AUTOMATIC GENERATION OF AMF COMPLIANT CONFIGURATIONS

ALI KANSO

A THESIS

IN

THE DEPARTMENT

OF

ELECTRICAL AND COMPUTER ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF APPLIED SCIENCE

CONCORDIA UNIVERSITY

MONTREAL, QUEBEC, CANADA

AUGUST 2008

Your file Votre référence
ISBN: 978-0-494-45338-4
Our file Notre référence
ISBN: 978-0-494-45338-4

Abstract

Automatic Generation of AMF Compliant Configurations

Ali Kanso

Nowadays, the demand for robust, reliable, and dependable telecommunication systems is higher than ever. End users expect services to be delivered with minimal to no interruption especially in cases where the effect of service outage can have catastrophic consequences such as loss of human lives and monetary losses. Examples of such applications include air traffic control and navigation systems, or systems that perform money transfer transactions such as VISA.

Systems are considered highly available if they are up and running 99.999% of the time. One solution to sustain such availability is for such systems to be deployed on specific middleware that allow the redundancy of the system components to ensure the availability of services they provide. However, most existing platforms are proprietary and platform dependent.

The goal of the Service Availability Forum (SAF) is to develop open specifications that aim to standardize the interface between the applications and the middleware from one side and the middleware and the underlying hardware from the other side. SAF specifications have also been developed to allow highly available applications to be built

using commercial off-the-shelf components. A key component of SAF is the Availability Management Framework (AMF), which is the middleware part responsible for managing the redundant resources of applications and therefore enables high availability.

AMF, however, requires a certain organization and groupings of those components known as an AMF configuration. Creating AMF configurations manually tends to be very difficult, error prone and sometimes impossible when the number of components forming the application and the cluster hosting the application is considerably high, which is the case for most real-world telecommunication systems.

In this thesis, we devise a solution for automatically generating AMF compliant configurations for applications. The proposed solution encompasses two techniques that vary depending on the way AMF entities are handled. We have implemented both approaches and applied one of them to a case study to demonstrate the applicability of our solution.

Acknowledgments

I would like to express my gratitude to:

- My family, especially my **sister** and my **mother** for their endless support.
- My supervisors, Dr. Khendek Ferhat and Dr. Abdelwahab Hamou-Lhadj for giving me the opportunity to pursue my thesis under their supervision.
- Dr. Maria Toeroe (Ericsson Canada inc.) for whom I am deeply indebted for her guidance and support throughout my stay at Ericsson.
- My girlfriend and friends for their encouragement.
- My colleagues and all those who supported me to achieve this work.
- Concordia University and Ericsson Canada for offering their facilities and resources and providing an adequate research environment.

Contents

List of figures

List of tables

List of algorithms

List of flowcharts

Abbreviations

AMF – Availability Management Framework

CSI – Component Service Instance

CST – Component Service Type

CT–Component Type

ETF – Entity Type File

IMM – Information Model Management

SAF – Service Availability Forum

SI – Service Instance

SMF – Software Management Framework

SG – Service Group

SU – Service Unit

Chapter-1

Introduction

In this chapter, we introduce the context of our research project, SAF and the concept of high availability. We define the problem and introduce the thesis contributions and organization.

1.1 SAF and High Availability

1.1.1 High availability

The reliability of a system is described by its failure rate [16] [17].The availability of a system depends on its reliability and on the time needed to repair the system in case of failure.

The availability of a system is measured using the following metric [17]

1. Mean Time Between Failures (MTBF): The failure rate of the system.
2. Mean Time To Repair (MTTR): The time required to repair the system in case of failure.

$$Availabili\ ty = \left(\frac{MTBF}{MTBF + MTTR} \right)$$

High availability is achieved when the system is available 99.999% of the time [17]. On a one year time frame, the system can be deemed highly available if it can only have a downtime of less than 5 minutes. In an ideal situation, when MTBF is significantly high and MTTR is low, high availability can be achieved. However, in practice, having within a system components with such level of dependability and that could be repaired instantly is not a realistic assumption, thus there is a need to resort to redundancy models [19].In order to minimize the impact of a faulty component on service delivery, "redundant" components must be added to back up the component providing the service. In case the active component fails, the standby can take over the assignment of providing the service. By having redundant components the MTTR of the system (not the component) will tend to zero, leading to an increase of the system availability.

It should be noted, however, that a fully integrated high-availability solution should also account for service continuity (see Figure 1.1). In other words, the state information of

the application sessions of each user must be preserved during the system recovery from a failure.

For example, if a user makes an online banking transaction, and an error occurs in the system, the user's session should be preserved during the repair phase. In other words, the transaction should be routed successfully to its destination.

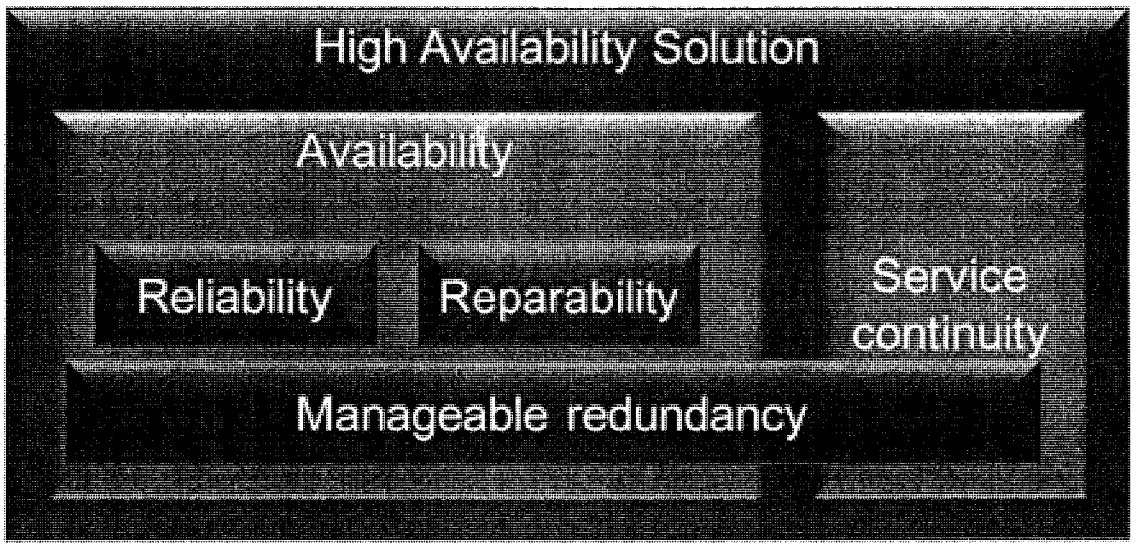


Figure 1-1High Availability Solution

1.1.2 Service Availability Forum (SAF)

Developing solutions that yield to highly available applications is not a recent work; software vendors have been developing such solutions for years (e.g. [18], [20]). However, existing solutions tend to be proprietary and platform dependent, hindering portability of applications from one platform to another.

The Service Availability Forum (SAF) is a consortium of telecommunications and computing companies that decided to join forces to develop open specifications that standardize high-availability platforms. The objective is to create an ecosystem for highly available applications that are compliant with these specifications.

SAF has defined two sets of specifications (Figure 1.2):

- The application interface specifications
- The hardware platform interface specifications

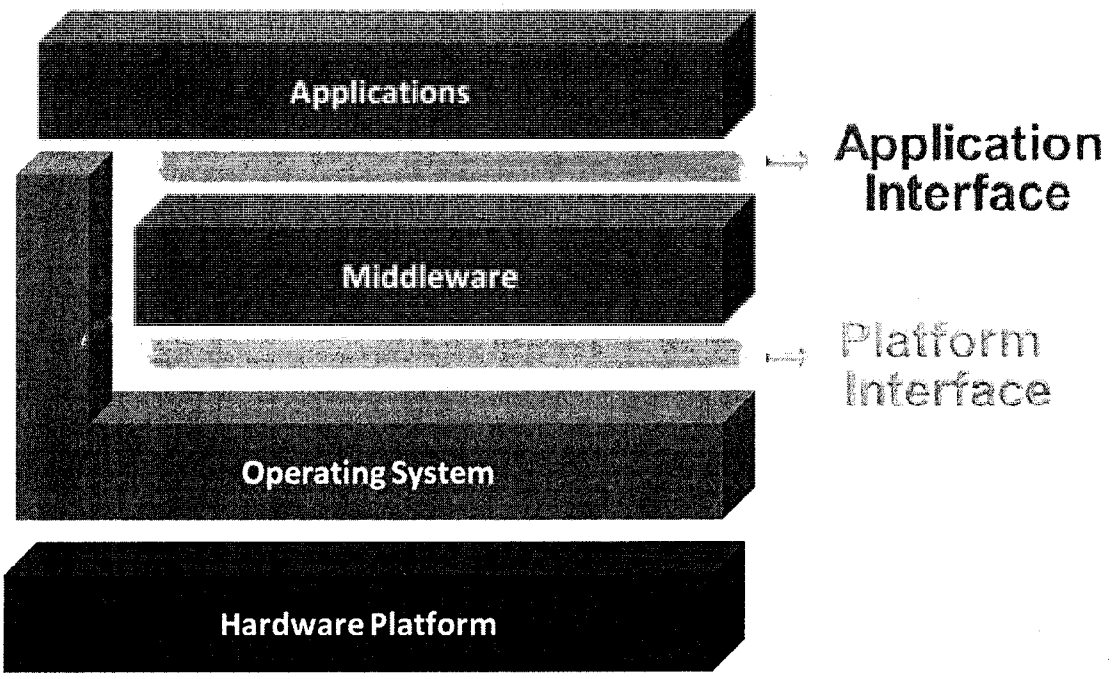


Figure 1-2 The Service Availability Interfaces

SAF middleware contains components that manage the redundancy of hardware and software components, for this reason it needs to interact with both, and therefore both interfaces are needed.

By having the standard interface, the applications can be developed independently of the underlying platform, thus allowing the application developers to concentrate on the application itself rather than on how to make it highly available, deferring this task to the middleware.

Another goal of SAF is to enable application developers to build highly available applications using commercial off-the-shelf components as building blocks of the entire system, which enhances design flexibility and portability [17]. In short, having a standard interface will increase flexibility, creativity and portability of the applications while decreasing the development complexity, the time to market, and maintenance efforts.

1.2 Thesis Motivation and Contributions

As aforementioned, AMF is the SAF middleware part that will manage the availability of the applications; the applications must be configured in an AMF compliant manner. In an AMF configuration, the software components of an application are abstracted into logical entities. These entities are described as objects of classes that represent the AMF

model (defined in the specification). The objects are stored as XML (Extensible Markup Language) files.

Most of AMF entities are typed, the description of these types is provided by the software vendors and they are also described in XML. The types have various capabilities, limitations and constraints. When creating an AMF configuration, a configuration developer must read these types, and select the ones that best meet his or her requirements, depending on the services one wants to provide, the availability level of the services and the work load assigned to the entities of this type. The work of the developer involves extensive search of types, analysis of type capabilities, dependencies, capabilities, constraints and limitations, etc. Generating a configuration even for a small application that involves a limited number of entities requires hours of human work and calculations.

The main motivation behind this thesis is the automation of the configuration generation process. Its main contributions can be summarized as follows:

- Methods of generating AMF configurations
    - A bottom up approach that builds a configuration from a lower granularity.
    - A top down approach that builds a configuration starting from the highest granularity and then goes further down.

- An Eclipse based prototype tool that implements both approaches.

1.3 Thesis Organization

The rest of the thesis is organized as follows: In Chapter 2, we provide the background knowledge to understand the parts of the SAF middleware specifications, necessary for this thesis. In particular, we describe AMF in more detail and how it relates to other SAF services. We then define an AMF compliant configuration, followed with a related work section. In Chapter 3, we present the two approaches for automatic generation of a configuration, the challenges and issues encountered, along with the solutions. The prototype tool and its architecture are described in Chapter 4 as well as a case study. Finally we conclude the thesis in Chapter 5 and present potential future directions.

Chapter-2

SAF Middleware and the Availability Management Framework

In this chapter, we introduce the SAF middleware and its structure. We then describe the Availability Management Framework (AMF) and the role it plays in the overall SAF middleware. Next, we present in depth the concepts of AMF configuration, AMF entities and types, as well as the entity type file, which carries important information used to create AMF configurations. Finally, we survey the literature for related work.

2.1 SAF Middleware

As mentioned in Chapter 1, the SAF middleware aims at providing high availability of network elements, systems and services through the usage of commercial off-the-shelf building blocks [17]. It consists of two main components (see Figure 2.1): The

Application Interface Specification (AIS) middleware and the Hardware Platform Interface (HPI) middleware. The objective of the AIS middleware is to handle high availability of the application's components, whereas the HPI middleware enables the monitoring and management of the underlying hardware.

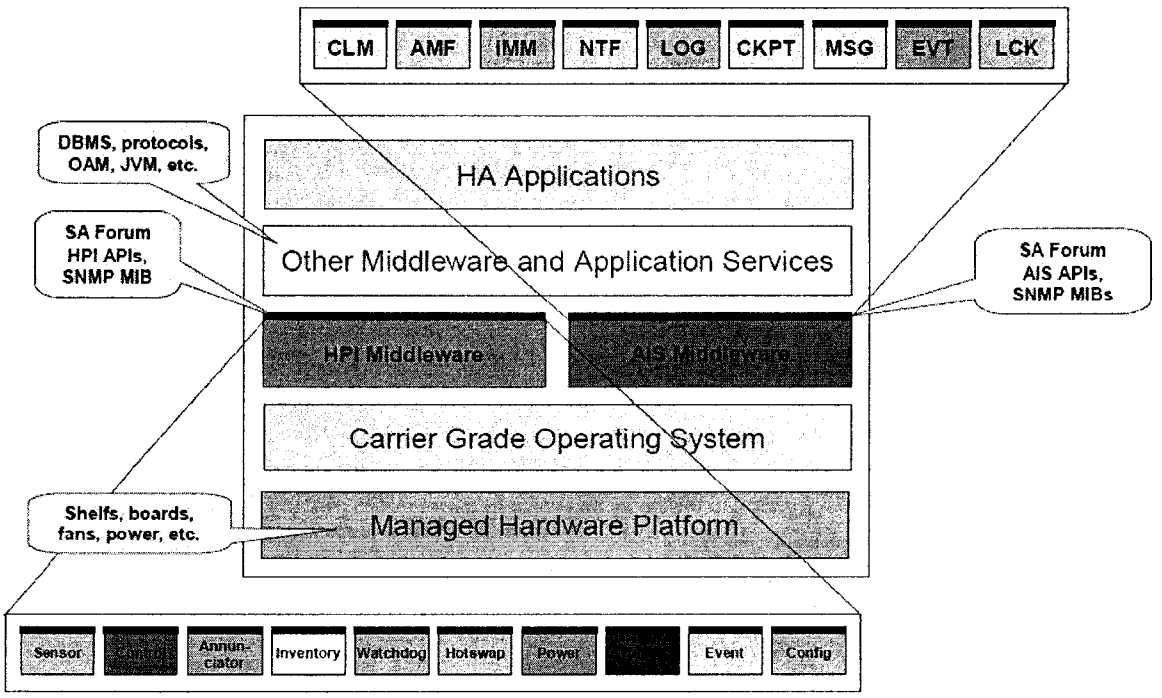


Figure 2-1Architecture of the SAF Middleware (taken from [2]).

As shown in Figure 2.1, the AIS middleware consists of several services that we briefly describe in what follows:

- **The Cluster Membership Service (CLM):** This service maintains the information about all the cluster nodes that are members of the CLM cluster, since

nodes join and leaves this logical cluster, it is important to keep track of the membership. CLM is also in charge of deciding which node can be a member of the cluster.

- **The Checkpoint Service (CKPT):** This service allows processes to record their checkpoint data. When a process recovers from a failure, it can use this data to resume the execution from the last recorded state. Alternatively, other standby processes use this information as well when they take over the active assignment.

- **The Event Service (EVT):** This service allows one or many publishers to publish an event through an event channel, where all the subscribers of this channel can listen to the published event.

- **The Message Service (MSG):** This service provides a communication mechanism that allows processes to exchange messages.

- **The Lock Service (LCK):** In a cluster where resources are distributed and shared, the lock service provides mechanisms used by application processes to coordinate access to shared resources.

- **The Log Service (LOG):** This service enables applications, or the system to create and output log records, there are various types of logs such as alarms, notifications, etc.

- **The Notification Service (NTF):** This service is used to report an incident or a change in status toward the system administration.

- **The Information Model Management (IMM):** This service maintains the system's information model and mediates the administrative operations performed on the objects of the model. We elaborate more about IMM in section 2.3, since AMF interacts with IMM to access the AMF configuration objects and modify their runtime attributes.

- **The Software Management Framework (SMF):** SMF is the SAF service that orchestrates and controls the upgrade/downgrade of a system from one configuration to another. SMF is introduced at a deeper level in section 2.2 to explain its role in maintaining the availability of the services by coordinating the changes applied to the system.

- **The Availability Management Framework (AMF):** It is the middleware part responsible for managing redundancy of the components of an application, hence ensuring high-availability of the application. AMF is discussed in depth in Section 2.4 to illustrate its role in the SAF middleware.

2.2 Software Management Framework (SMF)

During the life-cycle of a running system, it may need to accommodate certain changes/modification triggered by various factors, such as upgrading, downgrading, installing/removing/replacing new hardware/software, repairing defective components etc. Since highly available systems cannot afford to be out of service (e.g. in a shut down state) while the changes are performed, all of the system modifications must be performed while the system is online and providing services. In order to perform those changes dynamically while the system is running; a software management entity is needed. SAF defines the framework of such software entity by introducing SMF.

The migration of a system from one configuration to another must be done in a coordinated manner, and hence the *upgrade campaign* notion is introduced. An upgrade campaign is an XML file that describes all the steps and procedures to be performed in order for the system to reach a desired state. SMF is also responsible for the recovery of the system in case the upgrade fails by allowing the system to rollback to a previous configuration. SMF defines two sets of entities; the software bundle that represents a collection of software provided by the software vendor and an upgrade campaign. The description of the bundle includes the description of the types of its software entities. This information is stored in a file defined by SMF as the Entity Type File (ETF). ETF contains a platform-independent description of the content of a software bundle delivered to a SAF cluster. The data in ETF is described in XML according to an ETF schema defined in a SAF specification [7]. ETF will be further discussed in Section 2.8.

2.3 Information Model Management (IMM)

The SAF information model contains all the logical entities defined by various SAF services (e.g., an instance of a software execution is abstracted as an object called an AMF component, and a checkpoint is represented as a checkpoint object by the checkpoint service, etc.).

IMM service provides the means needed to access and modify the information model objects by system management applications which are *object managers* from IMM perspective. The changes required by object manager are implemented in the system by *object implementers*. IMM specifies a set of APIs to communicate with both object implementers and managers.

Due to the nature of the objects in the information model, some of them are runtime objects (e.g., the checkpoint object) that describe the system's current state and others are configuration objects that describe the system configuration (e.g., components). Configuration objects are managed by the object managers and the runtime objects are managed by the object implementers. Although configuration object can have runtime attributes (e.g., the attributes that describe their state at runtime) only the persistent part will be saved in IMM XML format which is standardized in schema [09].

2.4 Availability Management Framework (AMF)

AMF is perhaps the most important part of the SAF middleware since it has the responsibility of ensuring high-availability of applications. AMF uses logical entities to represent actual resources that are under its control. It also manages the life cycle of these entities. It assigns the workload each entity is supposed to handle. In case of failure, AMF automatically reassigns the workload of a faulty component to a healthy one, and isolates the faulty component while trying to repair it. AMF also defines the functions for health monitoring and error reporting.

In order for AMF to manage the redundancy, it requires a certain organization of the resources under its control. This organization is known as an AMF *configuration*. At system startup, the AMF configuration is loaded into IMM and made available to AMF through a set of APIs that enables AMF to access its objects and modify runtime attributes.

2.5 AMF Entities

In an AMF configuration, the resources are grouped into logical entities on which AMF can perform administrative operations and workload assignment. Figure 2.2 (taken from [2]) shows the UML class diagram of the AMF logical entities and their relations.

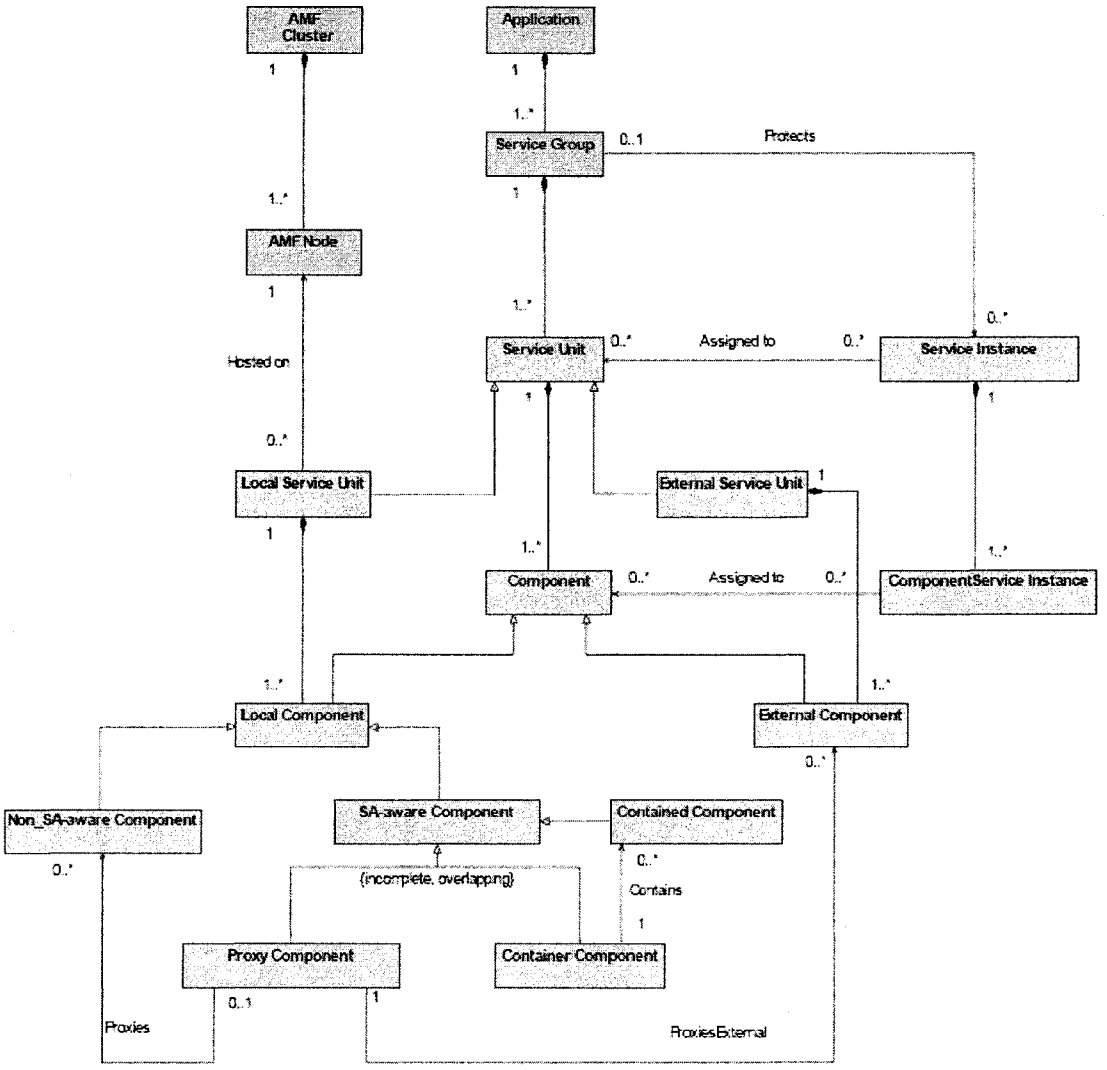


Figure 2-2AMF logical entities

The following subsections describe in detail the AMF entities:

2.5.1 Component

A component is defined as a set of hardware/software resources. It is the smallest entity on which AMF can perform error detection, isolation, and recovery. A component can be local or external depending on whether it resides on a node that is controlled by AMF or not. Depending on the resources the components encapsulate, they may have different properties and behaviors and therefore could be classified into different categories:

- **SA-Aware Components**: Service availability aware components are local components that are under the direct control of AMF. They implement the API function that enables them to register to AMF.

    - **Container and Contained Components**: contained components are components that are not executed by the operating system but rather by a controlled environment like virtual machines. The container components represent the environment where contained components are executed. They are all SA-Aware components.

- **Non SA-Aware Components**: unlike the sa-aware components, they are components that do not implement the API function that enables them to register directly with AMF. They are typically managed through an sa-aware component that acts as a *proxy* for these components. External components are always proxied.

- **Non SA-Aware, Non Proxied Components**: They are local components. However AMF's role consists of managing their life cycle, for example, instantiating and terminating them.

2.5.2 Component Service Instance (CSI)

The services provided by components are abstracted as component service instances. The notion of the services being abstracted separately from the service providers may sound counter intuitive for the first glance, but the CSI represents the workload that is dynamically assigned to a component. Abstracting the workload is essential for managing the high availability, since the ultimate objective behind having redundant components is to protect the workload assigned to them.

2.5.3 Service Unit (SU)

A service unit is a logical entity that aggregates a set of components, combining their individual functionalities into a higher level service. There are two categories of service units, local service unit composed of local components and external service units composed of external components. A service unit can have many components but a component can only be configured for one service unit. An SU can have the HA (High Availability) active state, the HA standby state or no HA state on behalf of an SI. Spare SUs are SUs that have no HA state on behalf of any SI. Note that in the rest of this thesis whenever the terms active/standby states are used, they reflect the HA active/standby states.

2.5.4 Service Instance (SI)

The service instance is an abstraction of the service provided by an SU. It aggregates a set of CSIs. When a service instance is assigned to a service unit, its CSIs are assigned to the components of this service unit. A service instance can have multiple CSIs but a CSI is configured for only one service instance.

2.5.5 Service Group (SG)

A service group aggregates a set of service units that collaborate in a redundant manner to protect the SIs assigned the service group. A service group can contain many SUs but an SU can only be configured for one service group. Each service group is characterized by a redundancy model that organizes the way SUs are protecting the SIs. The following is a list of the redundancy model defined by AMF:

- **2N Redundancy Model:** It requires two service units. The first one is active for all the SIs protected by this SG, whereas the other one acts as a standby:

  Figure 2.3 illustrates an example of an SG that has a 2N redundancy model consisting of two SUs that collaborate to protect two SIs assigned to them.

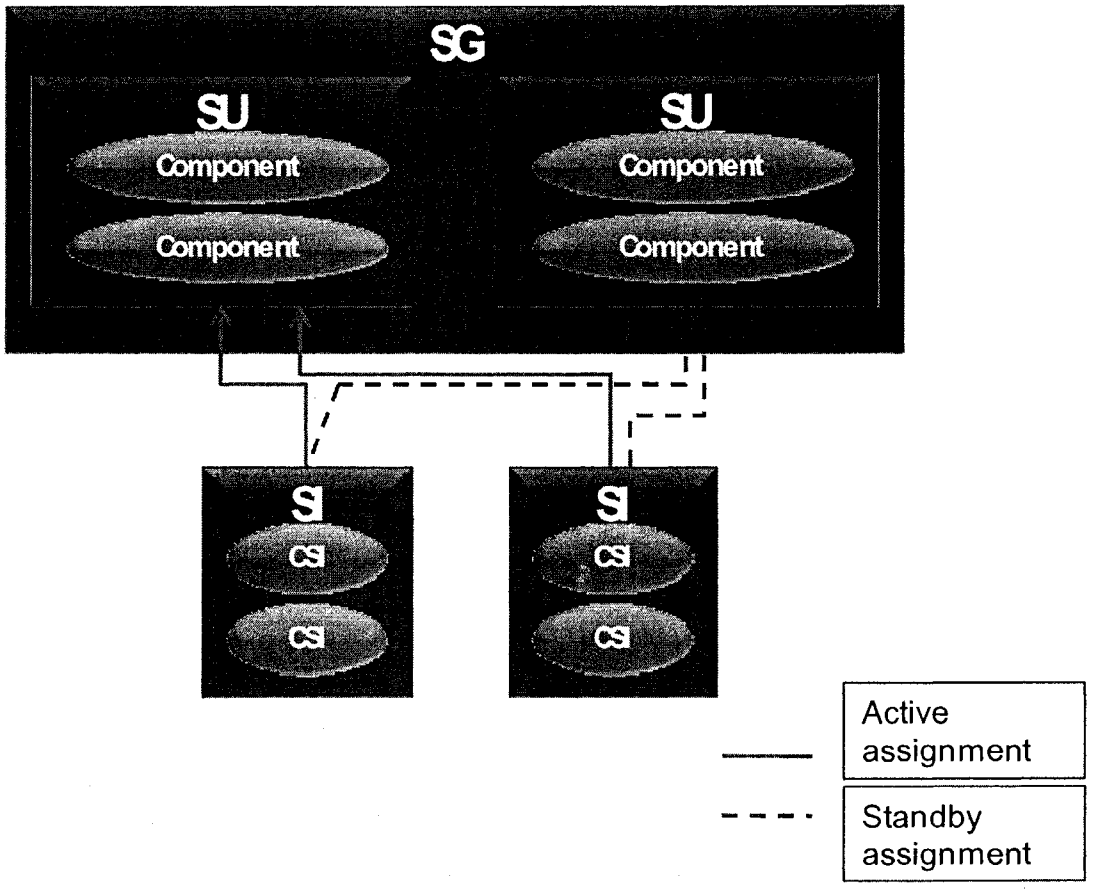


Figure 2-3An Example of a 2N Redundancy Model

- **N + M Redundancy Model:** This redundancy model consists of N service units that are in an active state for all SIs assigned, and M service units in a standby state for all SIs assigned. In addition to this, an SI can be assigned in the HA active state to at most one SU and can be assigned in the HA standby state to at most one SU.

Figure 2.4 shows an example of an SG that has N+M redundancy model. In this example, there are 2 SUs that are active and one SU that acts as a standby.

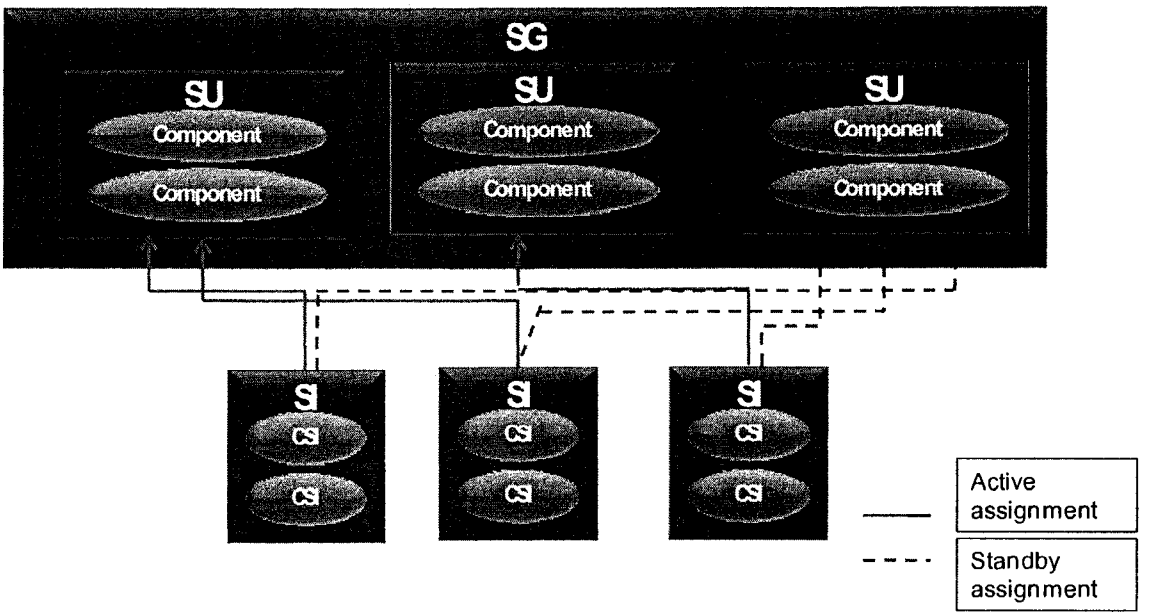


Figure 2-4An example of N+M redundancy model

- **N Way Redundancy Model:** It consists of N service units that can be in active and/or standby state (see Figure 2.3). In addition, an SI can have at most one active SU but one or more standby SUs.

  Figure 2.5 illustrates an example of an SG that has N Way redundancy model where each SI protected by this SG has one SU assigned in an active state and two SUs that act as standbys.

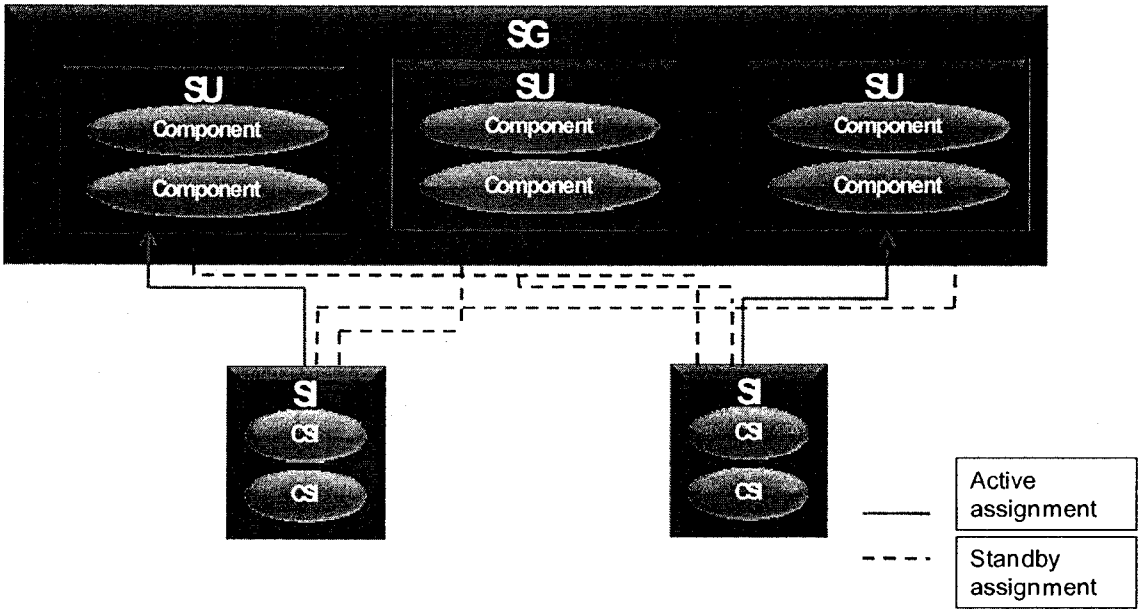


Figure 2-5An example of N Way Redundancy Model

- **N Way Active Redundancy Model:** It consists of N service units that handle the entire set of SIs protected by the SG in their active state. There are no standby assignments. In addition, an SI can be assigned to one or many SUs in the HA active state.

  Figure 2.6 illustrates an example of an SG that has N Way Active redundancy model and consists of three SUs where each SIs have two active assignments.

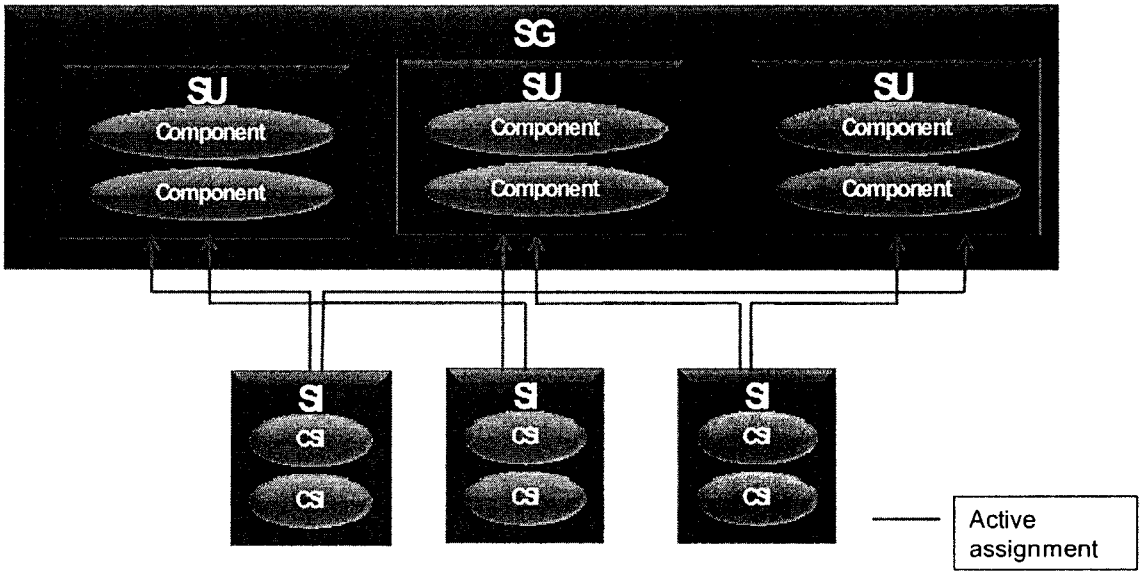


Figure 2-6An Example of N Way Active Redundancy Model

- **"No Redundancy" Redundancy Model:** It consists of one or many service units that handle the entire set of SIs protected by the SG in their active state. There are no standby assignments. Unlike the N Way Active redundancy model, an SI can be assigned to at most one SU in the HA active state. An SU can take at most one assignment. It may sound odd that in a highly available system we have no redundancy, but due to the nature of some of the service provided by the service unit in case no state information needs to be preserved, it is not necessary to have standby SUs, but instead we can have spare SUs that are able to take over the active assignment in case of a failure.

Figure 2.7 Illustrates an example of an SG that has "No Redundancy" redundancy model, where each SU is active for one SI and one SU is a spare.

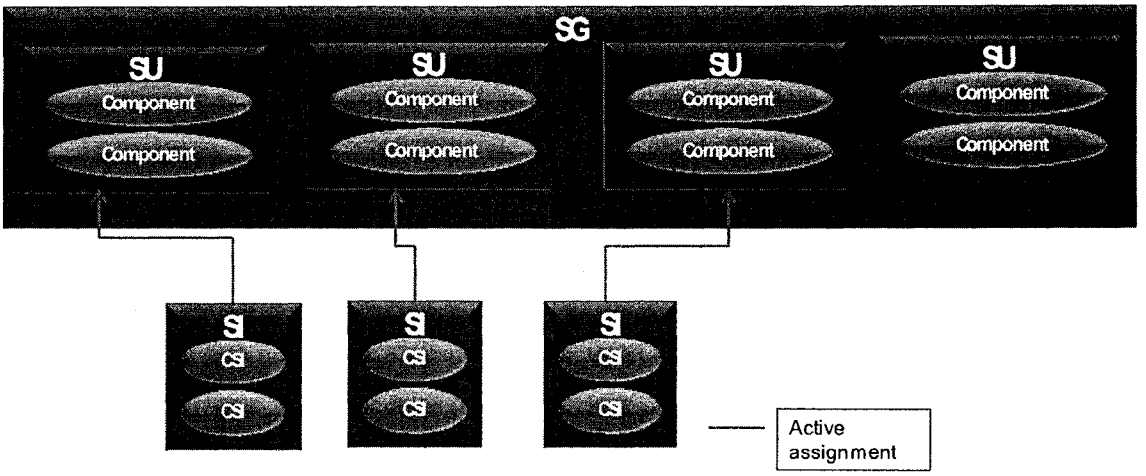


Figure 2-7An example of "No Redundancy" redundancy model

2.5.6 Application

An application is an aggregation of service groups. An application can have many service groups but a service group can be configured for only one application.

2.5.7 Nodes and Cluster

An AMF node is logical representation of AMF entities on a cluster node. An AMF cluster is a set of AMF nodes.

2.6 Example of an AMF Configuration

All of AMF entities are logical, they represent a certain organization of resources, since AMF is providing high availability for application software; the resources are the running processes of software entities. For example a component C1 can be a software execution of a billing application that processes phone calls and determines the cost according to predefined criteria. In addition, a database is needed to store the billing information, which can be represented by another component C2. Each of those two components will have CSIs assigned to them. For example, one of the CSIs for the billing process could be to process the phone calls made from Montréal region, and the other CSIs can represent the workload from other regions. C1 and C2 will form a service unit SU1, which provides billing services. The CSIs are aggregated into SIs that represent the billing workload an SU is supposed to handle. These SIs are assigned to the service unit SU1 in their active state. In order to protect these SIs, a replica of SU1 (a redundant SU), called SU2, will be placed in the same service group SG1 hosting SU1. SU2 will have the standby assignment on behalf of the entire set of SIs protected by SG1 and SU1 will have the active assignment, this is the case of a 2N redundancy model. SG1 will compose an application called App1 which is a simple application responsible for providing billing services. Figure 2.8 is an illustration of typical AMF configuration involving logical entities for this example application.

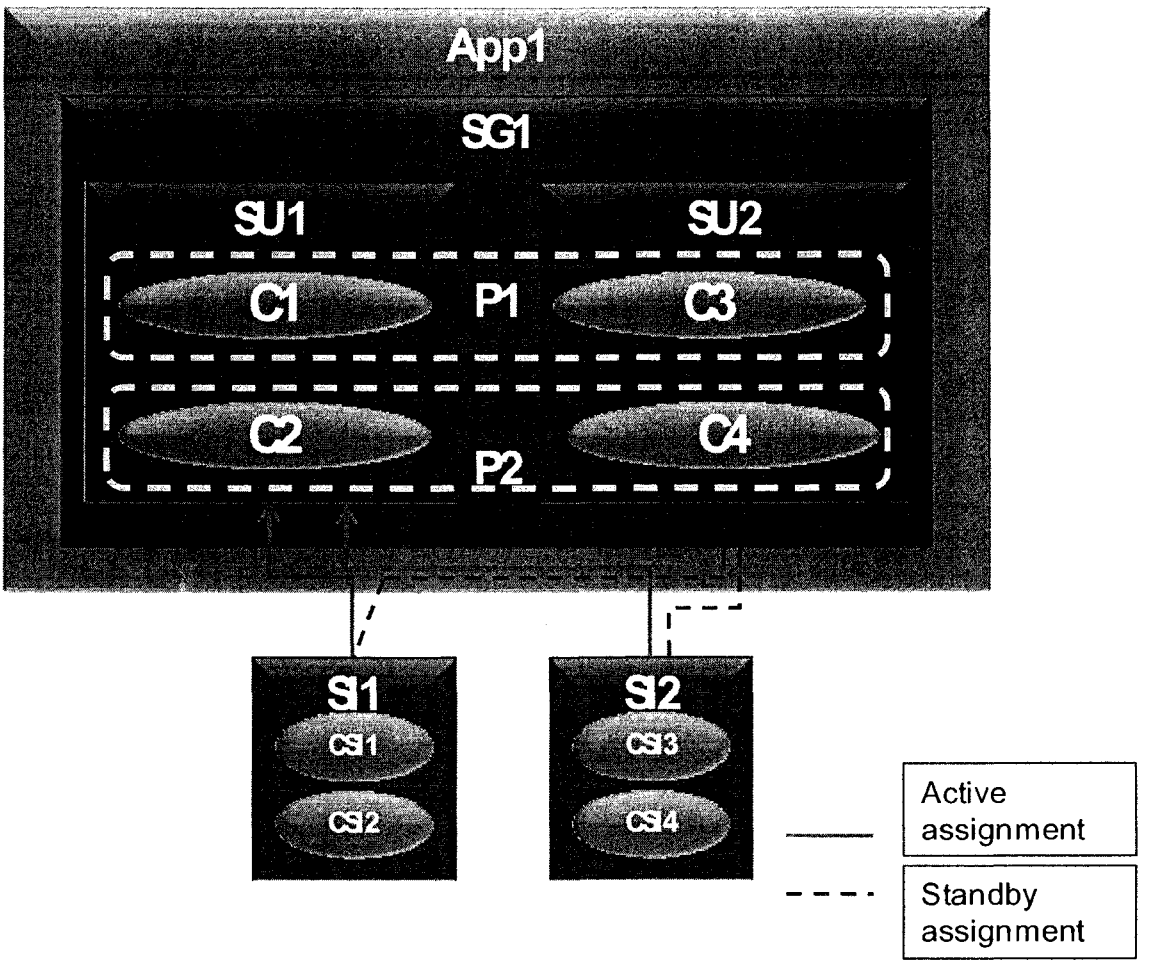


Figure 2-8 An Example of an Application

Every CSI in an SI will be supported by two components that will have respectively the active and standby state on behalf of this CSI. This set of components protecting the CSI is known as a protection group, each CSI has one protection group, a component can be a part of zero, one or many protection groups, the protection groups in Figure 2.4 are denoted by P1 and P2.

In case, the component C1 or C2 fails, the fault will cause their operational state to be disabled and hence the operational state of SU1 will be disabled AMF will transfer the

active state of the SIs to SU2 which will take over the active assignment of the SIs of SU1.

2.7 AMF Entity Types

Most of AMF entities are typed. The type of an entity contains vital information about the services this entity can provide (if it provides a service), as well as its limitations, compatibility, dependencies with other entities, etc. AMF types are derived from the Entity Type File (ETF) discussed in Section 2.2, that describes the features of the software application that will run on the top of AMF.

2.7.1 Component Service Type (CST)

The component service type is the type of service a component can provide. For example, if a component is a running instance of an FTP server, then the CST would typically refer to a file transfer service. The CST is referred to by a component type and a CSI, since a CSI is the logical AMF entity that describes the workload assigned to a component. Each CSI must refer to one and only one CST.

The CST specifies a set of attributes and their valid value range that describe a particular work load of the service. For example, for an FTP service provider, the attributes of the component service type and their values could be:

- Speed: high speed, regular etc.
- Security level: high, medium, etc.
- IP range: higher bound, lower bound, etc.

2.7.2 Component Type

A component type describes a set of hardware/software resource. The most important features of a component type are:

- The component service type(s) that the component of this component type can provide.
- The component capability model with respect to a particular component service type. The component capability is expressed in terms of the number of CSIs of a particular CST that a component of this type can support in their active and/or standby state. The following are the capability model defined by AMF:
    - X active and Y standby
    - X active or Y standby
    - One active or X standby
    - One active or one standby
    - X active
    - One active

- The component category such as sa-aware, proxied, and non proxied, non SA-Aware, or container and contained.

- A list of other component types that this component type can collaborate with in a redundant manner.

- A list of component types required by this particular component type in order to provide a particular CST.

2.7.3 Service Type

A service type is defined by a set of CSTs. It defines the type of services a service unit can provide. The service type imposes constraints on the number of CSIs of a particular CST that can exist in a service instance referring to this particular service type.

2.7.4 Service Unit Type

The service unit type is defined by a set of component types it aggregates. The SU type may impose a constraint on the maximum number of components of a particular type that can be included in a service unit of this type. An SU type also defines a set of service

types that a service unit of this type can provide. It also specifies whether this service unit type depends on other service unit types providing different service type(s).

2.7.5 Service Group Type

The service group type is defined by the set of service unit types it aggregates as well as the redundancy model of the SGs of this type.

2.7.6 Application Type

The application type is defined by the set of SG types it can be composed of.

2.8 AMF types Versus ETF types

ETF describes the types from SMF perspective; it describes the ways the software could be deployed and its various capabilities and limitations. AMF deals with types from a configuration and runtime management point of view in terms of dynamically assigning work load, performing error recovery actions etc. However, when ETF describes the dependencies, constraints, limitations and compatibility, it is because they are important to enable the deployment of a set of software in such a manner that will function properly and deliver the required service. For example, when ETF describes component dependency, it is very important in terms of creating a valid deployment configuration

and later upgrading or modifying the system, but AMF is agnostic with regards to this dependency because it is assumed that is has been taken care of at configuration time and thereafter it does not affect the way AMF is managing the redundancy.

As a result ETF defines ranges for some attribute values e.g. the attribute value of a CST, while AMF requires a specific value, from this perspective, ETF types act as meta-types for AMF types. From one ETF type, we can create multiple AMF types according to our needs. For example, if ETF defines a component type CT_A that provides CST_A and CST_B, and component type CT_B that provides CST_B, and that for our system configuration sake we choose the components of type CT_A to support the CSIs with CST_A, and the components of type CT_B to support CSIs with CST_B and place them in a service unit, we cannot be sure that AMF will assign the CSIs according to our preference, since components of type CT_A can support both CSTs. AMF might assign to them CSIs of both CST_A and CST_B, the optimal solution would be to create an AMF component type AMF-CT_A derived from CT_A, that can only provide CST_A, this way we can be sure that the components from this component type will only be used to serve CSIs of type CST_A. However if CT_A was chosen to provide both CST_A and CST_B then, AMF-CT_A will created in a such a manner that it supports both CSTs.

Another important point that distinguishes ETF types from AMF types is that the only mandatory types that need to be described in ETF are the component type and the component service types, and they form the building blocks for other types. The rest of the types are optional. If necessary they define a set of limitations and constraints on how

the component types and the component service types can be grouped and the way they must collaborate to provide a service in terms of dependencies and collocations. On the other hand, AMF requires the existence of all types in an AMF configuration since those types provide vital information about the AMF entities. For example the SU type defines the service type(s) an SU can provide and an SG type defines the redundancy model of an SG.

2.9 AMF Compliant Configuration.

Having defined AMF entities, AMF types, an AMF compliant configuration is the set of the following logical entities and their interrelations:

- AMF cluster and its member AMF Nodes.
- The set of SIs and their CSIs that need to be protected
- The set of applications and their SGs, SUs, and components
- The set of AMF types for each of the above entities except the cluster and nodes

The existence of all the objects representing the above entities and types does not define a valid AMF configuration unless the attributes of the above objects are populated with the correct values. Determining these values requires complex analysis and calculations as we will see in Chapter 3.

2.10 Related Work

The standardization at SAF is ongoing, existing service specification are reviewed and updated as necessary, and more of the services are being defined. The B.03.01 version of the AMF specification on which the reported work is based differs significantly from earlier versions as it introduced the AMF types to be aligned with the first release of the Software Management Framework specification [6].

The work on implementing the APIs is ongoing in different places; OpenAIS [14]. OpenSAF [15] and OpenClovis [16] are open source projects aiming at developing a SAF compliant middleware for high availability. These tools provide limited if any support for automatic generation of AMF configurations. In addition, none of them considers AMF types.

The closest research work to the contents of this thesis in the context of SAF has been reported in [10]. The authors in [10] apply the Model Driven Approach (MDA) to the design of AIS configurations. In this approach an initial AIS compliant configuration is devised using predefined design patterns, gathered from previous experiences. This initial configuration is referred to as the Platform Independent Model (PIM), which is then transformed and specialized automatically to a Platform Specific Model (PSM) to be used in a specific implementation of AIS. Meta-models are used for the transformation and for the validation of configurations. Our work is different from this approach, as we automatically generate this initial configuration or PIM. More work on configuration

generation has been done in the more general context of software configuration management, particularly using constraint satisfaction techniques and policies as reported in [11, 12]. Authors in [12], for instance, propose an approach for generating a configuration specification and the corresponding deployment workflow from a set of user requirements, operator and technical constraints, which are all modeled as policies. An example of constraints is, for instance, a given operating system can only run on certain processor architectures. Generating a configuration is formulated as a resource composition problem taking into account the constraints. Our approach is similar from this point of view; however, our focus is on the availability and AMF constraints instead of general utility computing environments. Challenging constraints, such as redundancy models to be provided, are not taken into account in [12].

Chapter-3

Configuration Generation

The overall picture of configuration generation is illustrated in Figure 3.1. Going from the input sets to the desired output requires several steps that are discussed in the coming sections of this chapter.

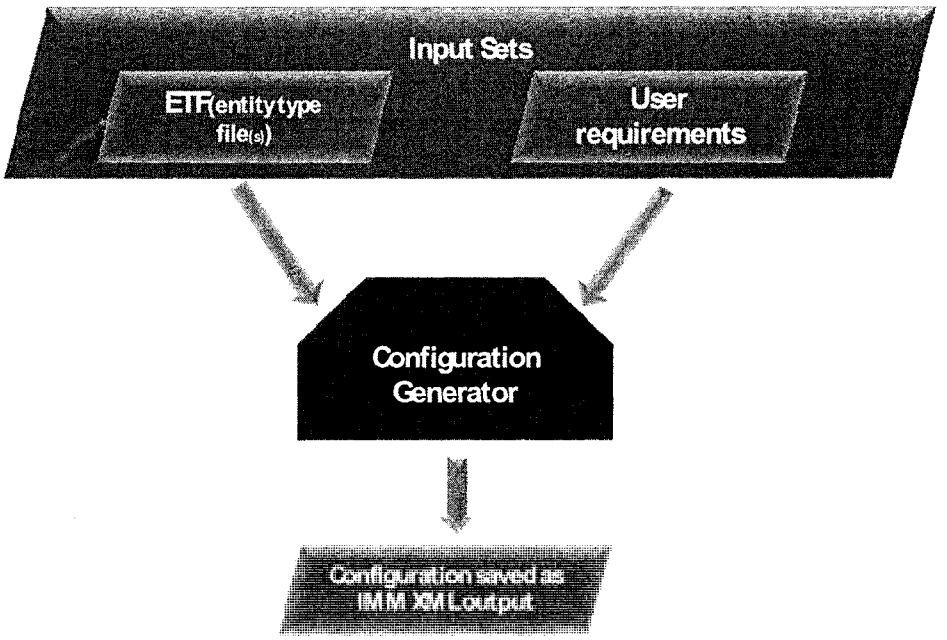


Figure 3-1 Overall Picture of Configuration Generation.

Figure 3.2 shows the main steps of our configuration generation algorithm. The algorithm takes as input the ETF provided by the software vendor and the configuration requirements provided by the configuration designer including the application services to be supported. The next step focuses on determining the AMF entity types that can support these required application services: namely the SU types, SG types, and application types. Once these types are determined, we proceed with creating their entities: components, SUs, SGs, applications. Then, entities and types' attributes are completed. Those attributes include the rank the SUs for SIs and hosting nodes for SUs. Finally, the generated configuration is specified in IMM XML.

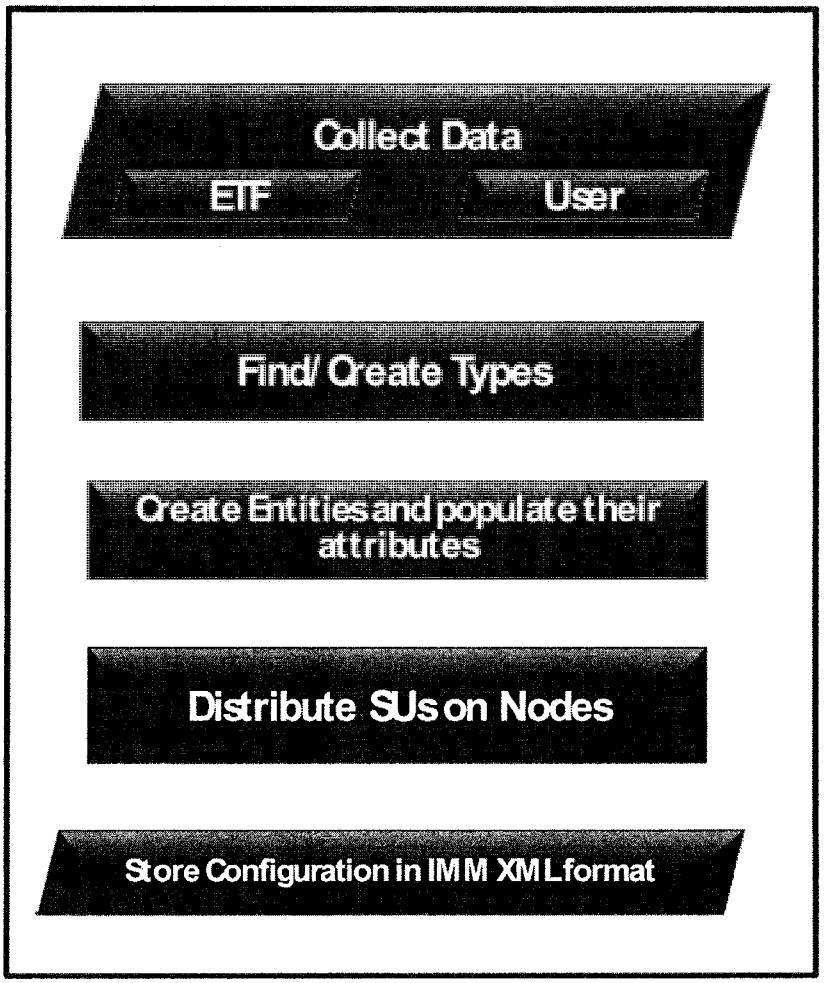


Figure 3-2 The Main Steps for Configuration Generation

3.1 Input Data and Validation

As shown in Figure 3.1, the input data needed to generate AMF configurations consists of two data sets:

- ETF types that describe the application to be deployed on top of the SAF middleware.
- User requirements that describe the services to be supported by the application as well as information about nodes and the cluster.

3.1.1 ETF Types

As discussed in the previous section, the ETF types describe the software application from the vendor's perspective. ETF must provide at least two types: the component types (CT) and the component service types (CST). Other entity types such as service types, SU types, SG types, and the application types may also be provided in order to capture limitations and constraints of the application. However, they are not necessarily provided in ETF.

In Figure 3.3, we show the typical relationships between ETF types. As shown in this figure, an application type refers the SG type(s) it can support; an SG type in turn can refer to SU type(s) which can also refer to component type(s). If a type is referred to by

another type this means that an entity of the referring type can support an entity of the referred type. If a type is not referred to by any type, we categorize it as an orphan type. In figure 3.3 examples of orphan types are "SG Type-3", "Comp Type-3", "CST-2" and "SU Type-2". As can be concluded from figure 3.3, the relations between types are not a parent child relationship, since a child can only have one parent, which is not the case here. In this thesis if we use the term parent type we mean one of the types that refer to our "child" type(s).

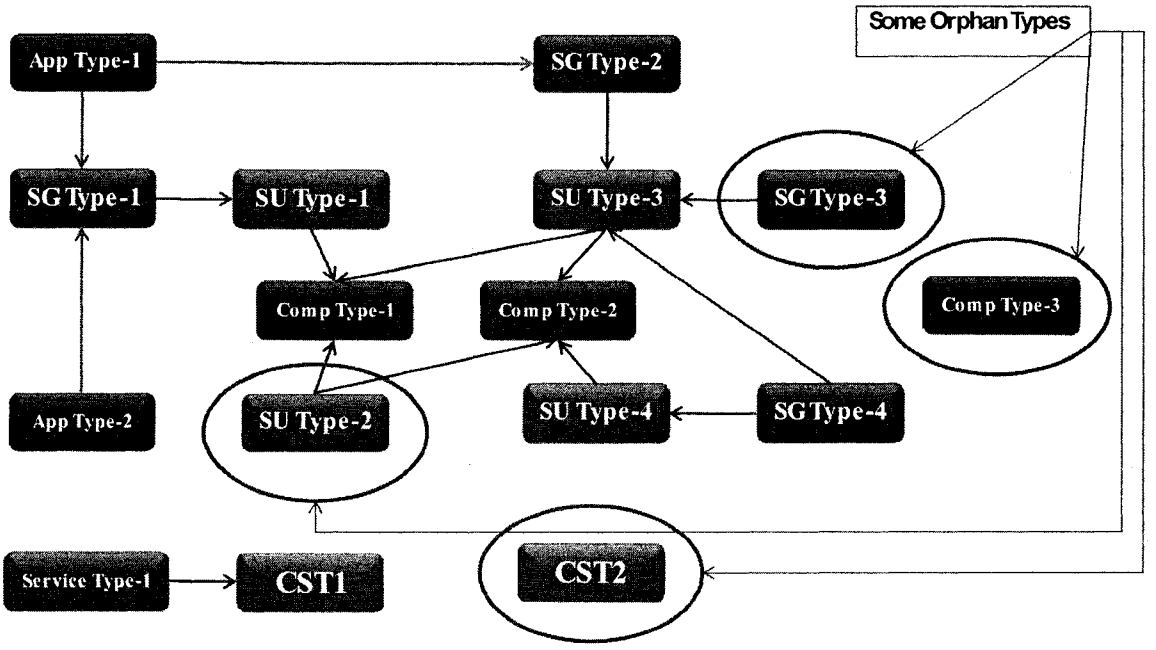


Figure 3-3Typical Relations Between Types and Some Orphan Types

We use the term *orphan type* to refer to a type that is not constrained by a higher type. For example, if a component type is provided without having an SU type that supports it then we call this component type an orphan type. It is important to note that since AMF expects all types to be present in a configuration, if an orphan type from ETF is selected

then there is a need to create all parent types. For example, in the case of a component orphan type, we will need to create an SU type, SG type, and an application type.

3.1.2 User Requirements

The second set of input data is provided by the configuration designer. The user enters two types of data:

- A set of services to be provided by the application, and
- A set of nodes on which the configuration has to be deployed.

We introduce the concept of templates to represent a group of multiple services and nodes that share common characteristics. This will also facilitate the data entry process. We define templates for CSIs, SIs, and nodes.

3.1.2.1 CSI Template

A CSI template contains the information needed to create a set of CSIs. A CSI template contains the following information:

- A template name that is used as a prefix for all CSIs names of the CSIs created from this template

- The number of CSIs to be created
- The component service type (CST) of the created CSIs
- The CST attributes' values and the offset that will determine the workload represented by CSIs of this type. For example if the attribute is an integer range representing the citizens social insurance numbers and the initial value of is X, and the offset is 1000, then CSI-1 will represent the workload of citizens X to X+1000, CSI-2 will be the workload of X+1001 to X+ 2000 etc.
- The relationships between the CSIs if there are any

3.1.2.2 SI Templates

Similar to a CSI template, an SI template contains the information needed to create a set of SIs, and the level of protection required. More precisely, an SI template contains the following information:

- A template name that will be used as a prefix for all the SIs created from this template
- The number of SIs to be created
- The service type of the SIs instantiated from this template
- Dependencies among SIs if there are any
- The set of CSI templates that constitute the set of CSIs each of the created SIs will contain.

- The redundancy model of the SG that needs to protect the SIs derived from this template.
- The following information needs to be provided in case the redundancy model is:
    - N +M:
        - The number of standby SUs (In the SG protecting the SIs generated from this template. For other redundancy models except 2N, this is set to zero)
        - The number of active SUs. (In the SG protecting the SIs generated from this template).
    - N way active:
        - The number of active assignments (meaning the number of SUs that will have the active assignment of behalf of a particular SI generated form this SI template. For other redundancy models, this is set to one)
        - The number of active SUs (should be greater than the number of active assignments per SI)
    - N way:
        - The number of standby assignments. (Meaning the number of SUs that will have the standby assignment of behalf of a particular SI generated form this SI template. This is set to one in case of 2N and N+M redundancy models and to zero for N way active and "no redundancy".

- The number of active SUs, which should be greater than the number of standby assignments +1.

- 2N or "no redundancy": all the above attributes that are specified according to the redundancy model can be deduced in this case, so there is no need to specify any.

In an N way active redundancy model the number of active SUs should be greater than the number of active assignments per SI because otherwise SIs will not have the required number of SUs assigned the active state on their behalf and hence the SIs' assignment state would be partially assigned instead of fully assigned. For example if an SI requires three active assignments and we have only two SUs, one assignment must be dropped. in case of the N way redundancy model. The same reasoning applies and that is why the number of active SUs should be greater than the number of standby assignments + 1 (the one is the active assignment).

Figure 3.4 illustrates the relation between a SI template and a CSI template. In this example, three SIs are generated from the SI template; each of them contains eight CSIs that derive from two different CSI templates.

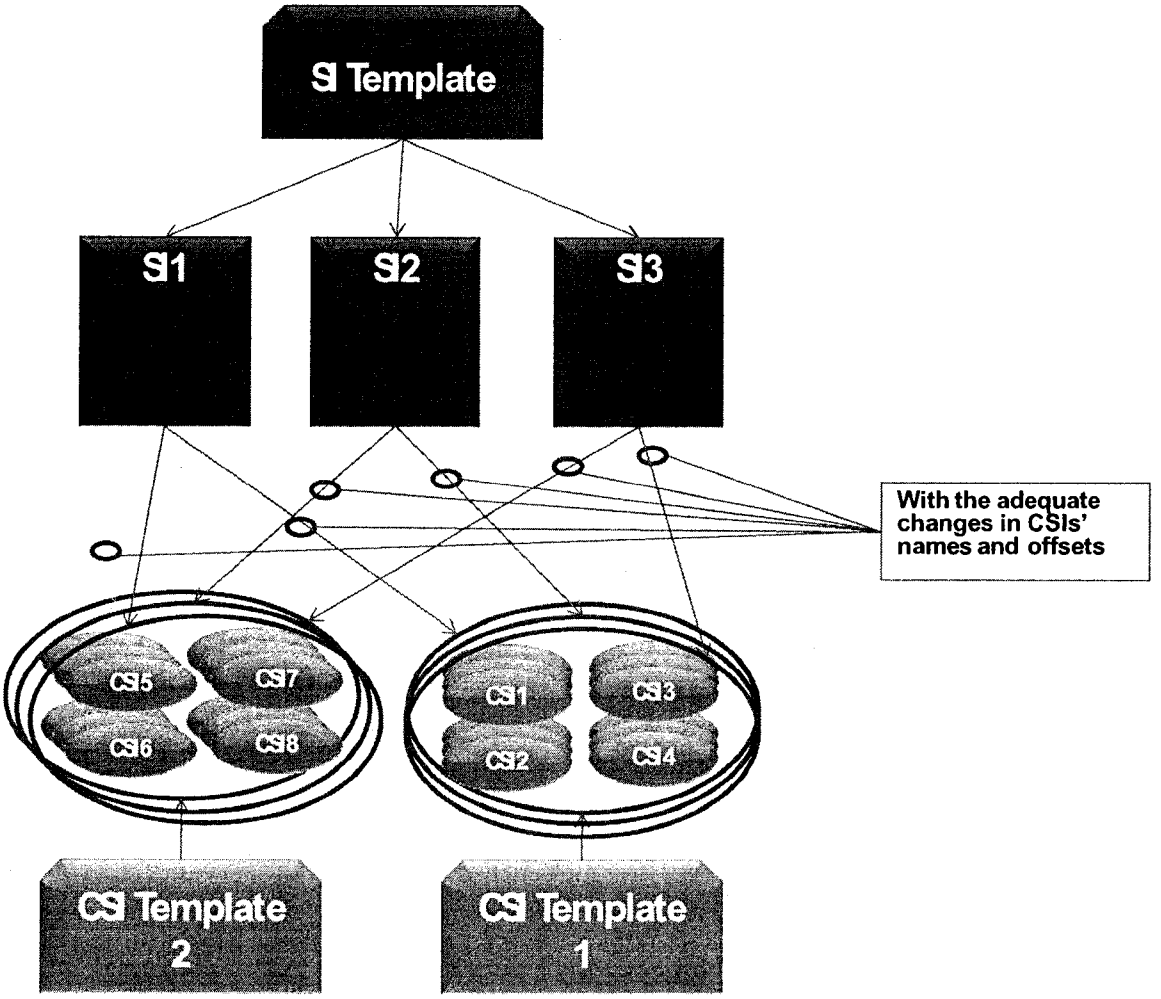


Figure 3-4Relation Between the SI Template and the CSI Template

3.1.2.3 Node Template

The node template is used to create AMF nodes that are identical, the templates include all the attributes defined by the AMF node class in addition to the number of nodes that needs to be created.

3.2 Selecting Types

The objective of this step is to determine the types of the entities that will provide the required services specified by the configuration designer. ETF types are examined to check if they are valid in terms of meeting the configuration designer requirements, if that is the case, AMF types are derived from the selected ETF types, however if some types are missing from ETF, or found not valid, AMF types must be built from existing orphan ETF types, if those types do not verify the requirements, then they cannot be used to build AMF types, and hence the configuration generation will fail.

In order to determine whether an SU type/component type is applicable to be used or not, we need first make sure that it provides the required service type/CST, then calculate the load of SI/CSIs that is anticipated to be assigned to it, and finally examine its capacity of providing the required service type and see whether it matches the calculated load. Section 3.3.1 illustrates how we calculate the expected load of SIs per SU at runtime.

3.2.1 Calculating the Expected Load of SIs per SU

Since each SU is expected to support an active load of SIs and/or a standby load of SIs, calculating the number of SIs per SU is a function of the active and standby assignments for those SIs depending on the redundancy model. Equations 3.1 and 3.2 are used to determine the SU load for active and standby assignments. The variables used in these equations are defined in what follows:

- *siTemplate*: refers to an SI template
- *siTemplate.numSIs*: refers to the number of SIs generated from a specific SI template
- *siTemplate.numSUs.sus*: refers to the total number of SUs participating in the SG that will protect the template SIs
- *siTemplate.numSUs.susAct*: refers to the number of SUs that will only have the active assignment on behalf of SIs. For example, in the case of a N+M redundancy model this number is N.
- *siTemplate.numSUs.susStdb*: refers to the number of SUs that will only have the stand by assignment on behalf of SIs. For example, this is number is M for the N+M redundancy model.
- *redMod*: refers to the redundancy model specified by the SI template
- *siTemplate.numAct*: the number of active assignment for each SI.
- *siTemplate.numStdb*: the number of active assignment for each SI.

$$suActLoad = \begin{cases} redMod \equiv nway \Rightarrow \text{ceil}\left(\dfrac{siTemplate.numSIs}{siTemplate.numSUs.sus - 1}\right) \\ redMod \equiv nwayactive \Rightarrow \text{ceil}\left(\dfrac{siTemplate.numSIs \times siTemplate.numAct}{siTemplate.numSUs.susAct - 1}\right) \\ redMod \equiv noredundancy \Rightarrow 1 \\ redMod \equiv 2n \Rightarrow siTemplate.numSIs \\ redMod \equiv nplusm \Rightarrow \text{ceil}\left(\dfrac{siTemplate.numSIs}{siTemplate.numSUs.susAct}\right) \end{cases}$$

Equation 3-1 The Active Load Formulas

$$suStdbLoad = \begin{cases} redMod \equiv nway \Rightarrow \text{ceil}\left(\dfrac{siTemplate.numSIs \times siTemplate.numStdb}{siTemplate.numSUs.sus}\right) \\ redMod \equiv nwayactive \Rightarrow 0 \\ redMod \equiv noredundancy \Rightarrow 0 \\ redMod \equiv 2n \Rightarrow siTemplate.numSIs \\ redMod \equiv nplusm \Rightarrow \text{ceil}\left(\dfrac{siTemplate.numSIs}{siTemplate.numSUs.susStdb}\right) \end{cases}$$

Equation 3-2 The Standby Load Formulas

An simple example of calculating the load can be illustrated by an SG that has three SUs providing services for five SI with a N way redundancy model, where each SI has one active assignment and two standby assignments (Figure 2.5). Using Equation 3.1, the active SI load per SU would be:

$$\text{ceil}\left(\frac{siTemplate.numSIs}{siTemplate.numSUs.sus - 1}\right) = \text{ceil}\left(\frac{5}{3-1}\right) = 3 \text{ SIs/SU.}$$

This means that each SU should have the capacity of being active for three SIs. The calculated standby load using Equation 3.2 would be:

$$\text{ceil}\left(\frac{siTemplate.numSIs \times siTemplate.numStdb}{siTemplate.numSUs.sus}\right) = \text{ceil}\left(\frac{5*2}{3}\right) = 4 \text{ SIs/SU.}$$

Each SU should have the capability of being standby for four SIs. Figure 3.5 illustrates graphically how the load is likely to be distributed.

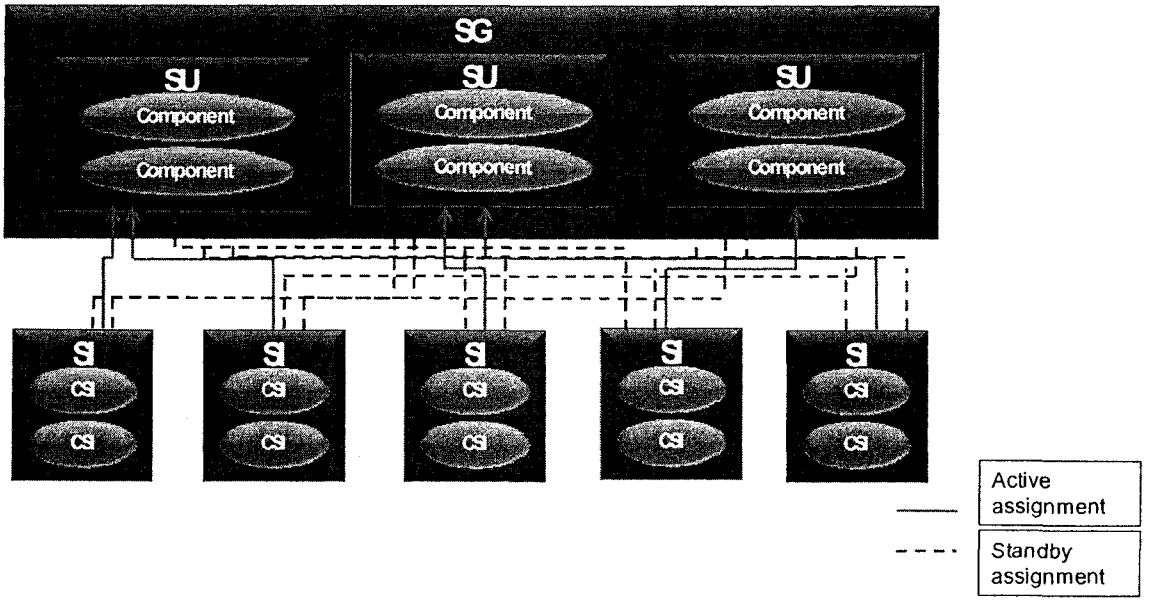


Figure 3-5 An Example of SUs' Load of SIs in N Way Redundancy Model

In Figure 3.5, each SU is assigned at most two SIs in their active state as opposed to three as computed by the above equations. The reason behind this discrepancy is that the equations calculate the load an SU is supposed to handle in a system with no single point of failure, meaning, if an SU fails, the other SUs should be able to carry its load of SIs. In the example above, if one SU is down, all the SIs will keep their active state assignments but will loose one of their standby state assignments, since the number of in service SUs, which is two, will not allow anymore for each SI to have one active SU and two standbys To maintain this assignment for the SI, three SUs are needed, one to be active, and two to be standbys.

It should also be noted that that the above equations assume equal load among SUs, i.e., the all SUs handle the same number of active and standby assignments. Load balancing of SUs is discussed in more detail in section 3.4.3.

3.2.2 ETF Types Selection

In this section we discuss how ETF types are selected. However, these selected types are not going to be a part of the AMF configuration, but rather, AMF types are going to be created or built based on the chosen ETF types.

3.2.2.1 ETF Component type selection

A component type is selected with respect to a set of CSIs generated from the same CSI template, and have the same component service type. Finding component types can be further divided into finding orphan CT and finding non orphan CT, This differentiation is due to the constraints put on non-orphan CTs by the parent SU type.

- Finding orphan component types:

  Finding an orphan component type is done by matching the CST specified in the CSI template with the CST(s) provided by this particular CT, if we have a match, the component capability model is checked if it is applicable with respect to the

redundancy model of the parent SI template of our CSI template, if that is the case, then it can be chosen to support the CSIs of this CSI template.

Table 3.1 illustrates the applicability of the component capability model of a component with respect to the redundancy model.

Note that Table 1 differs from the table described in the AMF specification [SAI-AIS-AMF-B.03.00.07] in the cells that are marked "- -", since according to the specifications, components can participate in a redundancy model where a standby assignments is needed even if they can only serve active assignments (e.g., components that have I active as a capability model), In practice this may lead to undesirable results such as loss of service in case state information cannot be synchronized with a standby component.

Redundancy model=> ---------- ---- Component capability	2N	N+M	N Way	N way Active	No redundancy
X active and Y standby	√	√	√	√	√
X active or Y standby	√	√	-	√	√
1 active or X standby	√	√	-	√	√
1 active or 1 standby	√	√	-	√	√
X active	--	--	-	√	√
1 active	--	--	-	√	√

-: aligned with the specification
-- : different from the specifications

Table 1 Applicable Component Capability Model with Respect to Redundancy Models

Algorithm 3.1 illustrates how orphan component types are selected from ETF based on the information found in a CSI template. The input of this algorithm is an SI template *siTemplate* and a CSI template *csiTemplate.* In this algorithm, we first match the CST of the component type with the one of the CSI template (line 5), in case we have a match, we check the required redundancy model by the SI template (line 6) then we check if the capability model of the component for the CST is applicable for the required redundancy model according to Table 1 (lines 8, 17), if this is case we return the component type.

```
1: BEGIN
2:      found = false
3:      FOR EVERY compt IN orphanCTS
4:           FOR every compt.csCapability IN compt
5:                IF currentCompt.csCapability.cst = csiTemplate.cst THEN
6:                CASE siTemplate.redMod OF
7:                     N way:
8:                          IF (currentCompt.csCapability.compCap IS NOT
9:                          x_active_and_y_standby) THEN
10:                              CONTUNUE TO next compt in orphanCTs
11:                         ELSEIF
12:                              RETURN currentCompt
13:                              SET found TO true
14:                              BREAK out of orphanCTs
15:                         ENDIF
```

```
16:                         2N OR N plus M:
17:                             IF (currentCompt.csCapability.compCap =
18:                             x_active OR 1_active) THEN
19:                                 CONTUNUE TO next compt in orphanCTs
20:                             ELSE
21:                                 RETURN currentCompt
22:                                 SET found TO true
23:                                 BREAK out of orphanCTs
24:                             ENDIF
25:                         OTHERS:
26:                                 RETURN currentCompt
27:                                 SET found TO true
28:                                 BREAK out of compt and orphanCTs
29:                         ENDCASE
30:                     ENDIF
31:             ENDFOR
32:     ENDFOR
33:     IF found = false THEN
34:             RETURN Null
35:     ENDIF
36: END
```

Algorithm 1 Description of "FindOrphanCT" Function.

The following are the notations used in the above and subsequent algorithms:

- *orphanCTs*: is the set of orphan component types
- *csiTemplate.cst:* is the CST specified in the CSI template.
- *compt*: is a component type.
- *currentCompt:* is the component type being read in the current iteration.
- *compt.csCapability:* describes the component type's capability model for the CST(s) it provides.
- *compt.csCapability.cst:* is a particular CST the component type supports
- *currentCompt.csCapability.compCap:* is the capability model the component type supports for a particular CST.

- Finding non orphan CT

A non orphan CT is found among the set of component types aggregated within an SU type. The same concept used to find an orphan CT is used to find a non orphan CT. However before determining whether the non orphan CT is valid or not, we should take into consideration that the number of components of the non orphan CT that could be included in a service unit may be limited by the type of this parent SU, so before selecting the component type we need to make sure that the number of components of this component type in a service unit is capable of supporting the load of CSIs of the particular CST that will be assigned to an SU of this type at runtime. The active load of CSIs is determined by the number of CSIs

in the CSI template multiplied by the SU active load of SIs of the parent SI template.

Algorithm 3.2, illustrates how the function findCT selects a component type among a set of component types aggregated by a certain SU type. The algorithm first matches the CST of the CSI template with the one of the component type (line 4) then it checks the component type capacity of serving CSIs (lines 9→12) and finally validates the component type capability model with respect to the required redundancy model according to Table 1 (lines 15, 16 and 25, 26).The input of this algorithm is an SU type *sut*, an SI template *siTemplate* and a CSI template *csiTemplate.*

```
1: BEGIN
2:      found = false
3:      FOR every compt IN sutCompts
4:           FOR every compt.csCapability IN compt
5:                IF currentCompt.csCapability.cst = csiTemplate.cst THEN
6:                     (CALL calculateSuActLoad WITH siTemp RETURNING
7:                     suActLoad)
8:                     (CALL calculateSuStandbyLoad WITH siTemplate
9:                     RETURNING StdbyLoad)
10:                    IF [(sut.Compt.maxComp *currentCompt.csCapability.maxActive >
11:                    csiTemplate.numCsi * suActLoad ) AND (sut.compt.maxComp *
```

```
12:                 currentCompt.csCapability.maxStdby>
13:                 csiTemplate.numCsi * suStdbyLoad)]THEN
14:                     CASE siTemp.redMod OF
15:                         N way:
16:                             IF (currentCompt.csCapability.compCap =
17:                             x_active_and_y_standb) THEN
18:                                 SET found TO true
19:                                 RETURN currentCompt
20:                                 BREAK from the sutCompts loop
21:                             ELSE
22:                                 GO to next compt
23:                             ENDIF
24:                         2N OR N plus M:
25:                             IF (currentCompt.csCapability.compCap
26:                             IS NOT x_active OR 1_active THEN
27:                                 SET found TO true
28:                                 RETURN currentCompt
29:                                 BREAK from the sutCompts loop
30:                             ELSE
31:                                 GO to next compt
32:                             ENDIF
33:                         OTHERS:
34:                             SET found TO true
```

```
35:                                         RETURN currentCompt
36:                                         BREAK from the sutCompts loop
37:                                 ENDCASE
38:                             ELSE
39:                                 GO to next compt
40:                             ENDIF
41:                         ENDIF
42:                 ENDFOR
43:     ENDFOR
44:     IF found = false THEN
45:             RETURN Null
46:     ENDIF
47: END
```

Algorithm 2 Description of "FindCT" Function.

The following notations as well as the ones defined before are used in the above and subsequent algorithms:

- *csiTemplate.numCSIs:* is the number of CSIs from the same template.
- *suActLoad/suStbdLoad:* are outputs of equations 3.1 and 3.2.
- *sut.compt.maxComp:* is the maximum number of components of component type *compt* an SU of a type *sut* can support.

- *Compt.csCapability.maxAct:* is the max number of CSIs of a particular CST a component of a particular type can handle in their active state.
- *Compt.csCapability.maxStdb:* is the maximum number of CSIs of a particular CST a component of a particular type can handle in their standby state.
- calculateSuActLoad/calculateSuStdbLoad: are the functions that implement equations 3.1 and 3.2
- *sutCompts*: is the set of component types that is being supported by the SU type *sut*.

It is required in certain situations discussed in the next section (3.2.2.2) to find within the SU type the component type with the highest capacity. What determines the non orphan component type active/standby capacity is the product of the maximum number of CSIs a component of this particular can support and the maximum number of components of this particular type the parent SU can support. The mentioned product is embedded into an algorithm similar to algorithm 2 (FindCT) where the component type capacity is examined, and where all the component types of the SU type are examined to select the one with the highest capability. The function that implements this algorithm is called "FindmaxCapCT". It is not illustrated in an algorithm in this thesis due to its resemblance to FindCT function however it is called in algorithm 3 (line 11).

3.2.2.2 ETF Service Unit Types Selection

A service unit type is selected to determine the type of the service units that will service a set of SIs generated from the same SI template and have the same service type. The selection of the SU type depends on the following criteria:

- The service types it provides and whether any of them matches the one specified in the SI template

- The component types this SU type aggregates, and whether those component types have the required capabilities and can support the CSIs determined by the load of SIs that will be assigned to SUs of the selected type.

In order to determine the load of SIs per SU defined in Equations 3.1 and 3.2, the number of SUs must be defined. This number can be determined by:

- **The configuration designer:** who can specify the number of SUs (active, standby) according to each of the redundancy models.

- **The redundancy model:** in case of 2N or "No redundancy" since in 2N the number of SUs with the HA state is limited to two, and in "No redundancy" the number of SUs with the HA state is equal to the number of SIs.

- **Computation:** We can compute the minimum number of SUs in situations where the configuration designer knows the services (SIs and CSIs) the system is

expected to provide but does not know the sufficient number of SUs required to support those SIs. This applies to N+M, N way and N way active redundancy models.

In the last case, we propose an algorithm that determines the minimum number of SUs required to support a defined number of SIs generated from the same SI template. In this algorithm, we investigate each SU type in a set of SU types, determine its capacity in terms of the number of SIs generated from the SI template it can support, and select the SU type that has the highest capability, and then deduce the number of SUs required.

The following algorithm finds a specific SU type for a specific SI template, the SI template is denoted by *siTemplate,* which in turn will contain CSI templates. *curremtCsiTemplate* denotes the CSI template being read in a specific iteration. Since we are looking for an SU type among a set of SU types, *actCap, stdbCap* are temporary variables that hold values that reflect the number of SIs the SU can support with respect to one set of CSIs of a certain template. However this number does not reflect the actual capability of the SU for the entire SI, since the SU may have lower capability when it comes to a different set of CSIs within the SI. The actual capacity of the SU type is reflected by *SuType.stdbCap/SuType.actCap* which is the minimum of the *actCap/stdbCap* values calculated with respect to each set of CSIs in an SI of the SI template. After determining the SU capability with respect to an SI, determining the number of required SUs can be calculated by dividing the number of SIs by the SU capacity. Since an SU cannot take two active or standby assignment for the same SI, the

required number of SUs is also dependent of the number of active/standby assignment an SI requires, for example if an SI requires four active assignments (one active and three stanbys), then the minimum number of SUs required would be four. Another issue addressed in our algorithm is that we are configuring a system with no single point of failure, and therefore we add one to the number of SUs in the redundancy models that have no standby SUs just in case one SU fails.

1: BEGIN

2: *maxCap* = max_integer_value

3: *suCap* = 0

4: *actCap* = 0

5: *stdbCap* = 0

6:*selectedSuType* = null // used to point to the SU type with the highest capability

7: FOR every su type that provides the required service type IN su type set

8: FOR every *CSI template* IN *siTemplate*

9: CALL FindMaxCT WITH *currentSuType, siTemplate currentCsiTemplate* RETURNING *compt*

10: $$actCap = \left(\frac{sut.compt.\max Comp \times compt.csCapability.\max Act}{currentCsiTemplate.numCSIs \times siTemplate.numAct} \right)$$

11: IF $siTemplate.numStdb \neq 0$ THEN // to avoid division by 0 (standbys are not required)

12: $$stdbCap = \left(\frac{sut.compt.\max Comp \times compt.csCapability.\max Stdb}{currentCsiTemplate.numCSIs \times siTemplate.numStbd} \right)$$

13: IF *maxCap* > *min(actCap, stdbCap)* THEN

14: SET *maxCap* TO *min(actCap, stdbCap)*

```
15:                 ENDIF
16:             ELSE
17:                 IF maxCap > actCap THEN
18:                     SET maxCap TO actCap
19:                 ENDIF
20:             ENDIF
21:             IF currentSuType.actCap > actCap THEN
22:                 SET currentSuType.actCap TO actCap
23:             ENDIF
24:             IF currentSuType.stdbCap > stdbCap THEN
25:                 SET currentSuType.stdbCap TO stdbCap
26:             ENDIF
27:     ENDFOR
28:     IF maxCap > suCap AND maxCap ≥ 1THEN
29:         SET selectedSuType TO currentSuType
30:         SET suCap TO maxCap
31:     ENDIF
32: ENDFOR
33: IF selectedSuType ≠ null THEN
34:     RETURN selectedSuType
35: ELSE
36:     SEND MESSAGE "No SU type was found and therefore the number of SU
37:                 cannot be determined"
```

38: ENDIF

39: CASE *SI_template.redundancy_Model* OF

40: **N+M:**

41: SET *siTemplate.numAct.SusAct* TO $ceil\left(\frac{siTemplate.numSIs}{selectedSuType.\max ActCAp}\right)$

42: IF *siTemplate.numAct.SusAct* < *siTemplate.numAct* THEN

43: SET *siTemplate.numAct.SusAct* TO *currentSiTemplate.numAct*

44: ENDIF

45: SET *siTemplate.numStdb* TO $ceil\left(\frac{siTemp.numSIs}{selectedSuType.\max StdbCAp}\right)$

46: IF *siTemplate.numSus.SusStdb* < *siTemplate.numStdb* THEN

47: SET *siTemplate.numSus.SusStdb*TO *siTemplate.numStdb*

48: ENDIF

49: **N way active:**

50: SET *siTemplate.numSu.SusAct* TO $ceil\left(\frac{siTemplate.numSIs \times siTemplate.numAct}{selectedSuType.\max ActCAp}\right)$

51: DO:

52: CALL calculateSuActLoad WITH *siTemp* RETURNING *suActLoad*

53: WHILE *selectedSuType.maxActCap* < *suActLoad*

54: INCREMENT *siTemplate.numSu.SusAct*

55: CALL calculateSuActLoad WITH *siTemp* RETURNING *suActLoad*

56: ENDDO-WHILE

57: IF *siTemplate.numSu.SusAct* < *siTemplate.numAct* THEN

58: SET *siTemplate.numSu.SusAct* TO *siTemplate.numAct*

59: ENDIF

60: SET *siTemplate.numSu.SusStdb* TO 0

61: **N way:**

62: (SET *siTemplate.numSu.SusAct* TO

63: $= ceil\left[\max\left(\left(\frac{siTemplate.numSIs}{selectedSuType.\max ActCAp}\right),\left(\frac{siTemplate.numSIs}{selectedSuType.\max StdbCAp}\right)\right)\right])$

64: DO:

65: CALL calculateSuActLoad WITH *siTemp* RETURNING *suActLoad*

66: WHILE *selectedSuType.maxActCap* < *suActLoad* DO

67: INCREMENT *siTemplate.numSu.SusAct*

68: CALL calculateSuActLoad WITH *siTemp* RETURNING *suActLoad*

69: ENDDO-WHILE

70: IF *siTemplate.numSu.SusAct* < *siTemplate.numStdb* + 1 THEN

71: SET *siTemplate.numSu.SusAct* TO *siTemplate.numStdb* + 1

72: ENDIF

73: SET *siTemplate.numSu.SusStdb* TO 0

74: ENDCASE

75: RETURN *selectedSuType*

76: END

Algorithm 3 Selecting SU Type and Calculating the Number of SUs

In case the number of SUs is already specified in the SI template, we first calculate the expected load of SIs to be assigned to an SU and then find an SU type that can handle the load and provide the required service type specified by the SI template. Algorithm 3.4 illustrates how a service unit types is selected among a set of SU types. The input of this algorithm is an SI template *siTemplate*.

```
1: BEGIN
2:      found = false
3:      FOR every sut IN sutSet
4:          FOR every st IN currentSut.Services
5:              IF st = siTemplate.st THEN
6:                  FOR every csiTemplate IN siTemplate
7:                  SET found TO false
8:                      (CALL findCT WITH siTemp, csiTemp, currentSut RETURNING
9:                      foundCt)
10:                     IF foundCt = Null THEN //this sut is not valid
11:                         CONTINUE TO next sut in the sutSet
12:                     ELSE
13:                         SET found TO true
14:                     ENDIF
15:                 ENDFOR
16:                 IF found = true THEN
17:                     RETURN sut //the sut in the current sutSet iteration
```

18: BREAK from the *sutSet* loop

19: ENDIF

20: ENDIF

21: ENDFOR

22: ENDFOR

23: IF *found* = false THEN//all the su types are not valid

24: RETURN Null

25: ENDIF

26: END

Algorithm 4 Selecting an SU Type Knowing the Number of SUs

Where:

- *sut*: is a service unit type
- *st*: is a service type
- *sutSet:* is the set of service unit types we are searching in.
- *sut.Services*: is the set of services provided by a service unit type

3.2.2.3 ETF Service Group Types Selection

A service group type is selected with respect to an SI template, for each SI template one service group type is selected if it satisfies the redundancy model specified by the user in the SI template. The SG type selected must also support a valid SU type with respect to the SI template.

3.2.2.4 ETF application Types Selection

An application type is selected if it supports an SG type that is valid to protect the set of SIs generated from a specific template.

3.2.3 AMF Type Creation

As previously mentioned, each AMF entity (expect cluster and node) have a type. Creating AMF types is accomplished in two ways:

- Creating AMF types from the selected ETF matching types. For example, an AMF SU type can be created based on a SU type that is selected from ETF, or

- Building an AMF type from scratch if we did not find the appropriate types in ETF.

The AMF types that are derived from ETF types, inherit most of the default values of the attributes defined for the ETF types how to determine the rest of the attributes is explained in Section 3.4 and 3.5.2.

3.2.3.1 Creating AMF Component Types

AMF component types are created based on the selected ETF component types. Once an ETF component type is selected to provide a certain CST, an AMF component type is created to capture all the information needed to provide this particular CST including dependencies, limitations, and capability. We carefully avoid having two components in the same SU that provide the same CST so as to make sure that the selected components provide the CST assigned the CSIs of this particular CST.

AMF component type creation is performed using a simple function called "createAmfCT" that takes as input a found ETF component type, and a CST for which this ETF type was found and creates an AMF component type that typically provides only this particular CST and inherits most of the ETF component type. Some attributes in the AMF component type are not present in the ETF component type, e.g., the instantiation level that determines which component is instantiated first. In Section 3.6, where we discuss the configuration of the attributes of the entities. The "createAmfCT" function is not defined in this thesis but it is called in Algorithm 5 (line 11).

3.2.3.2 Creating AMF SU Types

AMF SU types are created based on the selected ETF SU types (if they exist). Once an ETF SU type is selected to provide a certain service type, an AMF SU type is created to capture all the information needed to provide this particular service type, including dependencies and limitations. The AMF SU type is created in such a way that it provides only one service type. As a consequence, all SUs in the same SG will provide the same service type, which is the one specified in the SI template for which this SG will protect its SIs.

If there is no ETF SU type capable of providing the required service with the required capability, we build an SU type from orphan component types in such a way that it provides the same service type as the one required by the SI template. Algorithm 3.4 illustrates how an AMF SU type is built.

```
1: BEGIN
2:      Create a new AMF service unit type sut.
3:      ADD siTemplate.st TO sut.sutServices
4:      FOR every csiTemplate IN siTemplate
5:           (CALL findOrphanCT WITH currentCsiTemplate, siTemplate RETURNING
6:           foundCompt)
7:           IF foundCompt IS Null THEN
8:                SET sut TO Null // Delete the created SU Type
```

```
 9:                 BREAK
10:             ELSE
11:                 (CALL creatAmfCT WITH foundCompt, currentCsiTemplate.cst
12:                 RETURNING createdCompt)
13:                 ADD createdCompt TO sut.compts
14:                 SET sut. createdCompt.maxComp TO no limit
15:                 ADD sut TO createdSUTs
16:             ENDIF
17:         ENDFOR
18:         RETURN sut
19: END
```

Algorithm 5 Building AMF SU Type

3.2.3.3 Creating AMF Service Group Types

The approach used for creating SU types is also used to create service group types which can be derived from ETF SG types or built using orphan or created SU types. In this thesis, the created SG types contain only one SU type that provides one service type and given the redundancy model specified by the SI template for which the SG type is being created.

3.2.3.4 Creating AMF Application Types

Application types are created either from selected ETF application types or built from orphan or created SG types.

3.3 Creating AMF entities

The next step after determining the types to be used is to create the AMF entities including components, SUs, SGs and applications. Since these configuration objects that an integral part of the AMF configuration we are generating.

3.3.1 Creating Components

For each service unit, we need to determine the number of components to be created to support the CSIs of the SIs that are expected to be assigned to the SU hosting the components. The required number of components, *numOfComp,* to be created from a certain type to provide a particular CST with respect to a CSI template is determined in the following equation:

$$numOfComp = ceil\left\{\max\left\{\frac{\text{csiTemp.numCsi} \times \text{suActLoad}}{\text{comptCapability.maxAct}}, \frac{\text{csiTemp.numCsi} \times \text{suStdbLoad}}{\text{comptCapability.maxStdb}}\right\}\right\}$$

Equation 3-3the Number of Components of a certain type to be Created Within an SU

It should be noted that the "max" notation is used to capture the number of components required to support CSIs in both their active and standby state. For example, if five components can support all the CSIs in their active state, but ten components are required to support the standby state due to different component capability with regard to the standby assignment, then ten components must be created in every SU. The above equation provides the required number of components with respect to one CSI template, however if other CSI templates are served by the same component type the above number may increase if the number of components required to serve the CSIs of the other CSI template exceeds the one required to serve the first one.

3.3.2 Creating Service Units

SUs are created within an SG, the number of SUs to be created is specified in the SI template, or calculated as discussed in Section 3.4.2. SUs within an SG are created identical.

3.3.3 Creating SGs

SGs are created within applications; each SI template will have an SG created to protect the SIs generated from that template. The SG will have the redundancy model of the SG type, and will contain a specified/calculated number of SUs.

3.3.4 Creating Applications

The created applications can refer to one or multiple SGs, as long as the application type supports the SG types of those SGs. A cluster can contain multiple applications.

3.4 Populating the Entities' Attributes

As aforementioned Most of the attributes of the entities and their default values can be derived from the types of these entities. They can also be overridden if the configuration designer decides to do so. Some of the attributes depend on the deployment environment and have to be configured manually (e.g., the path prefix of a component, which specifies where the software of the component is located on the node).

Other attributes need to be computed. In this thesis, we limit ourselves to two important attributes that need to be calculated:

- saAmfSGMaxActiveSIsperSU attribute of an SG object, which represents the maximum active SIs per SU

- saAmfSuHostNodeOrNodeGroup attribute of an SU object which is used to determine on which node or node group the SU resides.

3.4.1 Max Active SIs per SU

When searching for an SU type, we calculate the maximum capability of this SU type and check if it can handle the load of SIs assigned to it. However, when creating the SU object, it is not necessary that we exploit the SU to full capacity; instead we opt for the least number of components that can handle the load of CSIs since on one hand at runtime it is AMF that chooses the components within an SU that will serve the CSIs, and thus putting extra components may lead to the case where some components are idle when others are at full capacity, on the other hand we don't have enough parameters for now to determine the optimum number of components within an SU, so we settle for the minimum required number. As a result, the SU actual capability must be recalculated since it may differ from the maximum capability. The following algorithm calculates the SU capability with respect to identical SIs generated from the same template.

1: BEGIN

2: *numOfComp =0*

3: *suActCap = 0*

4: *maxCap* = max_Interger_value

5: FOR every *CSI template* IN *currentSiTemplate*

6: CALL findCT WITH *currentSiTemplate, currentCsiTemplate, currentSu.sut* RETURNING *compt*

7: FOR every *component* IN *su.Components*

8: IF *currentComponent.compt* = *compt* THEN

9: INCREMENT *numOfComp*

10: ENDIF

11: ENDFOR

12: $suActCap = floor\left(\frac{numberOfComp \times compt.Capability.\max Act}{currentCsiTemplate.numCSIs}\right)$

13: IF *maxCap* > *suActCap* THEN

14: SET *maxCap* TO *suActCap*

15: ENDIF

16: ENDFOR

17: RETURN *maxCap*

18: END

Algorithm 6 Calculating Maximum Number of Active SIs per SU

Where *currentComponents* denotes the current component being read in a specific iteration, the type of this component is denoted by *currentComponent.compt. maxCap*, *suActCap* and *numOfComp* are temporary variables used to store integer values.

3.4.2 SU Host Node or Node Group

Each SU is hosted on a specific node, or on one of the nodes of a group of nodes known as node group. In this work, we assign each SU to a specific node so as to ensure load balancing regarding the number of SUs per node. This approach assumes that all nodes are identical and all SUs impose the same load on the node. The following functions

determine the node each SU is going to be assigned to, and the node group that will host the SG containing the SUs.

```
1: BEGIN
2:      assigned = false
3:      FOR every node IN the cluster
4:          IF |currentNode.hostedSus| < |firstNode.hostedSus| THEN // first node in the cluster
6:              ADD su TO currentNode.Sus
7:              ADD currentNode TO nodeGroup
8:              SET assigned TO true
9:              SET su.hostNodeOrNodeGroup TO currentNode.name
10:             BREAK out of cluster
11:         ENDIF
12:     ENDFOR
13:     IF assigned = false THEN
14:         ADD su TO firstNode.Sus
15:         ADD firstNode TO nodeGroup
16:         SET su.hostNodeOrNodeGroup TO firstNode.name
17:     ENDIF
18: END
```

Algorithm 7 Assigning SUs to Nodes, and Determining Node Groups

- The set *cluster* represents a collection of nodes.

- *currentNode* is the node being read in the current iteration
- *su.hostNodeOrNodeGroup* refers to the name of node or node group
- *nodeGroup* is a subset of the nodes of the cluster, the node group will host the SG containing our SUs
- *firstNode* is the first element is the collection of nodes

3.4.3 Ranking SUs for SIs

SU ranking for SIs is used to enable AMF to determine which SU(s) is assigned the high availability state on behalf a certain SI, among those contained in this SG.

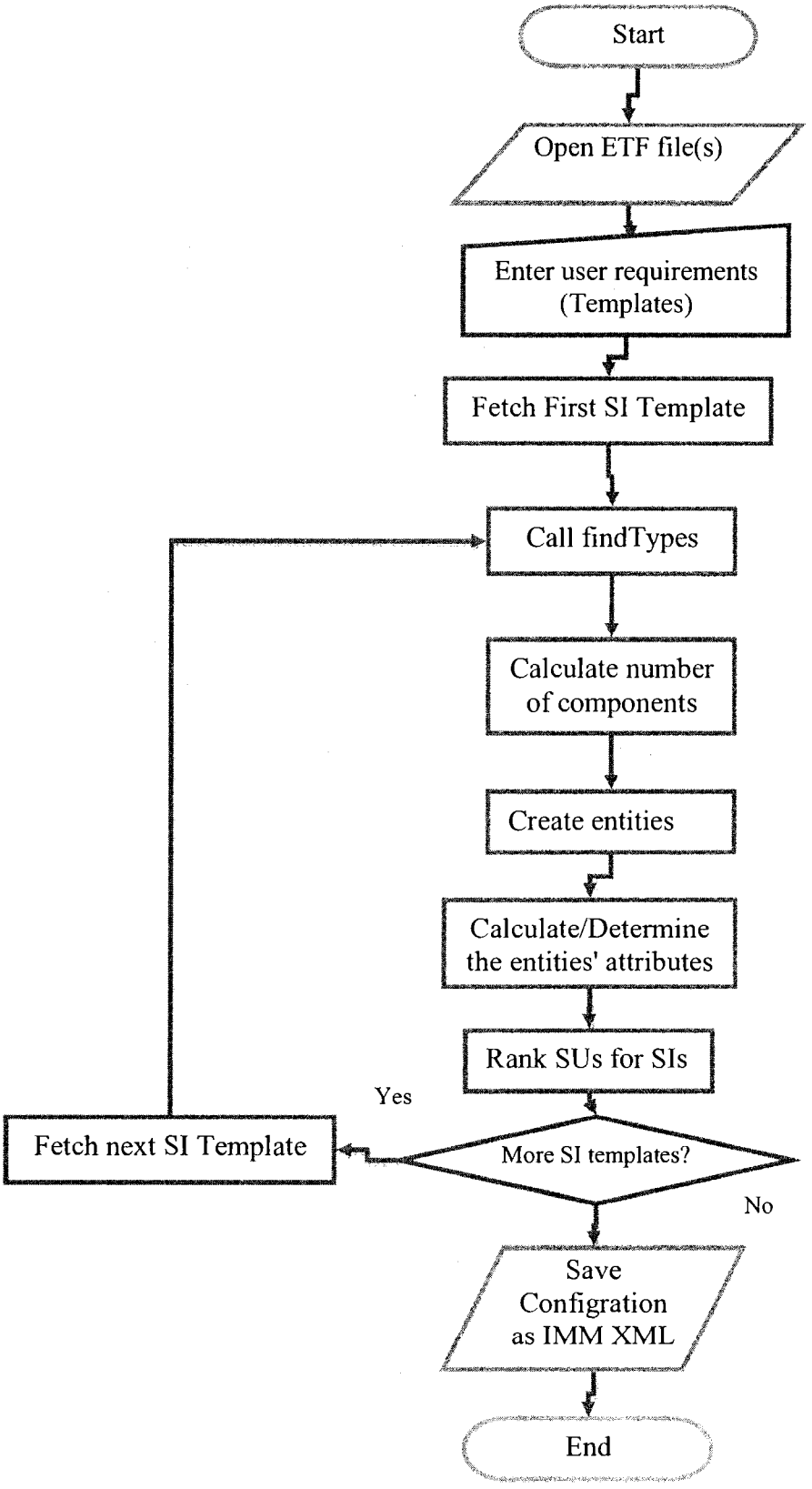
SU ranking for SIs works as follows: each SI will have a rank for every SU in the SG, the SU with the highest rank (which is the lowest value), gets assigned the active state of an SI, the SU with the second highest rank gets assigned the standby state, or the second active state depending on the redundancy model and so on.

The importance of SU ranking for SIs is that it is the only way we can balance the load distribution of SIs among the SUs of an SG. This ranking is the only attribute in AMF that allow us to express our preferences in terms of SI assignments. It should be noted that equal load of SIs among the SG's SUs has a significant impact on our process of generating a configuration, since most of our calculations are based on it (e.g. the active/standby load of SIs per SU, the number of required SUs, etc.)

Ranking of SUs could be applied to the N way, N way active and N+M redundancy models. The 2 N redundancy model we cannot balances the load among multiple SUs since we have only one active SU for all the SIs and only one standby. In a "no redundancy" redundancy model, an SU can take at most one active assignment, reducing the importance of the ranking.

3.5 Configuration Generation Processes

A configuration generation process determines the way the above steps (i.e., calculating the SU load, determining the number of SUs, selecting types and creating AMF types) are executed. We propose two approaches: bottom-up and top-down configuration generation processes. In Flowchart 1 we introduce the steps that are independent from any approach used.


Start
Open ETF file(s)
Enter user requirements (Templates)
Fetch First SI Template
Call findTypes
Calculate number of components
Create entities
Calculate/Determine the entities' attributes
Rank SUs for SIs
More SI templates?
Yes
Fetch next SI Template
No
Save Configration as IMM XML
End

Flowchart 1 General Steps in Configuration Generation

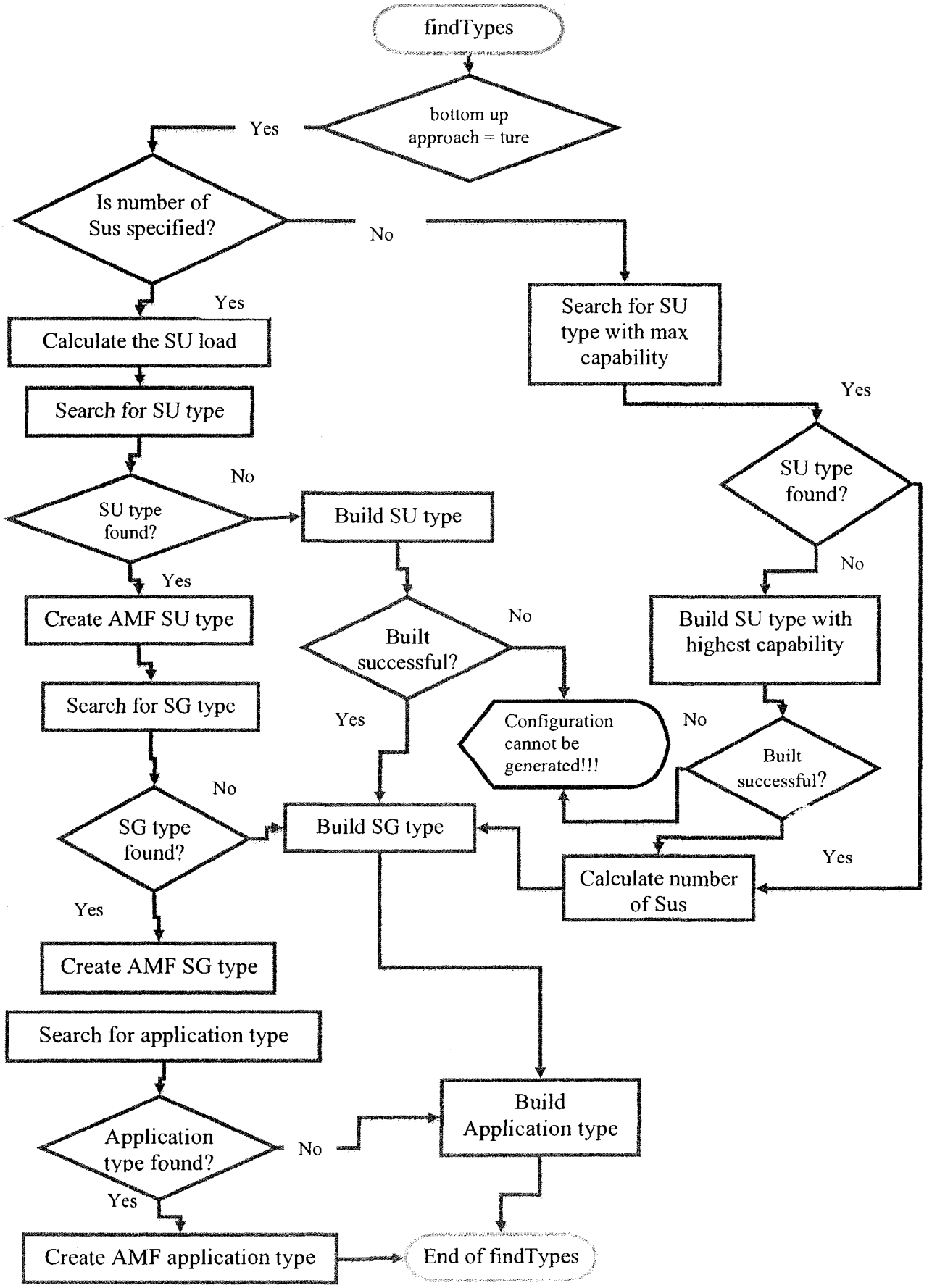
3.5.1 Bottom-Up Approach

The bottom-up approach consists of searching ETF for SU types for each SI template that provide the required service type specified in this template and that can support the SIs generated from this template. If an SU type is not found, we create one that satisfies the requirements and create the parent types as well (i.e., SG type and application type)

Next, we search for an SG type that supports the found or created SU type and that has the redundancy model specified in the SI template. If such SG type does not exist, we create an SG type that meets these requirements as well as the application type that supports this SG type. We apply the same process for an application type.

Once the types are determined, the AMF entities of these types are created and their attributes are populated just as described in previous sections.

Flowchart 2 illustrates how a configuration is generated using the bottom-up approach.


findTypes
bottom up approach = ture
Yes
Is number of Sus specified?
No
Yes
Calculate the SU load
Search for SU type
No
SU type found?
Build SU type
Yes
Create AMF SU type
Search for SG type
No
SG type found?
Build SG type
Yes
Create AMF SG type
Search for application type
Application type found?
No
Build Application type
Yes
Create AMF application type
End of findTypes
Built successful?
No
Configuration cannot be generated!!!
Yes
Search for SU type with max capability
Yes
SU type found?
No
Build SU type with highest capability
Built successful?
No
Yes
Calculate number of Sus

Flowchart 2Bottom Up Approach For Generating an AMF Configuration

The advantage of the bottom-up approach is that it allows maximum flexibility in terms of searching for SU types. If we want to inject certain preferences in the SU type selection (e.g., selecting SU types with highest capabilities, or that provide the most service types), the approach can easily accommodate these changes since the search is not bound by any constraints. In addition, we do not have the priority of searching for SU types that are supported by SG types and application types.

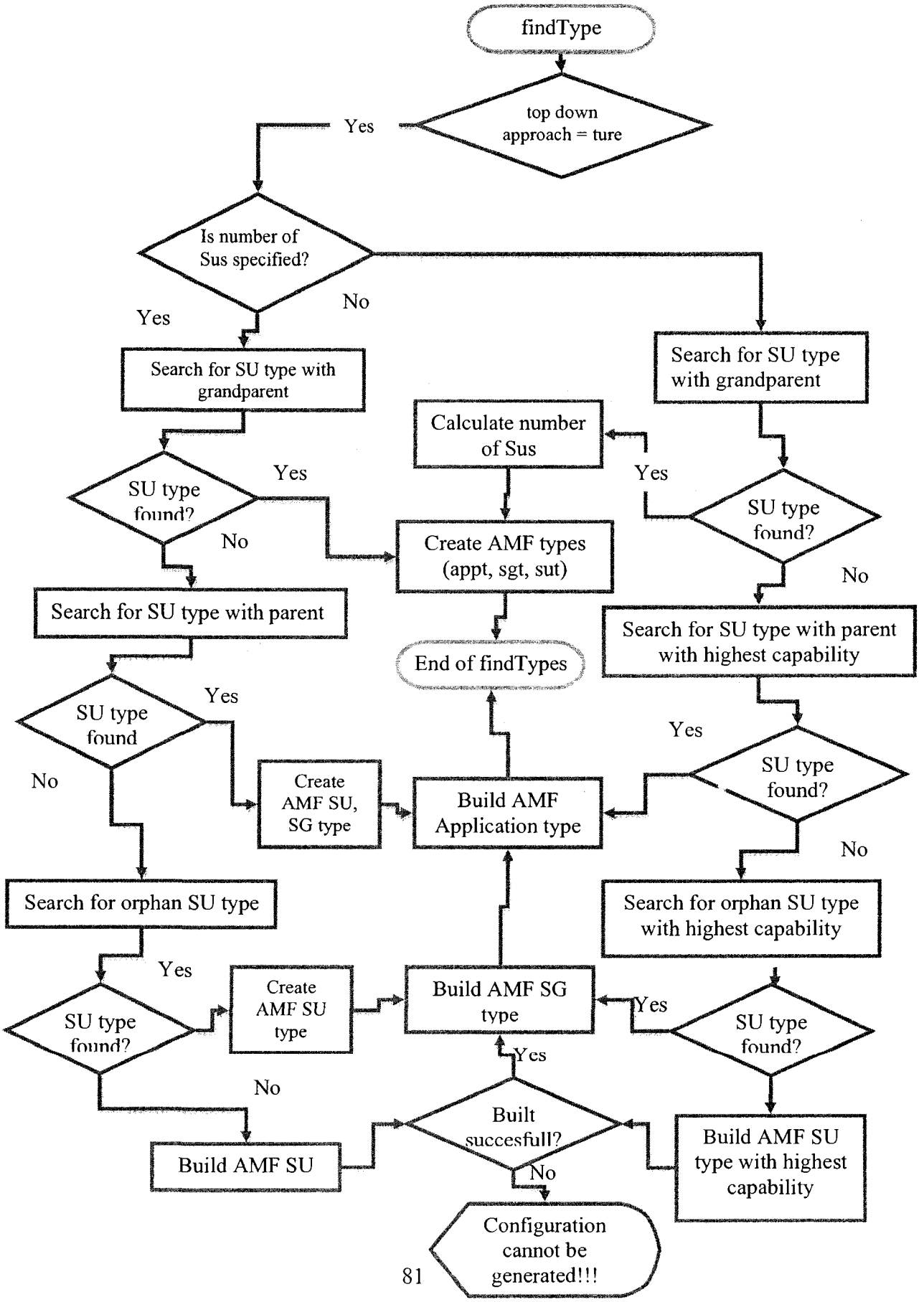
On the other hand, the disadvantage of this approach is that while creating our configuration we end up by building several AMF types, that could have been derived from ETF types had we gone for an extensive type search. The inconvenience of building AMF types is that it puts extra work on the configuration designer, since the attribute value of those types are no longer inherited from ETF type, but rather must be configured by the configuration designer. An example of such attribute is the "saAmfSgtDefSuRestartMax" of the SG type object which specifies the number of times an SU can be restarted within the service unit probation period before considering the SU faulty. As we can see, configuring such attribute requires a large knowledge of the system and the software entities, and may require testing and verification as well.

3.5.2 Top-Down Approach

The top-down approach has been introduced to make sure that selecting non orphan ETF types is given the priority over selecting the orphan ones. In other words, we only built our own AMF types after doing an extensive search of ETF types, and found that building AMF types cannot be avoided. An example to illustrate the difference of choice making in both approaches would be the case when we have two SU types that are eligible to provide a certain service type, but the first one is supported by an ETF SG type, and the second one is orphan, In the top-down approach we select the first one, since this choice minimizes the number of AMF types that needs to be built (in this case only the AMF application type must be built) however in the bottom-up approach we would have gone with either SU types, whichever was found first.

In the top-down approach, we start the type search at the application type level and we work our way down to the SU type.

Flowchart 3 illustrates how a configuration is generated using the top down approach.


findType
top down approach = ture
Yes
Is number of Sus specified?
No
Yes
Search for SU type with grandparent
Search for SU type with grandparent
Calculate number of Sus
Yes
SU type found?
Yes
No
SU type found?
Create AMF types (appt, sgt, sut)
No
Search for SU type with parent
Search for SU type with parent with highest capability
End of findTypes
SU type found
Yes
No
Yes
SU type found?
Create AMF SU, SG type
Build AMF Application type
No
Search for orphan SU type
Search for orphan SU type with highest capability
Yes
Create AMF SU type
Build AMF SG type
Yes
SU type found?
SU type found?
Yes
No
Built succesfull?
Build AMF SU
Build AMF SU type with highest capability
No
Configuration cannot be generated!!!

Flowchart 3Top-Down Approach for Generating an AMF Configuration

The advantage of the top-down approach is that we end up with a configuration where most of the attributes' values where derived from ETF types, and hence the configuration designer intervention in terms of populating those attributes is minimized.

It should be noted that in the above flowcharts, some details have been omitted to simplify the flowcharts and improve their readability. For instance, before creating a new AMF type, we need to check if there exists one that can perform the same task to avoid creating two identical types. Another issue is that we do not limit ourselves by one SI template per application In other words, if an application belongs to a type that can support the chosen SG type for an SI template; we use the same application and add a new SG.

3.6 Dependencies Among AMF's Entities and Types

Dependencies exist among various ETF types and entities at different levels. In order to generate a valid AMF configuration all those dependencies must be taken into consideration. Typical dependencies are between CSIs, SIs, component types and consequently components, SU types and consequently SUs.

3.6.1 Component Type Dependency

Dependencies among component types are either explicitly described in ETF, or implicitly deduced from the component type category, discussed in Section 2.5.1. Examples of such dependencies include the ones that exist between the container and contained component type and the proxy and proxied component type. These categories of components are out of the scope of this thesis.

The other type of dependency comes from the fact that some component types require another service provided by another component type in order to provide a certain service. Such dependency is explicitly specified in ETF.

The component type dependency is not expressed in AMF since this dependency does not affect AMF work. However, we can capture this dependency in the instantiation level of a component type, which is an attribute that specifies which components are instantiated first by AMF.

3.6.2 CSI Dependency

Part of the CSI dependency is derived from the component type dependency, since the independent component is required to perform a specific task described in a specific CSI in order to allow the dependent one to provide the required service. The other part of the dependency can be configured by the configuration designer. When the designer enters

an CSI template with a certain CST, the ETF component types can be checked to make sure that if a component type that provides this service depends on another one with a specific CSI, the configuration developer must enter this CST in a CSI template in the same SI template, since AMF states that the CSI dependency is within the containing SI.

3.6.3 SU Type Dependency

The SU type dependency is not explicitly defined in ETF as for the component type. A service unit type may require another service types provided by another SU types, in order to provide a particular service type. This type of dependency could also be thought of as service type dependency where one service type requires another service type to be provided. The dependent and independent SUs are loosely coupled in a configuration, they only need to coexist is the same cluster.

3.6.4 SI Dependency

Part of the SI dependency is derived from the SU type dependency, since the independent SU needs to provide a certain service type represented by an SI in order for the dependent SU to be able to support a certain SI. This can also be looked at from a service type dependency angle. Another part of the SI dependency can be configured by the configuration designer depending on the type of service required by the SIs. The SI

dependency is in the scope of the application, meaning the dependent and independent SI must reside in the same application.

3.7 Discussion

At several points in generating our configuration, we made various decisions that influenced the type selection and other aspects that lead to the generation of a specific configuration. For example when we check the applicability of a certain SU type, we examine its capability at its full capacity, however we could have optimized our choice by setting a certain threshold that would limit the number of SIs to be assigned to an SU. Another example is when calculating the required number of SUs, we search for SU types with highest capabilities in order to offer the minimum number of SUs required. However we could have selected SU types based on a different criterion, for example the ones that have average capability instead.

Our configuration is also limited to only one SG protecting SIs generated from the same template. However we could have split those SIs to multiple SGs if needed.

Another point is that the configurations generated are based on several assumptions made prior to the configuration such as assuming that all the SIs protected by an SG are identical, as well as assuming that they have the same priority in terms of their high availability state assignment.

In summary, the main objective of this thesis is to generate a valid AMF configuration. However, we intend in the future to investigate the generation of not only a valid configuration but also a optimal configuration. Finally, most of the work reported in this thesis was published in [21]

Chapter-4

The Configuration Generation Tool

During this research, we have developed a prototype tool that implements the algorithms presented in the previous section and therefore allows for automatic generation of AMF-compliant configurations. The tool was developed in Java, using the Eclipse environment. It is anticipated to make the tool as an Eclipse plug-in, so as to take full advantage of the various capabilities of the Eclipse integrated environment.

4.1 Description of the Tool

Figure 4.1 shows the overall flow of information in the prototype tool. A Graphical User Interface (GUI) allows the user to create and modify the various objects of the AMF model, needed to generate a configuration (e.g., SIs, CSIs, etc.) it also provides the user

with the necessary features to create, display and save a configuration. Our internal object model is based on the AMF information model described in the AMF specification [2], the ETF object model derived from the ETF schema [7], as well as additional utilities that have been created to map the entity types defined in ETF to the ones defined in AMF, and the templates populated by the configuration developer.

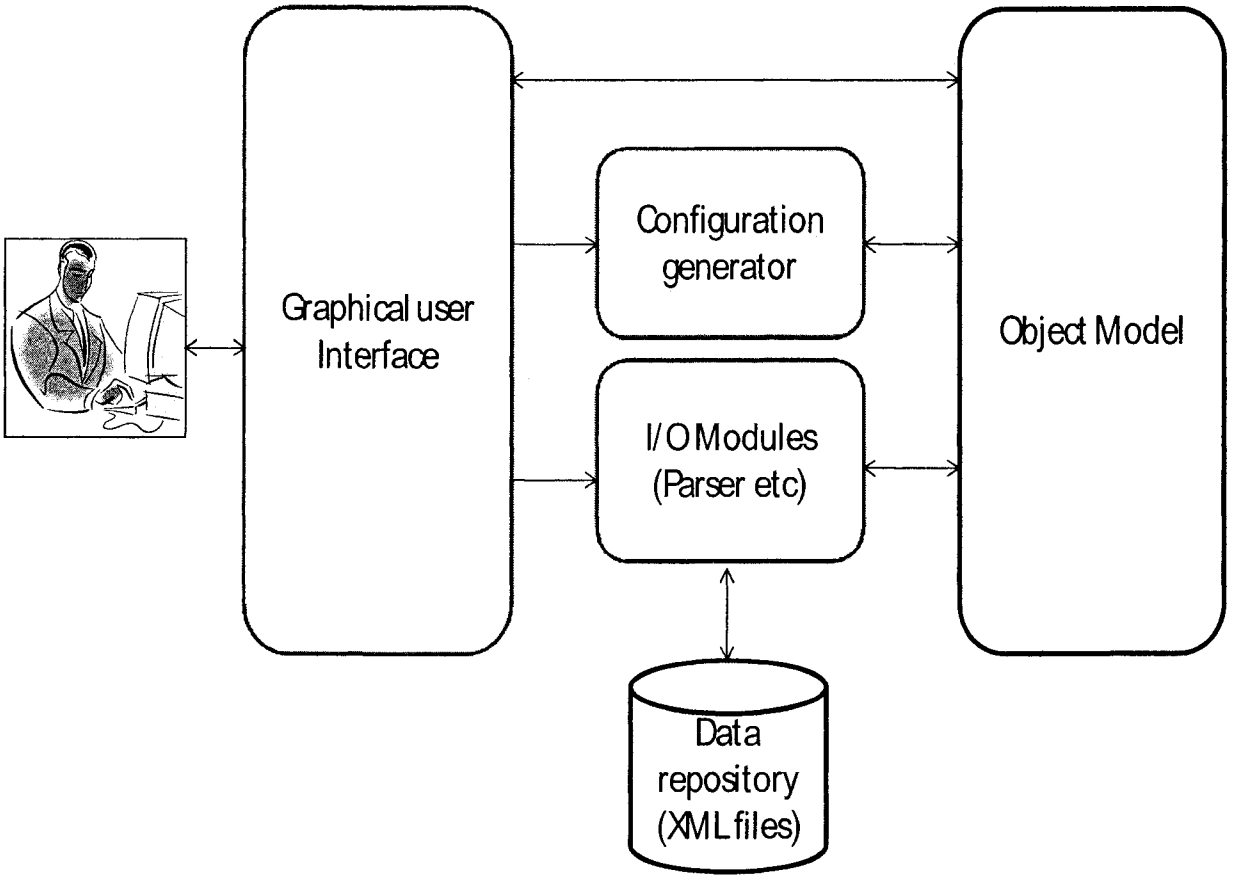


Figure 4-1The Prototype Tool Data Flow Diagram

The configuration generator component encompasses the configuration generation algorithms presented in the previous section which can be summarized in what follows:

- The type finder algorithm responsible for finding ETF types
- The type creator algorithm responsible for creating/building AMF types
- The calculation algorithm responsible for calculating the load of SUs, the required number of components, etc.
- The configuration generation algorithm that orchestrates the above algorithms to generate an AMF configuration.

The Input/Output component contains functions that save a configuration into an IMM XML. It also contains functions that read ETF and extract information from it. For this purpose, an ETF parser has been created. In addition to this, we created a data converter to convert the data represented by the AMF objects into IMM XML. We use the data repository to store the data necessary for generating configurations as well as the IMM XML

4.2 The Prototype Tool User Interface

A snapshot of the prototype tool graphical user interface is shown in Figure 4.2; it consists of four views: The Input view (the left pane), the Attribute view (the middle pane), the AMF Instance view (the upper-right pane). They are used to present the content of the object model from different aspects. And of course we have the menu that include the control elements of the tool that enable the user to open/save files, create entities/types, generate configurations etc.

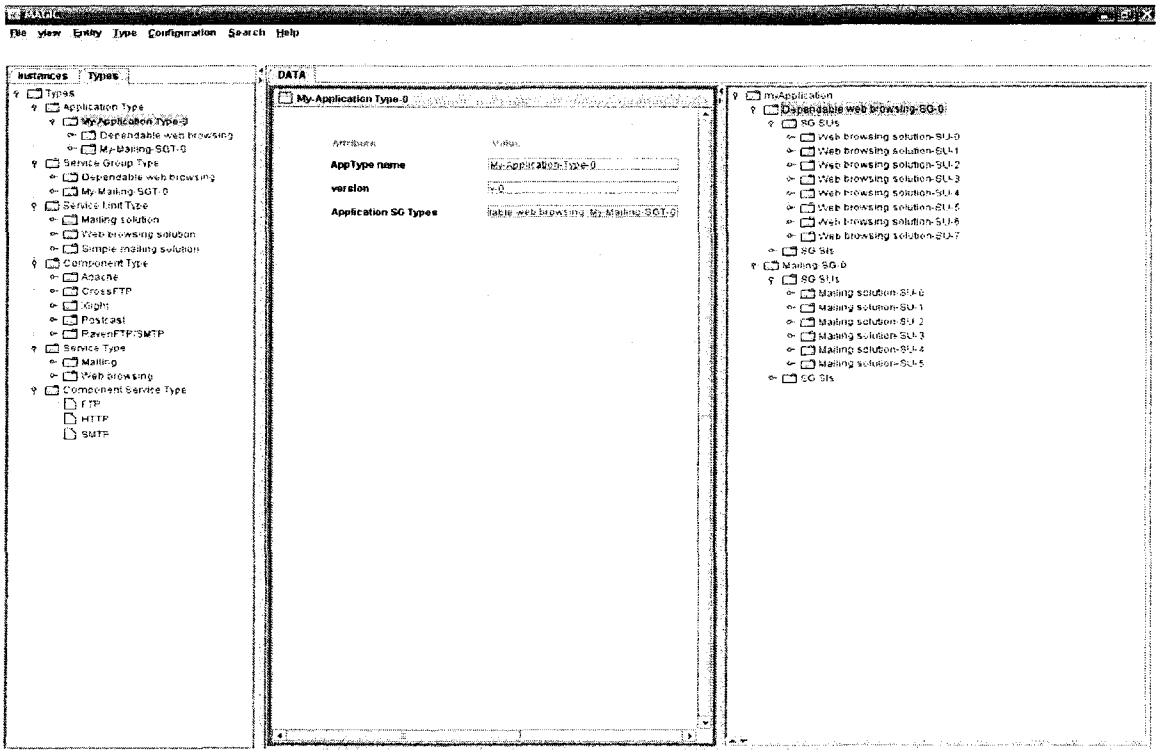


Figure 4-2 Snapshot of the Prototype Tool User Interface

The primary role of the Input view is to receive the input data for the configuration generation. It consists of two tabs: The Instances tab and the Types tab. Under the Types tab the AMF entity types read from ETF are presented to the configuration designer. We also allow the configuration designer to add new types that are not present in ETF. The Instances tab is used to allow a configuration designer to create SI and CSI templates together with the non-typed entities (i.e., cluster and node templates).

The AMF Instance view (right-upper pane) is based on the structure of the AMF Instance view defined in the specification [2]. Once the configuration has been generated, the Instance tab will contain all the entities involved in a configuration including the ones

specified by the designer (e.g., SIs, CSIs, etc.) and the entities created by the configuration generation algorithm (e.g., the components).

The Attribute view is used to display the attributes of the different objects selected either from the Input view or the AMF Instance view which ever selection is most recent.

4.3 Application Example

In this section, we demonstrate the generation of an AMF configuration with the prototype tool using the top down approach on a simple example. It is to deploy a Web service application that provides e-mail services using FTP and SMTP protocols, as well as web browsing services using HTTP and FTP protocols

The ETF file contains the following component service types: HTTP-CST, and SMTP-CST and FTP-CST. It also contains the components types: Apache, Cross FTP, Raven SMTP/FTP, and Xlight. The ETF file also describes the SG type "Dependable web Browsing" supporting one SU type "Web browsing solution" which in turn supports the Apache and Cross FTP component types. Figure 4.3 illustrate the constraints captured by this SG type.

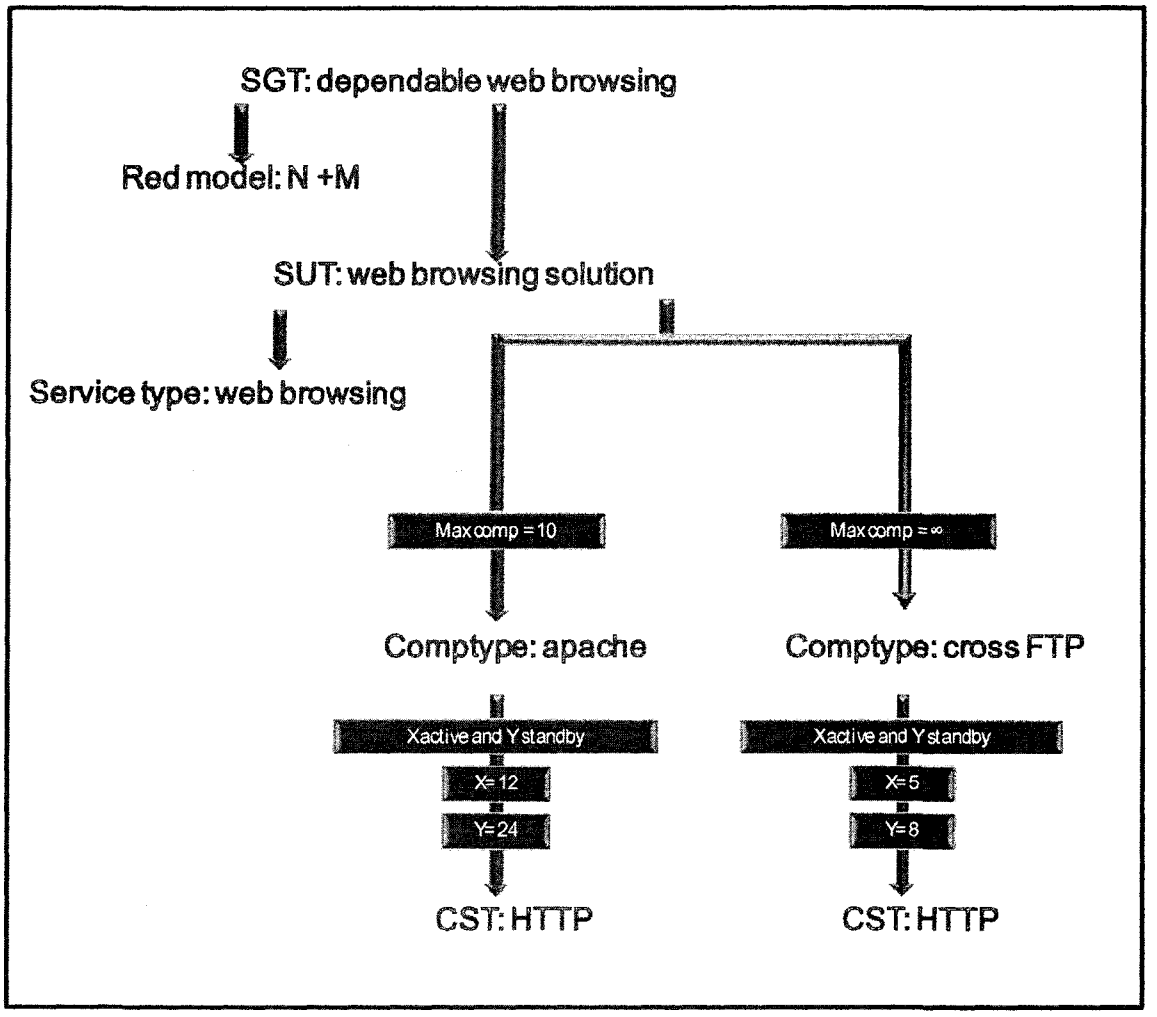


Figure 4-3 SG Type Described in ETF for the Example Application

Two orphan SU types are also defined in ETF, the "Mailing solution" SU type that composed of Postcast and Xlight component types, and the "Simple mailing solution" that composed of the Raven component type. The two SU types are described in Figure 4.4

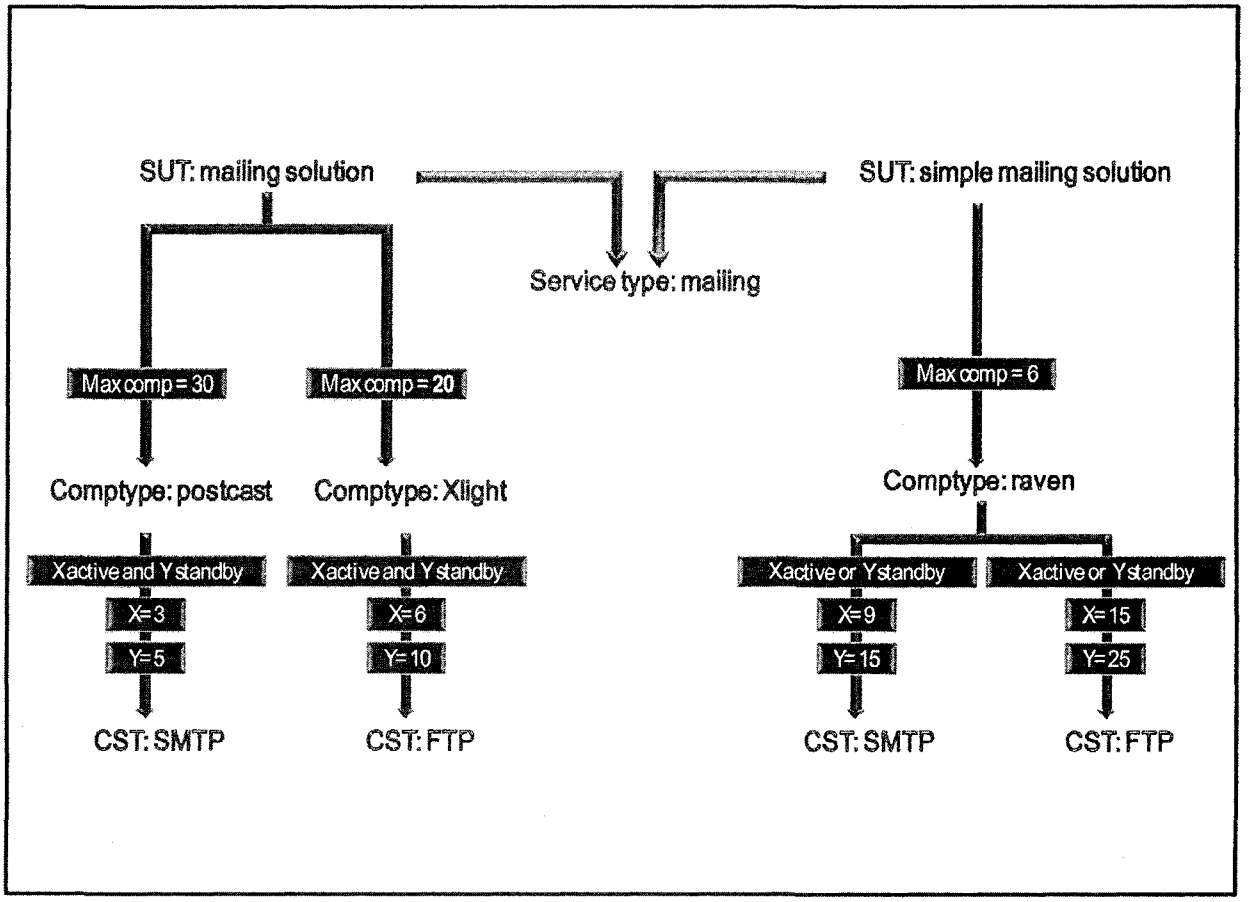


Figure 4-4SU Types Described in ETF for the Sample Application

The service types for this example are provided by ETF and each of them consists of two CSTs. Figure 4.5 shows two service types:

- "web browsing" that consists of two CSTs: FTP and HTTP, and
- "mailing" that consist of two CSTs: FTP and SMTP.

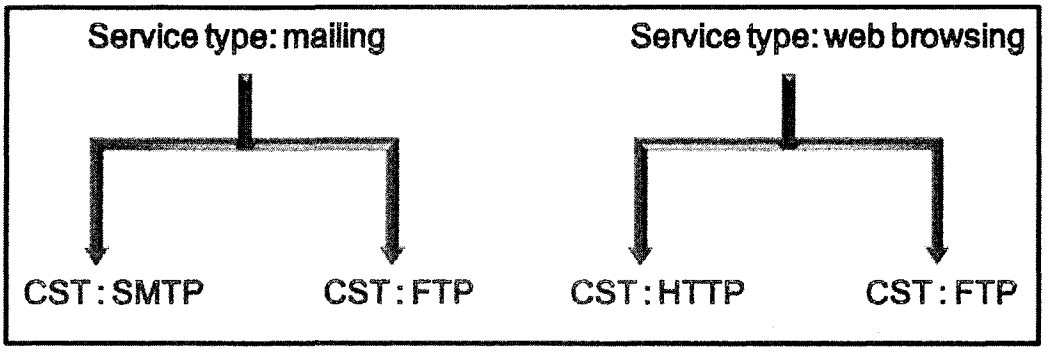


Figure 4-5 Service Types Described in ETF for the Sample Application

As discussed in Chapter 3, component service instances and service instances are created by the configuration designer using templates. We suppose that the user specifies the two SI templates "mailing services" and "web browsing services" as described in Figure 4.6, the "mailing services" template is configured for N way redundancy model, where its SI will have two standby assignments, and the number of SUs is undefined. The "web browsing services" Template is configured for N + M redundancy model with 5 active SUs and 3 standbys.

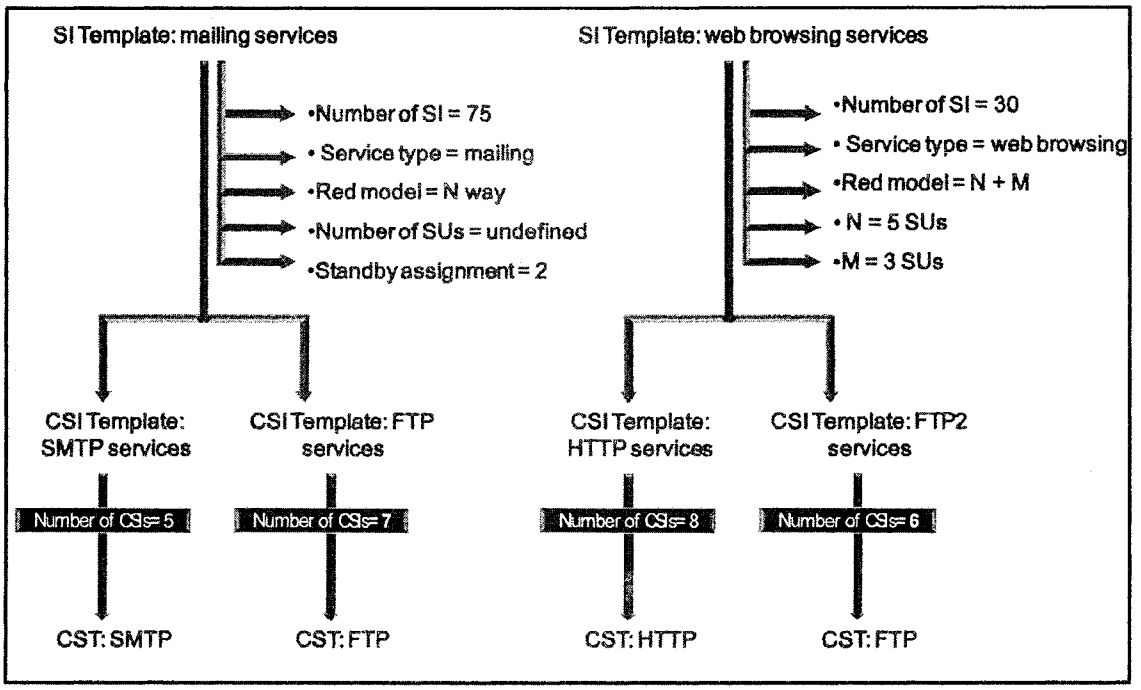


Figure 4-6 User Requirements Described in SI Templates.

Next, the cluster's configuration is entered by the designer. This includes the number of nodes in the cluster, fail over probation, etc.

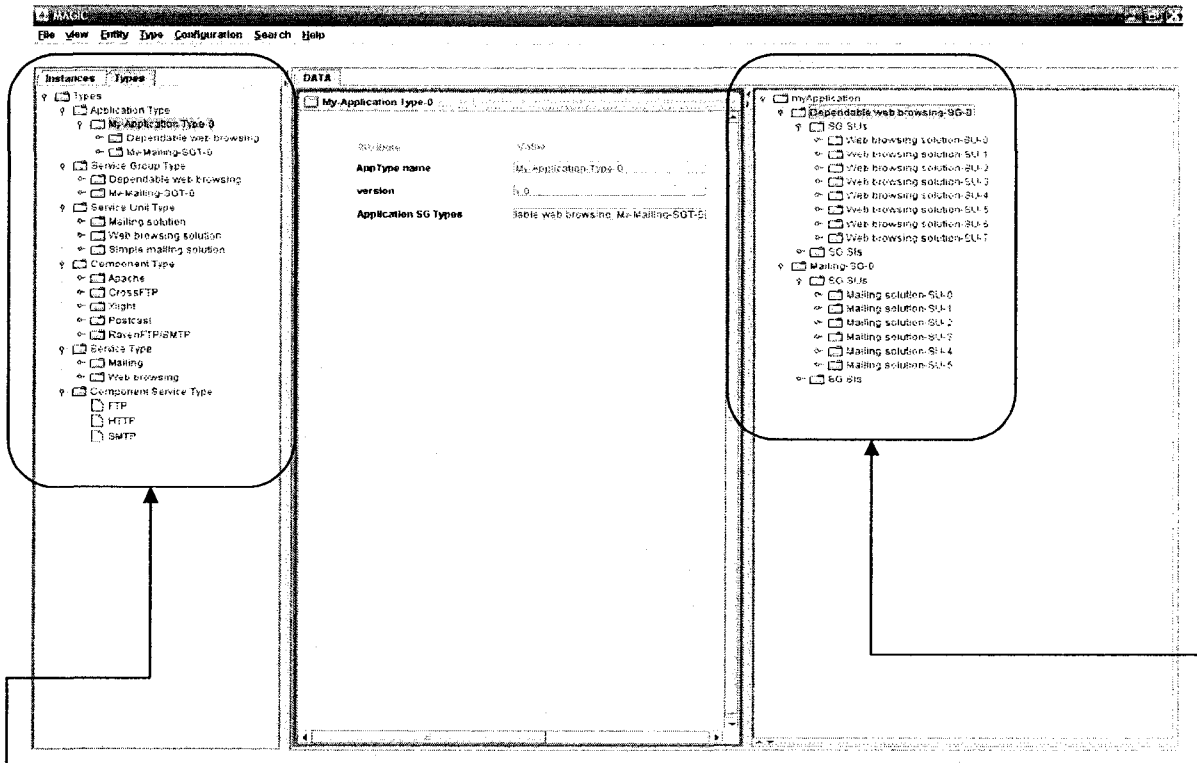


Figure 4.7.Snapshot of the configuration generated

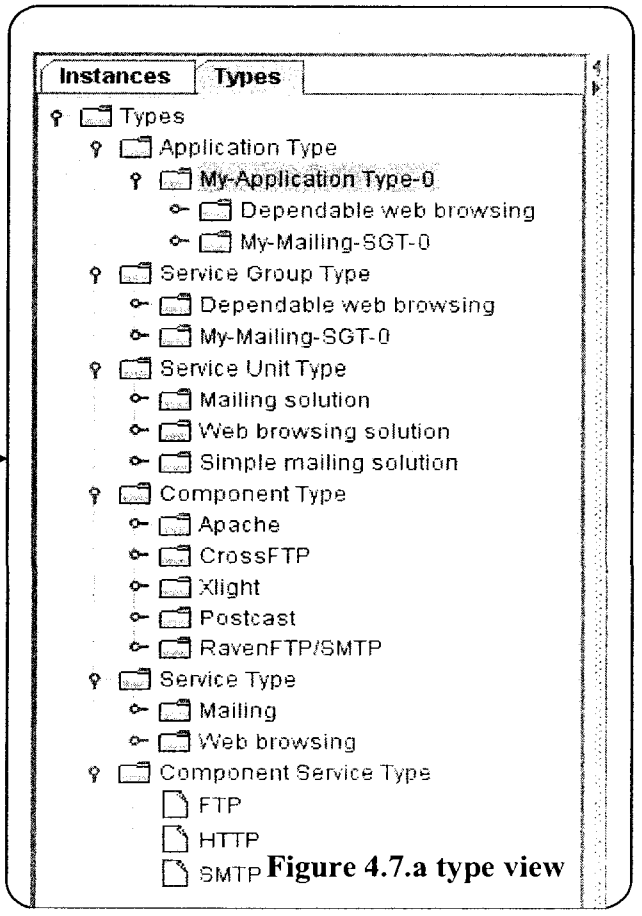


Figure 4.7.a type view

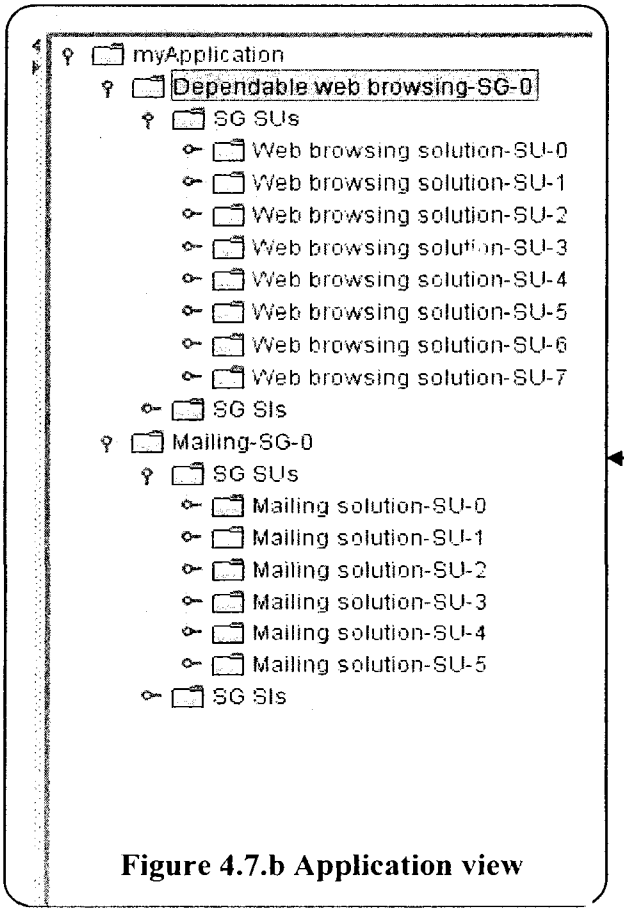


Figure 4.7.b Application view

The configuration that satisfies the above user and ETF requirements is shown in Figure 4.7. It contains one application that has two SGs (Mailing-SG-0 and Dependable web browsing-SG-0) with respectively 8 SUs (Web browsing solution-SU-0, ..., Web browsing solution-SU-7) as requested for the redundancy model in the SI template mailing services (5 plus 3), and 6 SUs in a N way redundancy model needed to support the SIs of the SI template "web browsing services". The details of this configuration are discussed in section 4.3.1. The configuration is saved by the tool as an XML file using the IMM XML schema. The size of the generated file for this simple configuration is around 213 KBs. Real life systems will require configuration files many times bigger.

4.3.1 Discussion

We start our discussion by examining the SI templates, for each SI template we discuss the choice of types and the number of instances of those types.

- SI template "web browsing services": for this SI template, the redundancy model required is N+M, the only SG type that supports this kind of redundancy is the SG type "dependable web browsing". The service type of the SIs of this template is "web browsing", the SU type that provides this service type within the above SG type is the SU type "web browsing solution". After finding an SU type that provides the required service type, we need to examine its capability in order to determine whether it can handle the load of SIs to be assigned to an SU of this type. Using equation 3.1, the active load of SIs is:

$$redMod \equiv nplusm \Rightarrow \text{ceil}\left(\frac{siTemp.numSIs}{siTemp.numSUs.susAct - 1}\right) = \text{ceil}\left(\frac{30}{5-1}\right) = \text{ceil}\ (7.5) = 8$$

Using equation 3.2, the standby load of SIs is:

$$redMod \equiv nplusm \Rightarrow \text{ceil}\left(\frac{siTemp.numSIs}{siTemp.numSUs.susStdb}\right) = \text{ceil}\left(\frac{30}{3}\right) = 10$$

The above result is used to find an SU type using Algorithm 4, which at this points validates the "web browsing solution" SU type to be used as the type of the SUs that will support the SIs of the above template. Since it provides the required service type and has enough capability to support the load of SIs

The next step is to determine the number of required components of each component type. By matching the CST of the CSI templates and the CST of the component types we can determine which component type will be used for each CSI template.

The component type used to support the CSI of the "HTTP services" CSI template is Apache. Using equation 3.3, the number of components of type Apache is:

$$numOfComp = ceil\left\{\max\left\{\frac{\text{csiTemp.numCsi} \times \text{suActLoad}}{\text{comptCapability.maxAct}}, \frac{\text{csiTemp.numCsi} \times \text{suStdbLoad}}{\text{comptCapability.maxAct}}\right\}\right\}$$

= CEIL (MAX ((8*8)/12),(8*10)/24)) = CEIL(MAX (5.33,3.33))= 6 components

Using the same equation, the number of components of type CrossFTP which will support the CSI of the "FTP service" CSI template is:

= CEIL (MAX ((6*8)/5),(6*10)/8)) = CEIL(MAX (9.6,7.5))= 10 components.

Figure 4.8.a and 4.8.b is taken by extending the SU item. As we can see the "web browsing solution" SUs have each 6 Apache components and 10 CrossFTP components

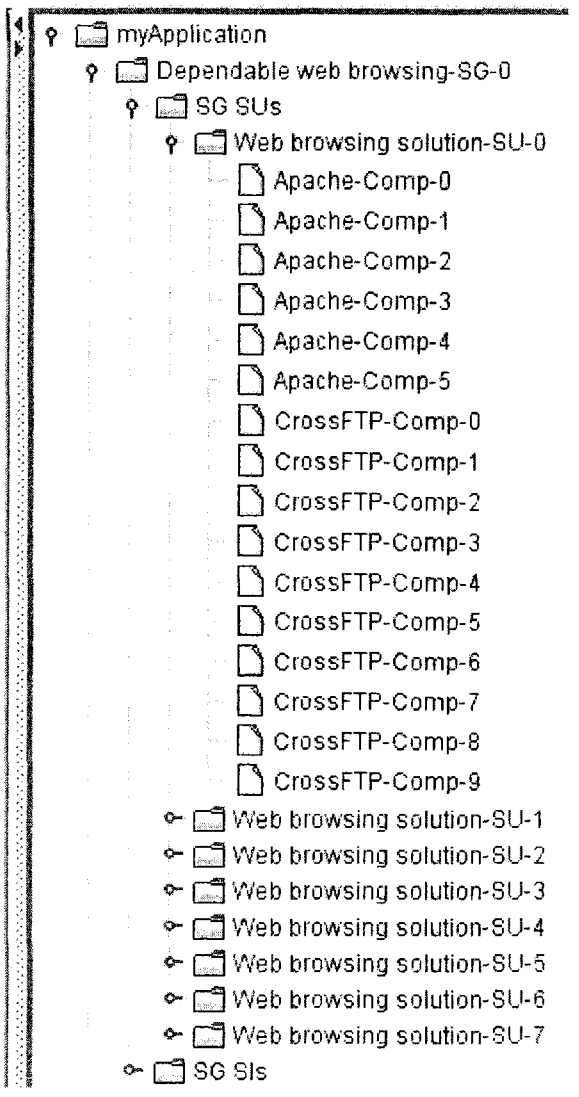


Figure 4.8 a. SG of type Dependable web browsing.

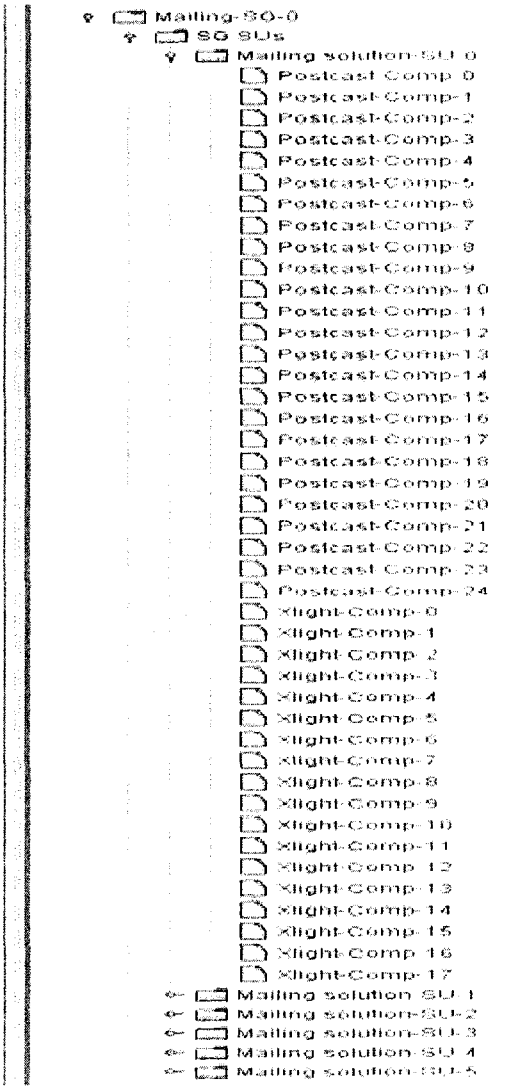


Figure 4.8 b. SG of type My-mailing-SGT-0

- SI template "mailing services": for this SI template, the redundancy model specified is N way, among the SG types provided; none of them supports this kind of redundancy, so an SG type must be created. Since the number of SUs is not specified in this SI template it must be calculated. The first step in calculating the minimum required number of SUs begins by finding the SU type that provides the "mailing" service type with highest capability of serving the SIs of our template. As discussed when presenting Algorithm 3.3, the capability of an SU type is

determined by the capability of each of its component types to serve CSIs of a particular SI template. Therefore, the active and standby capability must be calculated with respect to every selected component type in the SU type. The component type that will support the CSIs with "SMTP" CST is Postcast since it provides the "SMTP" CST and as we will see its capability allows it to support the CSIs that are expected to be assigned to it. The active capability of this component type is calculated using the equation from Algorithm 3

$$actCap=\left(\frac{sut.compt.\max Comp \times compt.csCapability.\max Act}{currentCsiTemplate.numCSIs}\right)=\left(\frac{30\times 3}{5}\right)=18$$

The standby capability of this component type is calculated using the following equation from Algorithm 3.

$$stdbCap=\left(\frac{sut.compt.\max Comp \times compt.csCapability.\max Stdb}{currentCsiTemplate.numCSIs}\right)=\left(\frac{30\times 5}{5}\right)=30$$

Using the same reasoning with the Xlight component type the active capability of the SU type in providing FTP CST would be:

$$actCap=\left(\frac{sut.compt.\max Comp \times compt.csCapability.\max Act}{currentCsiTemplate.numCSIs}\right)=\left(\frac{20\times 6}{7}\right)=17.4$$

The standby capability of this component type is calculated using the following equation from algorithm 4.

$$stdbCap=\left(\frac{sut.compt.\max Comp \times compt.csCapability.\max Stdb}{currentCsiTemplate.numCSIs}\right)=\left(\frac{20\times 10}{7}\right)=28.5$$

By taking the minimum of the active and standby capabilities of the SU type with respect to each of the CSI templates, the active capability of the SU is 17.4 and the standby capability is 28.5 SIs/SU.

The number of required SUs of the above SU type is determined by the following equation from Algorithm 3

$$ceil\left[\max\left(\left(\frac{currentSiTemp.numSIs}{selectedSuType.\max ActCAp}\right),\left(\frac{currentSiTemp.numSIs\times currentSiTemplate.numStdb}{selectedSuType.\max StdbCAp}\right)\right)\right]$$

$$=ceil\left[\max\left(\left(\frac{75}{17.14}\right),\left(\frac{75\times 2}{28.5}\right)\right)\right]=ceil[\max((4.37),(5.26))]=6 \text{ SUs}$$

Knowing the number of SUs will allow us to calculate the load of SIs per SU and determine the number of components required.

The active load of an SU is calculated using equation 3.1:

$$redMod \equiv nway \Rightarrow \text{ceil}\left(\frac{siTemp.numSIs}{siTemp.numSUs.sus - 1}\right) = \text{ceil}\left(\frac{75}{6-1}\right) = 15 \text{ SIs/SU}$$

The standby load of an SU is calculated using equation 3.1.:

$$redMod \equiv nway \Rightarrow \text{ceil}\left(\frac{siTemp.numSIs \times siTemp.numStdb}{siTemp.numSUs.sus}\right)$$

$$= \text{ceil}\left(\frac{75 * 2}{6}\right) = 25 \text{ SIs/SU}$$

The number of components to be created of each component type is calculated using equation 3.3. The number of components of Postcast is:

= CEIL (MAX ((5*15)/3),(5*25)/5)) = 25 components.

Using the same equation, the number of components of type Xlight which will be supporting the CSI of the "FTP2 service" CSI template is:

= CEIL (MAX ((7*15)/6),(7*25)/10)) = CEIL (17.5) = 18 components.

Note that the "simple mailing solution" SU type was not used because it has a lower capability than the "mailing solution", and therefore it would have required a larger number of SUs.

Two types had to be built during the course of generating the above configuration, the first type is the SG type "My-Mailing-SGT-0" built to adopt the "mailing solution" SU type, since this SU type did not have any parent, and an application type named "My-Application type-0" to adopt both "My-Mailing-SGT-0"SG type and "dependable web browsing" since they did not belong to any application type.

4.4 Conclusion

In this chapter, we presented a prototype tool that implements the configuration generation functions presented in this thesis. The tool is still in its evolving stage, since many new ideas and algorithms will be integrated into the tool as research in this area progresses.

Chapter-5

Conclusion

In this chapter, we conclude our thesis by discussing the main research contributions in Section 5.1. In Section 5.2, we elaborate on opportunities for future research to further improve the current approach. Finally in Section 5.3 we provide our closing remarks for this thesis.

5.1 Research Contributions

In this thesis, we presented our approach for generating automatically AMF compliant configurations from a set of entity types provided by the software vendor and from the configuration designer requirements, which include the service to be provided, its protection level indicating the redundancy model and the system to be deployed on. Our approach consists of sequence of steps and functions to be implemented; however the

order of implementation is described in two algorithms (bottom-up and top-down) that differ in the way ETF types are selected. The top-down approach starts by analyzing the application types, followed by SG types, and then SU types. The bottom-up approach, on the other hand, starts by analyzing SU types, then SG types, and finally application types. The algorithms share common functions for finding and creating types. The bottom-up approach had more flexibility in terms of types selection and creation but did not take all of ETF constraints into consideration, whereas the top-down approach was biased towards choosing the types that are most bounded by ETF constraints and types were only built as a last resort if no ETF type was found suitable.

Both approaches have been designed to integrate directly into the generation process a certain number of configuration decisions and constraints that come from the configuration designer as well as the software vendor of the application. Nevertheless, for a particular set of services and user requirements each approach could result in a different configuration. This is due to the fact that different decision choices were implemented in each approach.

In addition to this, we developed an Eclipse-based prototype tool that can be used by configuration designers to manipulate AMF entities, read ETF files, generate automatically AMF configurations, and save them as IMM XML files. To our knowledge, this is the first tool that is created for this purpose.

5.2 Opportunities for Further Research

A natural extension of this research is to work towards full automation of the configuration generation process by minimizing the input data that needs to be provided by the configuration designer.

Since the main concern of the configuration designer is the availability level of an application, we can, for example, limit the input data to the services and the level of availability required for those services. In order to achieve this goal, we need to investigate techniques that would allow us to evaluate the level of availability a particular configuration provides for an application. This level of availability is affected by many factors including the mean time between component failure and its correlation with the service load of the component, the time to recover the service when a switch or fail over occurs, the time to repair the faulty components, etc.

In the current approach, we consider the generation of only one AMF compliant configuration. This one configuration is created based on the strategy implemented in the generation algorithms during the selection or creation of different types, such as component types, SU types or SG types. However, different strategies can lead to different configurations with a choice of alternative component types, SU types, or SG types. The criteria of selecting the types can change due to changing the preference for some of the existing attributes or in the future in case further description of the types are provided, such as the resources required by each entity of a specific type, or the mean

time between failure for components, licensing cost, etc. Having these attributes will allow us to generate multiple configurations according to different criteria and thus exploring a wider space of configurations and choosing the one that best suits the environment of deployment and the designer requirements.

5.3 Closing Remarks

High availability is perhaps one of the most important requirements for building reliable communication systems. Managing availability through a standardized middleware, such as SAF middleware, simplifies and enhances the development of such applications. The part of SAF middleware responsible of managing availability, AMF, requires a certain configuration of the applications before they can be deployed on AMF. Generating such a configuration can be a tedious, and sometimes an impossible, task if performed manually. We believe that the configuration generation algorithms presented in this thesis and the tool that implements these algorithms will greatly facilitate the configuration generation process by relieving configuration designers from the complexity of handling AMF and ETF elements and the constraints associated with them.

Bibliography

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

2. Service Availability Forum, Application Interface Specification. Availability ManagementFrameworkSAI-AIS-AMF-B.03.01. http://www.saforum.org/specification/download/

3. Service Availability Forum, Application Interface Specification. Information Model Management Service SAI-AIS-IMM-A.02.01. http://www.saforum.org/specification/download/

4. eXtensible Markup Language (XML) at http://xml.org

5. Unified Modeling Language (UML) http://www.uml.org

6. Service Availability, Forum. Application Interface Specification. Software Management Framework SAI-AIS-SMF-A.01.01. http://www.saforum.org/specification/download/

7. SAI-AIS-SMF-ETF-A.01.01.xsd. (ETF schema describing the software bundle and the entity types' relations and features.)

http://www.saforum.org/specification/download/

8. SAI-XMI-A.02.00.09.18.xml.zip. (UML class diagram describing the AMF classes, their attributes and their relations.) http://www.saforum.org/specification/download/

9. SAI-AIS-IMM-XSD-A.01.01.xsd. (IMM schema describing the format in which objects must be saved) http://www.saforum.org/specification/download/

10. A. Kövi, D. Varró, "An Eclipse-Based Framework for AIS Service Configurations", *In Proc. of the International Service Availability Symposium (ISAS), LNCS Vol.4526*, pp. 110-126, Durham, NH, 2007.

11. T. Hinrich, N. Love, C. Petrie, L. Ramshaw, A. Sahai, S. Singhal, "Using Object-Oriented Constraint Satisfaction for automated Configuration Generation", *In Proc. of the 15th IFIP/IEEE International Workshop on Distributed Systems Operations and Management (DSOM), LNCS Vol. 3278*, pp.159-170, Davis, CA, 2004.

12. A. Sahai, S. Singhal, R. Joshi, V. Machiraju, "Automated Generation of Resource Configurations through Policies", *In Proc. of the 5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'04), Yorktown Heights, New York*, 2004.

13. G. Janakiraman, J. R. Santos, Y. Turner, "Automated Multi-Tier System Design for Service Availability", HP Laboratories Palo Alto, HPL-2003-109, May 22nd, 2003. http://www.hpl.hp.com/techreports/2003/HPL-2003-109.pdf

14. OPENAIS, URL: http://www.openais.org

15. OPENSAF, URL: http://www.opensaf.org/

16. OPENCLOVIS, URL: http://www.openclovis.org/

17. Service Availability Forum, Standards for a Service Availability™ Solution, http://www.saforum.org

18. W. Vogels, "*The Design and Architecture of the Microsoft Cluster Service-A Practical Approach to High- Availability and Scalability,*" Proc. 28th Symposium on Fault-Tolerant Computing, CS Press, 1998, pp. 422-431.

19. Chris Oggerino (2001). *High availability network fundamentals: a practical guide to predicting network availability*. Indianapolis: Cisco Press. 250.

20. R. Gamache, R Short, and Mike Massa, "Windows NT Clustering Service," *IEEE Computer*, October 1998, pp. 55-61.

21. A. Kanso, M. Toeroe, F. Khendek, A. Hamou-Lhadj, "Automatic Generation of AMF Compliant Configurations" *In Proc. of the International Service Availability Symposium (ISAS), LNCS Vol.5017*, pp. 155-170 Tokyo, Japan, 2008.

22. M. Reitenspiess,., "High Availability and standard interfaces – the way to go! Boards and Solutions", 2004, URL: http://www.embedded-control-europe.com/pdf/basapr04p34.pdf.

23. K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability and Programmability*, McGraw-Hill, 1993.

24. V.G. Kulkarni. *Modeling and Analysis of Stochastic Systems*, Chapman-Hall, New York, 1995.

25. E.D. Lazowska, J. Jzahorjan, G.S. Graham and K.C. Sevcik. *Quantitative System Performance: computer System Analysis Using Queueing Network Models*, Prentice-Hall, NJ, 1984.

26. J. Muppala, S. Woolet, B.R. Haverkort and K.S. Trivedi, "Composite Performance and Dependability Analysis" *Journal of Performance Evaluation*, Vol. 14, pp. 197-216, 1992.

27. G.F. Pfister, *In Search of Clusters: The Coming Battle in Lowly Parallel Computing*, Prentice Hall, Englewood Cliffs, NJ, 1998.

28. T. Robertazzi. *Computer Networks and Systems: Queueing Theory and Performance Evaluation*, Springer-Verlag, 1994.

29. Trivedi, Kishor S. 2002, *Probability and Statistics with Reliability, Queueing and Computer Science Applications,* Second Edition, John Wiley and Sons, New York, 848.