Cluster
    Sa Clm Node safClmNode=ClmNode1
    Sa Clm Node safClmNode=ClmNode2
```

Figure 6: Phase Case Study MAGIC StdAMF Configuration Model

Three implementation details bear note. First as one would expect from a model, all references, stored in IMM as SaStrings, are properly represented as references. We can see an example of this Figure 7, for the saAmfSutProvidesSvcTypes property. The second detail has to do with the DNs and IMM hierarchy. While one can note in Figure 7 that the SUType RDN property safVersion is present; there are no extra properties to store the DN. The way this is handled is by the addition of non-specification properties that reference child or parent classes. An example of that is the SUTcomponentTypes property that would aggregate all SutCompType child objects. Usually a reference to the parent exists, but Types are an exception.

This leads to the final interesting implementation decision in the StdAMF model. It is that all Types have a class inheritance relationship with their respective BaseType class. What that means is that each instance of a Type, instead of having a reference to a common BaseType, actually includes a copy within it. Since a BaseType's only property is its RDN, it can be recreated when converting this model back to IMMXML. The significant implication of this is that there are no direct instances of BaseType in any configuration we present, and as one can note in Figure 6.

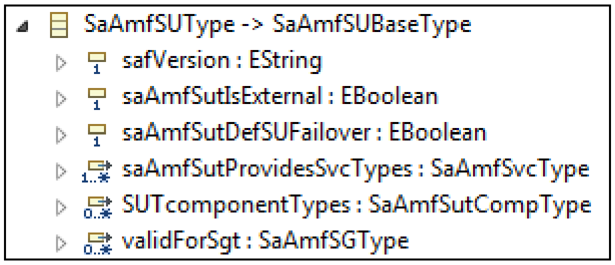

SaAmfSUType -> SaAmfSUBaseType
safVersion : EString
saAmfSutIsExternal : EBoolean
saAmfSutDefSUFailover : EBoolean
saAmfSutProvidesSvcTypes : SaAmfSvcType
SUTcomponentTypes : SaAmfSutCompType
validForSgt : SaAmfSGType

Figure 7: StdAMF SUType Implementation Model

2.2.4 MAGIC AMF ECore Model

A more advanced ECore model of AMF (MagicAMF) was made by Pejman Salehi and has been used in works by Ali Kanso. Unlike the standard AMF model that has as objective to be simple to program with, this model was made for analysis and validation. With that goal, it refines class definitions by creating child classes with extra data instead of just adding extra optional fields to the main classes. Since this is the output of MAGIC's Multiple Configuration Generator, this is what we will target as input for our implementation. However, we do not believe that an explanation of that model is needed to understand this thesis because the explanation of our algorithm is based on standard AMF, there is a one to one relationship in standard AMF and MagicAMF's classes, we do not use any of its added capabilities, and there is an automated tool developed by Jesus Garcia to convert the model to our standard AMF model (StdAMF) that we use for that purpose.

2.3 Related Work

There has been much research on the SA Forum's specifications by Concordia University's MAGIC project [CP10, GA09, GA09B, KA10, KA09, KA08, KA08B, KS09, PS10, PS10B, PS09]. As we have already seen in sections 2.2.3 and 2.2.4, models of the AMF configuration file have been made. The first being a direct transliteration of the specification and the second having been created after analysis and refinement of the model to be more apt for analysis.

Further work was done by Ali Kanso on configuration generation [KA10, KA09, KA08, KA08B]. He has developed both a single configuration generator, creating a suitable configuration, and a multiple configuration generator, creating more configurations for use by an analysis package to rank the configurations

The work by Setareh Kohzadi on upgrade campaign generation, also as part of the MAGIC project, considers the problem of matching Distinguished Names in new configurations [KS09]. However, it is simply left as a future problem to solve. Instead the upgrade campaign is based on an initial configuration and the upgrades to perform are specified by the user using a graphical user interface. With one exception, this eliminates the problem of having to know Distinguished Names: it is the user’s responsibility to interpret what entities have been updated and inform the campaign generator. The one exception is with regards to an assumption that is made with regards to consistency. All Components of the same type must have the same Relative Distinguished Name. This allows the upgrade campaign to perform rolling updates on all the affected components in one procedure.

Since a live AMF configuration is stored in IMM, it can be dumped as an IMMXML file. As previously mentioned in section 2.1.3.3, this is a text file in XML format. The subject of finding differences in text has been extensively studied and this is commonly referred to as the Longest Common Subsequence problem. Many algorithms exist that can solve this [BL00]. However the solutions are not adapted to this specific problem. There are three specific failures. The first failure is that the document is XML formatted. Many textual changes, such as white space and line wrapping, are not important to the meaning of the document. Altering the algorithms could fairly simply solve this. The second failing is that an AMF configuration is an unordered tree of objects. Only their hierarchy matters. The third failing is that the properties within each object are also unordered. Both of these can be solved by making the difference generator model aware, however this is not a trivial task. Works such as UMLDiff have attempted to solve this for UML [XZ05] and Eclipse’s EMF Compare tool has generalized it for Ecore models as we will see in the following paragraphs.

Eclipse contains a framework for the display of differences, Eclipse Compare. Originally, it is line based and was intended to be used by “team providers”: plug-ins that enable a team to work together. The main application for this is in source control systems. The dialogs allow the display of two or three versions of a file and display the difference by highlighting changes, and linking added and removed sections in between each file. The tool further allows merging of changes by copying code from one version to another.

This tool was further extended by Eclipse’s EMF Compare group that saw the same problems as previously mentioned when trying to compare their models. They further adapted the Eclipse Compare GUI to display models instead of text.

On the model comparison side, similarly to us as later explained in section Chapter 3, they have applied a two phase process to producing this difference. The first phase is devoted to matching objects based on key properties. The second phase is then producing the difference on the matched objects. Unlike our algorithm, theirs is generic such that it will work on all models. However this approach means that it is not able to value changes of different properties based on semantics, such as we do in our algorithm. Furthermore, the selective exclusion of object names from the match algorithm is not something handled by the tool.

It should be noted that the first general release of EMF compare as part of the Eclipse Platform happened in June 23rd 2010 as part of the Eclipse Helios release [ER09]. This is after the majority of the research of this thesis had been completed. This is why there is no attempt to integrate our prototype tool’s output with their viewer.

Chapter 3

Configuration Difference Generator

In this chapter, we will present our approach for the generation of the difference between two AMF configurations. Our approach is presented in an incremental manner. We will first give some preliminary definitions and outline a simple algorithm to generate the difference between two configurations using DN. We will then extract properties from AMF and IMM with the objective of performing difference generation with limited DNs knowledge. Next we will go through our reasoning for additional assumptions required for our solution. Then, we will present our algorithm. We conclude this chapter by listing some limitations.

3.1 Introduction

Our approach requires two AMF configurations: the initial (also called the source) configuration the cluster is currently deployed with, and the target configuration into which the administrator wants to upgrade the cluster. Some objects may represent common entities in between these two configurations. Once discovered, we will call these “matched” objects. The set of differences in between these two configurations will be referred to as the delta in this thesis.

The goal of this thesis is to determine automatically the delta between two AMF configurations. The differences may fall into 3 categories: there is a new object in the target, an object exists in the source but does not exist in the target, and an object from the source is changed in the target.

3.2 Configuration Difference Generation Based on Distinguished Names

As we have previously seen in the explanation of IMM, all objects in a configuration are uniquely identified by their DN. Generating a delta is thereby straight forwards and consists of two phases. The first phase is associating objects in between configurations, the second is property comparison within associated objects.

3.2.1 Object Matching Between Configurations

We start by iterating over all objects in the source configuration. For each source object, we take its DN and attempt to find a target object with the same DN in the target configuration by doing a linear search over all its objects. If we find such an object, we mark both the source and target objects for comparison in the next step. Objects not found are marked for removal. Finally, we iterate over all objects in the target configuration. Objects not previously marked either for removal or for comparison are marked for addition. This separation of the source and target configurations into three sets is illustrated Figure 8.

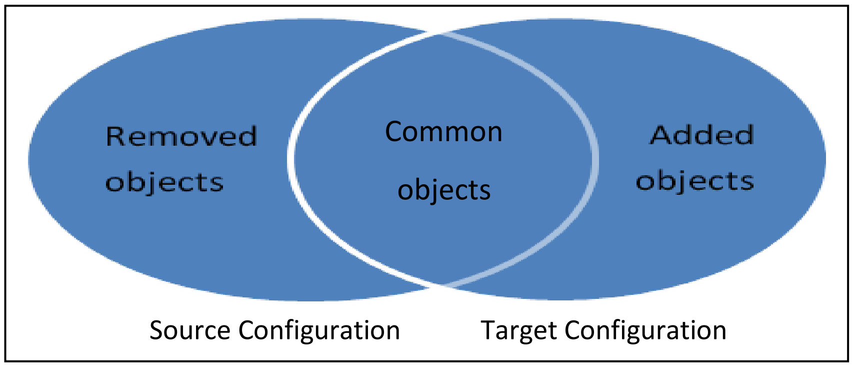


Figure 8: Object Sets

3.2.2 Property Value Comparison

After matching common objects, we need to compare them to see if their properties have been changed. This can be done as a property by property comparison since IMM and by extension the AMF model's most complex data type is a list.

3.3 Configuration Difference Generation without Distinguished Names

Unfortunately, the ideal situation where all DNs are properly populated in our target configuration cannot be relied upon. Below we detail the algorithm to perform the difference generation with restricted information. We will first introduce the challenges by explaining the limitations of difference generation based on DNs and analyse the AMF model to show that a more complex algorithm is needed. Next, we will make some assumptions to make this problem tractable. Next, we will show the algorithm by first giving a high level overview, then, in groups, we will give the mapping methods for each class. These groups actually reflect the mapping method used. They are: Single Instance Class, Generic Class, Types and finally Special Purpose. This last group contains the special purpose algorithms for Applications, SGs, SUs and Components. Next we will show how they all fit together by showing their dependencies. Finally, we will discuss value comparison and output modifications to fit the MAGIC Upgrade Campaign Generator.

3.3.1 Introduction and Reasoning

As part of the MAGIC project, we have run into a problem that we expect most automatic configuration generation systems will face: the Relative Distinguished Names of objects, and thus the Distinguished Names, are automatically generated and may not be consistent with either hand generated configurations or configurations generated with different requirements. Since SAF systems are expected to change over time, caused by software and hardware upgrades or new client requirements, we expect this problem to arise at some point in every cluster's lifecycle.

While the approach explained above in section 3.2 will still operate on these inconsistent configurations, the resulting change set will likely be similar to dropping the content of the cluster and reloading a new configuration. This would imply that unnecessary service downtime is required to perform the upgrade. This is simply not acceptable for Highly Available systems.

The objective of our delta generation algorithm is therefore to derive a set of differences between a source and target configuration; and to do so with limited knowledge about Distinguished Names.

In the explanation presented in section 3.2, the DNs are used to provide globally unique identifiers that allow elements in the source and target configurations to be uniquely matched. We need to find other means that will allow us to match objects in these configurations. We will do that by first analysing IMM and AMF to derive properties that we can use to perform these matches. These properties will fall into two categories: structural and value based arguments. We will then complement these with service based assumptions to allow us to complete our algorithm.

3.3.1.1 Structural Properties

We will be calling structural properties, anything that has to do with the IMM hierarchy. These arguments tend to be very basic, but are still noteworthy.

Property 1. IMM and by extension AMF object have a fixed class that cannot be changed during an upgrade.

This means that, for instance, an SU cannot become a Component. We can conclude from this that we do not need to compare objects of different classes with each other to see if they match. However, this should not be taken as meaning that object relationships cannot change: for instance, an SI can change what SG protects it.

Property 2. An object cannot change its parent.

IMM does not allow moving an object, though it does allow deletion and addition. While our objective is an algorithm without knowledge of DNs, there is no assumption that we are missing parts of the hierarchy. That is to say, our requirements do not imply that structurally significant objects are missing or that child objects are placed under incorrect parents. This gives us three important properties:

Property 3. If a parent object has been added or removed then all its children have also been added or removed, respectively.

Property 4. If two child objects match in the source and target configuration respectively, then their parents must match.

Property 5. Given that a parent object and its match have been identified in the source and target configurations, a child object's match must, if it exists, be a child of the parent's match.

Reading more into this last property, we get:

Property 6. Given matched parent objects, only a criterion that is unique amongst those children needs to be found to match those children.

Similarly, could the mere existence of a child object be a clue towards the parent? While an upgrade campaign does restrict in what state certain objects have to be in before a removal/addition is made, it does not actually pose any restriction on such changes. Since we only have snapshots of before and after an upgrade we cannot glean any information on child objects without having identified them.

Next, let us consider association objects. In AMF which objects are being associated is defined by the object's DN. While we have indicated we cannot use that DN, the association it represents is still valid. One side of the association is the parent object and the other is a reference to the target object. Since you can only have one association object representing a given relation in between two objects (otherwise the DNs of these association objects would be identical which is impossible), we obtain the following property:

Property 7. If two sets of objects match in their source and target configuration respectively, then the association object linking these, if it exists, must either be a match or be the result of a remove/add operation pair.

There is little difference in between the two cases expressed in the above property other than that read-only properties cannot be modified and that, therefore, any such changes indicate a remove/add operation pair. As a corollary, we get:

Property 8. If two association objects are found to match, then their endpoints must match.

However, it should be noted that we will not be using this property because matching up association object without having mapped the objects they associate is virtually impossible, as we will see in section 3.3.1.2.

Finally, there are certain objects that are expected to be unique. On the AMF side SaAmfCluster and SaAmfCompGlobalAttributes are both objects for which only one instance can exist. Expanding this to other services we will be using, SaClmCluster is also unique. We can easily assume that since a configuration can only have one of each of these, that they are the same. Strictly, that is not true. However, forcing a remove/add cycle on these objects has impacts comparable to wiping the cluster: removing SaAmfCompGlobalAttributes would mean having to remove all Components first; removing a cluster object would require removing all nodes from the environment. These are changes that are sufficiently large as to be unlikely. Furthermore, all configuration properties of these objects are writeable, so there is no appreciable value to dropping these, other than possibly changing their name. Future assumptions made in section 3.3.2 will further solidify this handling.

While the above properties cannot be directly used to make exact matches, other than for three specific objects, they are useful in limiting the possible matches. Let us look at property values to see if we can find identifying information.

3.3.1.2 Value Based Properties

Since AMF objects contain properties, an immediate question is if their values could help us match objects. The key problem with this is that the objective of this delta calculation is to be able to perform an upgrade. Certain values must be able to be changed. However, AMF does have read only properties. Clearly, if these read only properties do not match, then these objects cannot be the same. To analyse this, let us first classify all AMF properties into three main categories: runtime, RDN and configuration properties.

To begin, clearly runtime properties cannot be used for determining anything as our delta calculation algorithm works offline, just as the rest of the programs currently in the MAGIC project. Next RDN properties, that is to say properties that contain the RDN of the object, could actually be quite useful. Every AMF object has one such property; it is read-only and locally unique. Used in combination with Property 6 about matched parents and locally unique children, we would actually be able to use this to identify all objects, since even top level classes have these RDN properties. However, as we have indicated in section 3.3.1, these are not values we can trust as they may have been automatically generated.

This leaves us with configuration properties. We will further classify them into two categories: modifiable and read-only. Clearly, we cannot rely on the modifiable ones for identification as their values may change. This leaves the read-only ones. If we make a list of these, we find that, excluding association classes, only 4 classes have read-only configuration properties. These are: SaAmfComp (saAmfCompCmdEnv), SaAmfSGType (saAmfSgtRedundancyModel), SaAmfSUType (saAmfSutIsExternal), SaAmfCompType (saAmfCtCompCategory, saAmfCtSwBundle, saAmfCtRelPath(Instantiate|Terminate|Cleanup|Am(Start|Stop))Cmd). Furthermore, there is no expectation that any of these properties, or even the ones in association classes, are unique. Quite on the contrary, we would actually expect many of these to be the same amongst peers.

Based on this analysis, the model does not contain enough information for us to derive an exact set of matches. We need to add assumptions.

3.3.2 Assumptions

At all stages in designing this algorithm several different approaches and many different assumptions could be considered. Since SAF compliant systems are to be highly available, we believe it is to be taken for granted that the objective of the managed system is to provide continuous service. In that regard, maintaining service availability is a key requirement. First, we decided that our general guidelines for these choices should be this service availability. We can therefore eliminate possible paths that necessarily result in downtime. Second, we will be using a top down hierarchical method for matching objects. This is both based on the fact that most of our structural rules tend to go from parent to child and that, when we actually attempted to derive a bottom up approach, we found our-selves almost immediately going into probabilistic methods. Since a more structured approach was desired, we focused our-selves on the top down approach.

We will go through this approach by first looking at what a service is, then going through the AMF hierarchy starting from the top level classes down to Components, then handling the remaining type classes and miscellaneous classes.

3.3.2.1 AMF Services

Being able to maintain service availability means that we need to know what services are required to be available. SI, CSI, SvcType and CSType are the classes that define what services are. However, aside their RDN property, they have no read-only properties or associations, let alone ones that could positively and uniquely identify them. This leads to our initial assumptions:

Assumption I. We know the Relative Distinguished Names of Service Instances (SI) and they are globally unique.

Assumption II. We know the Relative Distinguished Names of Component Service Instances (CSI).

Assumption III. We know the Distinguished Names of Service Types (SvcType), Component Service Types (CSType).

As alternative to Assumption I, we could have chosen: “We know the DNs of Service Instances.” However, this would also immediately imply that we knew the DN of the parent Application. We wished to try to keep the number of known DNs minimal, so we excluded that. However, as we will see in the next section, since matching Applications is the step right after resolving services, very little would change if this alternate assumption were to be used instead.

3.3.2.2 Top Level Classes

The diagram Figure 9 gives us a view of the AMF hierarchy. We can see that there are a large number of top level elements. For the moment, let us ignore the Types, and instead focus on Software Bundles (SwBundle) and NodeGroups, then CLM and AMF Nodes and finally Applications.

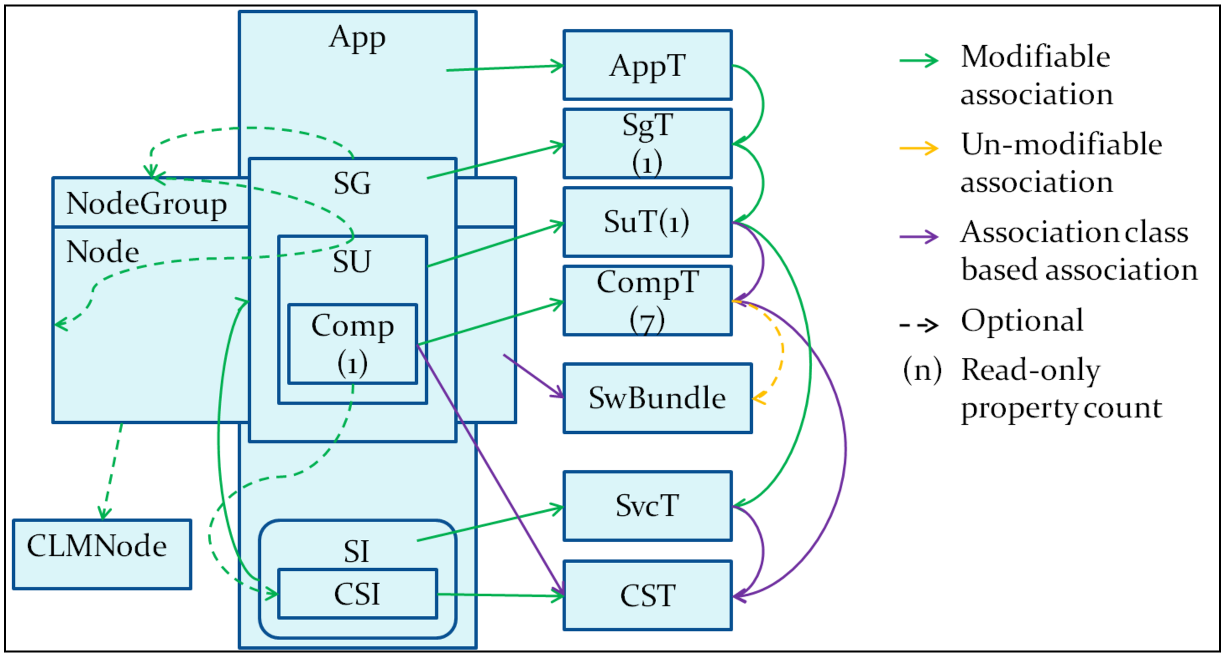


Figure 9: Hierarchy and associations of AMF classes

Unfortunately both SwBundles and NodeGroups have no read-only properties. Furthermore, while SwBundle does have an incoming read-only association from CompType, we will see in section 3.3.2.6 that they are utterly replaceable. In both cases, this means that we do not have enough information to map them without their names. This leads to the following assumption:

Assumption IV. We know the Distinguished Names of NodeGroups (NodeGroup) and Software Bundles (SwBundle)

CLMNodes represent actual hardware, be it virtual or not, in the system. We believe their names are auto-generable, so we can actually rely on their names as a source of information. Furthermore, they have an association with Node. While allowed by the specification, we do not believe that AMF Nodes switching physical location should be considered a simple property modification: software, data, etc... would need to be moved in between nodes. This should instead be considered as a remove/add cycle. This leads to the following assumptions:

Assumption V. CLMNode Distinguished Names are known.

Assumption VI. Nodes always specify CLMNodes and this property's modification is equivalent to the deletion/addition of the Node.

For Applications, from our service requirements, we already have mappings for SIs. However, SIs are child objects of Applications. Given Property 3, we can deduce the Application mappings based on our existing SI mappings.

3.3.2.3 Service Groups

Referring back to section 2.1.4.3.3, we see that SIs have a mandatory association to an SG. In fact, it indicates what SG provides its functionality and protects it. This property is not modifiable without incurring downtime. So we could make the following assumption:

Assumption VII. SIs cannot change SG

However, we can be more tolerant. While making this assumption, if we find that two SIs that used to be in the same SG no longer are, or if we find that two SIs that were in different SGs are now in the same SG, we can prompt the user to ask which service must not suffer downtime. This will allow us to exclude the SIs that moved and perform our matching using only SIs that satisfy our assumption.

3.3.2.4 Service Units

Unfortunately, if we look at SU's properties, none of them are read-only. However, at this point we have a mapping for SGs, Nodes and NodeGroups. SUs do have optional associations to a Node or NodeGroup. If not specified, the parent SG will have one or it is considered that this SU can be deployed on any node in the cluster.

In the case where only a NodeGroup is available, we have that every node in the group must be able to provide the service given that SU definition. Furthermore, all SUs in a NodeGroup must be able to support all SIs protected by the SG we are considering. Since all SUs need to work on all Nodes and all SUs provide the same functionality, it is not much of a stretch to assume that all SUs in an SG associated with the same NodeGroup are identical.

In the case where a specific Node is specified, one might expect that no other SU in the same SG would be assigned to the same node: it would lead to potential redundancy failure if active and standby instances were assigned on the same Node. However, since an overwhelming proportion of failures are software related, redundant SUs on the same node do make sense. We propose to resolve this conflict in the same way as NodeGroups. That is to say the assumption that all SUs in an SG associated with the same Node be identical. We obtain the following assumption:

Assumption VIII. All SUs in an SG associated with the same Node or NodeGroup are identical.

Also, we note that the SU's associated Node or NodeGroup can be changed. However, that can only happen with the SU locked. Furthermore, if software is needed to allow the SU's Components to function, these would have to be moved too. To us, that seems functionally equivalent to a remove/add cycle. This gives us our next assumption:

Assumption IX. SUs do not change Node or NodeGroups, they are removed and then re-added

3.3.2.5 Components

While internally Component has one read-only property (saAmfCompCmdEnv), there is no expectation that it be unique. We cannot rely on it alone. However, Component does have a noteworthy association class: SaAmfCompCsType. Services define what they need to function by indicating CSTypes that satisfy their needs. SaAmfCompCsType indicates what CSType a Component provides. So, from a service perspective, this association class indicates what potential CSI a Component can handle. We propose to classify based on that. However, this association both is not unique and is subject to changes in upgrades. We propose to solve this with the following assumption:

Assumption X. Components within a given SU and providing the same set of CSTypes (ignoring CSTypes not used by any CSI protected by the SG) are identical.

3.3.2.6 Types

Only four of the classes listed Figure 9 remain. These are AppType, SGType, SUType and CompType. We note that all associations going to them from objects of previously treated classes are modifiable. In fact, changing those associations is not service affecting. Therefore treating them all as objects removed from the source configuration and newly added to the target configuration satisfies our requirements. However, to attempt to keep a smaller output, we will attempt to map some of them. The method for this will be described in the Implementation section.

3.3.2.7 Remaining classes

A couple classes are not listed in Figure 9. These can be categorized into three groups: single instance classes, association classes and option classes. For single instance classes, we have indicated how we would handle matching them in 3.3.1.1: they must be the same object in the old and new models. For association classes, we will map them using the fact that we have matched both their endpoints as outlined in Property 7. Finally, what we call "option classes" are classes where the RDN is intrinsic to the meaning of the object: they define what the object does. Therefore we need to know the RDN. There are 3 such classes: CSIAttribute, Healthcheck and HealthcheckType. This gives us the following assumption:

Assumption XI. The Relative Distinguished Names of SaAmfCSIAttribute, SaAmfHealthcheckType and SaAmfHealthcheck are known.

3.3.3 Overall View of Algorithm

As explained in section 3.3.1, the algorithm is divided into two phases:

1) Find the objects in the target configuration that were in the source configuration. By extrapolation, the objects found in the source configuration and not found in the target configuration are marked for removal. Contrapositively, the objects found in the target configuration but not the source one are mark for addition.
2) Iteratively go through the matched object hierarchy from step 1; "adding" new child object hierarchies, finding property value differences in common objects and "removing" deleted object hierarchies.

3.3.4 Mapping Single Instance Classes

Several classes only have one instance object permissible within a configuration. As indicated in section 3.3.2.7, we assume these are the same objects in the old and new models. These are:

1. SaAmfCluster
2. SaClmCluster

3. SaAmfCompGlobalAttributes

As these are all root objects they can be matched immediately without having to process any parents.

3.3.5 Generic Mapping Algorithm

For the mapping of the remaining classes, the majority of the work can be done by two generic functions. The first one is Algorithm 1. It performs the following action: Given a set of objects from each of the source and target configurations, based on a reference property pointing to a mapped object, categorize the objects in the sets “new”, “old” and “common”. Furthermore, for this last category, retain the mapping information of the matched pairs.

```
Func generic_map( in guide, in old_set: Object[], in new_set: Object[],
                  out news: Object[], out olds: Object[], out commons: pair<Object>[]):

// First create a map based on old objects and the reference property
def associated_obj_map: α => Object
foreach old_obj in old_set:
    associated_obj_map.add(guide.get_associated_α_from_old(old_obj), old_obj)
```

continued on next page

```
// Now, try to match them up with the new set of objects.
foreach new_obj in new_set:
    def old_obj =
associated_obj_map.find(guide.get_associated_α_from_new(new_obj))
    if old_obj was not found:
            news.add(new_obj) // No old object matches? It is truly new. Add it.
    else
            commons.add(old_obj, new_obj)
// Remove the old object so that we have a clean list of what was mapped
associated_obj_map.remove(old_obj)

// At this point the only objects remaining in associated_obj_map are those that have
// no associated new object in the new set, since we have removed them all.
// This means it contains the set of old objects. Assign them as such.
olds = associated_obj_map.get_values_from_map()

end
```

Algorithm 1: Generic Mapping Function

In Algorithm 1, "guide" is used as an object of unspecified type that provides methods to get the association value out of the new and old objects. To illustrate the usefulness of this function, let us consider then sets of children of known mapped objects; children for which the RDN is known good. With α being a String and with the guide.get_associated_α_from_* functions simply returning the RDN of each object, this function allows the categorisation and mapping of these child objects.

In fact, all classes of objects, except Applications, SGs, SUs, Components, their Types and, for simplicity's sake, single instance classes, can be mapped using this function and the proper guide. We will demonstrate this by listing, in Table 25, each supported classes with its parent sets and its common local identification, thereby satisfying Property 6.

Class	Logical parent	Common name
SaAmfSI	Root (via. Application)	RDN; known and globally unique from Assumption I
SaAmfCSI	SaAmfSI	RDN; known from Assumption II
SaAmfSvcType	Root	DN; known from Assumption III
SaAmfCSType	Root	DN; known from Assumption III
SaClmNode	SaClmCluster	RDN; known from Assumption V
SaAmfNode	SaAmfCluster	SaClmNode, read-only from Assumption VI, unique by specification
SaAmfNodeGroup	SaAmfCluster	RDN; known from Assumption IV
SaSmfSWBundle	Root (entire DN is known and parent objects are not used, so we can ignore its hierarchy)	DN; known from Assumption IV
SaAmfCSIAttribute	SaAmfCSI	RDN; known from Assumption XI
SaAmfNodeSWBundle	SaAmfNode	SaSmfSWBundle, read-only and unique by specification
SaAmfSIDependency	SaAmfSI	SaAmfSI, read-only and unique by specification
SaAmfSIRankedSU	SaAmfSI	SaAmfSU, read-only and unique by specification
SaAmfSvcTypeCSType	SaAmfSvcType	SaAmfCSType, read-only and unique by specification
SaAmfCompCSType	SaAmfComp	SaAmfCSType, read-only and unique by specification
SaAmfSutCompType	SaAmfSUType	SaAmfCompType, read-only and unique by specification
SaAmfHealthcheck	SaAmfComp	RDN; known from Assumption XI
SaAmfHealthcheckType	SaAmfCompType	RDN; known from Assumption XI
SaAmfCtCsType	SaAmfCompType	SaAmfCSType, read-only and unique by specification

Table 25: Classes matched using Generic Map Algorithm

3.3.6 Generic Mapping of Remaining Types

The mapping of the remaining Types can also be handled in a generic manner. When looking at this algorithm, we would like to remind you that the operation of changing a Type association is not in and of its-self service affecting. It only affects a service if a requested change in the values of the Type affects service. Therefore the objective of the function is slightly different. Its objective is to minimise the number of differences found by the algorithm, thereby shortening output.

The function present in Algorithm 2 performs the following action: Given the sets of all types of a given class from each of the source and target configurations, categorize the objects in these sets as “new”, “old”, and “common”. Furthermore, in this last category, retain the information on the mapped pairs.

At this point, the criteria the algorithm uses are non-optimal, but good enough. The algorithm decides that two Types match by performing the following check: when iterating over the set of matched objects for which the class of Types under consideration is relevant, if neither the source nor the target Types are already in our common set and a compatibility checks passes, then they match and are added to our common set.

This is non-optimal because it is possible that a different mapping would produce fewer assignments. Consider an example with a source configuration containing 3 SUs of type A, and a target configuration with 1 SU of type B, and 2 of type C, all compatible. If this algorithm falls first on the SU of type B, it would result in two type assignments, instead of the opposite case where only the SU of type B would need the assignment. So why not optimize for this possibility? The issue is where to stop such minimisation considerations. For instance, SGTypes also have associations to SUTypes. Their updates would need to be minimized too. Further, the number of properties to be modified to obtain the new SUType, given that there may be several potential source SUTypes, should also be considered. These extra criteria vastly complicate the algorithm and the value of their improvement is debatable. For this reason, we chose to simply choose the first Type encountered.

```
Func generic_type_map( in guide, in common_objects: pair<Object>[],
            in old_type_set: Object[], in new_type_set: Object[],
            out news: Object[], out olds: Object[], out commons: pair<Object>[]):

// Try to match up types using pairs of mapped objects.
foreach old_obj, new_obj in common_objects:
    if new_obj.get_type in commons // if new type already mapped
            or old_obj.get_type in commons // or old type already mapped
            or guide.are_incompatible(new_obj.get_type, old_obj.get_type):
            continue
    // Types are compatible, say that they are the same
    old_type_set.remove(old_obj.get_type)
    commons.add(new_obj.get_type, old_obj.get_type)
    new_type_set.remove(new_obj.get_type)

// Since we have removed the common types in the new and old set of types,
// all that remains are the new and old types. Assign them as such.
olds = old_type_set
news = new_type_set

end
```

Algorithm 2: Generic Type Mapping Function

In the function presented in Algorithm 2, "common_objects" refers to the "commons" object output array for the class of objects for which are relevant to the class of Types we are trying to analyse. "Guide" is again of an unspecified type.

In this case, we see that "guide" only needs to provide one function that indicates whether two objects are incompatible. The incompatibilities can only arise in the case where there are read-only properties that are not equal. These are listed in Table 26.

Class	Read-only properties
SaAmfAppType	None
SaAmfSGType	saAmfSgtRedundancyModel
SaAmfSUType	saAmfSutIsExternal
SaAmfCompType	saAmfCtCompCategory saAmfCtSwBundle saAmfCtDefCmdEnv saAmfCtRelPathInstantiateCmd saAmfCtRelPathTerminateCmd saAmfCtRelPathCleanupCmd saAmfCtRelPathAmStartCmd saAmfCtRelPathAmStopCmd

Table 26: Type classes with read only properties

3.3.7 Handling of Remaining Classes

At this point the resolutions of only 4 classes of objects remain. These are Application, SG, SU and Component. All of them require special purpose algorithms so we will go over them one by one.

3.3.7.1 SaAmfApplication

From section 3.3.2.2 Top Level Classes, we have that Applications can be resolved through an association with SI. Following the same variable naming convention as previous functions, we get Algorithm 3.

```
Func map_application( in old_set: App[], in new_set: App [], in common_SI: pair<SI>[],
                out news: App [], out olds: App [], out commons: pair<App>[]):

foreach old_si, new_si in common_SI:
    if old_si.get_parent in old_set
                and new_si.get_parent in new_set:
                commons.add(old_si.get_parent, new_si.get_parent)
                old_set.remove(old_si.get_parent)
                new_set.remove(new_si.get_parent)
    else
                assert( commons.find(old_si.get_parent) == new_si.get_parent )

// Since we have removed the common Applications in the new and old sets,
// all that remains are the new and old Applications. Assign them as such.
olds = old_ set
news = new_set

end
```

Algorithm 3: Application Mapping Function

Note: if you actually look at the actions performed by Algorithm 3, they are extremely similar to Algorithm 2’s generic_type_map.

3.3.7.2 SaAmfSG

From Assumption VII we would have that, as with Applications, SGs can be resolved through an association with SI. The algorithm, presented bellow as Algorithm 4, would be virtually identical. However, we would like to be able to detect SI migrations and prompt the user for conflict resolution. To do this we need to:

- Group all SIs from the common object space into sets of SIs that map from a same source SG to a same target SG
- Create two maps. One associates all source configuration SGs with their potential target configuration SGs. The other associates all target configuration SGs with their potential source configuration SGs.
- If every SG in both sets only has at most one association, then no conflict exists. Otherwise we must ask the user to perform conflict resolution.
- Conflict resolution consists of presenting the user with a list of the SI sets, and asking which will and will not be suffering downtime. The answer is therefore ternary, with the third option being "unanswered".
    - For each SI set, if the answer is "no downtime" then we know that the source and target SG must be the same. This allows us to remove all questions that involve either of these two SGs and their corresponding associations.
    - If the answer is "yes downtime" then we simply remove this possible SG association.
    - Conflict resolution is completed when there is at most 1 forward and backward association for each SG.

- We now have a map with no conflicts. We need to copy it to the common set and populate the add and remove sets based on what is not in the common set.

```
Func map_SG( in old_set: SG[], in new_set: SG [], in common_SI: pair<SI>[],
                out news: SG [], out olds: SG [], out commons: pair<SG>[]):

def forward_map = new Map<SG,map<SG, list<pair<SI>>>>
def backward_map = new Map<SG,map<SG,list<pair<SI>>>>
def need_resolution = false

// Make forward and backward sets
foreach old_si, new_si in common_SI:
    forward_map[old_si.protectedBySG][new_si.protectedBySG].add(old_si,new_si)
    backward_map[old_si.protectedBySG][new_si.protectedBySG].add(old_si,new_si)

// Check for conflicts
foreach maps in forward_map:
    if maps.size > 1
                need_resolution = true
foreach maps in backward_map:
    if maps.size > 1
                need_resolution = true
```

continued on next page

```
if need_resolution
    perform_resolution(forward_map, backward_map)

// Conflict resolution is over, perform the merge
foreach old_sg, map in forward_map:
    foreach new_sg, si_list in map:
            if si_list.size() > 0:
                    // conflict resolution guarantees this will only iterate up to once per SG
                    commons.add(old_sg, new_sg)
                    old_set.remove(old_sg)
                    new_set.remove(new_sg)

// Since we have removed the common SG in the new and old sets,
// all that remains are the new and old SGs. Assign them as such.
olds = old_ set
news = new_set

end
```

Algorithm 4: SG Mapping Function

Note that in Algorithm 4, old_set and new_set must only contain SG from matched Applications. Otherwise both those SGs and their parents would be marked for addition.

The perform_resolution function is best served by a graphical interface, so we will not give specific code for it. However, we can give an algorithm for what actions to perform when one of the “yes downtime” or “no downtime” options are selected. It is presented as Algorithm 5.

```
Func mark_relation( in source: SG, in destination: SG, in downtime,
            inout forward_map: map<SG,map<SG,...>>,
            inout back_map: map<SG,map<SG,...>>):

if downtime == “no”:
    // Find all competing relations
    def forward_list = forward_map[source].keys
    forward_list.remove(destination)
    def backward_list = forward_map[destination].keys
    backward_list.remove(source)

    // Remove them from the maps
    foreach key in  forward_list:
            forward _map[source].delete(key)
            backward _map[key].delete(source)
    foreach key in  backward_list:
            backward _map[destination].delete(key)
            forward _map[key].delete(destination)
```

continued on next page

```
    // If an implementation requires to do additional steps on solved relations,
    // it should iterate again over both lists and check the following:
    //      map[key].size == 1 and other_map[map[key].first.key].size == 1
    // if it evaluates to true, then that relation is newly resolved.

else if downtime == "yes":
    forward _map[source].delete(destination)
    backward _map[destination].delete(source)

    // If an implementation requires to do additional steps on solved relations,
    // it should check the following:
    //      forward_map[source].size == 1
    //              and backward_map[forward_map[source]first.key].size == 1
    //      backward_map[destination].size == 1
    //              and forward_map[backward_map[destination]first.key].size==1
    // if either evaluates to true, then that relation is newly resolved.
```

Algorithm 5: Mark Relation Function

3.3.7.3 SaAmfSU

Assumption VIII and Assumption IX allow us to safely ignore any differences in SUs insatiable on a same Node/NodeGroup when trying to map SUs. Therefore the rough algorithm for mapping SUs is as follows:

- For each SG that is common to both the new and old sets:
    - Collect all SUs into groups based on their Nodes/NodeGroups. All SUs without a Node/NodeGroup go into the “cluster” group.
    - Arbitrarily map SUs within their given set.

Let has_mapping() and get_mapping() be functions that respectively indicate if there is and return the mapping of an object handled in a previous step. For simplicity’s sake, let us assume that these functions also properly resolve the “cluster” group. This gives Algorithm 7 for SU mapping. It requires a helper function to find what Node or NodeGroup will host the SU. It is presented as Algorithm 6.

```
// Helper function to find the associated node or node group of an SU
Func get_node_or_nodegroup(in su: SU): α
def key = su.get_NodeOrNodeGroup
if not key:
    key = su.get_parent.get_NodeGroup
if not key:
    key = “cluster”
return key
```

Algorithm 6: Helper Function to Find an SU's associated Node or NodeGroup

```
Func map_SU( in old_set: SU[], in new_set: SU[],
            out news: SU[], out olds: SU[], out commons: pair<SU>[]):

// First create a mapping based on Nodes/NodeGroups as reference property
def associated_N_map: α => SU[]
foreach su in old_set:
    def key = get_node_or_nodegroup (su)
    if key not in associated_N_map:
            associated_N_map.add(key, new SU list )
 associated_N_map.find(key).add(su)

// Now, try to match them up with the new set of objects.
foreach su in new_set:
    def key = get_node_or_nodegroup (su)
    if  has_mapping(key):
            def group = associated_N_map.find(get_mapping(key))
            if group:
                    def old_su = group.pop()
                    if old_su:
                            common.add(old_su, su)
                            continue
    news.add(su)
```

continued on next page

```
// At this point the only objects remaining in associated_N_map are those that have
// no associated new object in the new set, since we have removed them all.
foreach group in associated_N_map:
    foreach su in group:
        olds.add(su)

end
```

Algorithm 7: SU Mapping Function

3.3.7.4 SaAmfComp

Assumption X eliminates our need to exactly match Components. So, based on that assumption, we propose an algorithm that has the following general lines:

1) Create a CSType mask of mapped CSTypes used by SIs protected by our parent SG
2) Group our source Components based on provided CSTypes in the mask
3) Match them with target Components based on their set of covered CSTypes, taking into account property compatibility.

Again, we will assume we have the has_mapping() and get_mapping() functions. Furthermore, we will introduce two helper functions get_mask, shown as Algorithm 8, and get_cst_set, show as Algorithm 9, that respectively create a CSType mask from an SU and return what masked CSTypes a Component satisfies. Finally, our map_Component function, shown as Algorithm 10, should only be called for the components of matched SUs.

```
// Helper function to create the CSType mask
Func get_mask (in su: SU):  Set<CSType>
def cstype_set: Set<CSType>
foreach si in su.get_parent_sg.get_protected_si():
    foreach csi in si.get_child_csis():
        foreach cst in csi.get_cstypes():
            if has_mapping(cst):
                cstype_set.add(cst)
return cstype_set
```

Algorithm 8: CSType Mask Creation Helper Function

```
// Helper function to get the masked cstype set from the component
// This will map the CSTypes to the other (source/target) configuration if
// need_other is true
Func  get_cst_set (in comp: Component, in mask:  Set<CSType>,
                  in need_other: bool): Set<CSType>
def cstype_set: Set<CSType>
foreach comp_cst in comp.get_compcsts ():
      if comp_cst.get_cst() in mask:
if need_other:
      cstype_set.add(get_mapped(comp_cst.get_cst()))
else
      cstype_set.add(comp_cst.get_cst())
return cstype_set
```

Algorithm 9: CSType Mask Fetch Helper Function

```
Func map_Component( in old_set: Component[], in new_set: Component[],
                  in parent_su: SU,
                  out news: SU[], out olds: SU[], out commons: pair<SU>[]):

// First create a map based on associated CSTypes
def mask = get_mask(parent_su)
def associated_cst_map: Set<CSType> => Component[]
foreach comp in old_set:
      def cst_set = get_cst_set(comp, mask, false)
      if cst_set not in associated_cst_map:
                  associated_cst_map.add(cst_set, new Component list )
      associated_cst_map.find(cst_set).add(comp)
```

continued on next page

```
14 // Now, try to match them up with the new set of objects.
15 foreach comp in new_set:
16     def cst_set = get_cst_set(comp, mask, true)
17     if cst_set in associated_cst_map:
18         def group = associated_cst_map.find(cst_set)
19         if group:
20             foreach old_comp in group:
21                 if old_comp.saAmfCompCmdEnv == comp.saAmfCompCmdEnv:
22                     group.del(old_comp)
23                     common.add(old_comp, comp)
24                     continue line 15
25     // A match was not found, so this component is new
26     news.add(comp)
27
28 // At this point the only objects remaining in associated_cst_map are those that have
29 // no associated new object in the new set, since we have removed all those.
30 foreach group in associated_cst_map:
31     foreach comp in group:
32         olds.add(comp)
33
34 end
```

Algorithm 10: Component Mapping Function

Please note that as written, map_Component has a complexity of O(n^2). Changing associated_cst_map to have as right hand term a hash of Component arrays, with the value of saAmfCompCmdEnv as key, would drop this down to O(n).

3.3.8 Dependency in mapping objects

The mapping algorithms presented above all take as input a set of mapped objects. This implies an ordering and the possibility for dependency loops. To make sure no such loops exist, we have analysed the dependencies and we show the resultant dependency tree Figure 10. Additionally

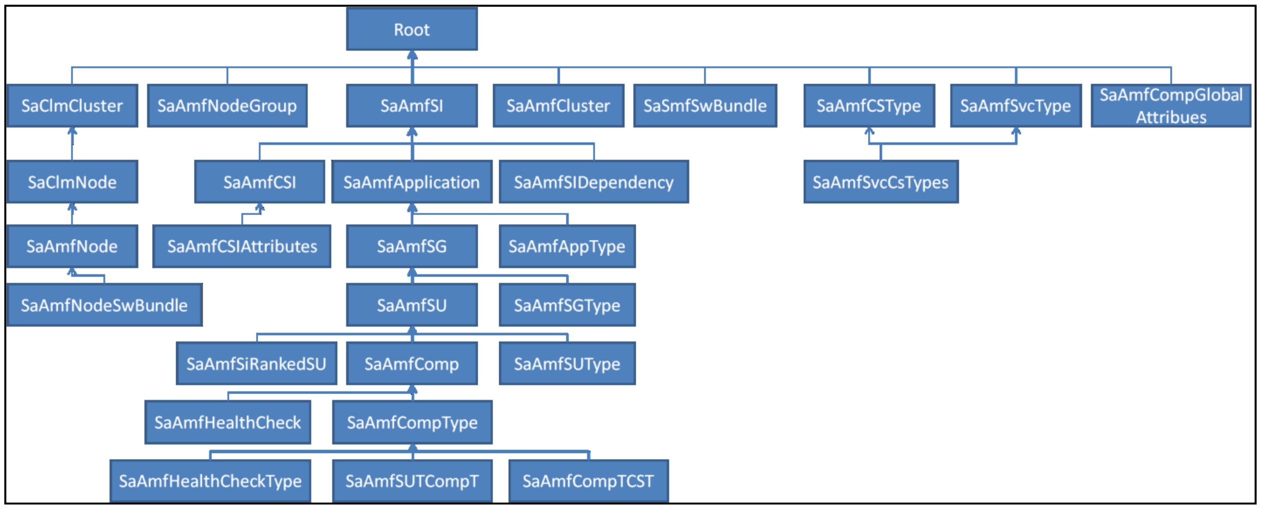


Figure 10: Critical Path Dependency analysis for parallel processing

it shows what classes of objects can be mapped in parallel.

3.3.9 Finding Differences

Phase two consists of iterating through the IMM hierarchy to create add/remove notifications and to compare common objects for differences.

Simultaneous hierarchical iteration through both source and target configurations can be performed using a recursive process. It starts with the root objects of each configuration. They are matched parent objects. For current matched parent object, the children are checked for inclusion in the add/remove lists.

Matched children can then be compared for property value differences. Comparing simple values, and lists thereof, is trivial. For references, the assumption of the existence of the has_mapping() and get_mapping() functions make their comparison equivalently easy. Each of these matched children can then be processed as matched parents, thus completing the recursive process.

3.3.10 Integration with the MAGIC Upgrade Campaign Generator

In a previous related work [KS09], Setareh Kohzadi has created a prototype Upgrade Campaign Generation (UCGen) tool. It consists of a Graphical User Interface (GUI) that requests a source configuration and then allows a user to select modifications through the use of a Wizard. This GUI supplies the inputted information to a backend as a set of tuples containing four data points: a source and target configuration object, a service group these objects belong to and a node list used for limiting where any automatically detected software installations will occur. Addition and removal of objects are handled by providing null references as source and targets respectively.

To be able to integrate our Delta Generator with this tool, we must convert our output to these tuples. It should be noted that some objects in which we can detect changes do not have related Service Groups. Unfortunately that will mean that UCGen cannot handle these changes directly. When indirect support is present we will attempt to use it, and when it is not we will simply have to inform the user that those changes were ignored.

Specifically, UCGen can only perform the following operations that we are interested in: Add SU/Component, Remove SU/Component, and Change Type on SU/Component. It will discover software changes caused by type changes by its-self and also add SGs and Applications if SU/Component adds are requested on non-existing hierarchical branches.

Integrating our Phase II with UCGen is fairly straight forwards. It expects the source model and a list of tuples as inputs. These tuples contain 4 pieces of information: Source, Target, SG, NodeList. The exact data given to it is listed Table 27 below.

	Source	Target	SG	NodeList
Add	ε	Object	Parent SG	γ
Remove	Object	ε	Parent SG	γ
Change Type	Source Object	Target Type	Parent SG	γ

Table 27: Upgrade Campaign Generator Tuple Data Entry

In most cases, the Parent SG value can easily be populated by traversing the IMM hierarchy upwards, looking for an SG. Since the Upgrade Campaign Generator can only handle tuples with an SG specified and, by extension, can only handle classes of objects that are descendants to SGs, for adding/removing parents, we simply need to go down to the SU/SI level and add/remove all their children. The Upgrade Campaign generator will create required parents and prune parents with no children. However, for the remaining classes of objects such as Nodes, we can simply skip tuple creation: their modification is unsupported.

For NodeList, we simply supply all nodes a given object can be hosted on; i.e. we travel up the AMF hierarch looking for values in the HostedNodeGroup and HostedNodeOrNodeGroup properties.

The Add and Remove tuple creation hook can be inserted into the Phase II IMM hierarchy iteration where we check if there are new and old object. The Change Type tuple creation hook can be inserted in the object comparison code. It can be generically trigger if we are presented with a changed property of single valued type EReference with a property name ending with "Type".

3.4 Chapter Conclusion

Our algorithm presented above generates the difference between two configurations given eleven assumptions. We believe that these assumptions are reasonable; however we do not expect that all reasonable configurations will abide by our assumptions.

Specific assumptions that are likely to be problematic are Assumption VIII and Assumption X. They both limit objects to be exactly the same, be they SUs or Components. The main risk with violating these assumptions arises when, for example, given that a component were to be using an exclusive resource "allocated" through a command line configuration parameter, then two Components or SUs swapping these arguments could result in overlapping use of the resource as these components/SUs may not be upgraded at the same time. While the source and target configurations are valid, the difference set generated is likely to induce an Upgrade Campaign Generator to create a campaign with intermediary states that may not be valid.

However, assumption based conflicts are not the only problems an Upgrade Campaign Generator may have to deal with. Component/software version incompatibility may mean that extra steps need to be inserted to do an upgrade to make sure that only compatible versions are running simultaneously.

The final subject we would like to cover is Base Classes. These are parent classes that all Types have. They are interesting in that they only contain a name. Thus their handling is somewhat problematic: if their values are not supposed to be usable, what do we do with them? Should we try to maintain possibly implied associations? The way we have handled them was guided by the MAGIC StdAMF Model implementation. At the time this research was done, it did not have separate instances for these objects, instead integrating each into their Type objects through a class inheritance. This means that, within that context, they have no structural meaning. Thus, we did not use them in our algorithm and our Type matching algorithm skirts around them.

Chapter 4

Prototype Implementation and Case Study

In this chapter, we will go over the implementation of the delta generation tool by seeing how the tool was divided up into separate sections. Then, we will see a brief walkthrough of the tool, followed by the examination of the same case study as that covered in Setareh Kohzadi's thesis [KS09].

4.1 Prototype Architecture

The base design principal of our prototype is the Model, View, Control pattern. Furthermore we split our control into two classes, based on the two phases of our algorithm. This is due to the phases being loosely coupled and that we considered the second phase to be a point of variability: it is likely that it will change as output requirements will be updated.

Based on this, our code is divided into 5 major classes, listed in order of execution:

- our Eclipse platform activator class (helper class)
- our Phase I controller (control)
- our SI-SG migration conflict resolver user interface (view)
- our MagicAMF to StdAMF mapping code (helper class)
- our Phase II controller (control)

The activator class integrates our implementations with the Eclipse platform. This class also mediates in between our Phase I controller and our SI-SG migration conflict resolver user interface by registering a callback function. Other than this, the class is mainly automatically generated and boilerplate code. We will therefore skip it in our explanation.

As mentioned in section 2.2.4, our implementation takes as input a MagicAMF configuration since MAGIC's Multiple Configuration Generators generates files in that format. Our prototype both outputs a complete difference set in its own textual format and is integrated with the MAGIC Upgrade Campaign Generator, which outputs an upgrade campaign file.

4.1.1 Phase I Controller

Our Controller class represents the Phase I algorithm described sections 3.3.3 to 3.3.7. Overall the only modifications made to the given algorithm were to accommodate a different calling semantic. These changes are twofold. The first change is, whereas the given algorithms list how to operate on the "children" of one object, our implementation operates on all object of the same class. On the conceptual level, this can be seen as encapsulating our main algorithm functions by a loop that iterates over all objects of the common set of the parent object class.

The second change is that, to more easily enable access to our mapping sets and to trivialise the implementation of the get_mapping() function, we merged our new, old and common arrays into a single generic helper class (OOMap). We further created an instance of this class for each of our AMF classes and have them stored as properties within the Controller class instance objects. What this means that all our member functions can access them directly.

The combination of these two changes means that, with the exception of the guides for our generic functions, both the input and output of our algorithms are contained within our class instance. This simplifies data flow in between functions: it is entirely done through our instance object.

To glue everything together, a “main” function exists. It simply calls all mapping functions in the right order and then hands off control to Phase II.

4.1.2 SI-SG Migration Resolution

SI-SG migration resolution is performed by the user through an interactive GUI. This obviously is a view and must be handled separately from our Phase I controller. To enable this, our activator class registers a callback when instantiating our Controller. This callback is what instantiates and invokes our SI_Resolver class. The arguments to the callback are the forward and backward SI to SG maps. On invocation these maps contain duplicates. As indicated in section 3.3.7.2, it is expected that on return from the callback, these duplicates be removed.

The actual SI_Resolver class implements a GUI in SWT, shown in Figure 13, that indicates the question “Will service availability be maintained throughout this upgrade on these SIs?”; provides a list of grouped SIs along with yes/no options; and an OK button. It should be noted that this question results in answers opposite to those listed in 3.3.7.2. In our following explanation, we will maintain our same definition for yes/no as in 3.3.7.2.

To ease interaction with the user, after responding to a question with “no”, all implicitly solvable questions are hidden. This has two side benefits: it denies the user the ability to give a conflicting answer and it eases answer maintenance as we do not need to remember if a “yes” answer was based on user selection or implication. Furthermore, the OK button is only activated once all duplicates are resolved.

The ability to change an answer is performed by maintaining a copy of the original input and then re-applying all the user’s answers to it, with the desired modification.

4.1.3 MagicAMF to StdAMF Mapper

In section 1.3 we mentioned that one of our motivations was to bridge the Magic Configuration Generator and Upgrade Campaign Generator. However, while the Configuration Generator uses the MagicAMF model, the upgrade campaign uses the StdAMF model. Similarly, while our Phase I implementation works on MagicAMF models, Phase II, which needs to conform to the Upgrade Campaign input, operates on the StdAMF model. Transforming from one model to the other has already been implemented by Jesus Garcia as separate project. However, we need an additional step. Our data structures indicate the mappings of objects in between two MagicAMF models. We need to make them work with the new StdAMF ones.

To this extent, we have created a mapping class that, based on RDNs and object hierarchy, will map MagicAMF object references to StdAMF object references. This is contained in out Magic2AMFmap class and is invoked as first step of Phase II. It operates generically by traversing simultaneously the MagicAMF and StdAMF hierarch and matching local names, storing resultant equivalencies in a hash. The traversal path used is given as a list of pairs of equivalent classes and naming functions, with child classes and access methods specified for each parent class.

4.1.4 Phase II Controller

Our Phase II class, named Delta, operates as describe in section 3.3.9, by traversing the hierarchy and comparing pairs in the common sets. It does this similarly to the MagicAMF to StdAMF mapper in that this traversal is directed by a data structure, this time only containing a list of parent classes, with lists of children classes and accessor functions. Before comparing two objects, their equivalents in StdAMF are found and then a comparison is done using ECore's reflection interface. This allows us to compare properties without having hard coded knowledge of their name and type, greatly simplifying the task.

Integration with the Upgrade Campaign Generator is done as explained in section 3.3.10. Tuple adaptation to meet SG parenting requirements is done after difference generation is completed. This allows the code to have a global overview of the changes to make any final decisions, and to not have to pollute the generic handling of the hooked code with corner cases.

4.2 Tool Walkthrough and User Interactions

In the following section, we will go over a simple usage of the tool. To be able to illustrate all aspects of the tool, we will be using a scenario different from our case study specifically to be able to demonstrate the usage of our SI Migration dialog. The source configuration is quite simple and has been automatically generated to contain one Application, with one SG and 10 SI's, named SI-sit-web0 through SI-sit-web9. The target configuration differs in that it has a new SG under the same Application and that SI-sit-web0 has been moved over to it.

1. The tool is invoked by clicking on a menu item, illustrated Figure 11.

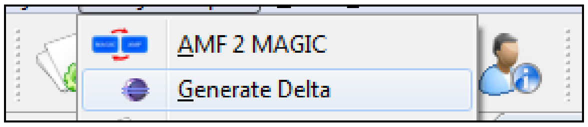
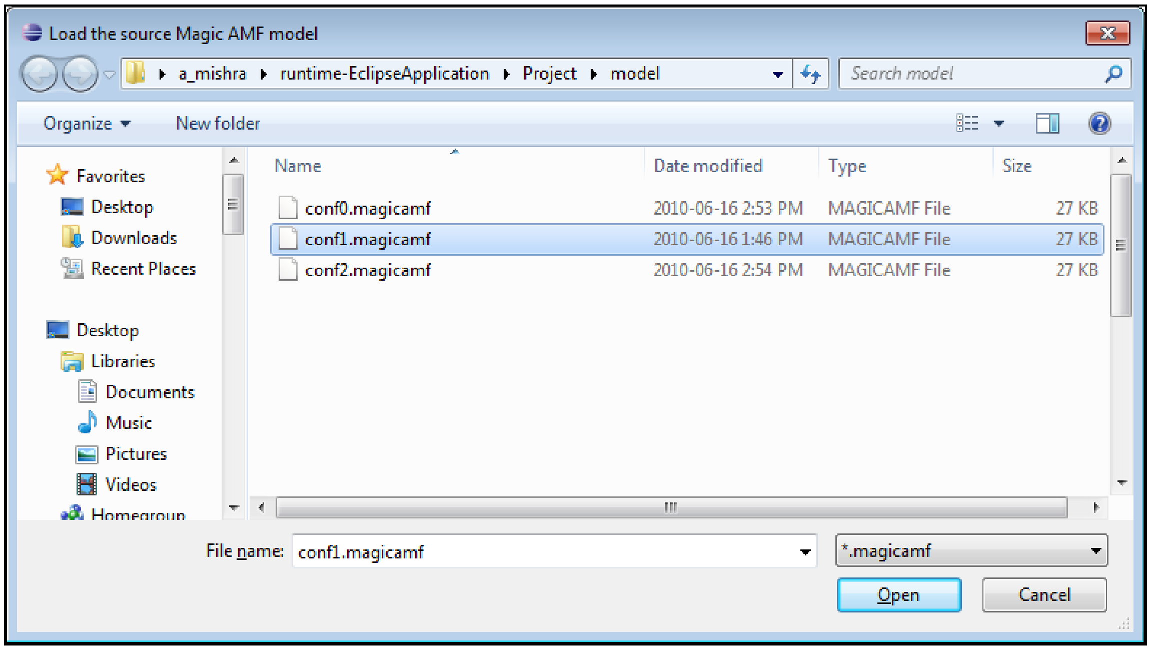


Figure 11: Delta Generation Menu Item

2. The tool then requests twice for input configurations. Once for the source configuration, shown in Figure 12; once for the target.

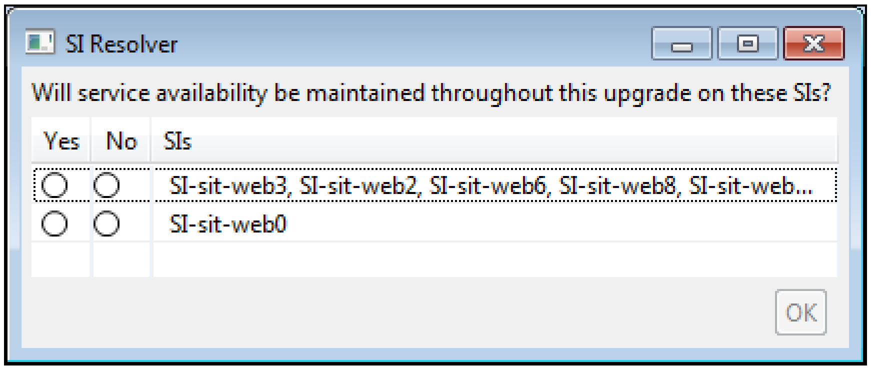


Name	Date modified	Type	Size
conf0.magicamf	2010-06-16 2:53 PM	MAGICAMF File	27 KB
conf1.magicamf	2010-06-16 1:46 PM	MAGICAMF File	27 KB
conf2.magicamf	2010-06-16 2:54 PM	MAGICAMF File	27 KB

File name: conf1.magicamf
*.magicamf

Figure 12: Source Configuration Selection Dialog Box

3. If needed, the SI SG migration dialog box appears. In this case it is needed and is shown in Figure 13. To obtain the results of this walkthrough, one would select “Yes” on the first line of the dialog box shown. The OK button would then become enabled.


SI Resolver

Will service availability be maintained throughout this upgrade on these SIs?

Yes	No	SIs
○	○	SI-sit-web3, SI-sit-web2, SI-sit-web6, SI-sit-web8, SI-sit-web...
○	○	SI-sit-web0

Figure 13: SI SG Migration Conflict Resolution Dialog Box

4. With all inputs to the algorithm completed, the delta is computed.

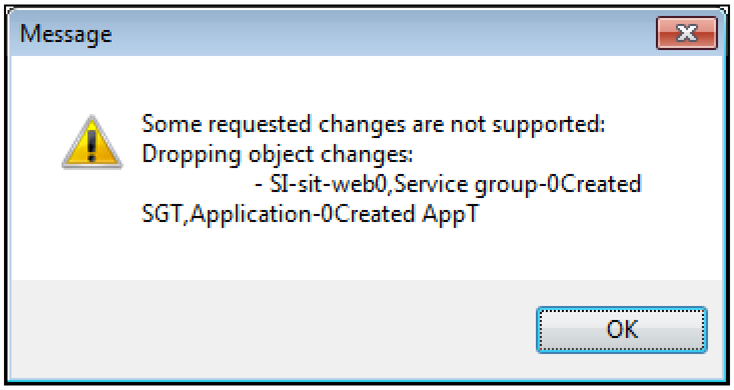
5. The first output is a detailed textual list of changes displayed on the console. The sample given Figure 14 has been edited not to show the details of the SG that has been added. The first line indicates which Application has been modified. The second line indicates the addition of the SG. This would normally contain a significant number of properties and child objects, but these have been removed for clarity. Line 5 indicates that SI-sit-web0 has been modified and line 6 indicates the modification: that its SG has been changed.

```
1   (Change SaAmfApplication: Application-0Created AppT
2          (Add SaAmfSG Service group-1Created SGT
3          [...]
4          )
5          (Change SaAmfSI: SI-sit-web0
6          saAmfSIProtectedbySG  :=  "Service  group-1Created  SGT,Application-
    0Created AppT";
7          )
8   )
```

Figure 14: Difference Generation Tool Textual Output (abriged)

6. Tuples are then created for the Upgrade Campaign Generator. However not all changes are supported. A dialog is shown that lists all unsupported changes that were requested.

This is shown in Figure 15. In this case, it indicates that SI-sit-web0's SG migration is unsupported.



```
Some requested changes are not supported:
Dropping object changes:
                - SI-sit-web0,Service group-0Created
SGT,Application-0Created AppT
```

Figure 15: Upgrade Campaign Unhandled Modification Message Dialog Box

7. Control is then handed over to the Upgrade Campaign Generator for it to perform its task. Excluding debug and Validation related dialogs; it simply prompts the user to save the upgrade campaign that results from the tuples that were provided.

4.3 Application to PHASE

For our case study, we will be reusing exactly the same case study of the Portable Highly Available Sensors [SAFS10] as was used in Setareh Kohzadi's thesis [KS09]. Furthermore, the same four example cases will be used. The results in her thesis will constitute the baseline for us to be able to compare our generated upgrade campaigns. Our objective is to demonstrate that, if the source and target configurations are available, it is significantly easier to simply use the Delta Generator frontend to the Upgrade Campaign Generator. In fact in each of these cases, only configuration input needed is those two configurations.

4.3.1 PHASE Application

The PHASE application is a set of sensor data collectors (SDC), data processors (DP) and processed data send (DS) units. As the names imply, the sensor data from multiple external sensors is collected by the SDCs; then sent to the DPs for processing; and results are given to the DS units to be sent out to any client. This is illustrated Figure 16. While two types of DPs exist, one with history and the other without, we will not distinguish them as their differences are not relevant to Delta Generation.

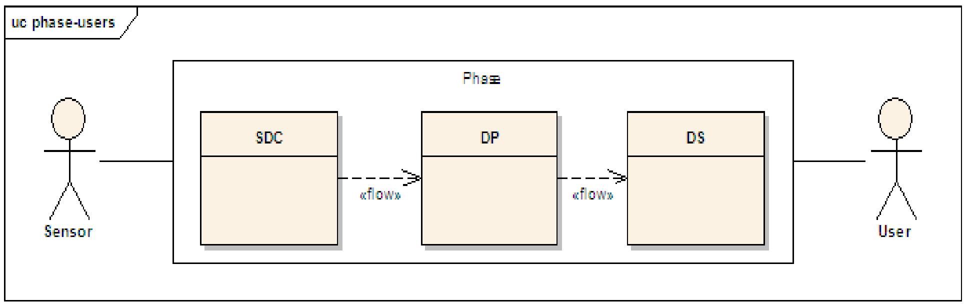


Figure 16: PHASE Ecosystem (taken from [SAFS10])

The specific configuration we will be using is illustrated both Figure 17 below and in more details in Figure 6 of section 2.2.3. It consists of one Application, containing one SG for each unit type, each with two SUs containing one component each. Each SU of each unit type is hosted on one of two Nodes. The SGs of SDCs and DPs both have 2 SIs to protect, each with one CSI. The DS SG only protects one DS SI, again with one component. All components start with version 1 of their software.

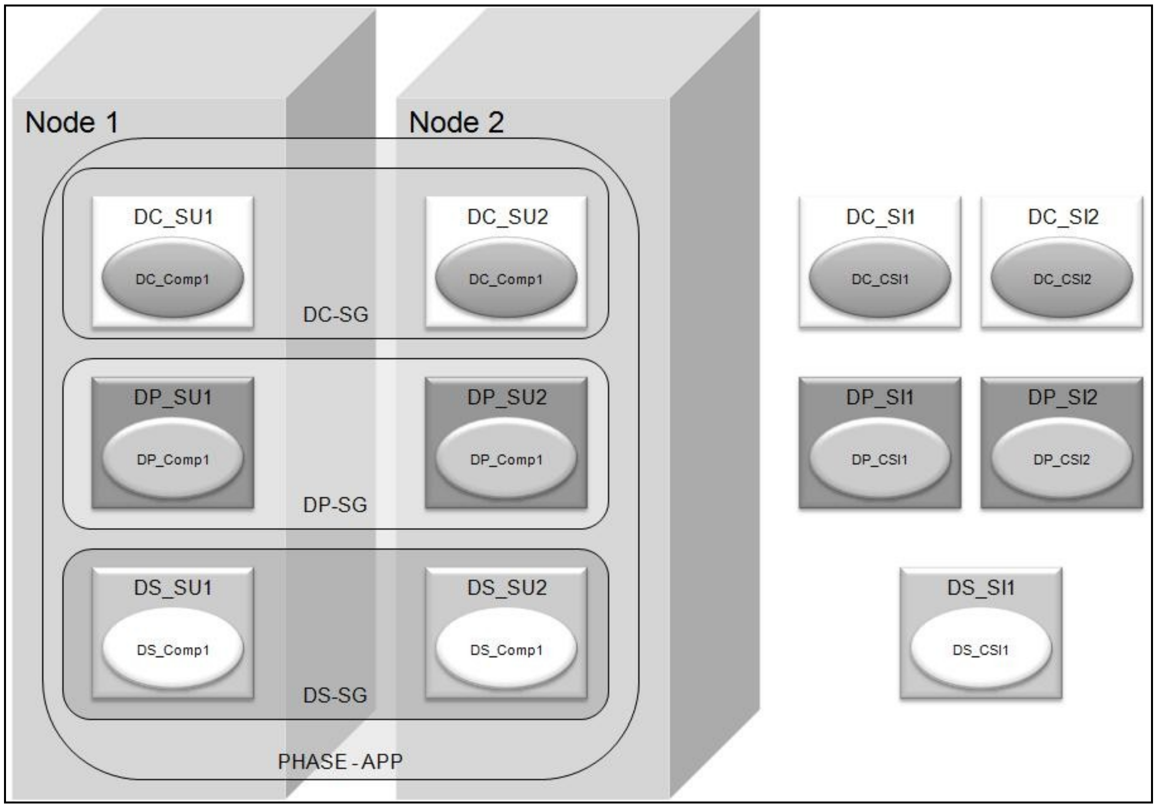
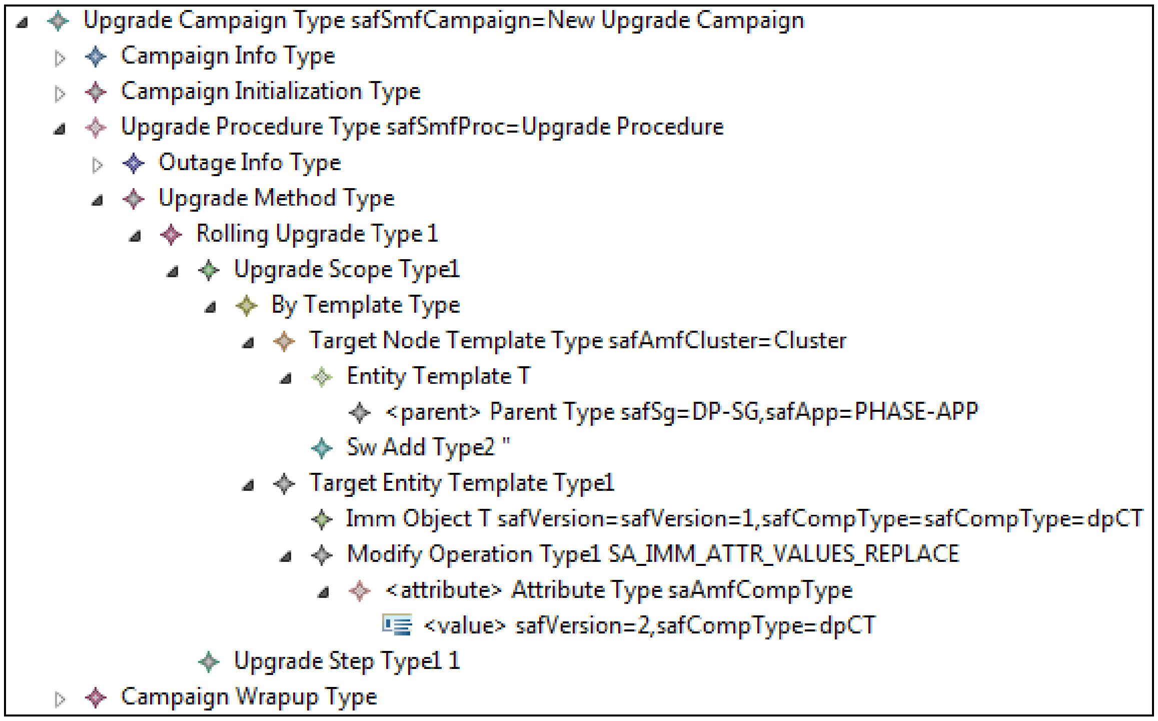


Figure 17: A Simple Configuration of PHASE (taken from [SAFS10])

4.3.2 Changing Type Scenario

In the Changing Type scenario, we wish to upgrade the DP components to version two of the software. To do this, we add the new Software Bundle and related Types to the configuration. We then change the CompType of each of the two affected components.

After feeding these configurations to the Delta Generator, the obtained upgrade campaign is shown in Figure 18. Just as in our baseline, a rolling upgrade is performed on the components. However, since the delta does not involve the modification of the parent SU’s type we do not have a rolling update of the SU’s Type.

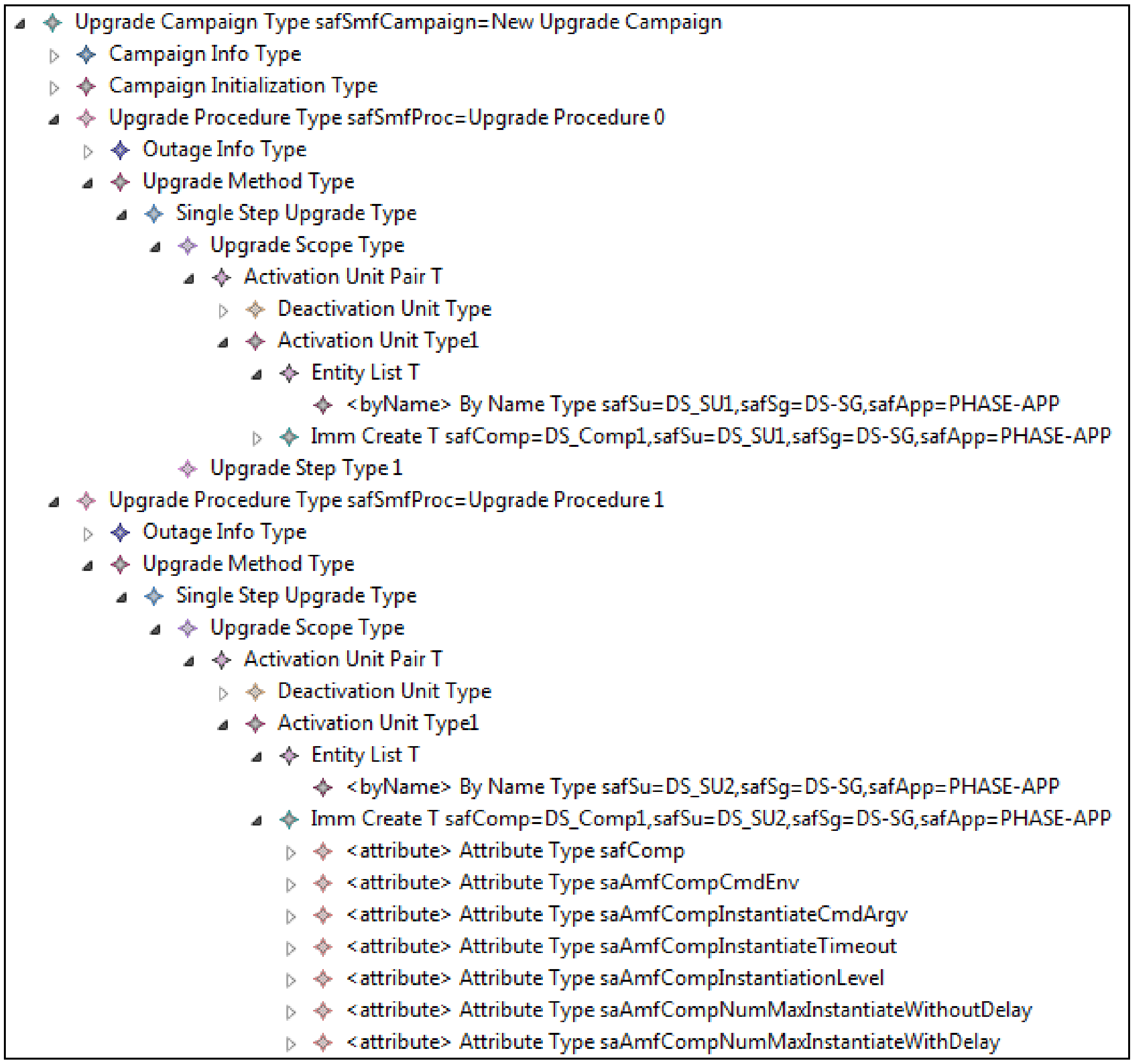


```
◢ Upgrade Campaign Type safSmfCampaign=New Upgrade Campaign
    ▷ Campaign Info Type
    ▷ Campaign Initialization Type
    ◢ Upgrade Procedure Type safSmfProc=Upgrade Procedure
        ▷ Outage Info Type
        ◢ Upgrade Method Type
            ◢ Rolling Upgrade Type 1
                ◢ Upgrade Scope Type1
                    ◢ By Template Type
                        ◢ Target Node Template Type safAmfCluster=Cluster
                            ◢ Entity Template T
                                  <parent> Parent Type safSg=DP-SG,safApp=PHASE-APP
                              Sw Add Type2 "
                        ◢ Target Entity Template Type1
                              Imm Object T safVersion=safVersion=1,safCompType=safCompType=dpCT
                            ◢ Modify Operation Type1 SA_IMM_ATTR_VALUES_REPLACE
                                ◢ <attribute> Attribute Type saAmfCompType
                                      <value> safVersion=2,safCompType=dpCT
                  Upgrade Step Type1 1
    ▷ Campaign Wrapup Type
```

Figure 18: Case Study, Upgrade Campaign Specification for Changing Type Scenario

4.3.3 Adding Entities Scenario

In Adding Entities scenario, our target configuration adds a second Component to each of our DS SUs. As these are of existing Types with Software already installed on each node, no other classes need to be added. The produced upgrade campaign is shown in Figure 19. Just as our baseline, this Add is split into two single step procedures. If compared line by line, the difference in between the two upgrade campaigns is simply due to an updated Upgrade Campaign Generator and display model.

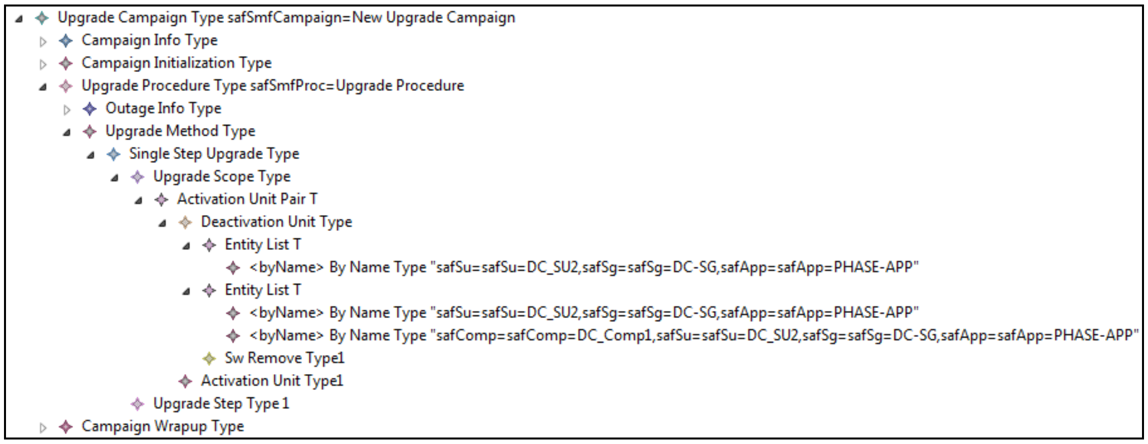


```
Upgrade Campaign Type safSmfCampaign=New Upgrade Campaign
  Campaign Info Type
  Campaign Initialization Type
  Upgrade Procedure Type safSmfProc=Upgrade Procedure 0
    Outage Info Type
    Upgrade Method Type
      Single Step Upgrade Type
        Upgrade Scope Type
          Activation Unit Pair T
            Deactivation Unit Type
            Activation Unit Type1
              Entity List T
                <byName> By Name Type safSu=DS_SU1,safSg=DS-SG,safApp=PHASE-APP
              Imm Create T safComp=DS_Comp1,safSu=DS_SU1,safSg=DS-SG,safApp=PHASE-APP
        Upgrade Step Type 1
  Upgrade Procedure Type safSmfProc=Upgrade Procedure 1
    Outage Info Type
    Upgrade Method Type
      Single Step Upgrade Type
        Upgrade Scope Type
          Activation Unit Pair T
            Deactivation Unit Type
            Activation Unit Type1
              Entity List T
                <byName> By Name Type safSu=DS_SU2,safSg=DS-SG,safApp=PHASE-APP
              Imm Create T safComp=DS_Comp1,safSu=DS_SU2,safSg=DS-SG,safApp=PHASE-APP
                <attribute> Attribute Type safComp
                <attribute> Attribute Type saAmfCompCmdEnv
                <attribute> Attribute Type saAmfCompInstantiateCmdArgv
                <attribute> Attribute Type saAmfCompInstantiateTimeout
                <attribute> Attribute Type saAmfCompInstantiationLevel
                <attribute> Attribute Type saAmfCompNumMaxInstantiateWithoutDelay
                <attribute> Attribute Type saAmfCompNumMaxInstantiateWithDelay
```

Figure 19: Case Study, Upgrade Campaign Specification for Component Addition Scenario

4.3.4 SU Removal Scenario

In the SU Removal scenario, our target configuration removes the second DC SU, located on Node 2. The produced upgrade campaign is shown in Figure 20. While the removal of the SU is still in a SingleStepUpgrade as compared to the baseline, the software removal step has not been split off to a separate RollingUpgrade phase. This is due to updated logic in the Upgrade Campaign generator and is intentional as the software removal is not service affecting.



```
Upgrade Campaign Type safSmfCampaign=New Upgrade Campaign
  Campaign Info Type
  Campaign Initialization Type
  Upgrade Procedure Type safSmfProc=Upgrade Procedure
    Outage Info Type
    Upgrade Method Type
      Single Step Upgrade Type
        Upgrade Scope Type
          Activation Unit Pair T
            Deactivation Unit Type
              Entity List T
                <byName> By Name Type "safSu=safSu=DC_SU2,safSg=safSg=DC-SG,safApp=safApp=PHASE-APP"
              Entity List T
                <byName> By Name Type "safSu=safSu=DC_SU2,safSg=safSg=DC-SG,safApp=safApp=PHASE-APP"
                <byName> By Name Type "safComp=safComp=DC_Comp1,safSu=safSu=DC_SU2,safSg=safSg=DC-SG,safApp=safApp=PHASE-APP"
              Sw Remove Type1
            Activation Unit Type1
        Upgrade Step Type 1
  Campaign Wrapup Type
```

Figure 20: Case Study, Upgrade Campaign Specification for SU Removal Scenario

4.3.5 Combined Scenario

The combined scenario performs all three operations at the same time. It results in the upgrade campaign shown in Figure 21. This upgrade campaign is essentially a merger of the three previous upgrade campaigns. This matches up very well with the baseline upgrade campaign. The only differences are the removal of the extra RollingUpgrade for the software remove of the SU removal scenario, just as in section 4.3.4; and the unchanged SU Types of section 4.3.2. This is as expected.

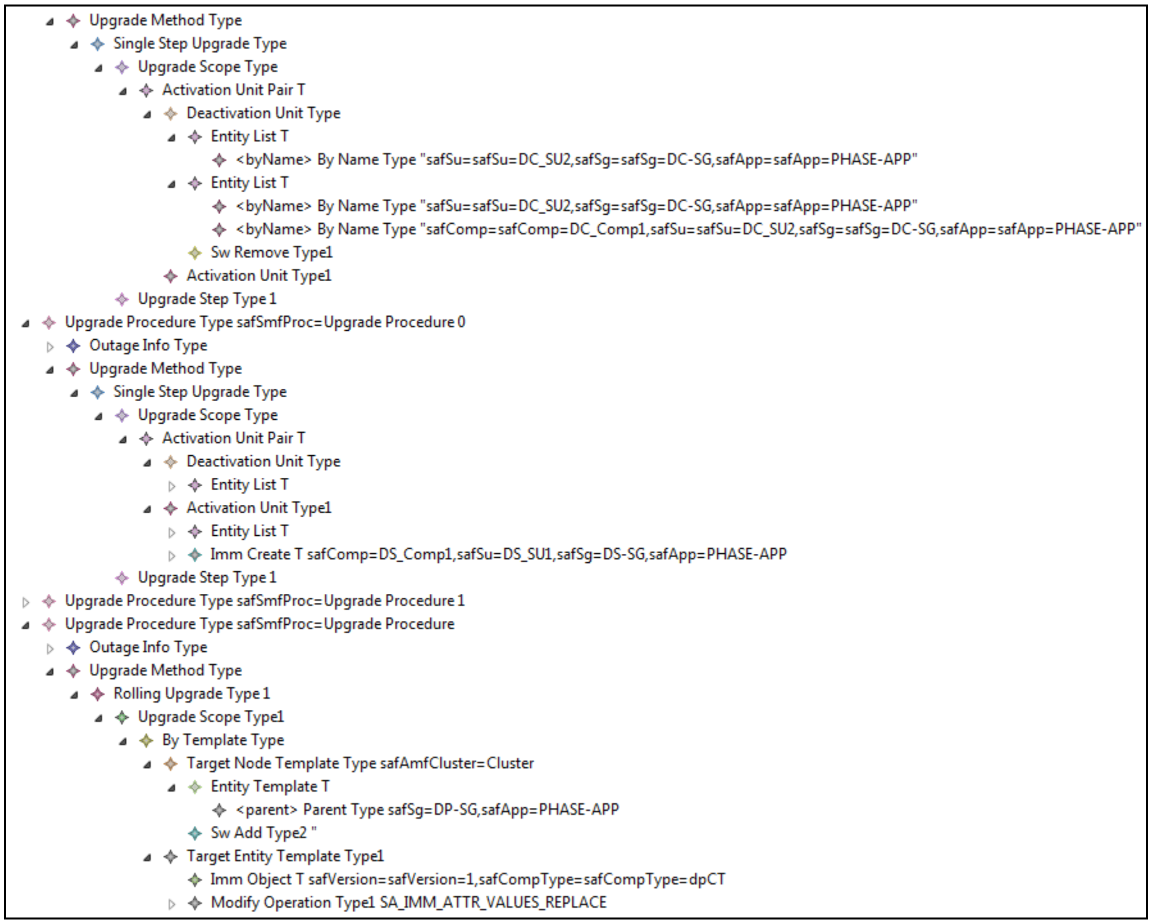

```
    Upgrade Method Type
      Single Step Upgrade Type
        Upgrade Scope Type
          Activation Unit Pair T
            Deactivation Unit Type
              Entity List T
                <byName> By Name Type "safSu=safSu=DC_SU2,safSg=safSg=DC-SG,safApp=safApp=PHASE-APP"
              Entity List T
                <byName> By Name Type "safSu=safSu=DC_SU2,safSg=safSg=DC-SG,safApp=safApp=PHASE-APP"
                <byName> By Name Type "safComp=safComp=DC_Comp1,safSu=safSu=DC_SU2,safSg=safSg=DC-SG,safApp=safApp=PHASE-APP"
              Sw Remove Type1
            Activation Unit Type1
        Upgrade Step Type 1
  Upgrade Procedure Type safSmfProc=Upgrade Procedure 0
    Outage Info Type
    Upgrade Method Type
      Single Step Upgrade Type
        Upgrade Scope Type
          Activation Unit Pair T
            Deactivation Unit Type
              Entity List T
            Activation Unit Type1
              Entity List T
              Imm Create T safComp=DS_Comp1,safSu=DS_SU1,safSg=DS-SG,safApp=PHASE-APP
        Upgrade Step Type 1
  Upgrade Procedure Type safSmfProc=Upgrade Procedure 1
  Upgrade Procedure Type safSmfProc=Upgrade Procedure
    Outage Info Type
    Upgrade Method Type
      Rolling Upgrade Type 1
        Upgrade Scope Type1
          By Template Type
            Target Node Template Type safAmfCluster=Cluster
              Entity Template T
                <parent> Parent Type safSg=DP-SG,safApp=PHASE-APP
              Sw Add Type2 "
            Target Entity Template Type1
              Imm Object T safVersion=safVersion=1,safCompType=safCompType=dpCT
              Modify Operation Type1 SA_IMM_ATTR_VALUES_REPLACE
```

Figure 21: Case Study, Upgrade Campaign Specification for All Three Scenarios Combined

4.4 Chapter Conclusion

In this chapter we have shown the implementation of our difference generation algorithms as a prototype tool. We have further recreated the upgrade campaigns from Setareh Kohzadi's thesis [KS09] while only requiring as input pairs of configuration files, completely bypassing its graphical interface. This significantly decreases the amount of required user interaction required to create and upgrade campaign.

Chapter 5

Conclusion

In this chapter we will restate our research contributions to the domain of AMF configuration difference generation and High Availability systems. We will then list some possible future research directions.

5.1 Contributions

In this thesis, we have shown how to create a list of differences between two configurations when the Distinguished Names of the objects are available. We have further shown that if Distinguished Names and Relative Distinguished Names are not available, that it is not possible to create a difference set with certitude of its accuracy.

To be able to perform difference generation under these conditions, we have proposed an approach with service availability as primary goal. Using a top-down methodology, we have formulated and explained assumptions to minimize the amount of downtime services will be obligated to incur during an upgrade to be able to go from a source configuration to the target configuration. Based on these assumptions we have created an algorithm to perform the difference generation with limited use of Distinguished Names.

Furthermore, we have created two special case optimisations to our algorithm. The first special case is to relax our assumption on SI migration. It allows prompting the user to be able to resolve a detected conflict. The second special case optimisation is to reduce the amount of output our algorithm generates with regards to Type replacement.

Finally, we have implemented our approach in a prototype tool on top of the Eclipse Project using the MAGIC AMF and StdAMF ECore models. Furthermore, we have adapted this prototype tool to be able to communicate with the Magic Upgrade Campaign Generator. Given a source and target configuration it thereby automatically outputs both derived difference set and an Upgrade Campaign file to apply those changes to a running system; as long as all required changes are supported by the Upgrade Campaign Generator.

5.2 Future Research

Two possibilities exist for extending the usefulness of the difference generator. Currently all the assumptions permit the algorithm to make firm decisions at each level of the algorithm. Using fuzzy searches may allow for a better match or, at the very least, less strict constraints. The disadvantage of this is that while that approach offers more flexibility it usually allows for unbounded error. Our approach allowed us to characterise and usually minimise the error, at the expense of this flexibility. The second possibility for increasing the types of allowed cases is to add more user interaction. Similarly to the way we prompt the user for information about SI migrations, we could ask for a user to review results after having performed an initial search.

On the Upgrade Campaign side of the automation chain, much work is needed. The current tool is limited. Most of the changes that the difference generator can detect are not supported. In many cases extending the tool's capabilities would be simple, such as the ability to add Nodes, SIs and CSIs.

Further analysis is also needed with regards to intermediate states required to maintain a high level of availability. For instance, while the difference generator will indicate that a component must be upgraded from A to C, it may be interesting for the campaign to transition the system through version B if that A and C cannot run simultaneously, while B is compatible with both.

Finally, the upgrade campaign generator's assumptions on locking for software installation and removal need to be removed and a full set of optimisations needs to be derived with finer granularity than Node level locks.

In the longer term, the difference generator and upgrade campaign generator will need to be bundled together or abstracted, to allow a configuration generator to be able to better judge the impact of its configurations during the update. For instance, while two configurations may be judged to have equivalent value, considering the impact of the required upgrade campaign may make one better than the other.

One ultimate goal of this automation is to enable automated reconfiguration of the cluster. Full and detailed impact analysis of the upgrade campaign is then critical since the system may end up having some reconfiguration happening for the entire active life.

Bibliography

BL00: Bergroth, L. and Hakonen, H. and Raita, T. *A survey of longest common subsequence algorithms*. Seventh International Symposium on String Processing and Information Retrieval, 2000. pages 39-48.

CP10: Colombo, P., Khendek, F., Lavazza, L. *Requirements analysis and modeling with Problem Frames and SysML: A case study*. Proceedings of the European Conference on Modeling Foundations and Applications (ECMFA), LNCS #6138, Springer. June 15-18, 2010. page 74-89.

EP10: *Eclipse Platform*. Eclipse Foundation. 21 Nov 2010. <http://www.eclipse.org/platform/>

ER09: *Helios/Simultaneous Release Plan – Eclipsepedia*. Eclipse Foundation. 3 June 2010. <http://wiki.eclipse.org/Helios_Simultaneous_Release>

ES10: *SWT: The Standard Widget Toolkit*. Eclipse Foundation. 21 Nov 2010. <http://www.eclipse.org/swt/>

GA09: Gherbi, A., Kanso, A., Khendek, F., Toeroe, M., Hamou-Lhadj, A. *A suite of tools for the validation and generation of AMF configurations*. Proceedings of IEEE/ACM Automated Software Engineering, Auckland, New Zealand. Nov. 2009.

GA09B: Gherbi, A., Salehi, P., Khendek, F., Hamou-Lhadj, A. *Capturing and Formalizing SAF Availability Management Framework Configuration Requirements*. Proceedings of Domain Engineering Workshop held in conjunction with CAiSE'2009, Amsterdam, Netherlands. June 2009.

HAF01: *Providing Open Architecture High Availability Solutions Revision 1.0*. High Availability Forum. Feb. 2001 <http://www.lynuxworks.com/products/whitepapers/ha-solutions.pdf>

KA10: Kanso, A., Khendek, F., Toeroe, M., Hamou-Lhadj, A. *Ranking Service Units for Providing and Protecting Highly Available Services with Load Balancing*. Proceedings of NOTERE'2010, Tozeur, Tunisia. May 31-June 02, 2010.

KA09: Kanso, A., Toeroe, M., Hamou-Lhadj, A., Khendek, F. *Generating AMF Configurations from Software Vendor Constraints and User Requirements*. Proceedings of ARES'2009, Fukuoka, Japan. March 2009.

KA08: Kanso, A., Toeroe, M., Khendek, F., Hamou-Lhadj, A. *Automatic Generation of AMF Compliant Configurations*. Proceedings of the International Symposium on Service Availability (ISAS), Tokyo, Japan. May 19-21, 2008

KA08B: Kanso, Ali. *Automatic generation of AMF compliant configurations*. Thesis (M.A. Sc.)--Dept. of Electrical and Computer Engineering, Concordia University. August 2008.

KS09: Kohzadi,Setareh. *Automatic generation of upgrade campaign specifications*. Thesis (M.A. Sc.)--Dept. of Electrical and Computer Engineering, Concordia University. October 2009.

PS10: Salehi, P., Colombo, P., Hamou-Lhadj, A., Khendek, F. *A Model Driven Approach for AMF Configuration Generation*. Proceedings of the 6th Workshop on System Analysis and Modelling (SAM), LNCS, Oslo, Norway, October 2010.

PS10B: Salehi, P., Hamou-Lhadj, A., Colombo, P., Toeroe, M., Khendek, F. *A UML-Based Domain Specific Modeling Language for the Availability Management Framework*. Proceedings of the 12th IEEE International High Assurance Systems Engineering Symposium (HASE), San Jose, CA. Nov. 1-4, 2010.

PS09: Salehi, P., Khendek, F., Toeroe, M., Hamou-Lhadj, A., Gherbi, *A. Checking for Service Instance Protection for AMF Configurations*. Proceedings of IEEE SSIRI'2009, Shanghai, China. July 2009.

SAF10: *Home.* Service Availability Forum. 21 Nov. 2010 <http://saforum.org/>

SAFA08: *Availability Management Framework B.04.01*. Service Availability Forum. 16 Oct. 2008. <http://www.saforum.org/HOA/assn16627/images/SAI-AIS-AMF-B.04.01.pdf>

SAFD08: *Dynamic Configuration Changes*. Service Availability Forum. December 2008.

SAFC08: *Cluster Membership Service B.04.01*. Service Availability Forum. 16 Oct. 2008. <http://www.saforum.org/HOA/assn16627/images/SAI-AIS-CLM-B.04.01.pdf>

SAFH09: *Hardware Platform Interface B.03.02*. Service Availability Forum. 4 Aug. 2009. <http://www.saforum.org/HOA/assn16627/images/SAI-HPI-B.03.02.pdf>

SAFI08: *IMM Initial XML Scheme A.01.01*. Service Availability Forum. 21 Oct. 2008. <http://www.saforum.org/hoa/assn16627/images/SAI-AIS-IMM-XSD-A.01.02.zip>

SAFO08: *Overview Document B.05.01*. Service Availability Forum. 16 Oct. 2008. <http://www.saforum.org/HOA/assn16627/documents/sai-overview-b.05.01.pdf>

SAFP08: *Platform Management Service A.01.02*. Service Availability Forum. 16 Oct. 2008. <http://www.saforum.org/HOA/assn16627/images/SAI-AIS-PLM-A.01.02.pdf>

SAFS08: *Software Management Framework A.01.02*. Service Availability Forum. 31 Jan. 2008. <http://www.saforum.org/hoa/assn16627/images/sai-ais-smf-a.01.02.pdf>

SAFS10: *The Software Management Framework: Basic Concepts Explained*, Service Availability Forum. 24 Aug. 2010. <http://saforum.org/HOA/assn16627/images/SMF%20White%20Paper.zip>

TS02: Tarkoma, Sasu. *Introduction to Service Availability Forum*. Seminar on High Availability and Timeliness in Linux. University of Helsinki, Department of Computer Science. 7 Oct. 2002. <http://www.cs.helsinki.fi/u/niklande/opetus/SemK03/Tarkoma.pdf>

W3C10: *Extensible Markup Language (XML)*. W3C. 6 Sept. 2010. 21 Nov. 2010 <http://www.w3.org/XML/>

XZ05: Xing, Z. and Stroulia, E. *UMLDiff: an algorithm for object-oriented design differencing*. ACM. Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering. 2005. pages 54-65.