AUTOMATIC GENERATION OF UPGRADE CAMPAIGN SPECIFICATIONS

SETAREH KOHZADI

A THESIS

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE AND SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF APPLIED SCIENCE (SOFTWARE ENGINEERING) AT

CONCORDIA UNIVERSITY

MONTREAL, QUEBEC, CANADA

OCTOBER 2009

Your file Votre référence
ISBN: 978-0-494-67318-8
Our file Notre référence
ISBN: 978-0-494-67318-8

Abstract

Automatic Generation of Upgrade Campaign Specifications

Setareh Kohzadi

The increasing reliance on computing systems has greatly impacted the customers' expectations from such systems; for which the need for reliable and highly available services has become an essential requirement. This has led service providers to seek for new ways to supply robust services in order to sustain their advantage in today's highly competitive markets.

A highly available system is defined as a system that is up and running 99.999% of the time. To gain such availability, a solution that has been widely employed is the usage of redundant components. However, solutions used in the past are proprietary and as a result applications had no portability from one platform to another.

The Service Availability Forum (SAF), a consortium of several computing and telecommunication companies, promotes standardized solutions for building highly available systems in which Commercial-Off-The-Shelf (COTS) building blocks can be used. The SAF middleware has many services each of which has a specialized role so that the whole middleware can manage the redundant components within the system to fulfill the service availability.

Like any other system, a SAF system may go through several upgrade and downgrade processes during its lifetime. Though due to the high availability requirement, a SAF system ought to be upgraded while ensuring minimum service interruption. Among the

SAF middleware services, the Software Management Framework (SMF) is responsible for this live upgrade.

In order for the SMF to perform an upgrade the road map of this migration, which is known as the Upgrade Campaign Specification, should be provided. However, due to the number of entities involved in an upgrade campaign and the complexity of the relationships among these entities, manual calculation of various steps of an upgrade campaign specification is time consuming and error prone.

In this thesis, we have devised an approach for automatic generation of upgrade campaign specifications to upgrade redundant entities of SAF systems. We have categorized possible upgrade variations into three main scenarios which consist of manipulating current entities of the system, removing or adding new ones. For each scenario we have recognized different criteria that impact the service availability. For each criterion, according to the different upgrade methods introduced by SMF, we have devised solutions to minimize the service availability interruption during the course of an upgrade.

Finally, we have created a prototype tool that supports the generation of upgrade campaign specification algorithms for each scenario. We have applied our approach to a case study to demonstrate its applicability.

Acknowledgements

I would like to express my deep gratefulness to:

- My mother, my father and my brother for their everlasting love, support and understanding,
- My aunt and cousin for their big, kind hearts throughout my stay as a member of their family,
- My supervisors, Dr. Ferhat Khendek and Dr. Abdelwahab Hamou-Lhadj for giving me the chance to benefit from their extensive knowledge and experience a new way of collaboration with my professors,
- Dr. Maria Toeroe (Ericsson Canada Inc.) whom I have learned a lot not only in the domain of high availability but also in the domain of life,
- My colleagues at MAGIC for all of their support and the good memories we have had,
- All of my friends whom their companionship have enlightened my heart and,
- Concordia University and Ericsson Canada for providing me with the opportunity to achieve this work.

Table of Contents

List of Figures

List of Tables

List of Flowcharts

Abbreviations

SAF – Service Availability Forum

AMF – Availability Management Framework

SMF – Software Management Framework

IMM – Information Model Management

ETF – Entity Types File

RDN – Relative Distinguished Name

CSI - Component Service Instance

CST – Component Service Type

CT – Component Type

SI – Service Instance

SU – Service Unit

SUT – Service Unit Type

SG – Service Group

SGT – Service Group Type

Chapter 1 - Introduction

In this chapter, we explain briefly the context of our research project. We introduce high availability, service continuity and the Service Availability Forum (SA Forum) [1]. We present the motivations behind this thesis and introduce its contributions.

1.1 High Availability

Availability is defined as the probability of service provision upon request, assuming that the time required for satisfying each service request is short [2]. The availability of a system is measured in terms of reliability of the system components and the required time to repair the system in case of failure:

- MTBF: Mean Time Between Failure: the failure rate of the system and,
- MTTR: Mean Time To Repair: the time to restore service,

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

If the availability of a system goes beyond 99.999% of the time (known as five nines) that system is considered a *highly available system.* That level of availability admits only 5.26 minutes downtime for the whole year.

As the availability formula states, high availability will be achieved when MTBF for the components of a system is considerably high and their MTTR is low. However, in a real system with a large number of software and hardware components it is hard to achieve high availability by increasing the MTBF [2]. Having components with high MTBF and low MTTR, almost null, is unrealistic and thus other solutions were sought out to increase the availability. Using redundant components that can substitute each other in case of failure is, so far, the best practical solution [1]. When a component fails the service provision will be carried on by the one backing it up, the MTTR of the system will tend to zero, thus increasing the availability.

1.2 Service Continuity

Service continuity is defined as maintaining the end-user sessions in spite of system components failure and during their recovery, maintenance and system management actions [2]. To provide service continuity a system should have redundant components that can preserve the state of the application session for each user [2]. Service continuity is achieved through cooperation between the components of the application software [2]. This cooperation requires communication and synchronization mechanism.

1.3 Service Availability Forum (SAF)

Traditionally, highly available systems were based on proprietary solutions and middleware. As a result, applications that were exclusively built considering the features of a specific platform were also proprietary and had no portability from one platform to

another. A single vendor supplied the hardware and software building blocks. Thus, the system enhancement and maintenance will depend on this vendor over its entire life.

The Service Availability Forum (SAF) [1] is a consortium of industry- leading communications and computing companies working together to develop and publish high availability and management software interface specifications [2].

As illustrated in Figure 1.1, the Service Availability Solution offered by SAF enables high availability together with the service continuity.

Figure 1.1. The SA Forum Service Availability Solution (taken from [2]).

The SAF specifications have been developed in two main areas (see Figure 1.2):

- The Application Interface Specification [3] and,
- The Hardware Platform Interface [4].

Both Application Interface Specification (AIS) and Hardware Platform Interface (HPI) are divided into smaller areas with specialized services that are used together to manage the redundant components of the applications and the underlying hardware. Details on AIS parts are provided in Section 2.1.

The SAF standard interfaces enable the use of Commercial-Off-The-Shelf (COTS) building blocks for the entire system; which results in enhanced portability and flexibility of the software and hardware components. They also reduced the complexity of

application development since the developer needs to only focus on the application logics itself.

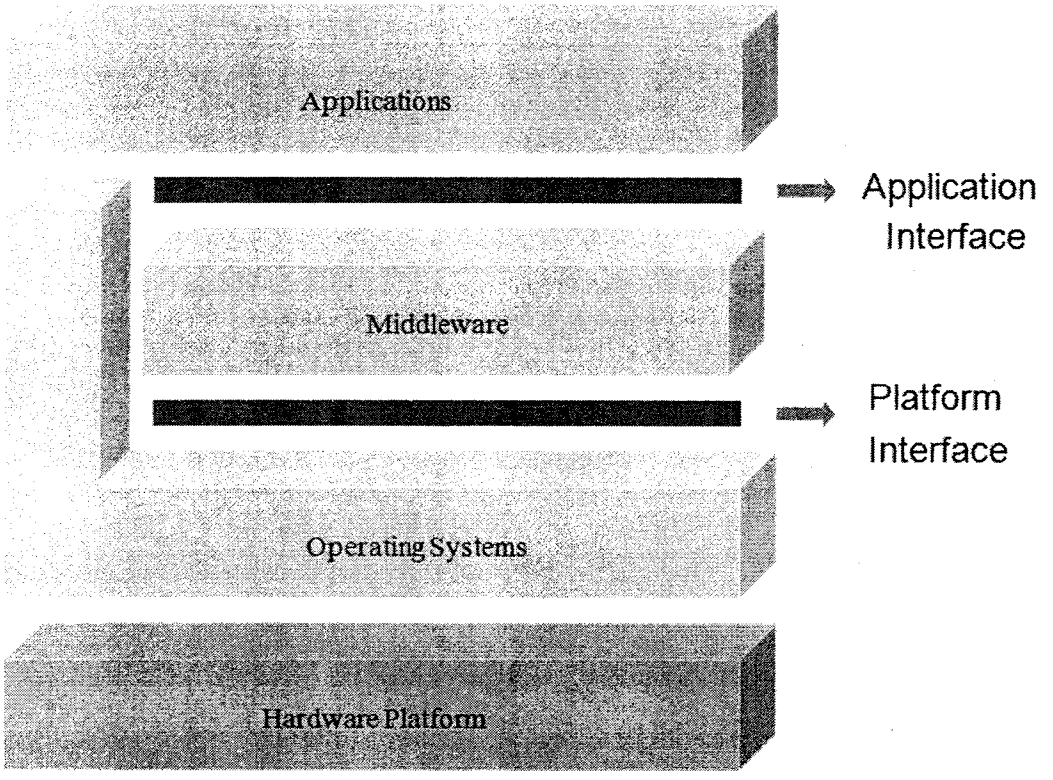


Figure 1.2. The Service Availability Interfaces (taken from [5]).

1.4 Thesis Motivation and Contributions

As mentioned earlier, AIS consists of various parts. The Availability Management Framework (AMF) [6] is the SAF middleware service that handles the availability of the applications. In the AMF compliant configurations of such applications, software components are abstracted into AMF logical entities and become instances of the AMF model. More details on AMF building blocks can be found in Section 2.3.

Software Management Framework (SMF) [7] is another SAF middleware services that is responsible of system upgrade or as its specification states, orchestrating the migration of

an AMF application from its current deployment configuration to another one. Like AMF, SMF defines a set of logical entities. SMF requires the specification of the step by step operations to take on the system entities to fulfill an upgrade. This specification is called the *Upgrade Campaign Specification* in the SMF terminology and is provided to SMF as an XML (Extensible Markup Language) [8] file.

In an AMF system with a large number of interdependent building blocks, manual generation of such an upgrade campaign specification is a time consuming, error prone and sometimes impossible task to perform. In addition to that there may exist different ways to upgrade a particular set of entities. Generation of all possible upgrade campaign specifications will give the opportunity to compare and analyse them according to different criteria such as the total duration of the upgrade, the impact of the upgrade on the availability of services, the ability to recover the system in cases where the upgrade is not successful, and so on.

Most of the decisions that affect the service availability at any point during the software upgrade period must be built into the upgrade campaign specification so that the whole upgrade process would cause the minimum possible service interruption. For that, we handle the system upgrade through three main scenarios and their combinations. These scenarios encompass the cases of manipulating or removing the existing entities of the system and adding new entities to it (see Section 3.2). For each scenario we have identified some criteria that influence the service availability during the system upgrade. For the identified criteria we have proposed solutions to minimize the service interruption by minimizing the set of affected entities at each point and building them into our upgrade campaign generation algorithms. These solutions are based on the different

upgrade methods introduced in the SMF specifications (see Section 2.4.1.2.9). When an AMF entity is given to be upgraded according to its upgrade scenario the relevant criteria is identified and solutions to minimize its upgrade impact are built into the upgrade campaign specification to upgrade this entity.

As a part of automatic upgrade campaign specification generation process user's input data (see Section 3.3.2) are checked to assure their consistency and completeness according to some criteria. Based on the provided input data also the ultimate status of the system will be assessed according to the AMF standard specification. By performing such operations we make sure that we will end up in a valid and coherent configuration of the system.

Finally we have developed an Eclipse plug-in for a prototype tool that implements all the proposed automatic upgrade campaign specification generation algorithms. Through different pages of a graphical user interface input data (see Section 3.3.2) is collected from the user. The provided input data is then analysed and the upgrade campaign specification XML file is generated.

1.5 Thesis Organization

The rest of this thesis is organized into four chapters. In Chapter 2, we provide the necessary background knowledge on AIS services. We particularly elaborate on AMF and SMF. We then give a brief review of the related state of the art. In Chapter 3, we present our upgrade campaign generation approach, the issues and challenges we faced and our solutions. The prototype tool and its architecture are explained in Chapter 4 along

with a case study. We wrap up this thesis in Chapter 5 and discuss possible future extensions for this research.

Chapter 2 - Background

In this chapter, we introduce the SAF middleware specifications. Then, we describe the Availability Management Framework (AMF) [6], its entities and entity types and the role it plays in ensuring availability of the services provided by the applications under its control. Next, we describe the Software Management Framework [7] and its role in the live upgrade of AMF applications. Finally, we discuss related research work.

2.1 SAF Middleware

Unlike traditional telecommunication systems that were built from proprietary hardware and software components, SAF middleware makes it possible to build a system from products of multiple vendors. It has two main parts to manage its commercial off-the shelf building blocks both at hardware and software levels: Application Interface Specification (AIS) and Hardware Platform Interface (HPI). AIS defines the services that handle high availability of the application's components, whereas the objective of HPI is to provide standard means to control and monitor hardware components.

As illustrated in Figure 2.1, AIS middleware defines several services, which are described in the following subsections:

The Availability Management Framework (AMF): It is the AIS service that ensures the availability of cluster applications through redundancy mechanisms. Further explanation of the AMF and its logical building blocks are discussed in Section 2.3.

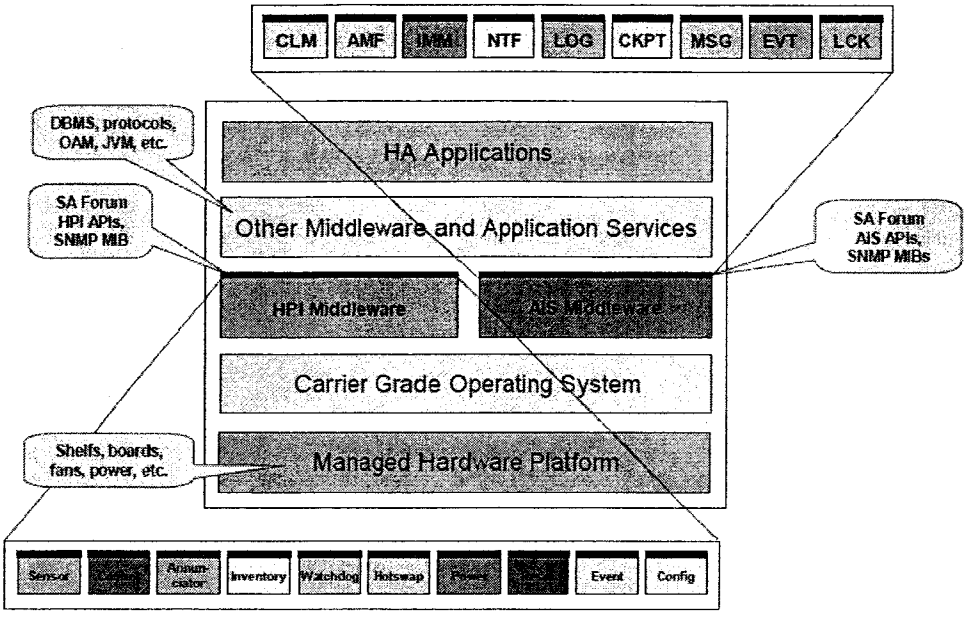


Figure 2.1. Architecture of the SAF middleware (taken from [2]).

The Software Management Framework (SMF): This service coordinates the live upgrade of an application that runs under the control of AMF from its current deployment configuration to another one. SMF structure and its mechanisms for maintaining the availability of applications during the upgrade process are discussed in more detail in Section 2.4.

The Information Model Management Service (IMM): This service manages the information model of a system. It defines specific APIs through which system administrators, applications and other AIS services can access and manage objects of the information model. We will elaborate more on the IMM in Section 2.2.

The Cluster Membership Service (CLM): This service provides users with information about the member nodes of the CLM cluster. It also keeps track of the nodes in the cluster.

The Checkpoint Service (CKPT): This service enables the processes to record their checkpoint data. For example, after a process failure this data will be used to retrieve the last state of the process to be able to resume its execution.

The Event Service (EVT): This service allows for asynchronous communication of one or more publishers with one or many subscribers. When a publisher publishes an event through an event channel, all the subscribers of that channel can receive the event.

The Message Service (MSG): This service provides a guaranteed communication mechanism that enables message exchange among processes that are on the same nodes or across multiple nodes.

The Lock Service (LCK): This service defines mechanisms for the processes to coordinate their access to shared resources.

The Notification Service (NTF): This service is based on the publish/subscribe pattern. Whenever an incident happens or the state of an entity changes this service delivers the notifications to the relevant subscribers.

The Log Service (LOG): This service enables the system and the applications to generate log files that, for example, contain alarms and notification information. System administrators can use these logs to trace the system. Various types of logs are defined in AIS such as alarms, notifications, and etc.

2.2 Information Model Management (IMM)

All the logical entities of an AIS compliant system (e.g., components of the AMF (see Section 2.3.1.1) or software entities of the SMF (see Section 2.4.1.1.2) are represented as objects of the AIS Information Model. The AIS Information Model Management service manages this information model and provides the means for other AIS services, administrators, and applications to access and modify its objects.

Information model objects are either configuration or runtime objects. A configuration object that describes the required system configuration (e.g., the component) contains configuration attributes and optionally runtime attributes. But a runtime object that describes the system's current state (e.g., the checkpoint object) only contains runtime attributes.

Object managers are system management applications for accessing and modifying information model objects. *Object implementers* implement these changes. Configuration objects are managed by object managers, while runtime objects are handled by object implementers. IMM service provides APIs to communicate with object managers and object implementers.

2.3 Availability Management Framework (AMF)

From the availability perspective, AMF is perhaps the most important service of AIS services since it is the service that ensures high availability of applications. To manage the redundant resources under its control, AMF uses an abstract object model, which consists of logical entities. This object model is known as the AMF configuration and is stored in the IMM repository.

At runtime, AMF accesses the configuration of the application it controls from IMM and assigns workloads to its entities. It constantly monitors their availability through health check and error reporting functions. In case of failure, AMF preserves the availability of the application services by turning over the workload of the faulty component to its standby one; meanwhile it isolates the faulty component, tries to repair it and put it back to service.

2.3.1 AMF Entities and Entity Types

To form the redundant resources, through an AMF configuration logical entities are grouped in different granularity levels. This way the AMF performs its management operations on and assigns the workload to them. The composition and relation of these logical entities are shown in Figure 2.2 and described in the subsequent section.

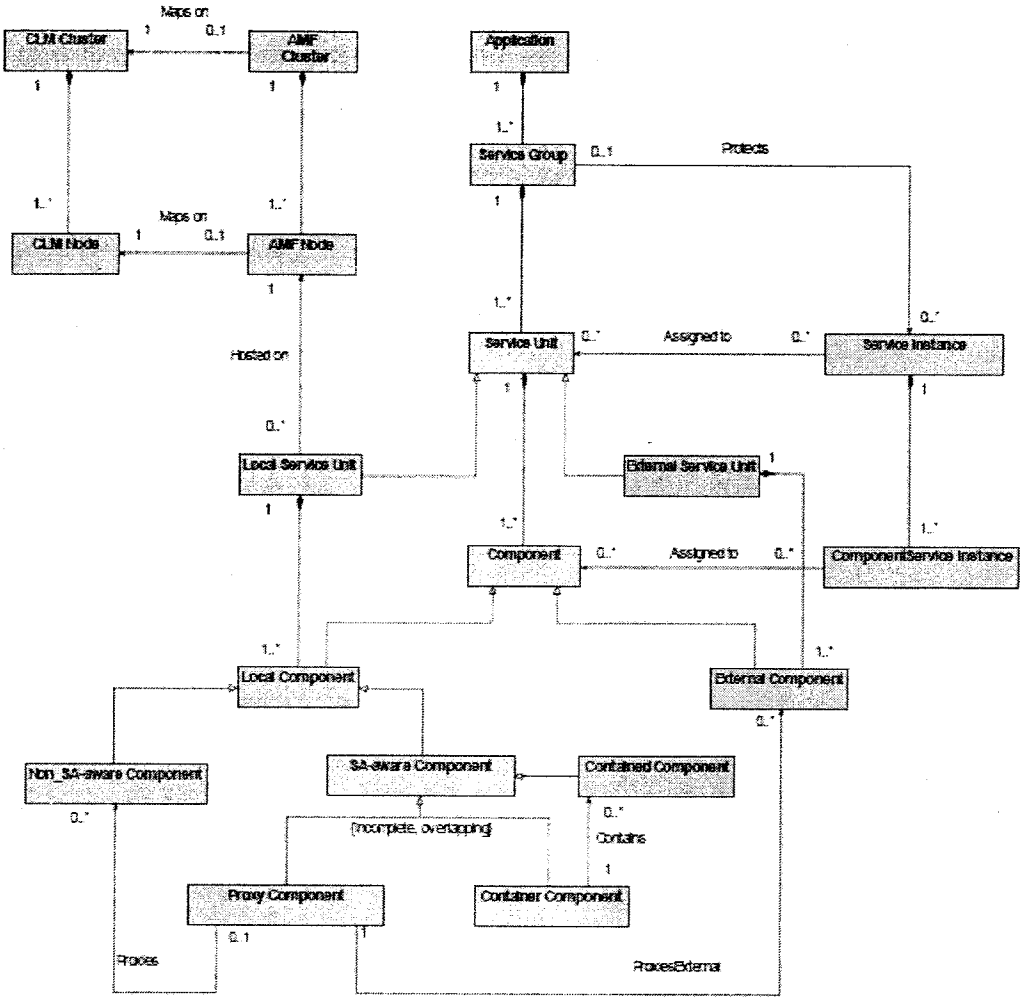


Figure 2.2. Logical Entities of the AMF (Taken from [6]).

2.3.1.1 Component

A *Component* represents a hardware or software resource that can provide a service. It is the smallest AMF logical entity on which AMF performs error detection and isolation, recovery and repair [6].

Each component has a name relative to its parent service unit (see Section 2.3.1.5). This naming is referred to as the component *RDN* (Relative Distinguished Name) by the AMF specification.

2.3.1.2 Component Service Instance (CSI)

The *Component Service Instance* represents the workload that AMF assigns to a component. AMF assigns High-Availability (HA) states of active and standby to components for handling their component service instances depending on whether the component is active (it is providing a service) or standby (used as a backup).

2.3.1.3 Component Type (CT)

Each component is typed and its type represents the particular version of the hardware or software used to build that component. It also specifies the component service types a component can support.

2.3.1.4 Component Service Type (CST)

A *Component Service Type* is the type of services a component provides. It is actually a generalization of similar component service instances that are equivalent from AMF perspective and hence handled in the same manner.

2.3.1.5 Service Unit (SU)

To provide a higher level service a set of components is aggregated into a *Service Unit*. While a component belongs to only one service unit, a service unit can have many components. A service unit is the unit of redundancy from AMF point of view.

Each service unit has a unique name in its containing service group. AMF specification referred to this name as the service unit RDN.

2.3.1.6 Service Instance (SI)

A *Service Instance* is the workload that AMF assigns to a service unit and it is an aggregation of the component service instances that are assigned to the components of that service unit. While a service instance can contain multiple component service instances, each component service instance belongs to only one service instance.

2.3.1.7 Service Unit Type (SUT)

Each service unit is typed and its type specifies the component types of the components that belong to the service unit of this type. The service unit type also specifies the maximum number of components of each particular type this service unit type can contain.

2.3.1.8 Service Type

A *Service Type* is the type of services a service unit can provide. It also refers to the component service types that are provided by the components of this service unit. For each component service type, the service type constrains the number of component service instances to handle.

2.3.1.9 Service Group (SG)

A set of service units is grouped into a *Service Group* to protect a particular set of services instances assigned to these service units. Any service unit of the service group must be able to serve any service instance that is assigned to that service group. While a

service group can have multiple service units, each service unit belongs to only one service group.

The service group also defines the level of protection applied to the service instances. This is achieved through five different redundancy models defined in AMF specifications [6]:

- **2N Redundancy Model:** A service group with 2N redundancy model has at most one service unit assigned active HA state for all of its service instances and at most one service unit as standby for the same service instances.

  In Figure 2.3 a service group with 2N redundancy model is shown. It has two service units; one with active and the other one with standby assignments for all of the service instances.

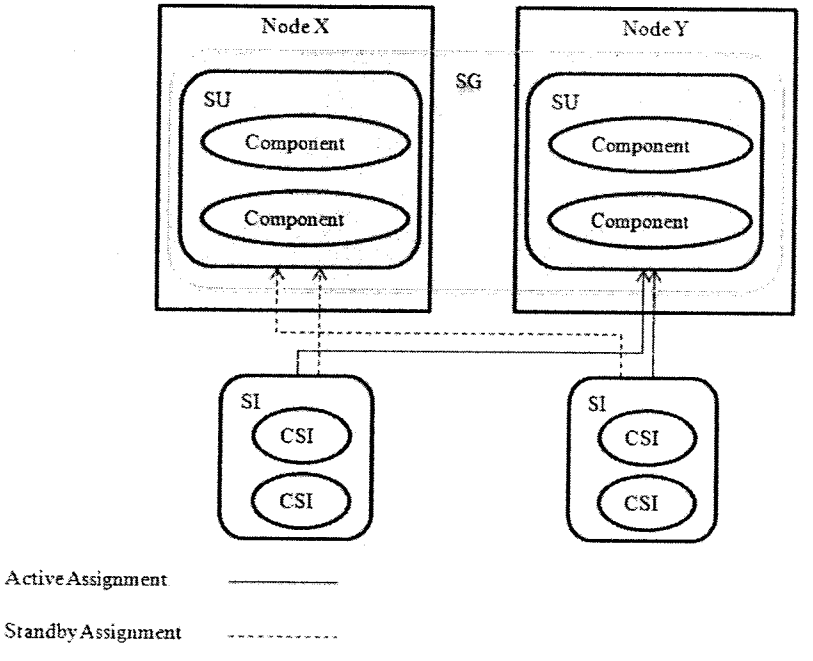


Figure 2.3. An example of 2N Redundancy Model.

- **N+M Redundancy Model:** A service group with this redundancy model has N active service units for the entire set of service instances assigned to the SG, and M service units assigned as standby for the service instances (Figure 2.4). For each service instance there is at most one active service unit and at most one standby service unit.

  Figure 2.4 shows a service group with N+M redundancy model. It has three service units; two of them are active for all the service instances assigned to the SG. The remaining service unit is standby for all of its assigned service instances.

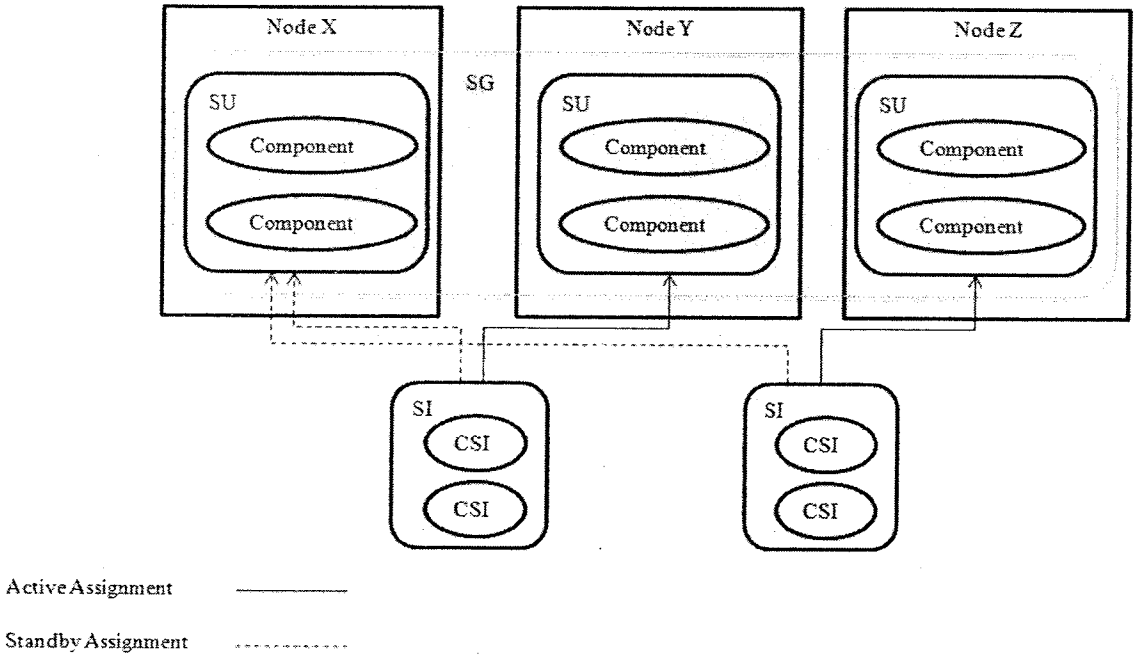


Figure 2.4. An example of N+M Redundancy Model.

- **N-Way Redundancy Model:** A service group with N-Way redundancy model contains N service units. Each service unit can have a combination of active and standby assignments (Figure 2.5). But each service instance can be assigned active

to only one service unit and standby to several service units. Figure 2.5 illustrates a service group with N-Way redundancy model. It has three service units that have both active and standby assignments. On the other hand, each service instance has exactly one active assignment and two standby ones.

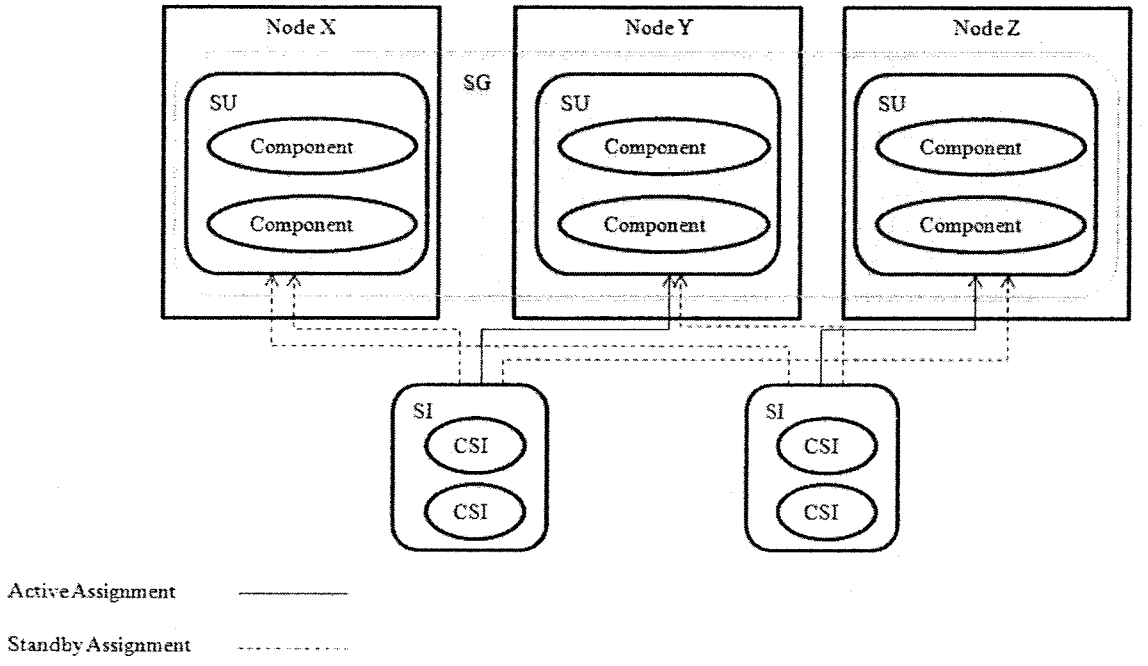


Figure 2.5. An example of N-Way Redundancy Model.

- **N-Way Active Redundancy Model:** A service group with N-Way Active redundancy model has N service units, which are assigned active HA state only (Figure 2.6). It has no service unit assigned the standby HA state. Furthermore, each of the service instances protected by this service group can be assigned to more than one service unit. Figure 2.6 shows a service group with N-Way Active redundancy. It has three service units that all of them have active assignment for the service instances assigned to them. On the other hand, one of the service instances has three active assignments while the other has two.

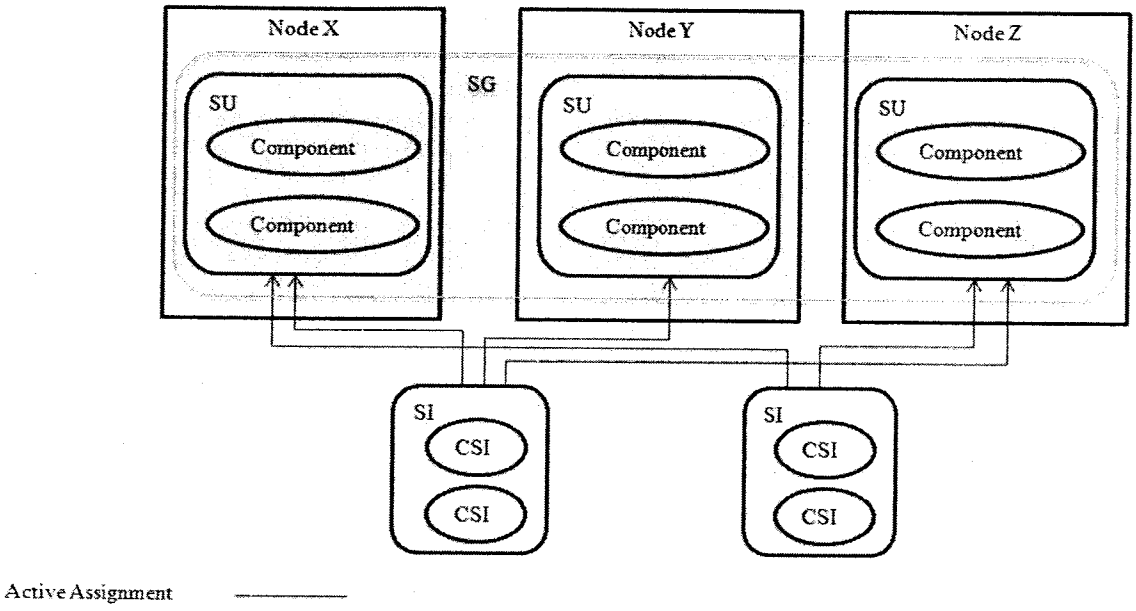


Figure 2.6. An example of N-Way Active Redundancy Model.

- **"No-Redundancy" Redundancy Model:** All the service units of a service group with the "No-Redundancy" redundancy model are assigned active HA state (Figure 2.7). The difference with the N-Way redundancy model is that in this case each service instance is assigned to at most one service unit and each service unit can protect at most one service instance. Figure 2.7 shows a service group with "No-Redundancy". As it is shown no service instance has standby assignment. In addition to that, each service instance is assigned to exactly one service unit and vice versa.

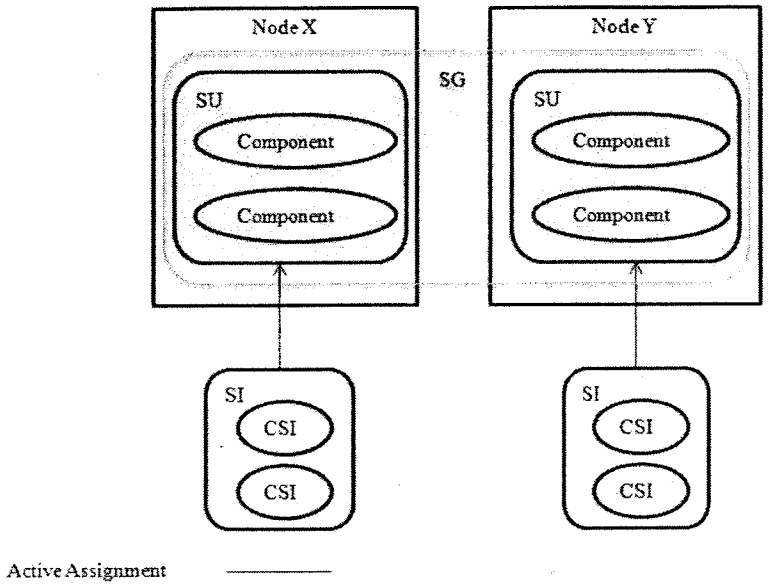


Figure 2.7. An example of "No-Redundancy" Redundancy Model.

2.3.1.10 Service Group Type (SGT)

A *Service Group Type* specifies the list of service unit types that a service group of this type can support. All the service groups of a specific type have the same redundancy model.

2.3.1.11 Application

To provide a higher level service, a set of service groups is aggregated into an *Application*. While an application can contain multiple service groups, each service group belongs to only one application.

2.3.1.12 Application Type

An *Application Type* specifies the list of service group types that an application of this type can support.

2.3.1.13 AMF Nodes and Cluster

Components, service units, service groups and applications are hosted on *AMF Nodes*. An AMF node is a logical entity on a cluster node. An *AMF Cluster* is a set of AMF nodes.

Each service group has a list of configured nodes that AMF specification referred to it as the *Node Group*.

2.4 Software Management Framework (SMF)

If a system is designed to function effectively for a long term, it needs to support configuration changes as it needs to be upgraded and downgraded several times. For an AMF system which is intended to provide services 24/7 such an upgrade and/or downgrade must be performed live while it is up and running with minimum disruption so as to minimize the service outage as much as possible.

The Software Management Framework is the AIS service that orchestrates the live migration of a highly available system from its current configuration to a new one. This migration is called the Upgrade Campaign in the SMF terminology (see Section 2.4.1.2.1).

In order for SMF to control the migration process, it needs to be provided the migration path which is called the Upgrade Campaign Specification (see Section 2.4.1.2.2). The upgrade campaign specification is an XML file that describes all the steps that need to be executed to upgrade the system from a source configuration to a target one.

During the course of an upgrade, SMF maintains the system backup and upgrade state model. It also monitors for potential errors caused by the upgrade and deploys recovery/ repair procedures when needed.

As it is shown in Figure 2.8, SMF does the upgrade with a tight collaboration with the AMF and IMM services. It also may use other AIS services if necessary.

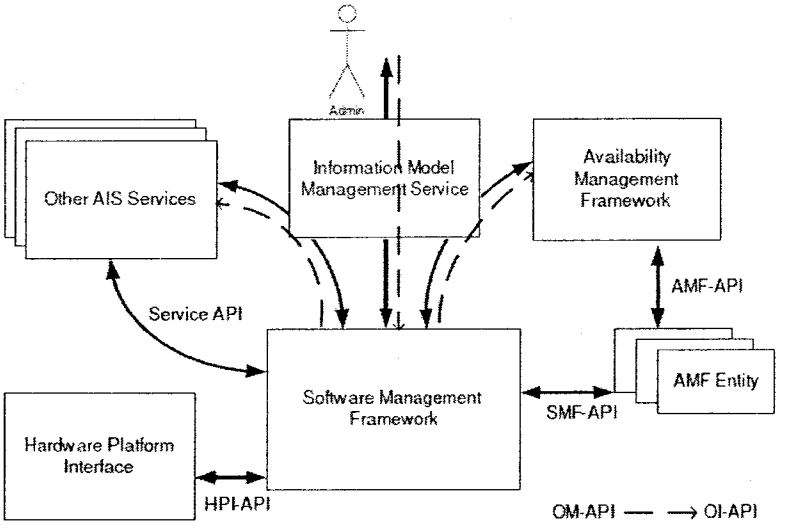


Figure 2.8. The SMF in the SAF Ecosystem (taken from [7]).

2.4.1 Software Upgrade in SAF Systems

SMF specification distinguishes two different phases for the software upgrade; *Software Delivery* and *Software Deployment*. Although these phases can be executed separately in time, the second phase usually requires the successful accomplishment of the first phase.

2.4.1.1 Software Delivery

This phase consists of delivering a new version of the software system that needs to replace the already installed version.

SMF keeps information about an application using different concepts that we discuss in what follows:

2.4.1.1.1 Software Catalog

The *Software Catalog* contains all the necessary information about the software entity types (see Section 2.4.1.1.3) that are currently available in the system. It has information about the versions of software entity types, references to those software bundles that delivered them and to the entities that deployed them.

2.4.1.1.2 Software Entity

A *Software Entity* is the SMF logical entity that represents the instance of software being manipulated by SMF. These entities are implemented by other AIS services such as AMF and SMF only configures them in terms of modifying, adding and removing.

2.4.1.1.3 Software Entity Type

The *Software Entity Type* is the generalization of similar software entities. SMF specification requires any software entity to have a certain type in order to manipulate it. Each Software Entity Type has a Base Entity Type and several Versioned Entity Types. A base entity type generalizes common functionality of executable pieces of software that

are versioned. A versioned entity type implements a base entity type and has specific information (e.g., the list of compatible and required versions of other versioned entity types) related to that particular version.

2.4.1.1.4 Software Bundle

The *Software Bundle* represents a set of interdependent software packages and their related files that will be deployed on the cluster. It is the smallest unit from SMF perspective.

2.4.1.1.5 Software Repository

To handle the software bundles SMF defines a logical single storage. This storage gathers all the available software bundles of the system. Newly delivered software bundles become available by being copied to the *Software Repository*. Similarly to make a bundle unavailable in the system, it should be removed from the software repository.

2.4.1.1.6 Software Installation and Removal

To use software bundles of the software repository on AMF nodes their executable form should be created on the target nodes. This process is the *Software Installation* after which the software can be used to instantiate software entities. Respectively the process of removing software bundles from the location on which they are installed is referred to as *Software Uninstallation.*

Depending on the nature of software bundles, SMF distinguishes two categories of installation and uninstallation operations: *online* and *offline*. An online operation does not violate the availability of other entities in the system and hence can be performed at any time. On the other hand, if the installation/uninstallation operation disturbs the functionality of the other entities of the system it must be performed offline. In that case, all the affected entities should be taken out of service.

The installation and uninstallation processes of software can contain online and offline portions. It is the responsibility of the software vendor to specify which portion of the software system can be installed /unistalled online or offline.

2.4.1.1.7 Ordering of the Operations for Upgrade

SMF specification defines an order for executing the installation/uninstallation operations. After successful completion of these steps the new software will be replaced with the old one:

1. online installation of the new software
2. offline uninstallation of the old software
3. offline installation of the new software
4. online uninstallation of the old software

2.4.1.1.8 Entity Types File (ETF)

Each software bundle is accompanied with an *Entity Types File*. This file is created by the software vendor according to the XML schema [9] to describe any implementation

specific constraint for deployment time. It also specifies software capabilities and limitations.

2.4.1.2 Software Deployment

This phase consists of system migration from its current deployment configuration to the new one. SMF specifies this migration as the upgrade campaign and has defined specific concepts for it:

2.4.1.2.1 Upgrade Campaign

The deployment configuration of a system may need to be changed at any point during its life cycle. These changes may comprise adding new entity types, modifying the current ones or removing existing instances from the system. Figure 2.9 illustrates the major activities during the course of an upgrade campaign.

At the initiation of the upgrade campaign the system creates a cluster-wide backup. During the upgrade process if any error occurred, this backup will be used to retrieve the system's initial configuration. Therefore the upgrade is either committed successfully or terminated due to some problems.

To preserve the availability, an upgrade campaign does not manipulate all the entities targeted for the upgrade concurrently. Instead the whole process is structured into steps and procedures.

At the end of each step, procedure and the upgrade campaign, verifications are performed. If the verification fails at the step level, the actions of the failed step will be

undone and the whole step will be retried. If the retry is unsuccessful or if the procedure verification fails, the upgrade campaign will be undone all the way to its initial point.

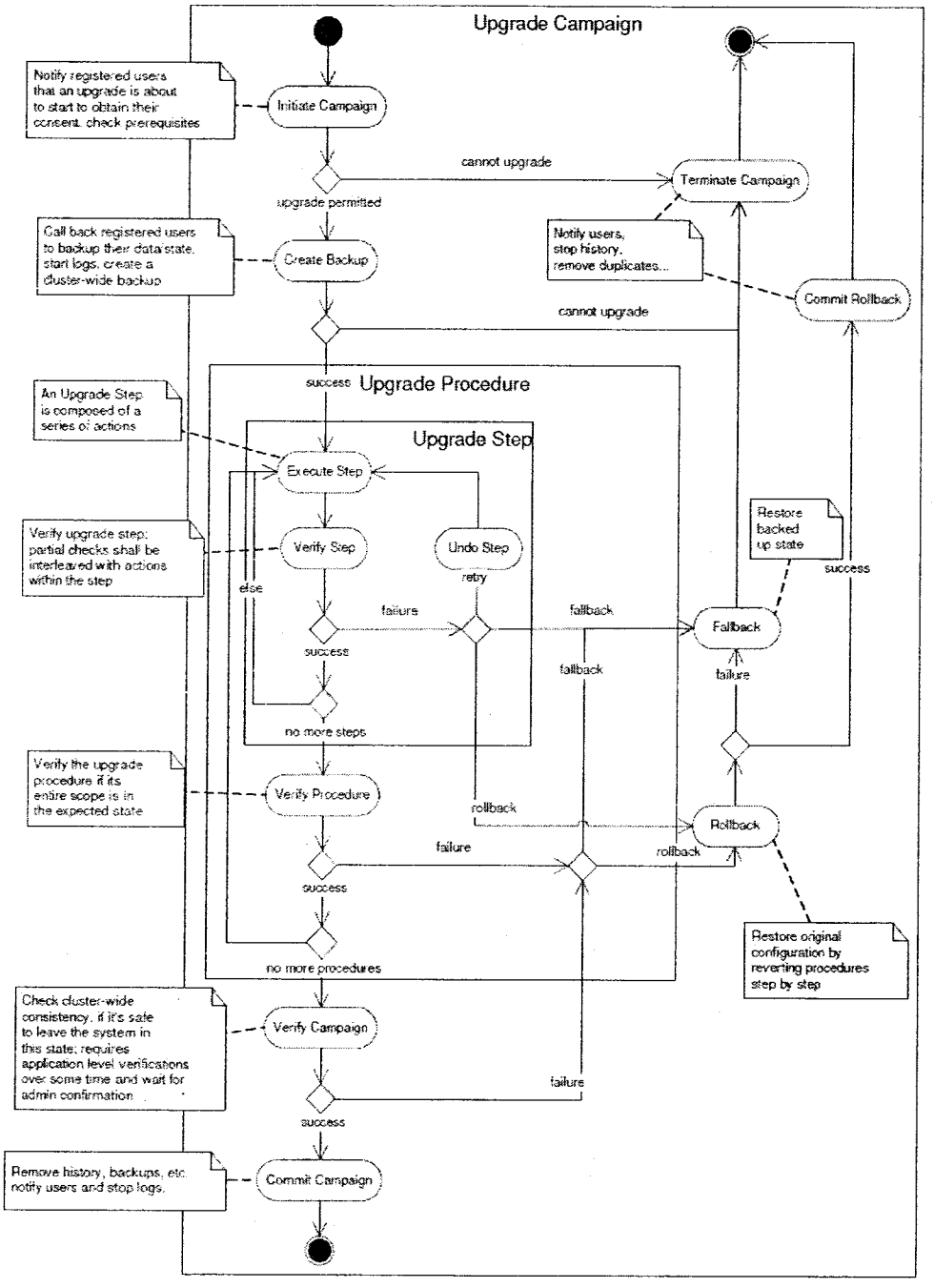


Figure 2.9. Upgrade Campaign Activity Diagram (taken from [7]).

2.4.1.2.2 Upgrade Campaign Specification

SMF only controls the execution of an upgrade campaign. Details on the selection of entities to be targeted by the upgrade, when and how to upgrade them are all built into the *Upgrade Campaign Specification*, which acts as the roadmap of this migration.

The upgrade campaign specification is provided to the SMF service as an XML file compliant to the XML schema [10] defined by the SMF specification.

From the upgrade campaign specification, a campaign object is created in the information model. As mentioned previously, this model is maintained by the IMM and is used to control campaign execution.

It should be noted that during an upgrade there are two configurations in effect; the initial configuration and the target configuration. It is the task of SMF to orchestrate between them according to the upgrade campaign specification.

2.4.1.2.3 Upgrade Step

An *Upgrade Step* is a set of logically related actions that are performed on a group of software entities. Consequently the software entities will be migrated to the target configuration. During an upgrade step some entities may be added to the configuration or removed from it.

2.4.1.2.4 Deactivation Unit

In addition to the entities being upgraded, the upgrade campaign may disturb the functionality of other entities. Therefore to control this migration, it is necessary to

identify these entities and take them out of service as well. The set of these entities is known as the *Deactivation Unit.*

If a software upgrade does not interrupt other entities functionality, its respective deactivation unit will be empty.

2.4.1.2.5 Activation Unit

At the end of the upgrade campaign the newly added entities and some of those which were taken out of service should be activated or reactivated. The set of all these entities creates the *Activation Unit.* If the upgrade campaign only removes the software entities from the system, the respective activation unit will be empty.

In case the upgrade campaign only modifies existing entities and does not add any entity to the configuration or remove them from it, the activation unit will be the same as the deactivation unit. Such an activation unit is called the symmetric activation unit by the SMF specification.

2.4.1.2.6 Actions of the upgrade step

SMF specification defines an ordered set of operations for an upgrade step:

1. Online installation of new software
2. **Lock deactivation unit**
3. **Terminate deactivation unit**
4. **Offline uninstallation of old software**
5. **Modify information model and set maintenance status**
6. **Offline installation of new software**

7. **Instantiate activation unit**
8. **Unlock activation unit**
9. Online uninstallation of old software

The bold actions should be performed within the upgrade campaign to ensure the availability of the system.

2.4.1.2.7 Upgrade Procedure

An *Upgrade Procedure* performs the same upgrade step on a set of similar software entities.

2.4.1.2.8 Upgrade Scope

For each upgrade procedure a deactivation scope and activation scope is defined as the composite set of their respective activation and deactivation units. The union of the deactivation and activation scopes creates the *Upgrade Scope*.

Entities of an upgrade scope usually have tighter dependencies among them.

2.4.1.2.9 Upgrade Methods

SMF specification defines two methods to perform the upgrade campaign: *Single Step* and *Rolling*. Each of them restricts the combination of upgrade steps into an upgrade procedure differently; so that the dependency among the entities is considered to gain the desirable availability during an upgrade.

2.4.1.2.9.1 Rolling Upgrade

In the rolling upgrade method the upgrade scope is divided into multiple deactivation-activation units. The upgrade iterates the same upgrade step over these pairs until the entire scope is covered.

This method takes out only one deactivation unit at a time while the rest of the system is running. Therefore it will lead to minimum service outage with respect to its upgrade scope.

Using this method for upgrade may end in collaboration of different versions of software for the upgrade period. Therefore if the old and new versions of software have incompatibility problems this method is not appropriate.

2.4.1.2.9.2 Single Step Upgrade

Contrary to the rolling upgrade, single step upgrade has only one deactivation- activation unit pair. It takes out all the entities of the deactivation unit simultaneously and then reactivates the entities of the activation unit concurrently at the end of the upgrade step.

Since this method upgrades entities of the upgrade scope all together, it loses the services being provided by those entities exclusively and has service outage.

However this method is indicated by SMF specification because it eliminates the compatibility issues among different versions of software entities. It is appropriate to be used for removing entities from the configuration or adding new ones to it. Since in the former case the provided services are not demanded anymore and in the latter case the services were not provided before and their availability is not meaningful.

2.4.1.2.10 Service Outage

During an upgrade some service instances may not be provided by the system at all. SMF specification refers to this service disruption as the *Service Outage*.

For each upgrade campaign specification an *Acceptable Service Outage* is defined. It is the threshold beyond which the upgrade results in the service disruption that is not tolerable.

The *Minimum Service Outage* for each upgrade campaign specification is calculated through matching its deactivation units with the deployment configuration when all of its service instances are fully assigned and it has no disabled entity. The upgrade campaign will only be initiated by SMF if the expected runtime service outage is lower than the acceptable service outage.

2.4.1.3 Typical Software Management Information Flow

As it is illustrated in Figure 2.10 the campaign builder needs some specific information to generate an upgrade campaign:

- An entity types file provided by the software vendor that describes the content of a newly delivered software bundle.
- Current configuration of the system that should be upgraded. This configuration is maintained by the IMM service.
- Other inputs that should be taken into account. This portion will be provided by the system administrator.

The upgrade campaign specification that is generated by the campaign builder is an XML file in accordance with the XML schema [10] provided in the SMF specification.

After the successful delivery of a new software bundle to the software repository, and once the upgrade campaign specification is provided to the SMF, the execution of an upgrade campaign will be started. During the upgrade SMF uses AMF and other AIS services to access, control and modify entities.

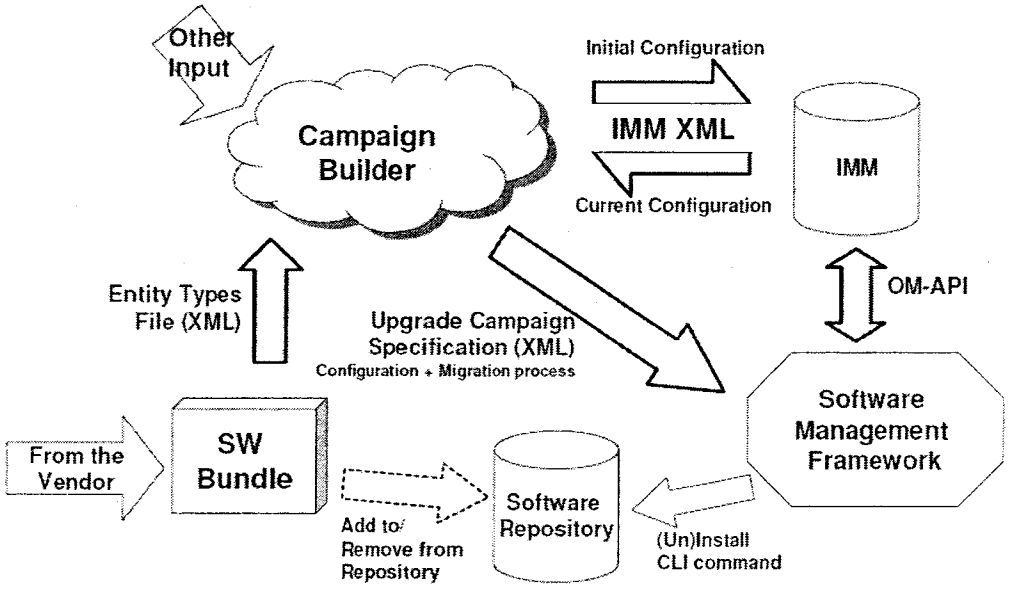


Figure 2.10. Typical Software Management Information Flow for an Upgrade (taken from [7]).

2.5 Related Work

Service standardization at the SA Forum is an evolutionary stream. As new services are being standardized, current specifications are being revised and updated by the Forum. Therefore several releases may exist for a specific service. However, up to now there is only one release for the Software Management Framework [7] on which our work is based.

Along with the SA Forum, Object Management Group (OMG) [11] and Java Community Process [12] are in the process of standardizing middleware for online upgrade [13]. The Java Community Process has developed a Java Specification Request (JSR), JSR #117 [14], which defines the programming model and runtime support for implementing J2EE applications requiring continuous availability. The OMG has also developed standard specification for an online upgrade of the Common Object Request Broker Architecture (CORBA). There, they have defined interfaces for objects that provide online upgrade capability. Among all of the interfaces *Upgrade Manager* is the principal management interface to achieve online upgrade. Methods of the Upgrade Manager are used to initiate, control, commit and revert upgrades of the objects. OMG has defined three scenarios for online upgrade: *Pushed Upgrade*, *Pulled Upgrade*, and *Smart Clients and System Management*. However OMG does not address availability and instead it recommends the combination of online upgrade and fault tolerant CORBA. Its objects are also more granular than the SA Forum entities; therefore their scenarios do not apply to our case.

In [15], an overview of the SA Forum Software Management Framework is given and the steps towards a software upgrade in such systems are explained. Though, they do not go further than introducing SMF upgrade methods. And the details on generating an upgrade campaign specification that results in minimum service outage are not addressed.

The most related work to our research in the SAF context is stated in [16]. The authors in [16] have focused on systems utilizing database-centric applications and they have defined five software layers to upgrade: *Operating System*, *HA Framework*, *Database*

Management System, *Database schema* and *Database Applications*. As it is specified in the SMF, entities dependency is a major problem in the rolling upgrade. In [16] they have devised an *Upgrade Food Chain* (*UFC*) to capture the dependencies among components. UFC is a directed graph which each of its nodes is a component and the edges are pointing to other components that are dependent to the former component. Intuitively the upgrade must be started from the outmost component and traverse the UFC in reverse direction. They have explained some methods to remove the cycles in the UFC. Finally for each software layer that was introduced they have defined in details the steps on performing the upgrade.

However [16] differs from our work in the sense that we do not concentrate on the systems which utilises specific type of applications as they did. Besides at the current stage we have addressed a different aspect of the problem domain and we did not take into account the components dependencies in our methods for generating an upgrade campaign. Moreover in addition to the rolling upgrade we have devised some algorithms to generate single step upgrade campaign specifications that will lead to minimum service disruption.

Chapter 3 - Upgrade Campaign Generation

In this chapter, after a brief introduction and a few definitions, we present the different scenarios we handle for upgrading AMF configurations. After providing an overall picture of the upgrade campaign generation approach, we discuss the different challenges we faced during this research and our solutions. The upgrade campaign generation processes for the different scenarios are then described in details along with the respective specific algorithms. We conclude this chapter with the limitations of our solutions and with the assumptions we made in order to tackle several issues we faced during this research work.

3.1 Assumptions and Definitions

The building blocks of a highly available system are tightly interdependent. Therefore to upgrade these systems one should be careful and take into account a lot of details. Throughout this research work, we have made some assumptions to limit the scope of problem and facilitate the cases we handle. We presume that:

- Equivalent components of different service units of a service group have the same RDN.
- Components are being added to the existing service units of the configuration.
- All the service units of a service group are identical.

- Removing a single service unit will result in the removal of its containing components.
- Removing service units by specifying the service unit type will result in the removal of the application along with its service group, service units and containing components if the application related to this service unit type has only one service group. If the application has other service groups, only the service group that contains this service unit type will be removed along with its service units and containing components.
- A single service unit will be added to an existing service group of the configuration and its containing components will be added consequently.
- Adding service units by specifying the service unit type will result in the addition of an application with all of its service groups, service units and containing components.
- If software has an offline portion within its installation or uninstallation operations, we do not separate its online and offline processes. Instead the software will be installed or removed completely offline and the whole node will be affected.
- In the rolling upgrade procedure the scope of upgrade within each step is the entire node.
- If we lock a single node in an upgrade step due to the redundancy within the system, the availability of the services will not be interrupted.

The ultimate upgrade campaign generator should be able to compare the current and the target configurations and determine automatically the difference and then generate the upgrade procedures to be performed to move the system from the current to the target configuration. Such a comparison could be done through entity names or their features to find equivalent ones in both configurations. However, when we generate a target configuration, since all the names are generated and the features are changing due to the upgrade, finding automatically the equivalent entities and determining the difference between two given configurations is not a straightforward task. Therefore, in this thesis, we assume that the user has the target configuration in mind and will provide the set of necessary modifications as input to move the system to the target configuration. This keeps the entity names consistent between the current and target configurations. We also assume their configuration will be provided as an additional one whenever entities of new types should be added to the system. The later assumption replaces the ETF with the additional configuration in the set of required input for generating an upgrade campaign.

We use the term *Upgrade Tuples Set* to refer to the set of configuration changes the user would like to perform. *Upgrade Tuples Set* is a set of upgrade tuples, each has the following format and meaning: *(Source, Target, ChangeSet, NodeList)*

- *Source*: It is an entity/type from the current configuration to be upgraded.
- *Target*: It is an entity/type from the current or the additional configuration.
- *ChangeSet*: The service group that contains the *Source* and/or will contain the *Target*. It is used as a refinement for applying the entity/type selection.
- *NodeList*: The list of impacted nodes on which the software installation and removal should be performed for a particular *Source* and *Target*.

Source and *Target* can contain an instance of the component *RDN* (The component's name relative to its parent service unit, for more details see Section 2.3.1.1), the component type, the service unit *RDN* (The service unit's unique name within its containing service group, for more details see Section 2.3.1.5) or the service unit type. It should be noted that when a type is specified in the *Source,* all the entities of that type are selected in the source configuration. However, indication of an entity's *RDN* will reduce the set of entities of a selected type to the subset that has the specified *RDN.*

3.2 Upgrade Scenarios

In this thesis, we handle three basic upgrade scenarios and their combinations:

- Changing the type of existing entities to another type,
- Removing entities from the configuration, and
- Adding entities to the configuration.

In the following subsections we describe these scenarios in detail.

In all cases the *Source* and *NodeList* = *{nodes}* are part of the current configuration. The *Target* and *ChangeSet* could be part of either of the current configuration or the additional configuration that is provided by the user depending on the operation.

3.2.1 Changing Type Operation

In this scenario, the upgrade targets the current entities' types to be changed to other ones. Each row in the Table 3.1 shows a possible variant of upgrade tuples for this scenario.

The upgrade tuple in the first row indicates that the service units of *SG* of current type *SUT1* that is specified as the *ChangeSet* need to be upgraded to type *SUT2*. It is also required to replace the old software with the new one on the nodes that are specified by *{nodes}*. Precisely, software bundles referenced by component types of *SUT1* need to be removed and software bundles referenced by the component types of *SUT2* need to be installed on these nodes.

Table 3.1. Upgrade tuples variations of "Changing Type" scenario.

Source	***Target***	***Change Set***	***Node List***
SUT1	*SUT2*	*SG*	*{nodes}*
CT1	*CT2*	*SG*	*{nodes}*
SURDN	*SUT2*	*SG*	*{nodes}*
ComponentRDN	*CT2*	*SG*	*{nodes}*

The second upgrade tuple specifies that all components of type *CT1* in *SG* need to be upgraded to *CT2*. Similar to the first case, it requires also the replacement of the old software of *CT1* with the new one of *CT2* on the nodes that are specified in the *NodeList*.

The third upgrade tuple is equivalent to the first one as it will target all service units of the *SG* and upgrade them to *SUT2*. This is due to the consistency assumption we have made that all service units of a service group need to be of the same type.

Finally, the forth upgrade tuple will target the component with the specified RDN in each service unit of the *SG* and upgrade them to *CT2*. The component type is also identified through its RDN. The upgrade will also replace the old software with the new one on the nodes specified in the *{nodes}*.

Details on generating upgrade procedures for this scenario are discussed in Section 3.3.5.4.

3.2.2 Removal of entities

In this case, the objective of the upgrade is to remove entities from the current configuration. Each row in the Table 3.2 specifies a possible variant of the upgrade tuple for the Removal scenario. As we are removing entities and types from the configuration the *Target* is empty in all cases.

Table 3.2. Upgrade tuple variants for the Removal scenario.

Source	***Target***	***Change Set***	***NodeList***
SUT	ε	*SG*	*{nodes}*
CT	ε	*SG*	*{nodes}*
SURDN	ε	*SG*	*{nodes}*
ComponentRDN	ε	*SG*	*{nodes}*

The first upgrade tuple specifies a *SUT* in a *SG* to be removed. Since we assume that each service group has only one service unit type, this tuple will result in removing the whole *SG* from the configuration. If that is the only service group of the related application, the application will be removed consequently. The software that is referenced by component types of the *SUT* will also be removed from the nodes specified in the *NodeList*.

The second upgrade tuple specifies the removal of all components of the specified type from the *SG*. It will also uninstall the software referenced by this component type in the nodes of *NodeList*.

The third upgrade tuple will result in the removal of the specified *SU* together with its components from the *SG*. It will also uninstall the related software from the nodes of *NodeList*. The nodes will be removed from the *SG*'s node group as appropriate.

Finally, the last upgrade tuple targets all components with the specified RDN and specifies their removal from each service unit of the *SG*. As a result related software bundles will be removed from the nodes specified in the NodeList.

Details on generating upgrade procedures for this scenario are addressed in Section 3.3.5.

3.2.3 Addition of entities

In this scenario, the upgrade adds new entities to the current configuration. Table 3.3 shows the possible variants of upgrade tuples for this scenario. In this case, the complete configuration of entities to be added should be provided as an additional configuration; where all the entities and their appropriate entity types are specified.

It is noted that in Table 3.3 the *Source* is empty. In addition to that, adding new components cannot happen with the specification of their types only; since they need to be encapsulated within a service unit in an AMF configuration.

As we have already addressed in the assumptions, the first upgrade tuple specifies the addition of a new application with all of its service groups, service units of type *SUT* and

their contained components. The related software bundles need to be installed on the nodes specified in the *NodeList*.

Table 3.3. Upgrade tuple variants for the Addition scenario.

Source	***Target***	***Change Set***	***NodeList***
ε	*SUT*	*SG*	*{nodes}*
ε	*SURDN*	*SG*	*{nodes}*
ε	*ComponentRDN*	*SG*	*{nodes}*

The second upgrade tuple specifies the addition of one service unit with the specified RDN along with its contained components to an existing service group of the configuration that is specified in the *ChangeSet*. This will require the addition of the respective software to the nodes in *NodeList*.

Finally the last upgrade tuple specifies the addition of a component with the specified RDN to all service units in the service group *SG*. The service group and its service units must exist in the configuration. The upgrade will install related software on the nodes in the *NodeList*.

Details on generating upgrade procedures for this scenario are addressed in Section 3.3.5.

3.3 Upgrade Campaign Generation Approach

3.3.1. Overall Approach

The overall upgrade campaign generation approach is illustrated in Figure 3.1. As shown in this figure, the current configuration of the system to be upgraded is provided by the user to the *Upgrade Campaign Generator*. In addition, the user provides the set of modifications to the current configuration, specified in the *Upgrade Tuples Set*.

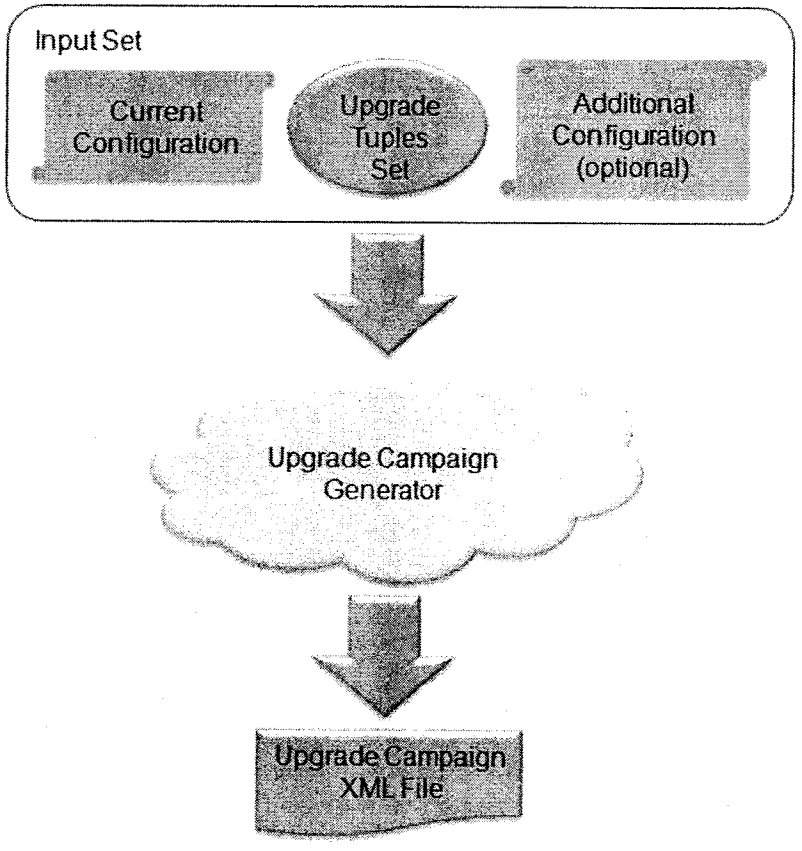


Figure 3.1. Upgrade Campaign Generation: Overall picture.

Optionally, if any new entity needs to be added to the current configuration, its complete configuration should be provided.

The *Upgrade Campaign Generator* processes the Input Set and generates the necessary upgrade procedures that are saved into an XML file.

The different steps of the Upgrade Campaign Generator are shown in Figure 3.2. After collecting the input from the user, the first step consists of checking if the provided upgrade tuples are consistent and complete. It calculates any additional set of entities that need to be upgraded because of dependencies and adds them to the set of upgrade tuples.

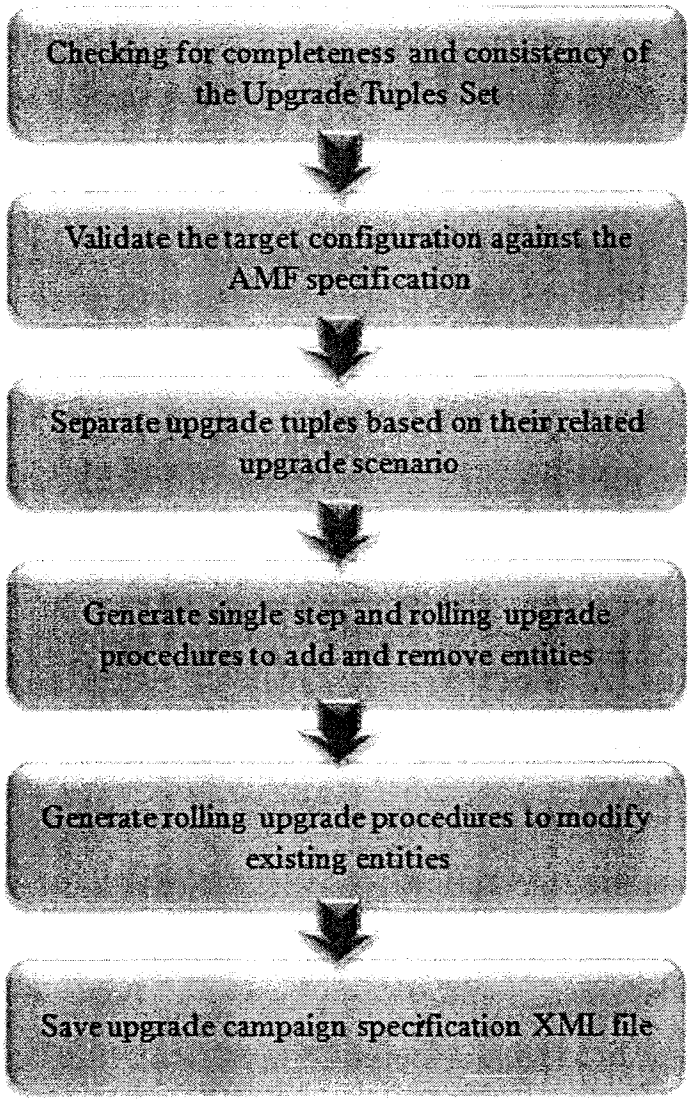


Figure 3.2. Main steps of the upgrade campaign generation.

The second Step manipulates a copy of the current configuration, and according to the *Upgrade Tuples Set* creates an instance of the target configuration and validates it against the standard AMF model.

If the target configuration is not compliant to the standard AMF model, the whole process is interrupted. In the case where the target configuration is valid, upgrade tuples are separated based on their relevant upgrade scenario. If there are upgrade tuples of Addition and Removal scenarios, single step and if necessary rolling upgrade procedures will be generated to add entities and their respective software to the system or to remove entities and the related software bundles from the system. For the upgrade tuples of the Changing Type scenario, if any exists, rolling procedures will be generated also, to modify existing entities of the configuration and replace the software with a new version. In the end, the generated upgrade campaign specification will be saved as an XML file, to be provided to the SMF.

3.3.2 Input Data

As mentioned previously, the required input for the upgrade campaign generation consists of three parts:

- The current configuration of the system that needs to be upgraded,
- Optionally an additional configuration of entities and their types that needs to be added to the system and,
- The modifications to the current configuration provided as a set of upgrade tuples.

The current configuration of the system to be upgraded is represented as an IMM XML file and contains AMF entities and their types.

As discussed in Chapter 2, ETF provides entity types and describes the application from the software vendor perspective. To utilise an ETF for generating an upgrade campaign, AMF entities and entity types should be generated from this ETF. The new entities and types can then be added to the system's current configuration.

Despite SMF specification that requires the ETF as an input for the upgrade campaign generator; our approach does not use ETF raw types. We assume that a configuration has been generated based on the newly delivered ETF. To add entities to the system, this new configuration with new AMF entities and types will be provided as input. Notice that providing a new configuration is mandatory and useful only when new entities or entity types are to be added to current configuration.

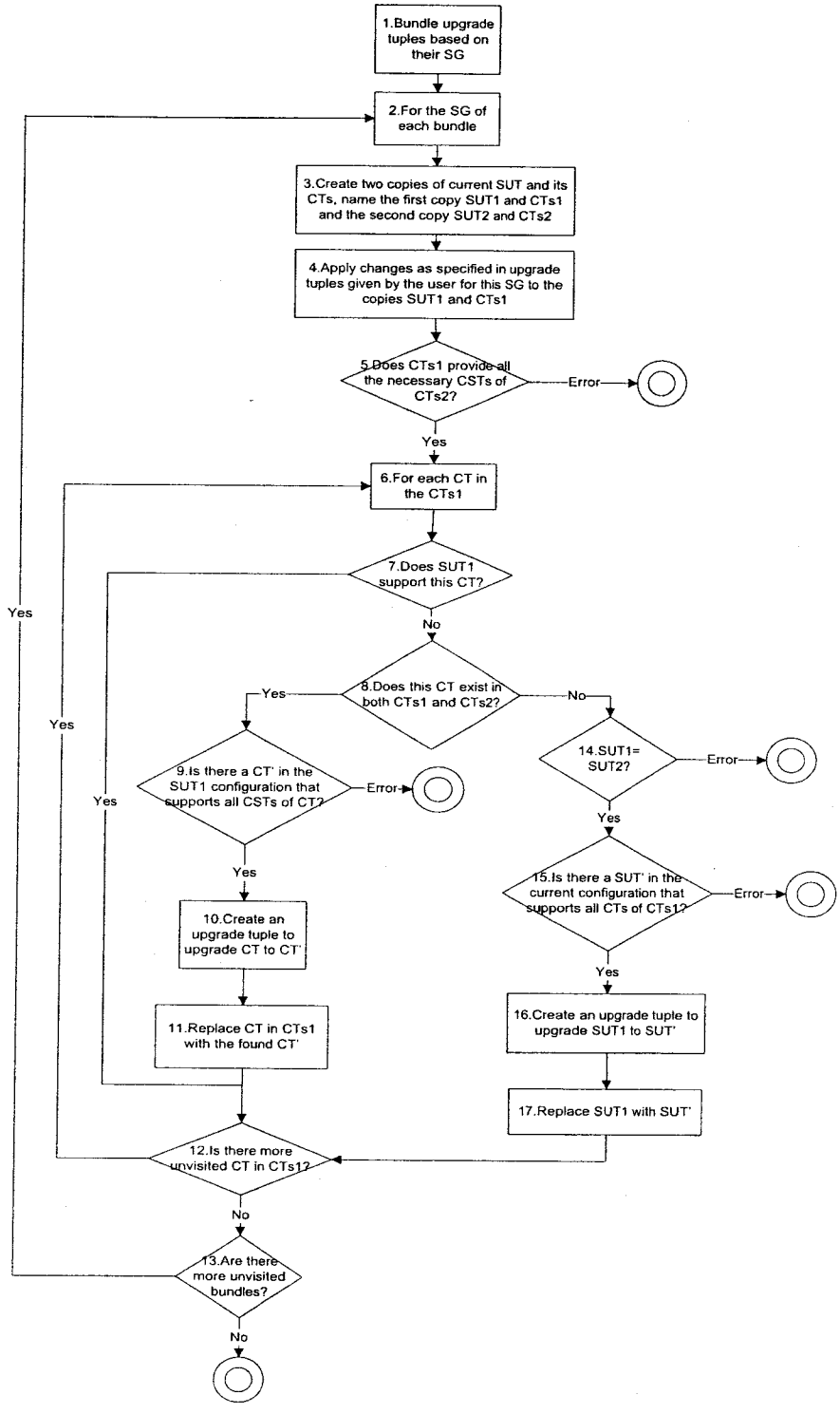
The set of modifications to the current configuration is provided with the *Upgrade Tuples Set*. Each upgrade tuple has the format discussed in Section 3.1. The different variants of upgrade tuples are also discussed in details in Table 3.1 through Table 3.3.

3.3.3 Checking for Completeness and Consistency

As already mentioned, due to the complexity in determining the difference between two configurations, we rely on the user to provide a set of upgrade tuples. However, the user may overlook some dependencies. This set may be incomplete e.g. a component is being upgraded to a new type that is not supported by its current service unit type and the upgrade tuple to upgrade the service unit type to an appropriate one is missing. The

Upgrade Tuples Set may also contain inconsistencies. For example, a component type and its containing service unit type being upgraded but the new service unit type does not support the new component type.

Before proceeding with generating the upgrade procedures, we need to make sure that the *Upgrade Tuples Set* is complete and consistent; meaning that it has all the necessary upgrade tuples and they do not contradict each other so that the target configuration will not have any inconsistency within its types. For that purpose, we have devised an algorithm which takes the *Upgrade Tuples Set* as input and checks for its completeness and consistency according to the criteria discussed in Section 3.3.3.1. If there is any incompleteness within the *Upgrade Tuples Set*, if possible, this algorithm will generate additional upgrade tuples and add them to this set. If this algorithm cannot generate the additional upgrade tuple we will reject the campaign generation; e.g. if a component type upgrade triggers a service unit type upgrade but the appropriate type for the latter upgrade cannot be found within the source configuration. In addition to that, if there is any inconsistency in the *Upgrade Tuples Set* this algorithm will find it and reject the campaign generation. This algorithm sorts and processes all the upgrade tuples in the *Upgrade Tuples Set* to determine the completeness and consistency of the set. Any additional upgrade tuple that is generated through this algorithm is added to *Upgrade Tuples Set*. This algorithm is described with Flowchart 3.1.



Flowchart 3.1. Checking for Completeness and Consistency of the Set of Upgrade Tuples.

Due to the interdependency between the building blocks of an AMF system, modifications to these entities and types that are provided in terms of upgrade tuples of the *Upgrade Tuples Set* are consequently interdependent. Therefore, to check for completeness and consistency of the upgrade tuples we need to consider their effect on the source configuration entities and types all together and not individually. Since there can exist upgrade tuples that complement each other's effect and unless we do not process all of them, we will not have the complete picture of the result. As an instance, consider four upgrade tuples three of which upgrade component types of a service group and the remaining one, upgrades its service unit type. Each of these upgrade tuples may result in inconsistency if processed separately from the others while their complete set, if they do not contradict each other, will not result in such an inconsistency. For that purpose, in the first four Steps of the Flowchart 3.1, for each service group a copy of its target service unit type and its containing component types is generated by manipulating its current service unit types and containing component types according to the relevant upgrade tuples.

As it is mentioned earlier, the problem that we seek to solve through this algorithm has its origins in the incompleteness or contradiction within the provided *Upgrade Tuples Set*. Having the target service unit type and its containing component types we cannot distinguish the cause of inconsistency since for each specific type we do not know whether it has remained intact or been manipulated through an upgrade tuple. But if we have an image of what were the initial situation of service unit type and its containing component types by comparing the two sets we will find those entities that have been changed. As you can see in Flowchart 3.1, in the third Step a copy of the current service

unit type and its component types is maintained, later on through the Steps 6, 7 and 8 this copy is used to capture the source of inconsistency.

3.3.3.1 Criteria for Completeness and Consistency Check

Checking for consistency of the *Upgrade Tuples Set* is a complex and thorny issue because of the different cases we have identified:

a) Identicalness of service units within a service group,

b) Consistency among a service unit type and its component types,

c) Consistency among component types of a service unit,

d) Maximum allowed number of components of a specific type within a service unit,

e) Software bundles that exist on a node and so on.

However, in this thesis we only handle cases (a) and (b) which allow us in some cases to resolve inconsistencies by addition of new upgrade tuples. To handle all the other cases we need to do the validation of the target configuration. This completes the consistency checking.

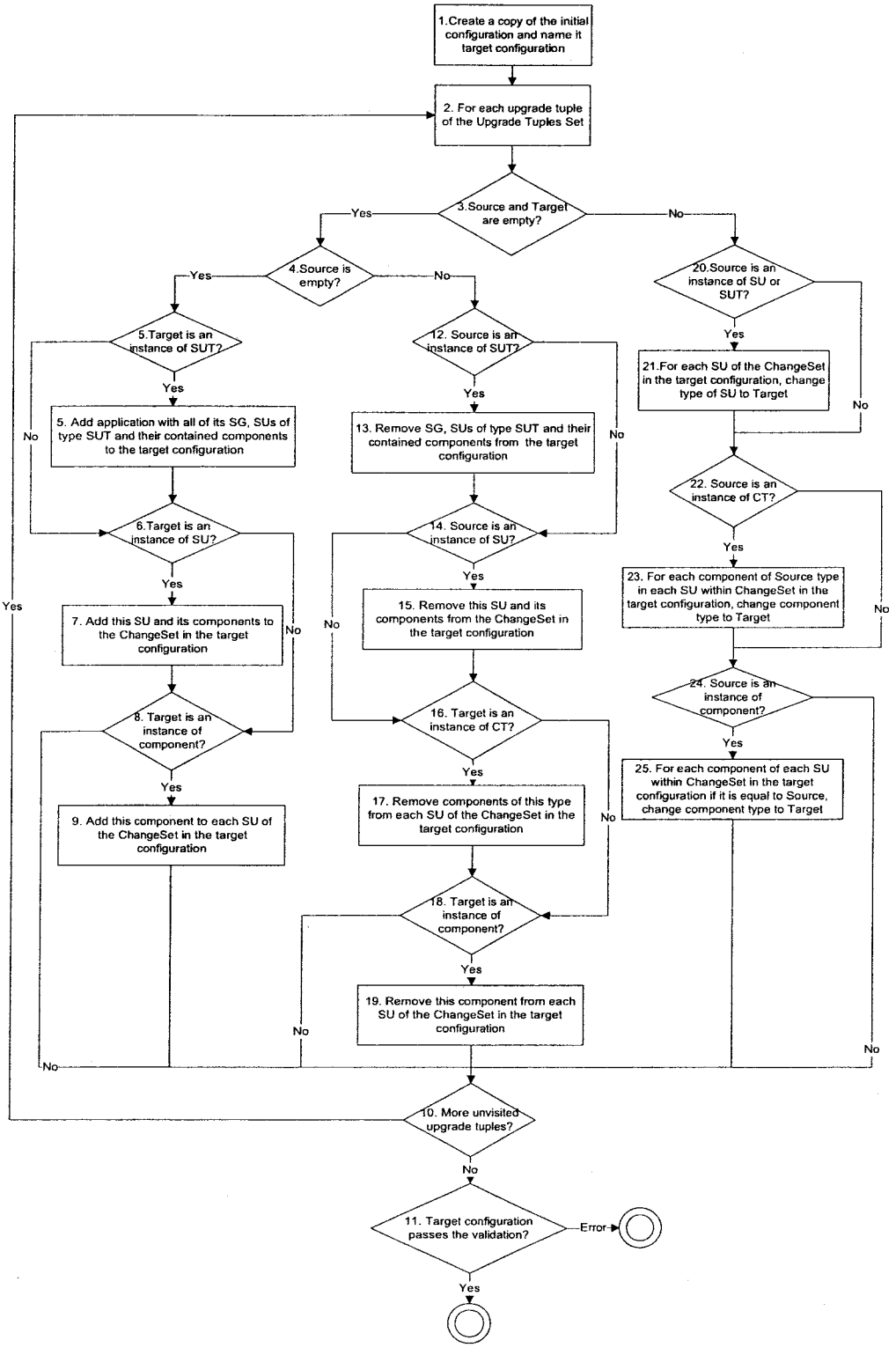
3.3.4 Validation of Target Configuration

Since the completeness and consistency criteria are not exhaustive it may fail noticing some dependencies and potential problems with the target configuration; we proceed with the validation of the target configuration. As an instance consider the case when the user specifies that a component should be removed from a specific node along with its related software bundle, however there is another component within a different service group but

on the same node that is not being upgraded and has been configured to use the same software bundle. Consequently removing this software bundle will interrupt the functionality of the second component and the whole configuration is not valid any more. For that purpose, the target configuration is generated by manipulating the current configuration according to the *Upgrade Tuple Set* (containing the upgrade tuples provided by the user in addition to the ones we generated automatically for completeness). We feed the target configuration to the **validator** tool [17], which is capable of checking the compliance of any given configuration with the AMF specification. In case the target configuration is not valid, the campaign generation will be interrupted and the user will be referred to the generated log file that contains a list of all errors. When the target configuration is valid, we proceed with the generation of the upgrade procedures.

The algorithm to generate the target configuration from the source configuration and the *Upgrade Tuples Set* is described by Flowchart 3.2.

Note that in addition to the changes that are explicitly derived from the upgrade tuples (e.g. changing type of a component from its source to the target), sometimes the upgrade tuples contain more implicit changes as well that we need to deduce from them. For example if we are removing a component from service units of a service group the related component service instances should be removed as well. Only this way the configuration that is gained through the above algorithm can become a correct representation of the target configuration.



Flowchart 3.2. Validation of Target Configuration.

3.3.5 Generating Upgrade Procedures

3.3.5.1 Minimizing Service Outage

During the upgrade, service interruption is highly probable. As an upgrade campaign designer one should try to minimize this service outage as much as possible. In this section we introduce the causes of service outage in an upgrade campaign and our solutions to eliminate them or reduce their impact. Later on, in the following subsections, these solutions are further explained with flowcharts.

SMF specification suggests single step upgrade procedure to add new entities to the configuration or to remove existing entities. However, this could result in a great amount of outage if the upgrade scope is not calculated carefully; since all the entities within the upgrade scope will be taken out of service at once. To overcome this problem, we propose to divide the upgrade scope into many smaller ones, so that at each point of time a smaller number of entities are taken out of service. For instance, adding or removing a component from a service unit might result in losing the services being provided by that service unit. Now if the component is being added to or removed from all service units of a service group through one single step upgrade procedure, all the services being protected by this service group will lose their availability. In order to break up the upgrade scope into many smaller ones we have separated operations on components from the ones on entities other than components. To add and remove components we generate a series of single step upgrade procedures that imitates the rolling upgrade procedure. So that at any point only one service unit of the targeted service group is taken out of service

while the rest of its service units are up and providing services. The related algorithms are further described in Sections 3.3.5.3.2 and 3.3.5.3.5.

Another problem that may lead to service outage in single step upgrade procedures is the software installation and removal if the software has an offline portion. In such situations, an installation and removal may harm the functionality of other entities running on a node targeted by the upgrade. To solve this problem we separate software with offline portion from the ones without offline portion. The algorithm generates single step procedures to install or remove the software without offline portion. For installation and removal of software with offline portion additional rolling upgrade procedures will be generated so that at each point only one node of the service group's node group is affected. It is mentioned that in software installation and removal the set of impacted entities may be the service unit or the entire node. For simplicity, our algorithm is presented with the assumption that the scope is the entire node. Further optimization are devised to reduce the number of times we lock a node to perform the operations on it. Addition and Removal scenarios are addressed in Sections 3.3.5.3.3, 3.3.5.3.4 and 3.3.5.3.6 and Changing Type scenario is discussed in Section 3.3.5.4.

3.3.5.2 Upgrade Tuples Classification

If there is no error in the previous steps, the upgrade tuples are categorized into two sets. The algorithm is given in Figure 3.3.

The upgrade tuples of the Upgrade Tuples Set (the ones provided by the user and those we added later for completeness) are divided into two sets by examining the Source and

Target existence in the tuple. The rollingTuples set contains the upgrade tuples that change type of existing entities; i.e. have both Source and Target. The otherTuples include the upgrade tuples to add and remove entities.

```
1: BEGIN
2:      FOR (each upgrade tuple of the Upgrade Tuples Set)
3:          IF (Source is not empty AND Target is not empty)
4:              Add this upgrade tuple to rollingTuples
5:          ELSE
6:              Add this upgrade tuple to otherTuples
7:          END IF
8:      END FOR
9: END
```

Figure 3.3. Upgrade Tuples Categorization.

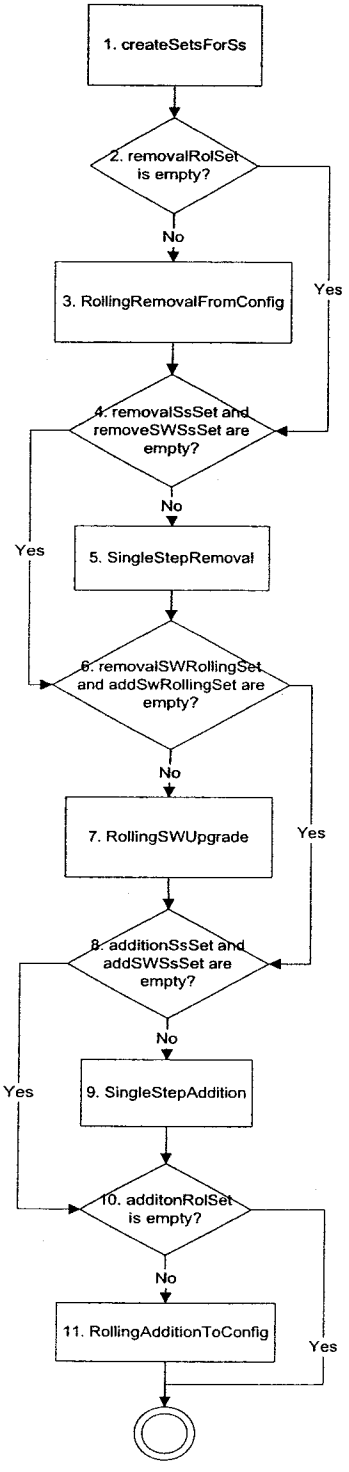
To generate a complete upgrade campaign, if the otherTuples set is not empty single step and rolling procedures will be created as appropriate to add and remove entities. Similarly, if the rollingTuples set is not empty based on its upgrade tuples, rolling upgrade procedures will be generated to upgrade entities' types.

3.3.5.3 Addition and Removal Scenarios

As we have discussed in Section 3.3.5.1, careful calculation of the scope of a single step upgrade procedure will result in minimizing the service outage. The processing of the upgrade tuples of the Addition and Removal scenarios and the generation of appropriate upgrade procedures is illustrated in Flowchart 3.3.

In Flowchart 3.3, the first Step of this algorithm upgrade tuples of the Addition and Removal scenarios are further divided into many sets. This separation is done according to the nature of the entity the tuple will add or remove, or the nature of the software bundle it will install or remove. The reasoning for such a division and the contents of each set are further discussed in the following sections.

When removing entities the related software, if needed, should be removed from the nodes ***after*** the entities have been removed completely from the configuration. In contrast when adding entities the related software should be installed on the node ***before*** the entities being added to the configuration. This is the timing we had to consider while generating upgrade procedures for Addition and Removal scenarios. We decided to generate separate procedures for adding and removing entities as these operations typically have no service impact since their services are also being added or removed. However since software installation and uninstallation may have such impact, as shown in Flowchart 3.3, in Step 7 we tried to merge the addition and removal of software that has an offline portion so that each involved node will be lock the minimum possible times.



Flowchart 3.3. Addition and Removal Scenarios.

We could have followed other solutions: For instance we could have not separated processing the upgrade tuples of Addition and Removal scenarios and then our process could have steps as follows: installing software with offline portion, adding entities to the configuration and removing entities from it, installing and removing the software without an offline portion and finally removing software with offline portion.

Though in our approach we try to minimize the number of upgrade procedures and times we lock a node in the offline installation and removal, for the second approach we could have minimized the number of upgrade procedures and times we lock a node to add entities to it or remove them from it. But comparing these two approaches and precisely determining their strength and weakness points needs a formal investigation and proof that is beyond the scope of this work.

3.3.5.3.1 Analysis and Separation of Upgrade Tuples

As discussed earlier when generating single step upgrade procedures to add and remove entities the upgrade scope has a great impact on the service outage it causes. For that reason we have studied all the entities we handle in our Addition and Removal upgrade scenarios and the upgrade scope they create. Initially we have separated them into three streams of changes:

- Changes to the service units and service unit types,
- Changes to the components and component types and.
- Changes to the software bundles.

For each set we then have studied the upgrade scope that they create to see if it is greater than these entities or stays the same:

- Service unit type addition: As we have already mentioned in our assumptions (see Section 3.1) this will result in the addition of a new application along with its service groups, their service units and containing components. No service was provided previously and therefore no service instance will be interrupted.
- Service unit type removal: As discussed in the Section 3.1 the service unit type removal can result in the removal of the service group along with its service units and containing components if the application has other service groups. It also can result in the removal of the application as well as the service group and service units and their containing components if the application has no other service group. In both cases services of the service group are not needed any more and their removal will not affect the availability.
- Service unit addition and removal: Since the service unit is being added to or removed from a service group that supposedly has other service units handling the assigned service instances the service availability will not be harmed.
- Component type addition and removal: Adding components of a specified type to a service unit or removing them from it may interrupt the functionality of that service unit. If the components are being added to or removed from all the service units of a service group at once, the whole service group will lose its functionality. So we need to target a smaller number of service units at each point of time. Since the number of running service units of a service group may vary, we decided to target only one of them at a time to be upgraded.

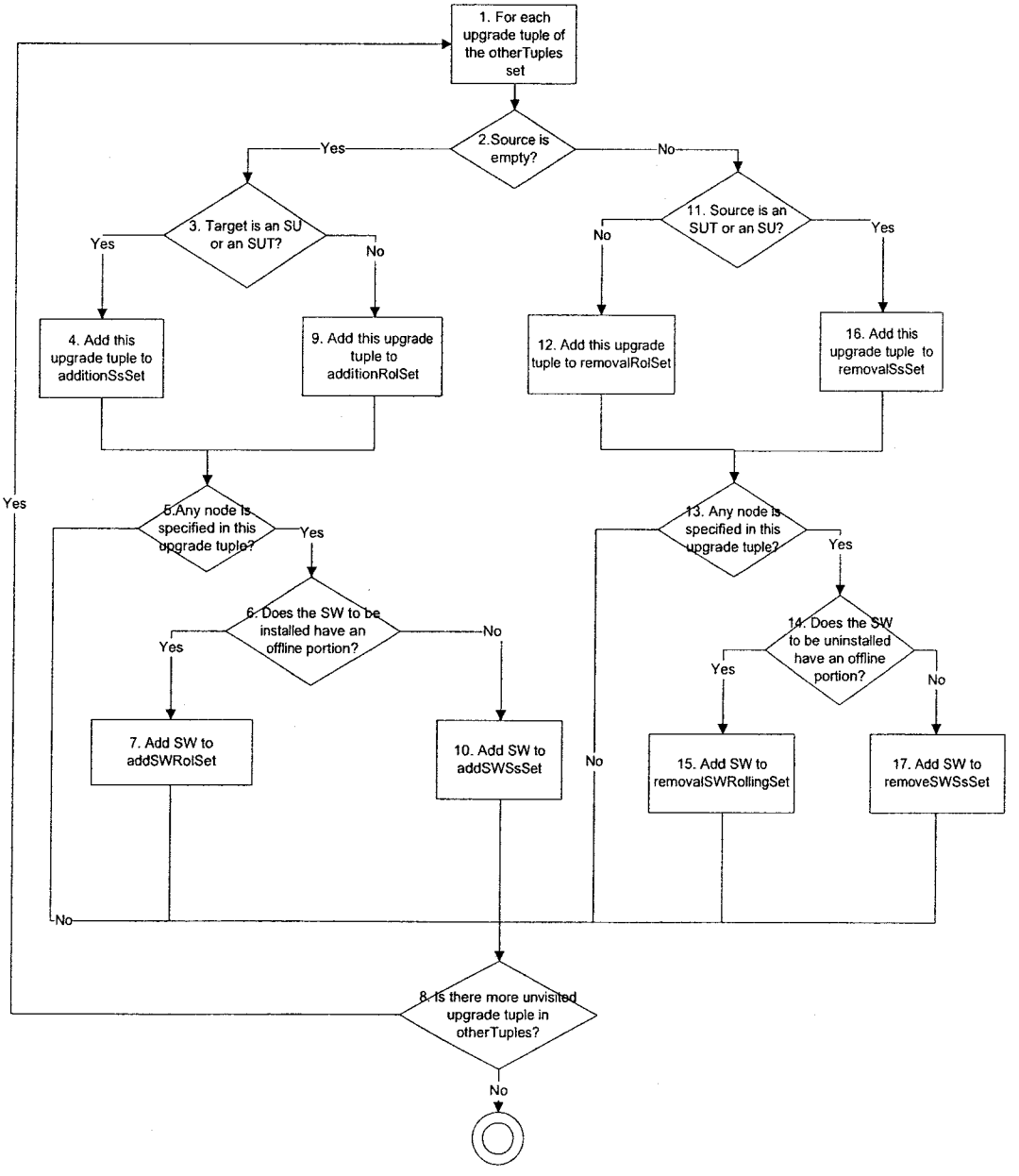
- Component addition and removal: Has the same situation like the component type addition and removal.
- Software installation and removal: If the software has offline portion in its installation and removal process, as discussed in Section 3.1 performing this operation will affect the whole node. If such software is being installed on or removed from many nodes at the same time, all other entities that are configured on these nodes will be affected and lose their functionality. Therefore, we should separate the installation and removal of the software with offline portion from the ones without an offline portion.

The algorithm that performs this separation based on the discussed criteria is shown in Flowchart 3.4.

As seen from the algorithm presented in Flowchart 3.4, the upgrade tuples of the Addition and Removal scenarios are separated into eight sets as follows:

- additionSsSet containing upgrade tuples to add entities other than components,
- additionRolSet a set of service groups and their related upgrade tuples to add components addSWRolSet a set of nodes and their related software having offline portion,
- addSWSsSet a set of nodes and their related software without any offline portion,
- removalRolSet a set of service groups and their related upgrade tuples to remove components,
- removalSsSet containing upgrade tuples to remove entities other than components,

- removalSWRolSet a set of nodes and their related software having offline portion and ,
- removeSWSsSet a set of nodes and their related software without any offline portion.



Flowchart 3.4. Analysis and Separation of Upgrade Tuples.

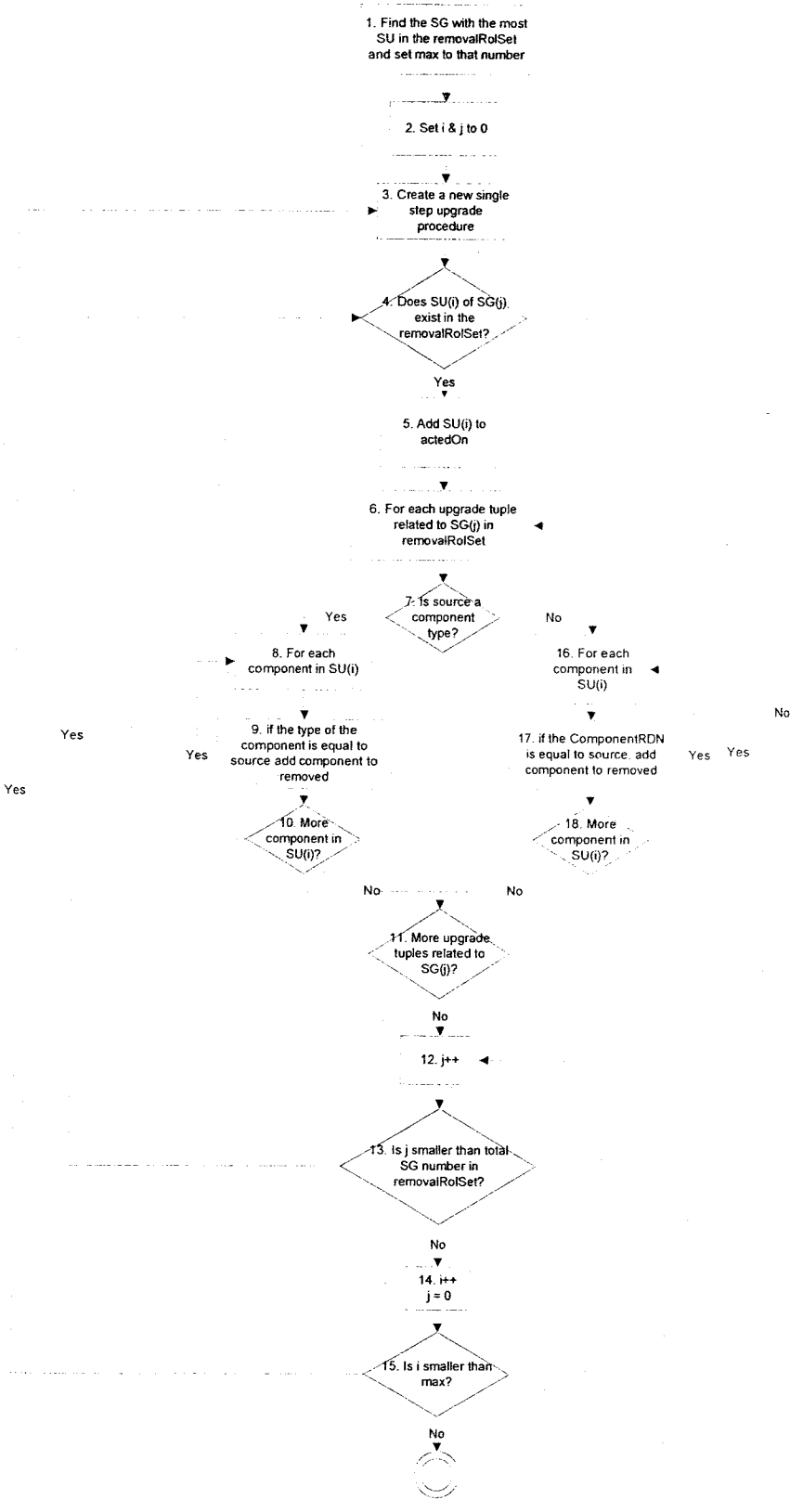
Later on each of these sets will be used for generating the appropriate upgrade procedures. In the following sections the generation of these upgrade procedures are discussed through flowcharts.

3.3.5.3.2 Removing Components

To preserve the availability while removing components from the configuration, the upgrade process is reduced to generating a series of single step upgrade procedures that imitates the rolling upgrade procedure; e.g. for a service group with five service units, five single step upgrade procedures are created and each of them targets one service unit. This decision was made due to the fact that rolling upgrade procedure template that is presented in the upgrade campaign schema [10], the only possibility is to change the type of existing entities, add and remove the related software bundle. Entities can only get added or removed through single step upgrade procedures.

The algorithm that processes the upgrade tuples of *removalRolSet*, the set of service groups and their related upgrade tuples, and generates the related single step procedures is shown in Flowchart 3.5.

As one may have noticed in the Flowchart 3.5, some optimizations are built into the algorithm to minimize the number of created upgrade procedures. Instead of creating one procedure for each service unit of each service group, the minimum number of upgrade procedures is created by targeting in each of them a service unit of each service group.



Flowchart 3.5. Removing Components.

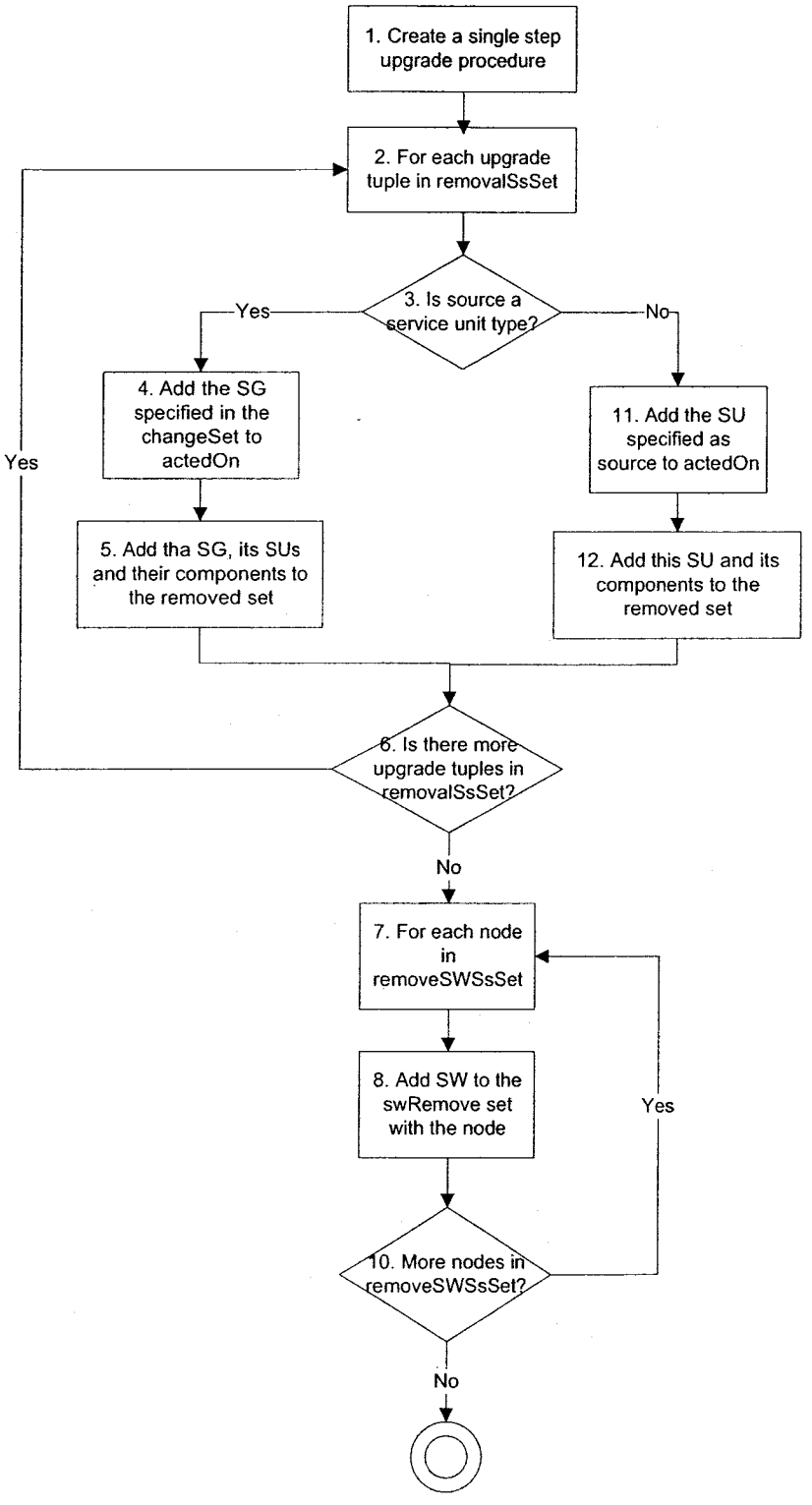
This way in each procedure more than one service unit is targeted and since they belong to different service groups, the availability will not be impacted. The minimum number of upgrade procedures that should be created is equal to the number of service units of the service group with the highest number of service units in the *removalRolSet*. This number is referred to as max in Flowchart 3.5.

3.3.5.3.3 Removing Entities other than Components

To remove entities other than components our optimization consists of minimizing the number of generated upgrade procedures. Since removing these entities and the software bundle does not expand the scope of upgrade to other entities, we can perform all the required operations through generating only one single step upgrade procedure.

The algorithm described by Flowchart 3.6 is used to generate a single step upgrade procedure used to remove entities other than components and software without offline portion from the configuration. This algorithm takes as input *removalSsSet* and *removeSWSsSet* sets that were created by the algorithm described in Section 3.3.5.3.1. It processes these upgrade tuples to generate a single step upgrade procedure that removes service units and service unit types and software without an offline portion from the configuration.

It is noted that the entities related to the software being removed in this algorithm can be components that are themselves being removed in rolling manner. However as discussed earlier, since their related software has no offline portion, it is put into the *removeSWSsSet* and therefore is removed in single step.



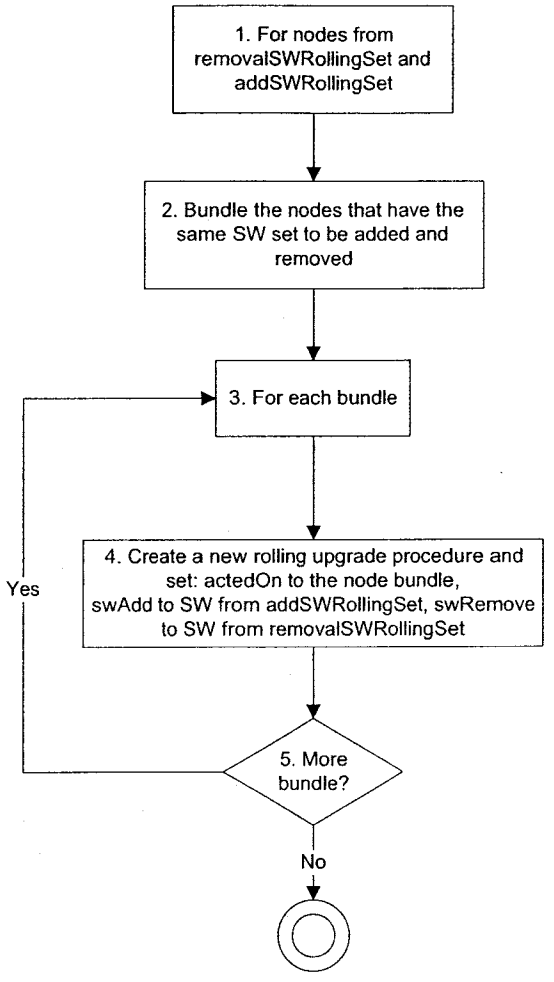
Flowchart 3.6. Removing Entities other than Components.

As illustrated in Flowchart 3.6, if a tuple is to remove the service unit type, since each service group has only one service unit type, in Step 5 the service group, its service units and the contained components will be added to the set of entities to be removed. But if the upgrade tuple is to remove the service unit, only that service unit and its contained components of the specified service group in the *ChangeSet* is added to removed set in Step 12.

3.3.5.3.4 Handling Software Containing Offline Operation

As we discussed earlier, to minimize the outage, we have separated the addition and removal of software with offline operation portion from the ones without offline portion. The software without an offline portion was removed by the algorithm described in Section 3.3.5.3.3. The algorithm that illustrated in Flowchart 3.7 generates rolling upgrade procedures to install and remove software that has an offline portion. It takes the *removalSWRollingSet* and *addSWRollingSet* as the input (see Section 3.3.5.3.1).

In *removalSWRolling* set, each node has a complete list of software with offline portion that should be removed from it. Similarly, in *addSWRolling* set each node has a list of software with offline portion that should be installed on it. As you noticed in Flowchart 3.7 for each node of the *removalSWRolling* and *addSWRolling* sets all the related software bundles, either to be removed or installed, are extracted. This categorization allows us to lock each node the minimum possible time to perform the operation on it instead of locking the node for each single software bundle to be installed or removed.



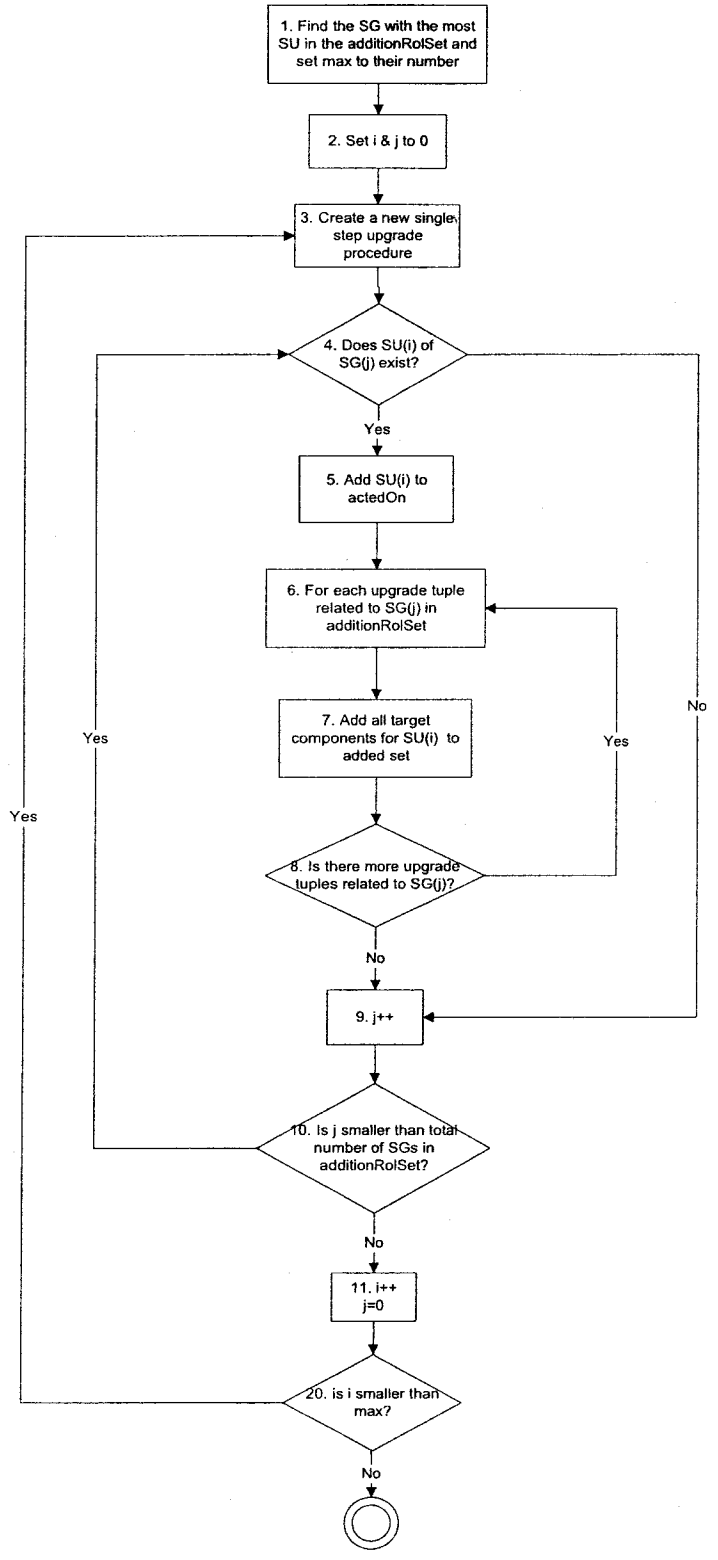
Flowchart 3.7. Upgrade Software Containing Offline Operation.

As another optimization for the generation process, this algorithm creates a minimum number of rolling upgrade procedures; by identifying those nodes that have identical sets of software to be installed and removed in Step 2 and bundling them together. Then, through the rest of the algorithm a rolling upgrade procedure is created for each bundle and the nodes, software to be installed and removed are added to the upgrade template as appropriate, until no bundle remains unvisited.

3.3.5.3.5 Adding Components

Similar to our approach in the Removal scenario, to break down the upgrade scope to smaller ones a series of single step upgrade procedures add components to the configuration. By doing so the rolling upgrade procedure is imitated. The algorithm that generates these procedures is illustrated in Flowchart 3.8. It gets the *additionRolSet* as the input (see Section 3.3.5.3.1). A single step upgrade procedure is generated then, for each service unit of a specified service group; e.g. if components should be added to a service group with five service units, five single step upgrade procedures will be generated respectively.

In Flowchart 3.8, the same optimization as for the algorithm that removes components (see Section 3.3.5.3.2) is built into the algorithm. Instead of creating one procedure for each service unit of each service group, the minimal upgrade procedures are created and within each of them a service unit of each service group is targeted. Therefore, in each procedure more than one service unit is targeted and since they belong to different service groups the availability will not interrupted. The minimum number of upgrade procedures that should be created is equal to the number of service units of the service group with the most number of service units in the *removalRolSet*. This number is set to max in the first Step of Flowchart 3.8.



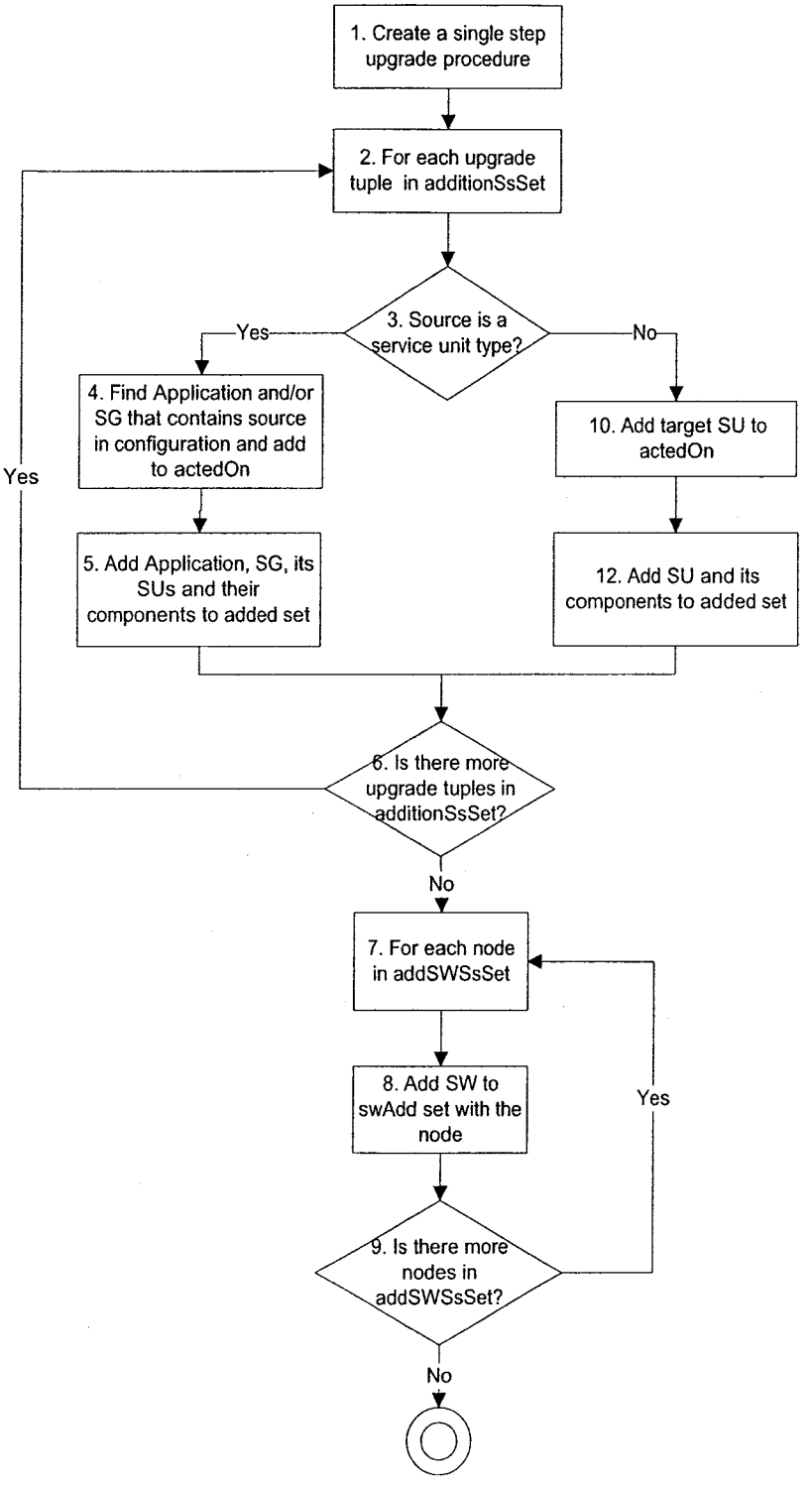
Flowchart 3.8. Adding Components.

3.3.5.3.5 Adding Entities other than Components

Similar to the Removal scenario, to minimize the outage, the addition of entities other than components is handled separately. For each node the software without an offline portion was also collected. Since the addition of these entities and the software bundles will not expand the upgrade scope to other entities, the upgrade scope does not need to be broken into smaller ones. Therefore a single step upgrade procedure is generated by the algorithm described in Flowchart 3.9. It gets *additionSsSet* and *addSWSsSet* as the input (see Section 3.3.5.3.1). Then by processing the upgrade tuples of these two sets, this algorithm creates a single step upgrade procedure to add service units and service unit types and software without an offline portion to the configuration.

As it is shown in the Flowchart 3.9, since each service group has a unique service unit type, in Step 5 the application, service group, its service units and their contained components are put into the added set. On the other hand if the upgrade tuple adds a service unit, in Step 12 only that service unit and its contained components are added to the service group specified in its *ChangeSet*.

It is noted that the software being added by this algorithm are not necessarily related to upgrade tuples of *additionSsSet*. As it is discussed earlier, the software can belong to any upgrade tuple but it has no offline portion and therefore can be installed in single step.



Flowchart 3.9. Adding Entities other than Components.

3.3.5.4 Changing Type Scenario

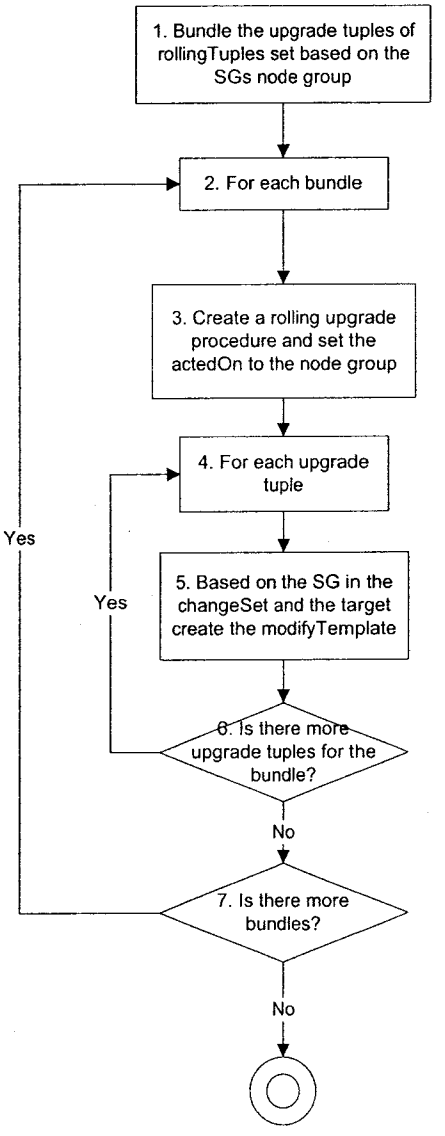
As it is proposed in the SMF specification [7] changing type of the existing entities should be done through rolling upgrade procedures. In Section 3.1 we have assumed that the scope of upgrade for each step within a rolling upgrade procedure is a node. For that a rolling upgrade procedure targets one single node of the given service group's node group at a time, therefore the upgrade scope is limited to the entities on that node and as we have assumed, due to the redundancy within the system we will not lose the availability. The algorithm to generate upgrade procedures for this scenario is illustrated by Flowchart 3.10. This algorithm takes *rollingTuples* (see Section 3.3.5.2) as the input.

To optimize the number of generated rolling upgrade procedures, this algorithm tries to lock each impacted node the minimum number of times possible. As it is shown in the Flowchart 3.10 this algorithm starts by identifying the service groups (specified in the upgrade tuples of the *rollingTuples* set) that have nodes in common in their node groups and create bundles out of them. Each bundle that is created in the first Step belongs to either of the following categories:

- a bundle of service groups and a set of nodes that all these service groups are deployed on all of those nodes, or
- a bundle of a service group and its nodes that are not in common with any other service group specified in the upgrade tuples of *rollingTuples*.

It is noted that a service group can be in both categories of bundles; while each node can only belong to one of them. For instance, let us consider two service groups A and B that are specified in upgrade tuples of the *rollingTuples* set. Let us assume the node group of

service group A contains nodes N1, N2, N3 and N4 and the service group B is deployed on nodes N3, N4 and N5. If the two service groups are provided to this algorithm three bundles will be created out of them: One contains both of them and nodes N3 and N4 that are in common, the second contains service group A with its remaining nodes N1 and N2 and the third bundle contains service group B with the node N5.



Flowchart 3.10. Changing Type Scenario.

However we need to make sure that these procedures, when given to the SMF, will not be executed simultaneously; as they may take out two nodes from the same service group and interrupt the availability.

3.4 Discussion

To lay the foundation for an automatic upgrade campaign generation the approach proposed in this thesis handles only the upgrading component types and service unit types, besides the removal and the additions of components and service units.

Since this upgrade campaign generation approach does not target the service group upgrade, we do not deal with the upgrading of redundancy models. Moreover, in this work, dependencies among component service instances and within service instances are not considered. However, we do recognize and handle dependencies between a service unit type and its related component types.

As discussed in Section 3.1, there are a certain number of assumptions that influenced our campaign generation approach. For instance, instead of using an ETF we assumed that the user has already generated a configuration based on an ETF and provides us with the new configuration. This assumption has been made to let us handle only AMF types instead of their ETF counterparts. We also assumed that whenever the user wants to add components, the service units these components are being added to them are part of the current configuration. Similarly when the user states a service unit to be removed, the script to remove its components will be generated consequently.

In the Addition Scenario, we eliminate the case in which the user can add components by specifying their type. For that if the user wants to add two components of the same type to the service units of a service group two separate upgrade tuples should be specified; each of which contains the component RDN of those components.

There are also other criteria for completeness and consistency check as we mentioned earlier. For example the number of component instances of a specific type should not go beyond a defined threshold. The algorithm could check for that number and if the user tries to add components more than the specified number, it interrupts the campaign generation process.

Chapter 4 - The Upgrade Campaign Generation Tool

Based on the algorithms introduced in the previous chapter, we have developed a prototype tool as a proof of concept for automatic upgrade campaign generation. The prototype tool was developed in Java [18] as an Eclipse [19] plug-in. In this chapter we first describe the prototype tool and then demonstrate the upgrade campaign generation following different scenarios for a given AMF configuration.

4.1 The Prototype Tool Description

Figure 4.1 shows the overall data flow in the prototype tool. The user interacts with the tool through a Graphical User Interface (GUI). Using this GUI the user provides all the aforementioned Input Data (as discussed in Section 3.3.2). First the user specifies the current configuration. The contents of this configuration are made available graphically to the user in order to create the upgrade tuples of the Upgrade Tuples Set. The user also can provide the additional configuration through this GUI when necessary. If any error occurred during the upgrade campaign generation, the user is informed using appropriate error messages. Necessary features for saving the log file generated by the validator and the upgrade campaign specification are provided.

The object model is based on the AMF information model described in the AMF specification [6] and the upgrade campaign object model derived from the upgrade campaign schema [10].

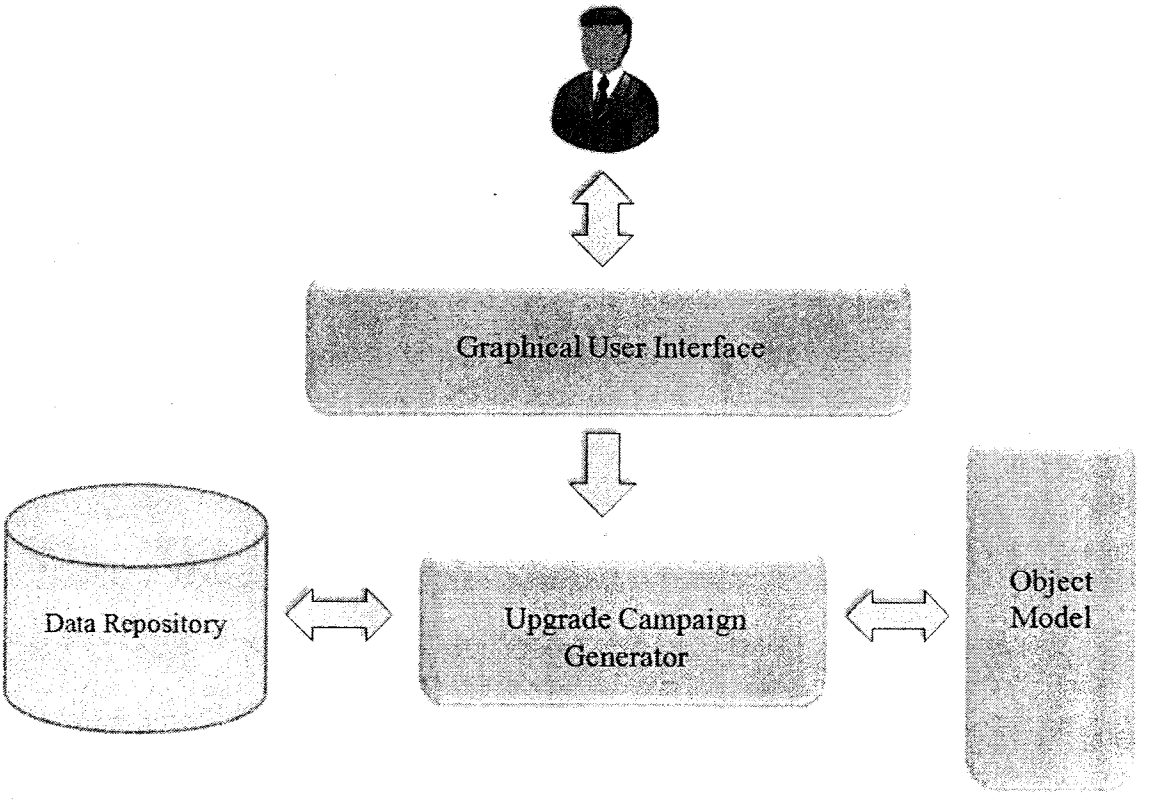


Figure 4.1. Dataflow Diagram of the Prototype Tool.

The upgrade campaign generator module encompasses the algorithms that have been described in details in Chapter 3 and its main parts are as follows:

- The algorithm to calculate necessary additional upgrade tuples and to check for consistency,
- The algorithm to manipulate a copy of the current configuration to create an instance of the target configuration for validation purpose and,

- The upgrade campaign generation algorithms that generate the upgrade procedures as appropriate.

4.2 The Prototype Tool Graphical User Interface

The graphical user interface of the prototype tool consists of several pages. The visibility and contents of some of these pages are static while for the rest they are dynamic and depend on the earlier choices that the user has made. Figure 4.2 through Figure 4.9 show snapshots of different pages of the prototype tool GUI.

The upgrade campaign generation process starts with the page illustrated in Figure 4.2. In which the user provides the current configuration to upgrade.

The user is provided with the second page shown in Figure 4.3. As we can see, there the user specifies what kind of upgrade scenario he wants to proceed with. He also indicates the way he wants to specify the entities, i.e. either by their RDN or by their type.

The decisions the user makes in this page influence the following pages and their contents. If the user selects the *changing type* option, the page shown in Figure 4.4 is provided next. Note that the contents of this page depend on the choice the user has already made in the previous page. If the *by type* option is chosen, the first table is populated with all service unit types and component types of the service group shown in the first combo box. Otherwise, the first table is populated with the list of service units RDN and their contained components RDN of the specified service group in the first combo box.

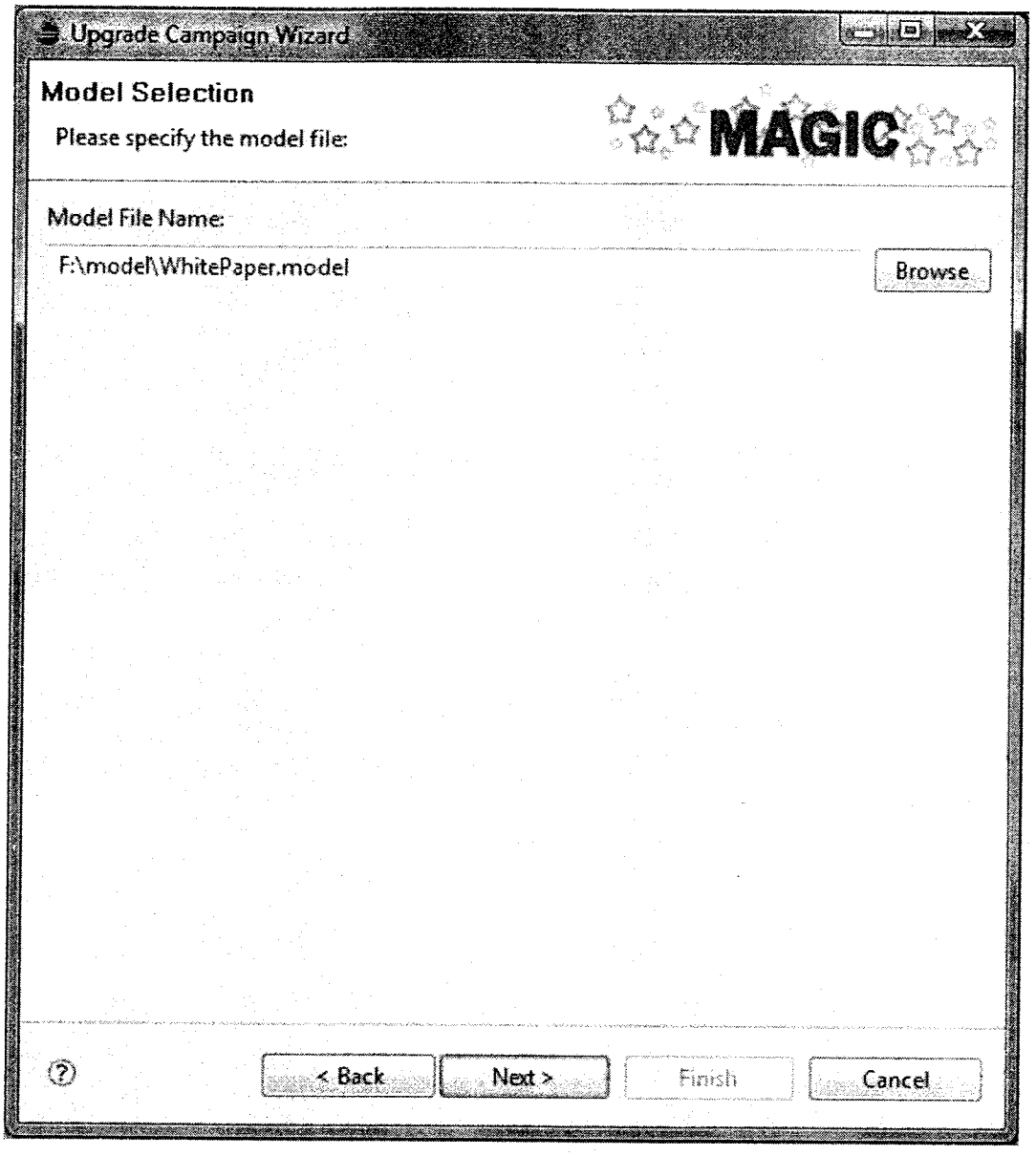


Figure 4.2. The Current Configuration Selection Page.

The second table shows the list of nodes of the chosen service group's node group where the user can state the nodes he wants to upgrade the software on. Finally, the second combo box shows a list of available entity types in the current configuration to be chosen as the Target entity.

If the user selects the option of *adding entities* in the upgrade intent page (Figure 4.3), the page illustrated in Figure 4.5 is provided for specifying the additional configuration.

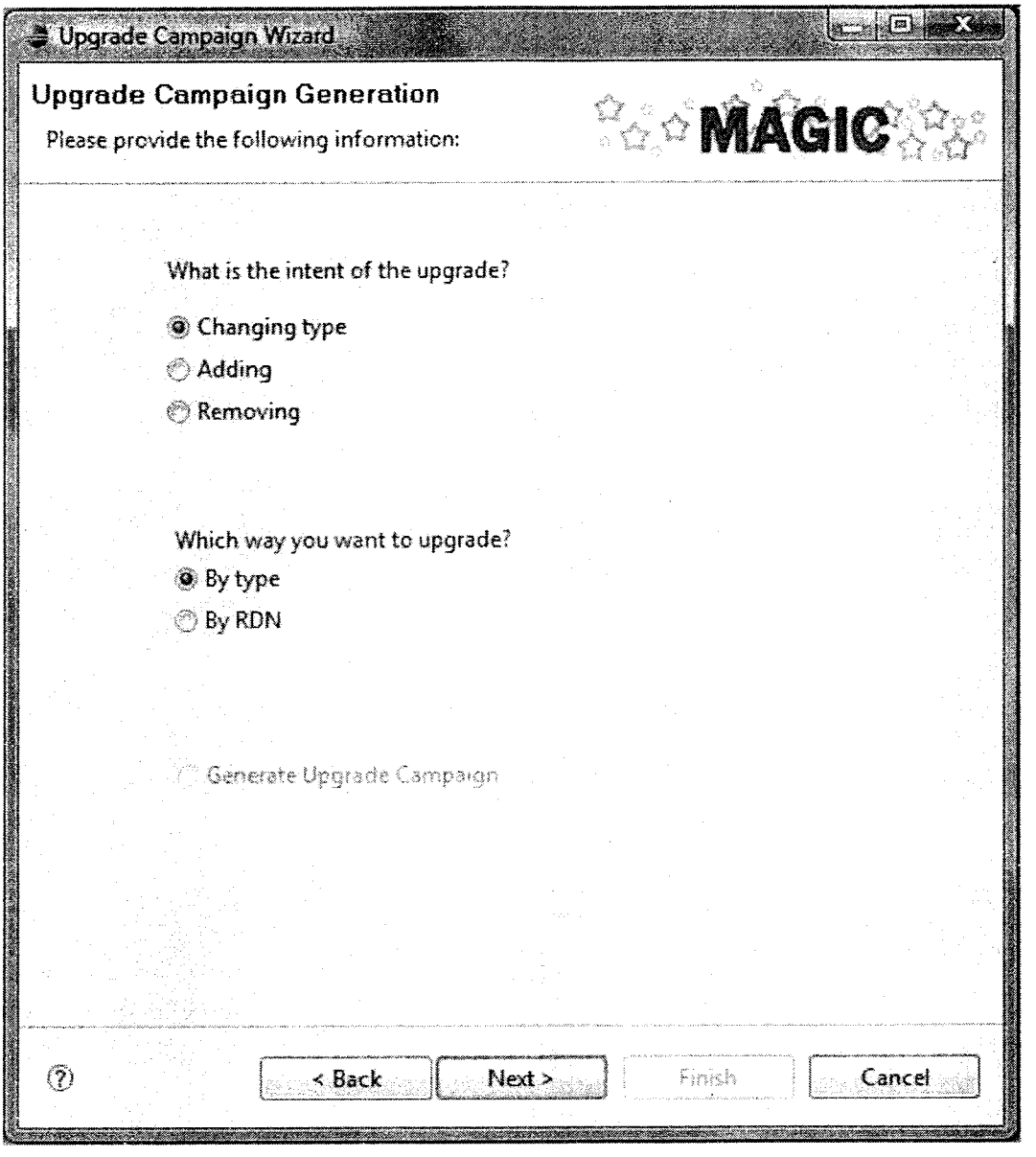


Figure 4.3. The Upgrade Intent Page.

After providing the additional configuration the user proceeds to the page shown in Figure 4.6. In the first combo box, the user specifies the service group in which he wants to perform his operations. As mentioned previously this service group may be a totally

new service group or a service group from the current configuration. As for the upgrade by type page (Figure 4.4) the first table is populated by entity types or by their RDNs depending on the user selection in the upgrade intent page (Figure 4.3). The second table contains the list of nodes from the chosen service group's node group, on which the user wants to install the software.

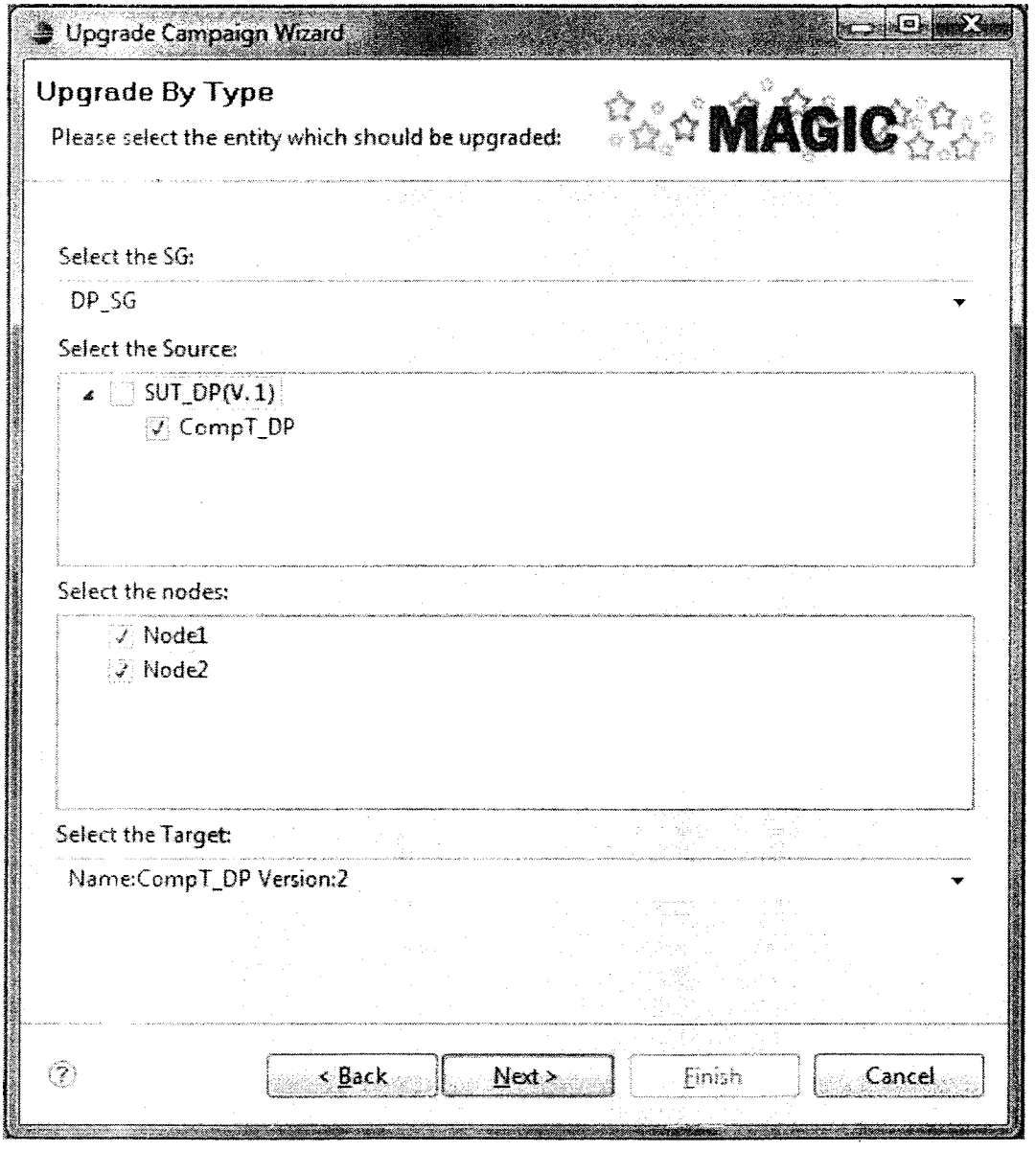


Figure 4.4. The Page for the Upgrade by Type Scenario.

If the user selects the *removing* of entities option in the upgrade intent page (Figure 4.3), he is provided the page shown in Figure 4.7. In this figure, in the first combo box he should specify the service group he wants to remove entities from. The first table contains the entities of the chosen service group. These entities are displayed either by their types or by their RDNs depending on the user selection in the upgrade intent page.

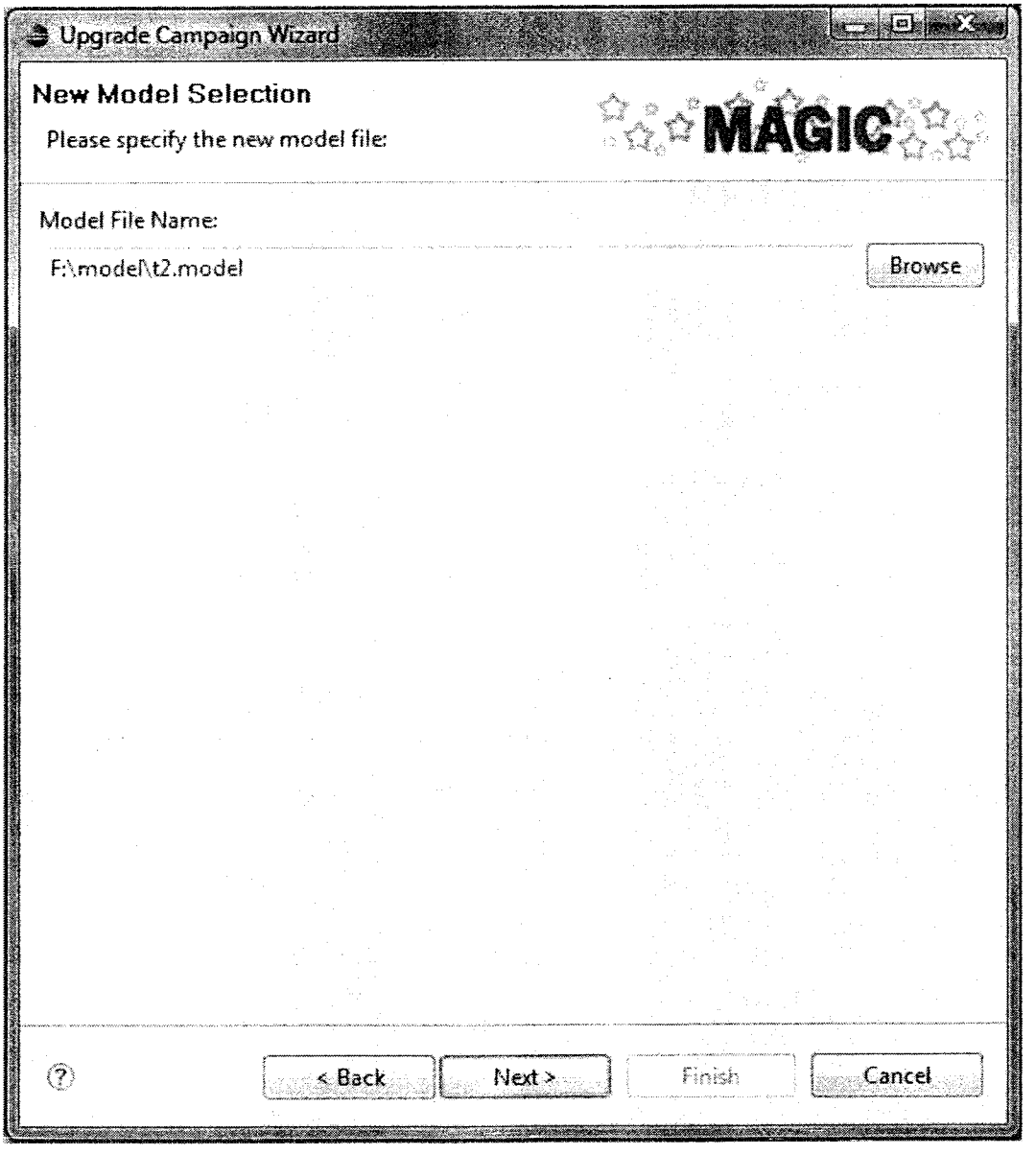


Figure 4.5. The Additional Configuration Selection Page.

As for the previous cases, the second table contains the list of nodes of the chosen service group's node group. The user selects those nodes he wants to remove the related software from.

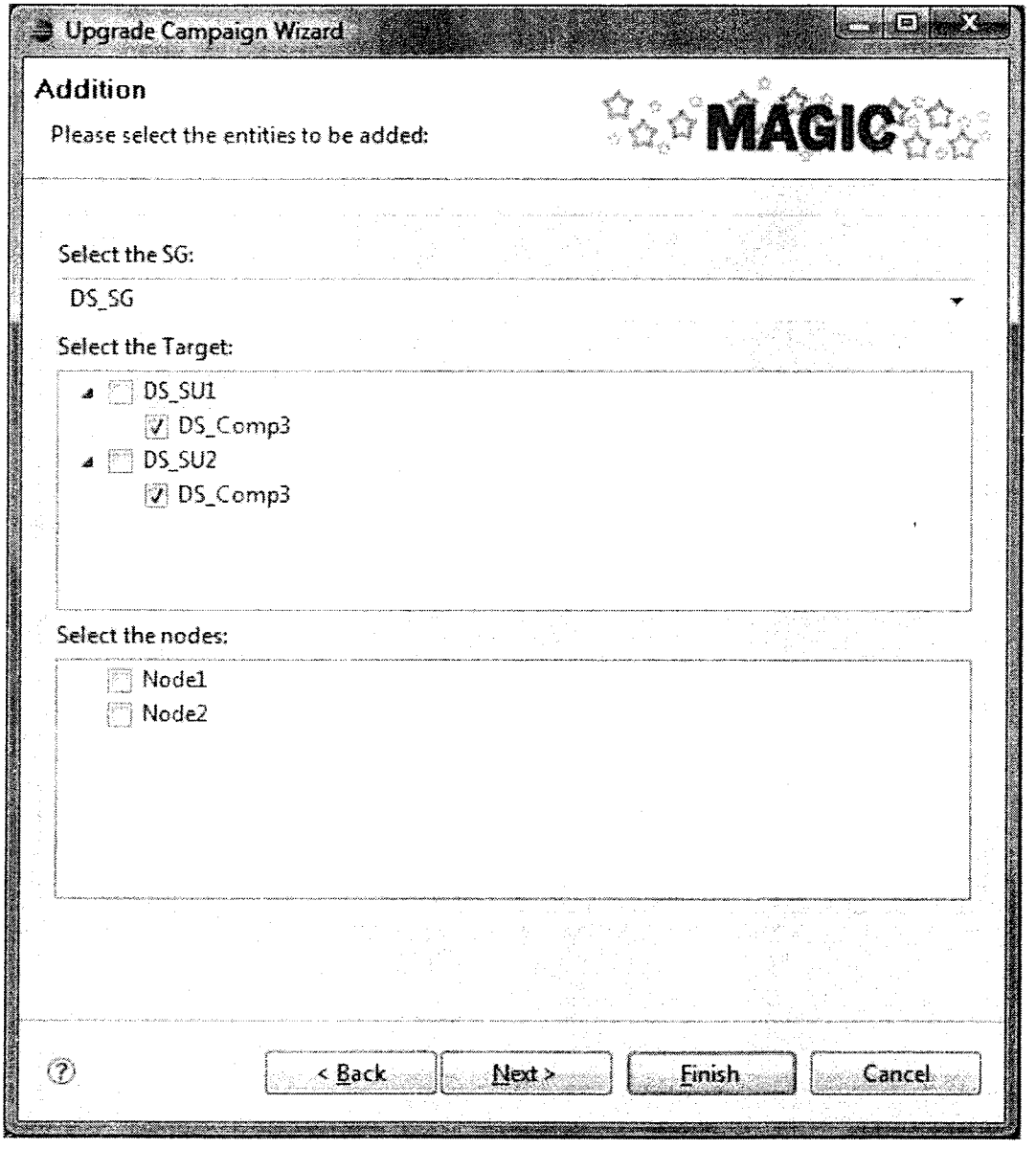


Figure 4.6. The Page for the Addition Scenario.

After specifying an upgrade tuple through the upgrade by type page or adding or removing pages, the user is brought back to the upgrade intent page (Figure 4.3) where he

can choose to continue on specifying other upgrade tuples through the same process or stop creating new upgrade tuples and start the upgrade campaign generation by selecting the *Generate Upgrade Campaign* option.

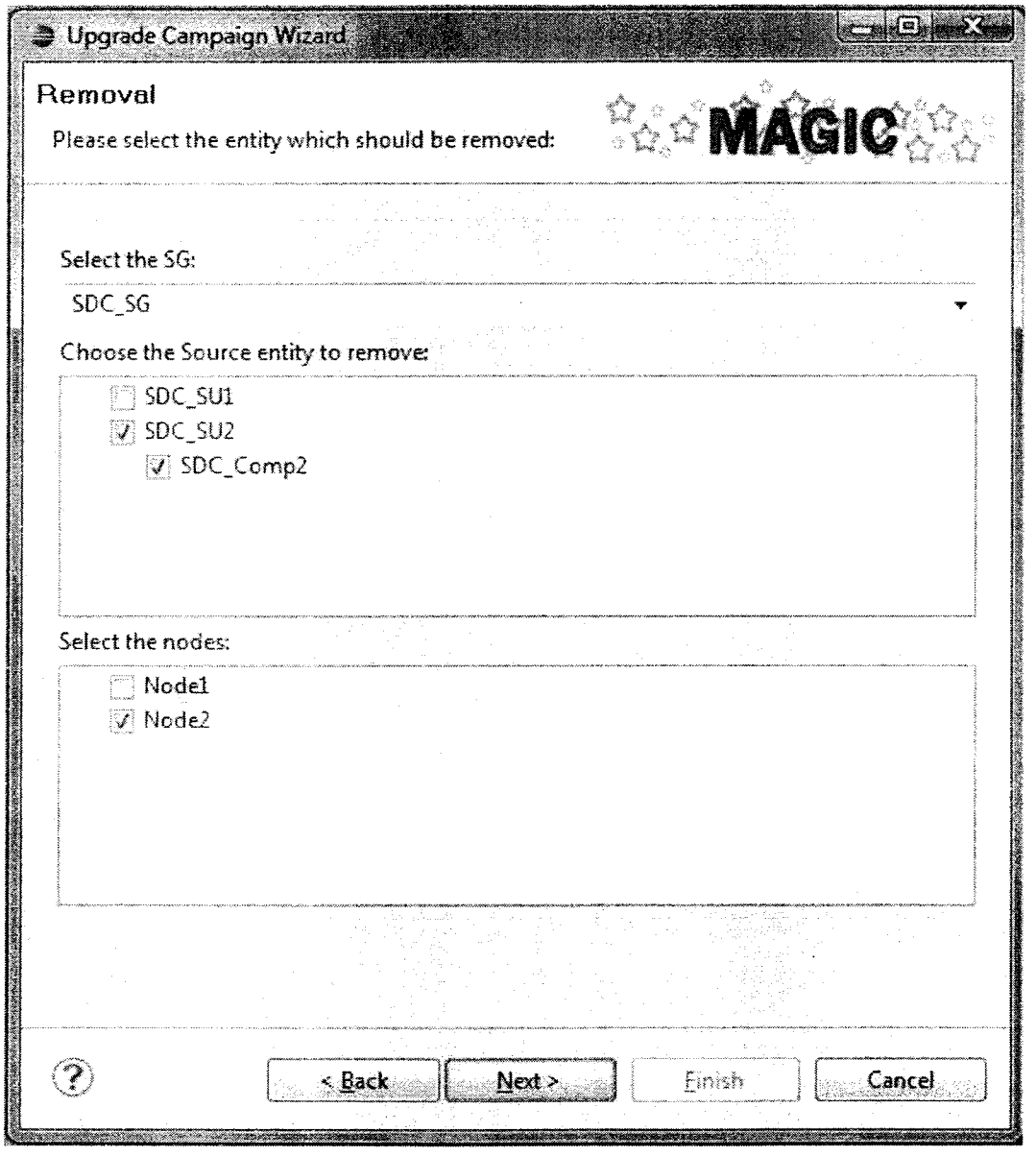


Figure 4.7. The Page for the Removal Scenario.

When the user is done with providing the upgrade tuples and selects the Generate Upgrade Campaign option, the upgrade tuples will be examined for their completeness

and consistency. If there is any error the campaign generation process will be rejected and the user will be notified through appropriate error messages. If not, the target configuration which is created as we have discussed in Section 3.3.4, will be fed to the validator. At this point if the target configuration is not valid, again the campaign generation process is interrupted and the user will be referred to the generated log file. Otherwise an upgrade campaign specification will be generated and saved as an XML file.

4.3 Case Study

In this section we demonstrate the upgrade campaign generation with the Portable Highly Available Sensors (PHASE) application. For that purpose we first describe an AMF configuration for PHASE and then generate upgrade campaign specifications for various upgrade scenarios.

4.3.1 PHASE Application

PHASE application consists of collecting, processing and distributing sensor measurement data [20]. As illustrated in Figure 4.8, it receives the measurement data from an external sensor, processes it and sends the processed data to the external user.

In PHASE application, the Sensor Data Collector (SDC) part receives and groups the data transmitted by the external sensor. It then forwards the, now grouped, data to the Data Processor (DP). The data processing is actually performed by two kinds of data

processors; data processors with history and data processors without history. Finally, the Data Sender (DS) part provides the processed data to the user upon request.

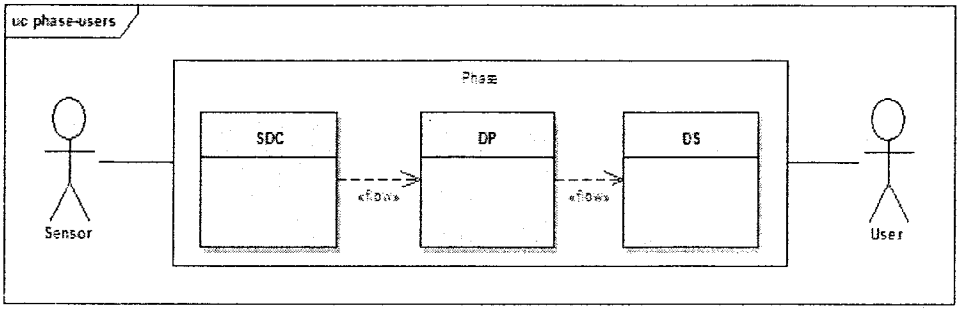


Figure 4.8. PHASE Ecosystem (taken from [21]).

A simple configuration of the application is presented in Figure 4.9. The PAHSE-APP application of Figure 4.9 has three service groups, one for each part, with different redundancy models to serve their respective service instances; the SDC-SG has "No Redundancy", the DP-SG has 2N and the DS-SG has N+M redundancy models. In addition to that, each service group has two service units that are built from the version v1 of their appropriate service unit types. Finally each service unit contains one component built from the version v1 of the suitable component type.

On the other hand, the SDC-SG service group has SDC-SI1 and SDC-SI2 to serve. The DP-SG service group has DP-SI1 and DP-SI2 to serve and protect. While the DS-SG service group has only one service instance, DS-SI1, to take care of. Each of the service instances contains a component service instance of the type to be served by the appropriate component of their assigned service group.

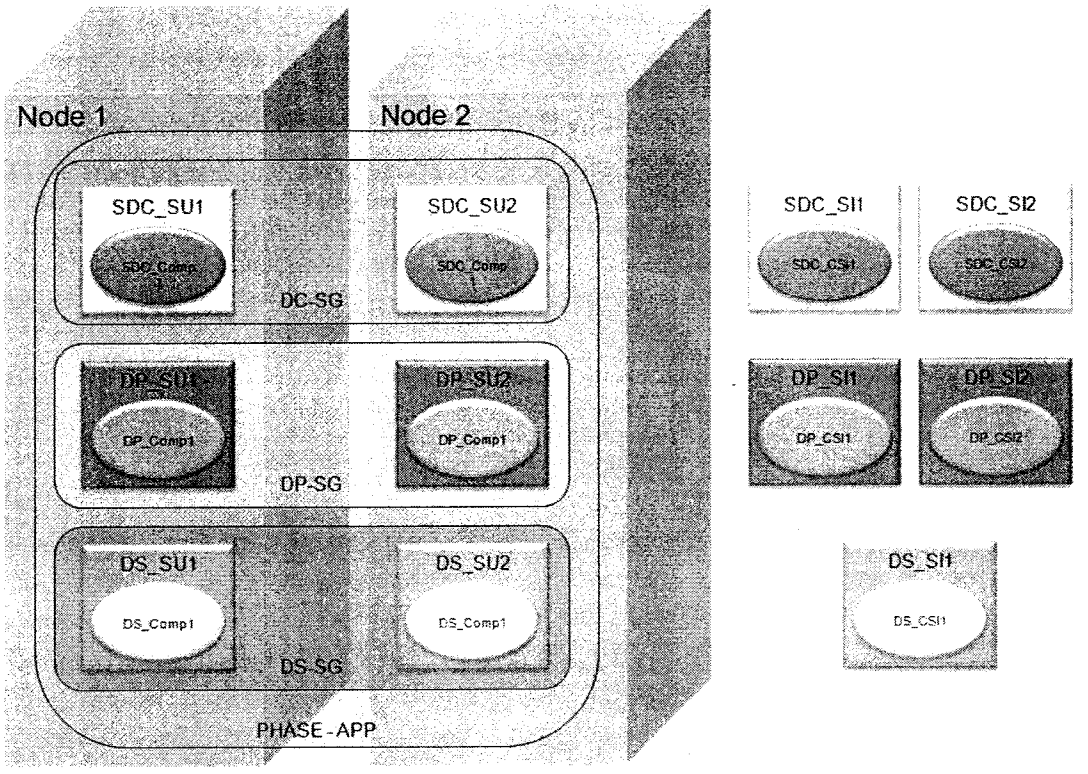


Figure 4.9. A Simple Configuration of PHASE (take from [21]).

4.3.2 Generating Upgrade Campaign Procedures for PHASE-APP

In what follows, using the prototype tool, we will generate upgrade procedures for PHASE-APP according to different upgrade scenarios.

4.3.2.1 Changing Type Scenario

Assume that the version v2 of component type CompT_DP has become available and the PAHSE-APP configuration should be upgraded to it. For that, the user provides the following upgrade tuple:

(CompT_DP(V.1), CompT_DP (V.2), DP_SG, {Node1, Node2}).

The upgrade campaign specification generated for this input is illustrated in Figure 4.10.

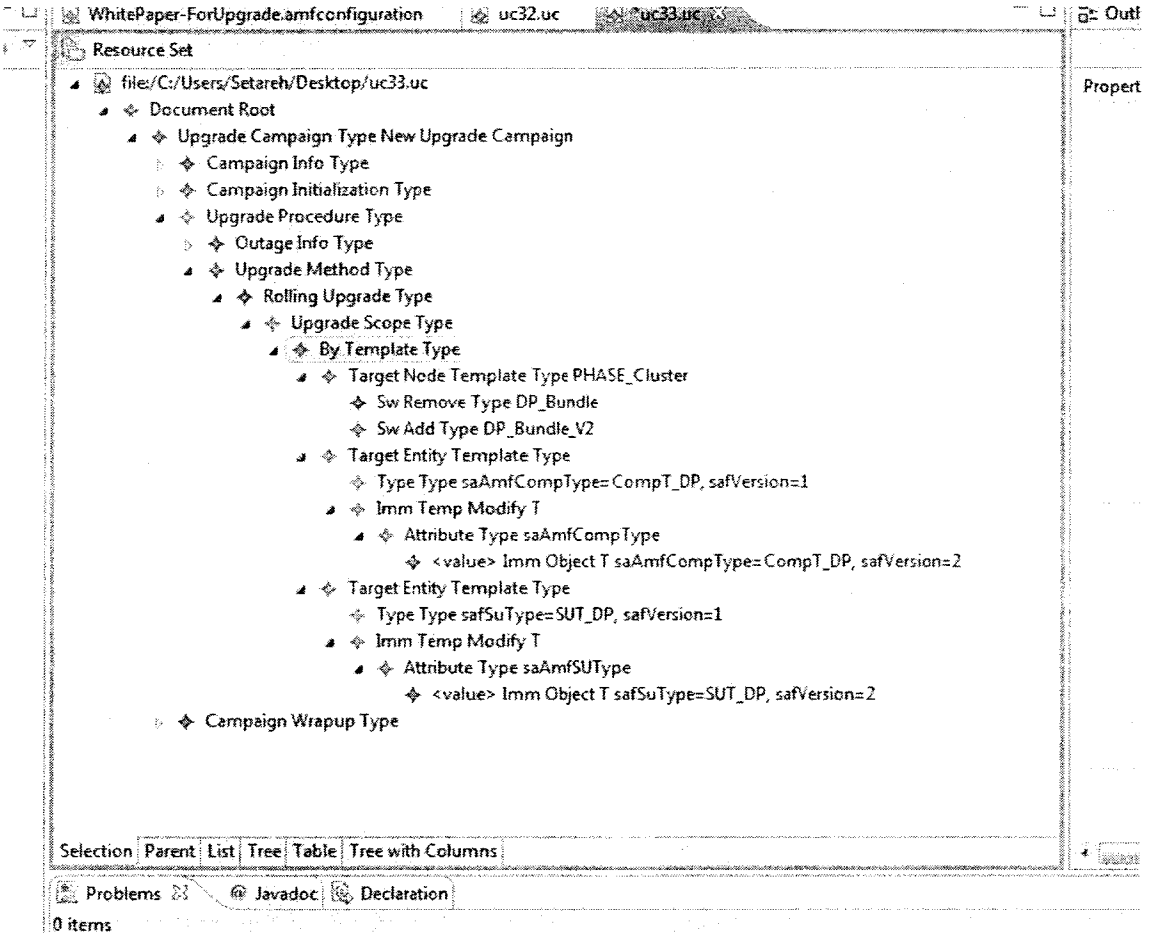


Figure 4.10. Upgrade Campaign XML file for Changing Type Scenario.

As you will notice, a rolling upgrade procedure is generated to upgrade the component type of component DP from v1 to v2. However, since the current service unit type SUT_DP of version v1 does not support component type CompT_DP of version v2 the

service unit type had to be upgraded as well. The latter upgrade tuple is added by our algorithm that checks upgrade tuples for completeness and consistency. The resulting upgrade template contains another portion for upgrading service unit type SUT_DP from version v1 to v2. Since the user has specified nodes Node1 and Node2, the Target Node Template portion of the Upgrade Procedure contains the script to replace the old software with the new one.

4.3.2.2 Adding Entities Scenario

For this scenario assume that the user wants to add a new component to the service units of service group DS_SG. The user specifies the following upgrade tuple:

(ε, DS_Comp3, DS_SG, ε).

The generated upgrade campaign XML file is shown in Figure 4.11. The DS_SG has two service units. Therefore, as discussed in Chapter 3, to minimize the service outage two single step upgrade procedures are generated where each of them adds a new component to a service unit of service group DS_SG. Note that each of the components is added by specifying all of its attributes. As the nodes already contain components of this type there is no need to install the software on them and therefore the user did not specify any node in the upgrade tuple.

Figure 4.11. Upgrade Campaign XML file for the Adding Entities Scenario.

4.3.2.3 Removing Entities Scenario

Here suppose the user wants to remove the service unit SDC_SU2 and its contained component SDC_Comp2 from the SDC_SG along with its related software from the Node2. For that he specifies the following upgrade tuple:

(SDC_SU2, ε, SDC_SG, {Node2}).

The generated upgrade campaign XML file is shown in Figure 4.12. As you can see two upgrade procedures are generated; a single step upgrade procedure to remove the service unit SDC_SU2 and component SDC_Comp2 from the PHASE-APP, and a rolling

upgrade procedure to remove the related software from Node2. Since the related software contains offline portion, therefore as discussed in Chapter 3, it should be removed in a rolling manner.

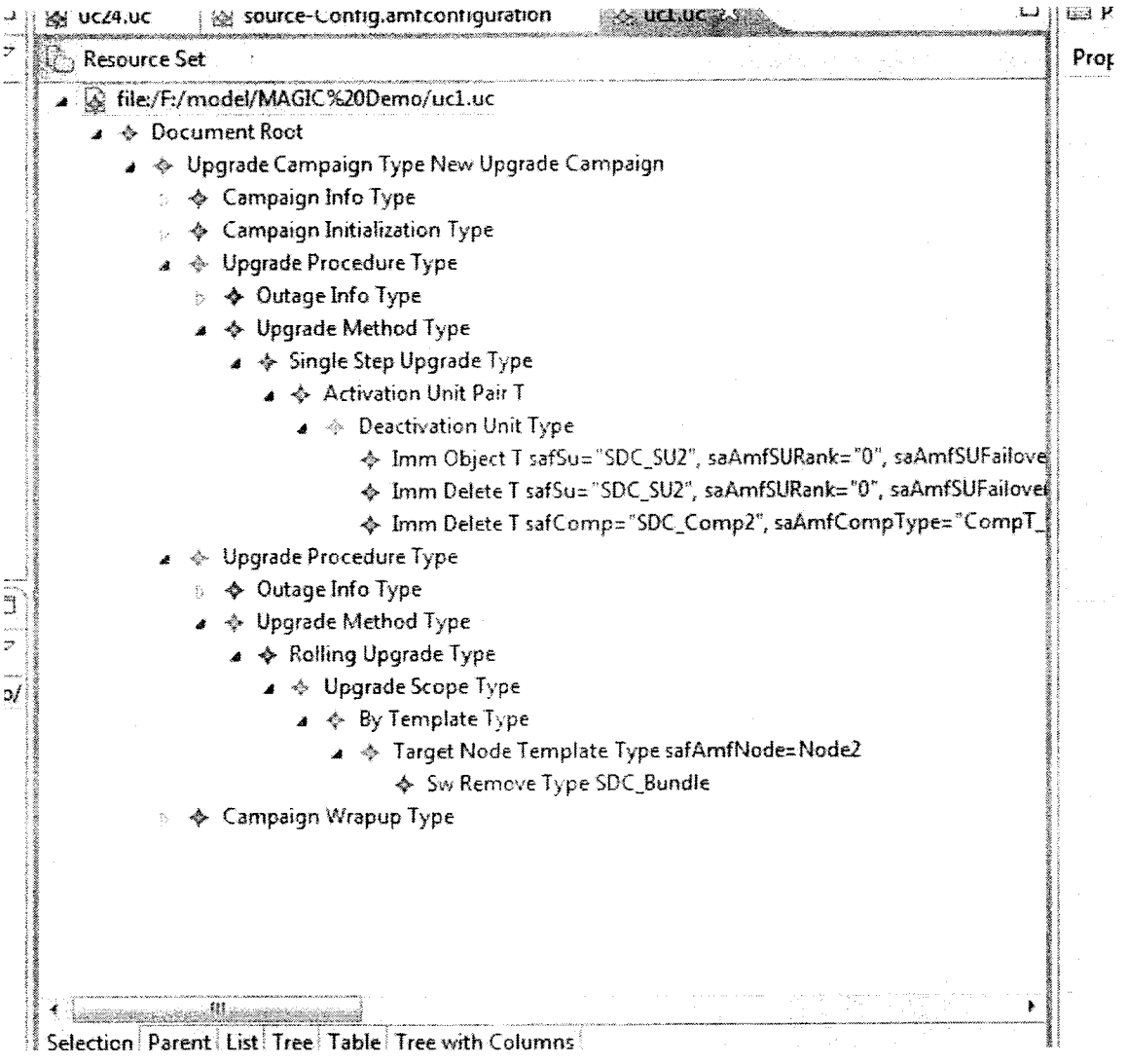


Figure 4.12. Upgrade Campaign XML file for Removing Entities Scenario.

4.3.2.4 Combining Three Scenarios

Let us now consider together all of the three scenarios discussed in previous sections. For that the user provides the following upgrade tuples:

(CompT_DP(V.1), CompT_DP (V.2), DP_SG, {Node1, Node2}),

(SDC_SU2, ε, SDC_SG, {Node2}) and,

(ε, DS_Comp3, DS_SG, ε).

The generated upgrade campaign XML file is illustrated in Figure 4.13.

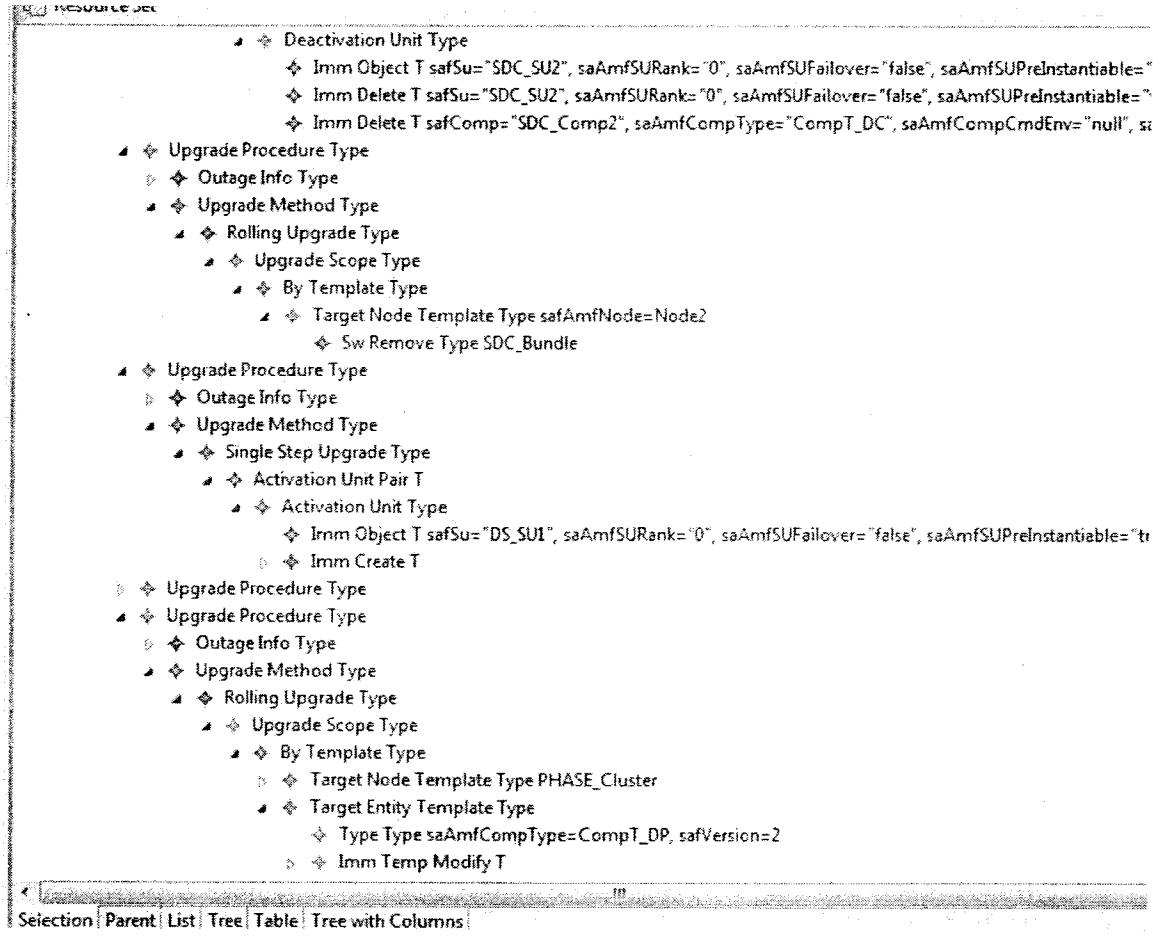


Figure 4.13. Upgrade Campaign XML file for Combining Three Scenarios.

As Figure 4.13 shows, five upgrade procedures are generated; three single step and two rolling upgrade procedures. Two of the single step upgrade procedures add a new component to service units of the DS_SG. While the third single step upgrade procedure removes the SDC_SU2 from the SDC_SG. As we have discussed it in Section 4.3.2.3 the first rolling upgrade procedure that is generated will remove the software bundle related

to SDC_SU2 from Node2. While the second rolling upgrade procedure upgrades the component types and service unit types of DP_SG from the version v1 to the version v2 of their respective types.

4.4 Conclusion

As it was seen through this chapter necessary data is collected through different pages of the GUI to be provided to the upgrade campaign generator algorithms. The creation and control flow of the GUI pages were done through the wizards of the Window Builder [22]. However using Window Builder wizard facilitates our work, from the coding perspective, using SWT [23] APIs, it uses lots of code that we do not have control of. So the debugging of this portion and connecting it to the campaign generation algorithms was tricky and took the most time and effort.

Using this tool for generating upgrade campaigns will relieve the user from the burden of exploring the source configuration and considering all of the related dependencies and making the right decisions while generating such upgrade campaigns manually. If the user has the basic knowledge about the nature of upgrade tuples and our upgrade scenarios, using this tool is straight forward. Since the appropriate messages will guide him from the beginning of campaign generation process toward the end.

The size of the generated upgrade campaign file may vary depending on the complexity of source configuration and the upgrade the user wants to perform that affects the number of generated upgrade procedures and their contents. But as an example the fourth upgrade campaign that we generated for the PHASE-APP in this work is almost 13 KBs.

The upgrade campaign that was generated when the upgrade tuples where given all together was exactly the combination of their respective upgrade campaigns when each of them was given separately. In that case the generated upgrade campaign will mainly change when there are overlapping nodes in addition and removal scenarios for software installation and removal. For that as we discussed all the software related to a single node is collected and the node will be locked the minimum times possible. But the tool does not perform such optimization among the upgrade campaigns of the Changing Type scenario and the other two.

In the Changing Type scenario we only upgrade the types of entities. However there can be the case that due to the upgrade attributes other than type are also changing. This latter case is missing from the ones we handle in our scenario.

Finally, as one could notice that we have fixed ways to generate the upgrade campaigns that do not consider other possibilities that will lead to generating all potential upgrade campaigns for a specific given input. Instead, for each given upgrade tuple according to the upgrade scenario it matches, certain upgrade procedures are created; e.g. for an upgrade tuple related to the changing type scenario always rolling upgrade procedures will be created. As a result, an upgrade campaign that is generated according to a specific upgrade tuple will have no variations; something that should be considered for the analysis purposes.

Chapter 5 - Conclusion

In this chapter we will conclude our work. We will again state the contributions we have made to the domain of SMF upgrade campaign generation for a highly available system. We will then list possible directions for expanding this work in the future. Finally, we give our closing remarks for this thesis.

5.1 Research Contributions

In this thesis, we presented our approach for automatic generation of an upgrade campaign for an AMF system from a set of input data provided by the user. The input data set consists of the current configuration of the system under consideration, the set of modifications to be performed and optionally an additional configuration if there is any entity to be added to the current configuration. We check the modifications to be performed for completeness and consistency. Before proceeding with the generation of the upgrade procedure, we make sure that the target configuration is valid with respect to the standard AMF specification.

The upgrade campaign generation consists of a series of algorithms that handle three upgrade scenarios and their combinations: changing types of current entities, removing entities, or adding new entities to the configuration. Our main contribution in this part was to generate an upgrade campaign with the minimum service outage possible. For that we identified the causes of the service outage in each upgrade scenario and then built our

solutions as a set of algorithms that generate upgrade campaign procedures for that scenario. We have separated the addition and removal of the components from the other entities. Since for addition and removal of the components their enclosing entities will be affected as well, for that a series of single step upgrade procedures are generated that imitates the rolling upgrade procedure. However for addition and removal of the rest of the entities a single step upgrade procedure is generated. First we have separated the installation and removal of software from the related entities. Later we have also separated the installation and removal of software with an offline potion from the one that does not have any offline portion. For the former case rolling upgrade procedures are generated and for the latter one single step upgrade procedures. Finally, whenever appropriate, in our algorithms we have collected the set of modifications to the software of each node and generated the upgrade procedures that lock each node the minimum possible time.

A prototype tool has also been developed as an Eclipse plug-in that implements all the aforementioned algorithms of the automatic upgrade campaign generator. The tool takes the input data set from the user, processes it and generates upgrade campaign specification XML file. To our knowledge this is the first tool which is capable of fulfilling this purpose.

5.2 Possible Directions for the Future Research

In the current approach, we have limited the extent of the upgrade to the redundant entities of the configuration, i.e., component types and service unit types. However, this work can be extended to eventually cover the upgrade of an AMF configuration with all

of its entities, types, redundancy models and services. Obviously the larger the upgrade, the more dependencies among entities will come to the picture. Automatic generation of upgrade campaign specifications with respect to such dependencies and the upgrade of non-redundant entities (e.g. service groups and applications) becomes a challenging issue to handle.

As we have discussed also in Chapter 3, the ultimate goal for an automatic upgrade campaign generation is an algorithm that can calculate the difference between two given configurations and generate upgrade procedures to be performed to move the system from its current configuration to the target one. To firmly prove the claims we made in this thesis the final extension for this work could be the formal analysis of the optimizations we built into our algorithms.

5.3 Closing Remarks

In today's computing and communication systems, high availability of the services is among the most important user requirements. The SAF middleware, a standardized middleware to manage the availability, aims at simplifying, enhancing, and speeding up the development of highly available applications. It also enables solutions to upgrade and downgrade a SAF system while preserving its availability. SMF, a key component of SAF middleware that is responsible for orchestrating the migration of the system from its current deployment configuration to a newer one, requires a certain specification of operations to perform them step by step. Manual generation of such an upgrade campaign specification is a tedious, error prone and sometimes an impossible task to perform. We

believe that the upgrade campaign generation algorithms presented in this thesis and the prototype tool which implements these algorithms will greatly facilitate the upgrade campaign generation process by relieving the campaign designer from the complexity of handling the different upgrade issues.

Bibliography

[1]. Service Availability Forum at: http://www.saforum.org.

[2]. Service Availability Forum, Overview Tutorial SAI-Overview at: http://www.saforum.org/link/linkshow.asp?link_id=227891.

[3]. Service Availability Forum, Overview SAI-Overview-B.05.01 at: http://www.saforum.org/link/linkshow.asp?link_id=222259&assn_id=16627

[4].Service Availability Forum, Hardware Platform Interface SAI-HPI-B.03.01 at: http://www.saforum.org/link/linkshow.asp?link_id=222259&assn_id=16627.

[5]. Kanso, Ali., 2008, Automatic Generation of AMF Compliant Configurations, Master thesis, Concordia University, Montreal.

[6]. Service Availability Forum, Application Interface Specification. Availability Masnagement Framework SAI-AIS-AMF-B.03.01 at: http://www.saforum.org/link/linkshow.asp?link_id=222259&assn_id=16627.

[7]. Service Availability Forum, Application Interface Specification. Software Management Framework SAI-AIS-SMF-A.01.01 at: http://www.saforum.org/link/linkshow.asp?link_id=222259&assn_id=16627.

[8]. eXtensible Markup Language (XML) at: http://xml.org.

[9]. Service Availability Forum, Application Interface Specification. Software Management Framework, Entity Types File SAI-AIS-SMF-ETF-A.01.01 at: http://www.saforum.org/link/linkshow.asp?link_id=222259&assn_id=16627.

[10]. Service Availability Forum, Application Interface Specification. Software Managemant Framework, Upgrade Campaign Specification SAI-AIS-SMF-UCS-A.01.01 at:

http://www.saforum.org/link/linkshow.asp?link_id=222259&assn_id=16627.

[11]. Object Management Group at: http://www.omg.org.

[12]. Java Community Proces at: http://jcp.org/en/home/index.

[13]. L. E. Moser, P. M. Melliar-Smith, L. A. Tewksbury, "Online Upgrades Become Standard", COMPSAC, pp.982-988, 26th Annual International Computer Software and Applications Conference, 2002.

[14]. Java Community Process, J2EE APIs for Continuous Availability. JSR 117 at: http://jcp.org/jsr/detail/117.jsp.

[15].Toeroe, M., Frejek, P., Tam, F., Penubolu, S. and Kasturi, K.,"The Emerging SAF Software Management Framework", In Proc. of the 3rd International Service Availability Symposium (ISAS), LNCS Vol. 4328/2006, pp. 253-270, Helsinki, Finland.

[16]. Wolski, A., Laiho, K., "Rolling Upgrades for Continuous Service", In Proc. of the 1st International Service Availability Symposium (ISAS), LNCS Vol. 3335/2005, pp. 175-189, Berlin, Germany.

[17]. A. Gherbi, A. Kanso, F. Khendek, M. Toeroe and A. Hamou-Lhadj, "A Tool Suite for the Generation and Validation of Configurations for Software Availability", To Appear in the Proc. of the International Conference on Automated Software Engineering (ASE), 2009.

[18]. Java at: www.java.com/en.

[19]. Eclipse at: www.eclipse.org.

[20]. Source Forge, safapp-phase at: http://sourceforge.net/apps/trac/safapp-phase/.

[21]. White Paper in Preparation for the Software Management Framework, to appear in the Service Availability Forum at: http://www.saforum.org.

[22]. Instantiations, Software Tools for Professional Developers at: http://www.instantiations.com.

[23]. Eclipse, SWT: The Standard Widget Toolkit at: http://www.eclipse.org/swt.