Automated Configuration Design and Analysis for Service High-Availability

by

Ali Kanso

A Thesis

in

The Department of Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy at

Concordia University

Montreal, Quebec, Canada

May 2012

© Ali Kanso, 2012

CONCORDIA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Ali Kanso

Entitled: Automated Configuration Design and Analysis for Service High-Availability

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

Doctor of Philosophy in Electrical and Computer Engineering

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

Signed by the final examining committee:

Dr. C. Mulligan	Chair
Dr. Dorina C. Petriu	External Examiner
Dr. J. Rilling	External to Program
Dr. O. Ait Mohamed	Examiner
Dr. M. Debbabi	Examiner
Dr. F. Khendek, Dr. M. Toeroe	Thesis Supervisor

Approved by

Chair of Department or Graduate Program Director

Dean of Faculty

ABSTRACT

Automated Configuration Design and Analysis for Service High-Availability

Ali Kanso, Ph.D.

Concordia University, 2012

The need for highly available services is ever increasing in various domains ranging from mission critical systems to transaction based ones such as online banking. The Service Availability Forum (SAForum) has defined a set of services and related Application Programming Interface (API) specifications to address the growing need of commercial-off-the-shelf high availability solutions. Among these services, the Availability Management Framework (AMF) is the service responsible for managing the high availability of the application services. To achieve this task, an AMF implementation requires a specific logical view of the organization of the application's services and components, known as an AMF configuration. Any AMF configuration must be compliant to the concepts and constraints defined in the AMF specifications. The process of defining AMF configurations is error prone and requires extensive domain knowledge. Another major issue is being able to analyze the designed AMF configuration to quantify the anticipated service availability. This requires a different set of modeling and analysis skills that system integrators might not necessarily possess. In this dissertation we propose the automation of this process. The premise is to define a generation method within which we embed the domain knowledge and the domain constraints, and by that generating AMF configurations that are valid by construction. We also define an approach for the service availability analysis of AMF configurations. Our method is based on generating an analysis stochastic model that captures the middleware behavior and the application configuration. This model is thereafter solved to quantify the service availability.

Acknowledgments

I would like to express my deepest gratitude to:

- My mother and sister for their endless support and all my family for their encouragement,
- My supervisor Dr. Maria Toeroe for helping me grow both professionally and personally through her guidance and knowledge sharing, and for offering me her friendship which means a lot to me,
- My supervisor Dr. Ferhat Khendek for believing in me, and guiding me throughout this work, as well as for being there for me whenever I needed his support in every way,
- For my colleagues (past and present) for their support and friendship,
- For all my friends that accompanied me in my journey and shared with me the good and the ‘not so good’ memories,
- For Ericsson Canada that offered me a place to learn, grow and interact with professionals that willingly shared their knowledge and expressed their support,
- For Concordia University for offering me the academic environment and the opportunity to achieve this work.

This work is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), Ericsson Research, and Concordia University.

Table of Contents

1 Introduction 1

2 Background 7

2.1 A Brief Look-Back 7

2.2 Service Availability and the SAForum Middleware 8

2.3 The Availability Management Framework 8

2.3.1 The AMF Configuration 9

2.3.2 The Dynamic Behavior of AMF 17

2.3.3 The Entity Types File 19

2.4 Availability Analysis 20

2.4.1 Deterministic and Stochastic Petri Nets 21

3 Related work 26

3.1 Configuration Generation 26

3.2 Availability Evaluation 30

4 Generating AMF Compliant Configurations 38

4.1 Overview of Creating AMF Configurations 39

4.2 Overview of Automatic Generation of AMF Configurations 40

4.3 The Configuration Requirements (CR) 41

4.3.1 Why a CR Model? 42

4.3.2 The CR Domain Model 43

4.3.3 Specifying and Analyzing Dependencies 46

4.4 Generating AMF Configurations 58

4.4.1 Determining the SI-Load an SU Is Expected to Support ... 59
4.4.2 Top-Down Type Selection Criteria ... 64
4.4.3 Creating AMF Types and Entities ... 68
4.5 From One to Multiple Configuration Generation ... 75
4.5.1 Approach for Multiple Configuration Generation ... 77
4.6 Summary and Discussion ... 78
4.6.1 Compliancy with the AMF Specifications ... 79
4.6.2 Selection of Orphan Types ... 80
4.6.3 Selection of Higher Level Types ... 80
4.6.4 Multiple Configuration Generation ... 81
5 Workload Balancing through AMF Configurations ... 82
5.1 Introduction ... 82
5.2 The Ranking Mechanism for Runtime SI Assignment ... 83
5.3 Predefined Workload and Workload Assignment: a Motivating Example ... 85
5.4 Existing Workload Balancing Solutions ... 86
5.5 Workload Balancing in NwayActive ... 88
5.5.1 The Problem with Conventional Load Balancing Algorithms at Configuration Time in NwayActive ... 88
5.5.2 A Ranking Solution to Ensure Load Balancing Before and After One Failure in NwayActive Redundancy ... 90
5.6 Workload balancing in N+M ... 102
5.6.1 The Problem with Conventional Load Balancing Algorithms at Configuration Time in N+M ... 102
5.6.2 Can We Find a Solution that Satisfies all the Requirements? ... 104
5.6.3 Ranking Solutions Targeting Workload Balancing Before and After One Failure in N+M Redundancy ... 105

5.7 Conclusions and discussion 119

6 Configuration Based Service Availability Analysis 122

6.1 Introduction 123

6.2 The Service Outage 123

6.3 The Service Recovery 124

6.3.1 Recovery Altering Attributes 127
6.3.2 Issues and Challenges 129
6.4 The Availability Analysis Framework 129

6.4.1 Extending the Standard AMF Model 130
6.4.2 Actual Recovery Analysis 131
6.4.3 Defining the Stochastic Analysis Model 137
6.4.4 Mapping the Configuration to a DSPN Model ─ Overall Process 138
6.4.5 The DSPN Templates 148
6.4.6 Availability Analysis Discussion 206
7 Tooling Framework and Case Study 209

7.1 Tooling Framework 209

7.2 The Media Streaming Case Study 211

7.3 Configuration Generation Example 215

7.4 Availability Analysis Example 223

8 Conclusion and Future Work 233

8.1 Summary of Challenges and Contributions 233

8.2 Future work 236

8.2.1 Mapping Non-Functional User Requirements to the Configuration Requirements 236

8.2.2 Designing Configurations that Satisfy the Non-Functional Requirements 237

8.2.3 Defining Ranks that Maintain the Workload Balanced After Multiple Failures 238

8.3 Closing remark 238

9 *References* 243

List of Acronyms

AIS	Application Interface Specification
AMF	Availability Management Framework
ATL	Atlas Transformation Language
CSI	Component Service Instance
CSType	Component Service Type
CR	Configuration Requirements
DSPN	Deterministic and Stochastic Petri Nets
EMF	Eclipse Modeling Framework
ETF	Entity Types File
IMM	Information Model Management
OCL	Object Constraint Language
OMG	Object Management Group
SAForum	Service Availability Forum
SG	Service Group

SGType	Service Group Type
SI	Service Instance
SU	Service Unit
SUType	Service Unit Type
SvcType	Service Type
SQL	Structured Query Language
UML	Unified Modeling Language

List of Figures

Figure 1-1 The SAForum middleware specifications 2

Figure 1-2 The use of the AMF configuration 3

Figure 2-1 An AMF configuration example 13

Figure 2-2 The types versus the entities referring to these types 14

Figure 2-3 The standardized AMF configuration model (taken from [4]) 17

Figure 2-4 A two-states Markov model 20

Figure 2-5 A DSPN example 24

Figure 2-6 Analytical results of solving the DSPN example 25

Figure 4-1 Overall view of the AMF configuration generation 41

Figure 4-2 Example OCL constraint for the SG template 45

Figure 4-3 The UML class diagram of the CR model 46

Figure 4-4 The dependency specification approach 52

Figure 4-5 A dependency relation between two sets of services 56

Figure 4-6 Configuration generation process 59

Figure 4-7 Multiple configuration generation illustration 78

Figure 5-1 An example of NwayActive SG 84

Figure 5-2 The SU back up graph using the round robin algorithm 90

Figure 5-3 A mapping from the assignment table row to a ranked sub-list. 101

Figure 5-4 A snapshot of our ranked list of SUs. 102

Figure 5-5 The balanced assignment tables 113

Figure 5-6 The process of deriving the ranked list of SUs 113

Figure 5-7 The balanced standby assignment table in the ‘complete balance before failure’ approach 118
Figure 5-8 The backup-standby assignments for the ‘the complete balance before failure’ approach 118
Figure 5-9 The process of deriving the ranked list of SUs in the ‘complete balance before failure’ approach 119
Figure 6-1 Service outage time 124
Figure 6-2 The extension to the standard AMF model 131
Figure 6-3 Recovery mutation path 133
Figure 6-4 The main recovery flowchart 135
Figure 6-5 The component restart recovery flowchart 136
Figure 6-6 The component failover recovery flowchart 137
Figure 6-7 Example an application mapping to its corresponding DSPN 140
Figure 6-8 The cluster DSPN template 149
Figure 6-9 The node DSPN template 154
Figure 6-10 The application DSPN template 163
Figure 6-11 The SU DSPN template 165
Figure 6-12 The component DSPN template 171
Figure 6-13 The comp-CSI DSPN template 185
Figure 6-14 The SU-SI DSPN template 194
Figure 6-15 The SI DSPN template 200
Figure 7-1 The data flow diagram in the configuration generation tool 210
Figure 7-2 Overview of the media streaming example workflow 212

Figure 7-3 Dependency graph in the media streaming example.................................... 214
Figure 7-4 Overview of the ETF content (Component type and CS type) for the example application.. 215
Figure 7-5 A snapshot of the input frames... 216
Figure 7-6 A snapshot of the SG, SI, and CSI template 217
Figure 7-7 A snapshot of creating a service type... 218
Figure 7-8 A snapshot of the dependency specification 220
Figure 7-9 A snapshot of the rankings generated for the SIs of the WebSIT 222
Figure 7-10 A snapshot of the configuration generated... 223
Figure 7-11 The configuration portion under analysis 224
Figure 7-12 The DSPN instance for the availability analysis.. 230
Figure 7-13 The availability chart of the streaming SI 231
Figure 7-14 The availability chart of the streaming SI with reversed proxy SI assignment 232

List of Tables

Table 2-1 AMF Entities brief description 16

Table 4-1 Service provider type dependencies 48

Table 5-1 A ranking example 84

Table 5-2 A ranked list of SUs using a round robin algorithm 88

Table 5-3 The assignment table blueprint 94

Table 5-4 The assignment table before balancing. 99

Table 5-5 The assignment table after balancing. 100

Table 5-6 The standby assignment table 108

Table 5-7 The backup-standby assignment table 110

Table 5-8 The backup-standby assignments for the ’one active for one standby for one backup’ approach 114

Table 6-1 The cluster states 150

Table 6-2 Cluster transitions 150

Table 6-3 Cluster guard conditions 152

Table 6-4 The node states 154

Table 6-5 Node transitions 155

Table 6-6 Node guard conditions 159

Table 6-7 Application states 163

Table 6-8 Application transitions 163

Table 6-9 Application guard conditions 164

Table 6-10 The SU states 165

Table 6-11 The SU transitions 166

Table 6-12 The SU guard conditions 168

Table 6-13 The component states 171

Table 6-14 The component transitions 173

Table 6-15 The component guard conditions 179

Table 6-16 The Comp-CSI states... 185

Table 6-17 The Comp-CSI transitions 186

Table 6-18 The Comp-CSI guard conditions... 189

Table 6-19 The SU-SI states 194

Table 6-20 The SU-SI transitions 194

Table 6-21 The SU-SI guard conditions 195

Table 6-22 The SI states 200

Table 6-23 The SI transitions... 201

Table 6-24 The SI conditions... 203

Table 7-1 List of values of certain attributes of the SGTemplates specified for the media streaming application 218

Table 7-2 List of the values of certain attributes of SITemplates and CSITemplates of the media streaming application 219

Table 7-3 Parameter values for the configuration under analysis.................................. 225

List of Equations

Equation 5-1 Backup assignment for each SU 93

Equation 5-2 Active assignment for each SU 96

Equation 5-3 Active load in N+M 107

Equation 5-4 Standby load in N+M 107

1 Introduction

In today's technological and information based world nearly everyone depends upon the availability of services provided by various systems. Therefore, the high availability of services has become a necessity in various domains ranging from mission critical systems to transaction based ones. When such systems fail, the results can be catastrophic, leading to substantial damages in reputation, finance, and sometimes injuries or even loss of life.

The high availability of a system depends greatly on its reliability. However all systems fail eventually, and therefore another important factor to sustain the high availability is the reparability of the system. In order to avoid service outage during the system repair, fault tolerance is needed. Fault tolerant systems are capable of providing the expected services even in the presence of a failure. This is typically achieved through deploying redundant components within the system and then managing this redundancy through implementing health monitoring, checkpointing, recovery mechanisms etc. There exist several proprietary solutions for rendering systems fault tolerant [1][2]. However, these solutions fall under two categories, (1) they are platform specific, and hence any application designed to run on these systems has to conform to the platform and the specificities of the underlying technology and may not be easily ported to other platforms. (2) They are based on virtualization, where fault tolerance is achieved by having the standby mimic the active which tend to be penalizing for the performance, and

lacks protecting against software failures running within the virtual machine. In either cases the cost of enabling and maintain high availability is high. In order to address the growing need of commercial-off-the-shelf (COTS) affordable high availability solutions, the Service Availability Forum (SAForum) [3], a consortium that develops and promotes open specifications for carrier-grade and mission-critical system, has defined a set of specifications for middleware services. The objective is to enable an ecosystem for highly available platforms. Hence any application built from COTS components can be enabled for high availability if these components are compliant with the SAForum middleware (shown in Figure 1-1), i.e. they can interface with the middleware services.

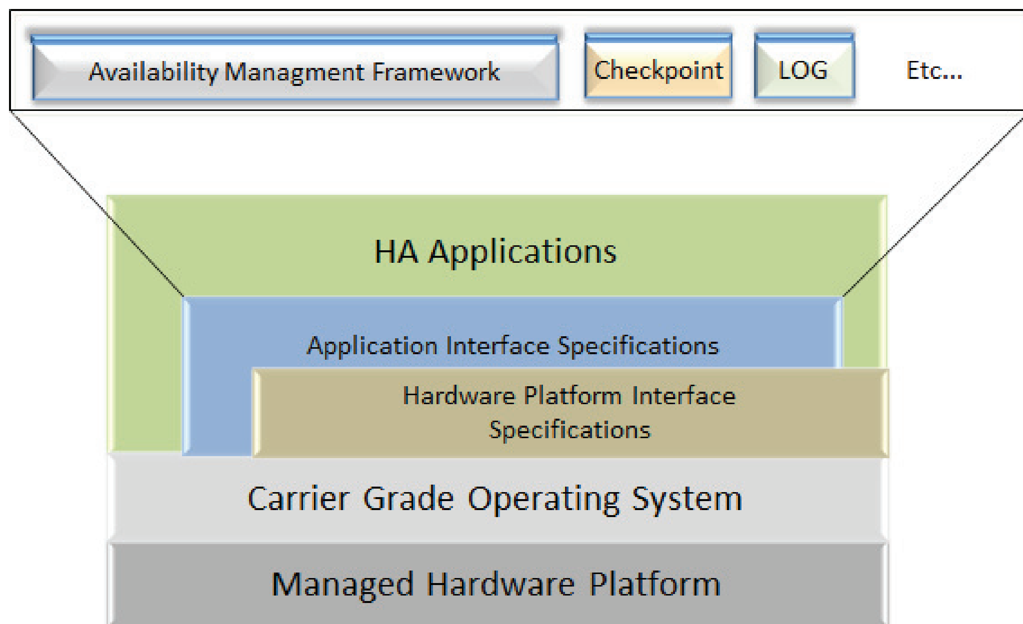


Figure 1-1 The SAForum middleware specifications

Among these services, the Availability Management Framework (AMF) [4] is the service responsible for managing the high availability of the application services by coordinating redundant application components. To achieve this task, an AMF

implementation requires a specific logical view of the organization of the application's services and components known as an AMF configuration. AMF uses the information specified in the AMF configuration (as shown in Figure 1-2) to manage the availability of the services; and hence the runtime behavior of AMF in terms of service assignment and the service recovery in case of failure depends greatly on the information specified in the AMF configuration.

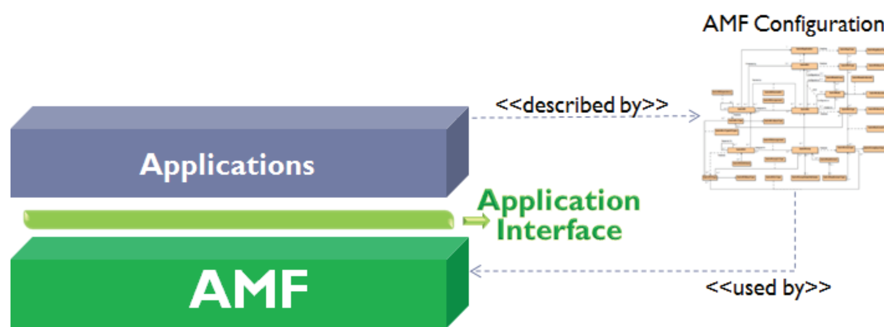


Figure 1-2 The use of the AMF configuration

The AMF configuration is specified according to a standardized configuration model. The configuration includes all the elements (e.g. software components, computing nodes, etc.) needed to describe the AMF managed system (including the applications). Each element in the model has a set of attributes that, when configured, describe the properties of a particular instance of this element. The AMF configuration is then considered as a collection of interrelated elements. The AMF model introduces over 200 attributes. In an actual system configuration typically composed of hundreds of elements, the number of attributes is by the thousands. These attributes include the description of the dependencies among elements, as well as the recovery/protection policies that an AMF

implementation must apply at runtime in order to protect and recover the services provided by the system in case of failures.

Hand-crafting AMF configurations is an error prone process that incorporates several risks. First the complexity of the process renders it virtually impossible to achieve. Second even if we master the complexity by creating configurations that are syntactically compliant to the AMF standards, this does not guarantee that the configuration is semantically sound. A semantically sound configuration must (1) capture the various dependencies among elements, (2) ensure that the system will be able to provide and protect the needed services.

Our first contribution in this dissertation is to define a method for automating the design of AMF compliant configurations. A companion contribution is a workload balancing technique, for improving the quality of the configuration in providing and protecting services, and which can be embedded in the configuration generation process. Our method generates multiple configurations that satisfy the same set of requirements. This raised the question that if more than one configuration can satisfy the requirements, how can we rank them? And according to which criteria? As we are dealing with highly available systems, the most relevant evaluation criterion is the availability level that each configuration can offer to the services. Therefore, the research question that we embark to answer is: can we quantify (based on the AMF configuration) the expected runtime availability of the services (described in the configuration) which are managed by AMF? By answering this question we would be able to judge the quality of an AMF configuration with respect to the availability criterion. In view of that, our final and main

contribution in this dissertation is defining a method for the service availability analysis of AMF configurations. Our contributions can be summarized as follows: at the configuration generation input level, we have defined and implemented a model used to structure and validate the configuration requirements. We have defined and implemented a configuration generation method where we embedded the domain knowledge and the constraints to generate AMF compliant configurations. We have complemented this method with a workload balancing technique that we defined to guarantee workload balancing before and after a failure. Finally we have defined an approach for the availability analysis of AMF managed services. Our analysis is based on transforming AMF configurations into a stochastic model that captures the runtime behavior of AMF. We have defined our stochastic model using Deterministic and Stochastic Petri Nets (DSPNs) [5], an extension of the Petri Net formalism.

Our implementation of the configuration generation has been adopted by the development teams of our industrial partner to produce a commercial-grade software product. The users of this product are system integrators who need to configure their AMF managed systems.

The aim of the availability analysis that we defined is not only to annotate configurations with availability figures, but to also constitute the corner stone of the future research concerning generating "optimal" configurations with respect to predefined criteria (mainly availability and usage of resources), whereas we would be able to generate configurations that, by constructions, meet the availability criteria.

This dissertation is organized as follows, in Chapter 2 we introduce the domain in more details, and illustrate the AMF configuration concepts. In Chapter 3 we survey the literature and discuss the related work. In Chapter 4 we present our configuration generation approach. In chapter 5 we introduce our workload balancing method. In Chapter 6 we present our availability analysis approach. Chapter 7 illustrates our tooling framework and presents the prototypes implementations. In chapter 8 we summarize our contributions and discuss the domains of their applicability, and discuss the potential future work, and how it would integrate with the work that has already been realized.

2 Background

2.1 A Brief Look-Back

The computing industry has witnessed a huge growth over the past two decades. The paradigm of one firm does all, as in the old IBM structure, has been replaced by the vertical integration paradigm. The latter paradigm favors the Component Based Architecture (CBA)[6] for system development. One of the key enablers of CBA is standardization and interoperability among components. For example the same hardware can run multiple operating systems and each operating system can host a variety of compatible applications. In CBA the components interact through interfaces. As a result, a system can be built using COTS components, as long as they can interface with each other. The main goal of the SAForum is to enable building highly available applications − using COTS components − that are portable among multiple platforms. For this purpose the SAForum specifications define a set of middleware services accessible through a set of standardized APIs. As a result, any component that implements the required interface can interact with the middleware, and consequently the availability of the services of applications that are built using such COTS components can be managed by the SAForum middleware.

2.2 Service Availability and the SAForum Middleware

Service availability is defined as the probability that a service is available at any point in time [7]. High availability is attained when the services are available 99.999% of the time (aka five nines of availability) [3]. In highly available systems, the service availability should not be correlated with the health of the component providing the service. In other words, the service is abstracted as the useful functionality provided by the component. In case the component providing the service fails, another redundant replica of the component (that can provide the same functionality) should take over the assignment of providing the same service. When this shift is executed in a swift manner, the service receiver should merely experience a negligible outage. One way to achieve this is to employ availability management software, capable of monitoring the service providing software, and making sure that, in case of failure, the service provisioning is promptly resumed. For this purpose, The SAForum has developed the Application Interface Specification (AIS), for a set of middleware services which includes AMF. The role of AMF is to manage the availability of the service provided by an application.

2.3 The Availability Management Framework

The role of AMF is to manage the availability of the service provided by an application. This is achieved through the management of its redundant components. In fact, the role of AMF in the service recovery management can be summarized as follows:

(1) Failure detection: AMF detects or gets notified that a failure has occurred.

(2) Failure isolation: AMF isolates the failure by cleaning up the faulty component(s).

(3) Service recovery: AMF recovers the services by failing them over to healthy components.

(4) Repair: AMF attempts to repair the faulty components by restarting them or by restarting the node. It should be noted that sometimes the repair is executed as part of the recovery itself. E.g. when the recovery is in fact a component restart.

In order for AMF to manage the availability of the applications it needs an explicit description of the application's components and services, including their groupings, dependency, etc. as well as the recovery policy to be enforced in case of failure. We refer to this configuration as the AMF configuration.

2.3.1 The AMF Configuration

The AMF configuration is a logical grouping of entities constituting the AMF managed system. The AMF concepts described in the configuration are better explained through an illustrative example. We present a highly available web application example shown in Figure 2-1 (a) the corresponding mapping of this example to an AMF configuration is shown in Figure 2-1 (b). The application's architectural workflow is as follows: the user requests are forwarded to an active HTTP server which in turn will examine the requests, and forward the dynamic ones to the application server. The application server will determine whether the request is cached in the memory, or whether it needs to gather the needed information from the database server, and subsequently dynamically generate the HTML code and return it to the HTTP server which will finally provide the processed data back to the client. The HTTP server and the database synchronize their state information with their redundant replicas to keep them up-to-date. In the following we

will present the AMF entities and show how the resources in our example map to these entities.

The smallest entity AMF manages is the *component*. It represents a set of hardware and/or software resources that provide some functionality. AMF manages each component through APIs. For instance, the component can abstract the process representing an instance of the HTTP server. A *service unit* (SU) is a logical entity that consists of one or more components that combine their functionalities into some service. For instance, since the HTTP server and the Application server collaborate closely on each request, they are grouped together and the collaborative service they provide can be considered a web service. The workload associated with providing or protecting some functionality and which can be assigned to the component is represented by a *component service instance* (CSI). For instance the workload of processing the HTTP request that include a source IP address within a specific range of IP addresses can represent the workload assigned to the HTTP server (namely HTTP-CSI). A set of CSIs (required for a service) that need to be assigned to the components of the same SU is represented by a *service instance* (SI). Thus, the SI is the workload that is assigned to an SU by AMF at runtime either to actively provide the service represented by the SI or protect it as a standby. E.g. a Web service can be formed by aggregating the HTTP server CSI and the Application server CSI (namely AS-CSI).

AMF maintains the availability of the SIs by managing their assignments among a set of redundant SUs. For this purpose, SUs are grouped into service groups. A *service group* (SG) consists of a set of SUs that collaborate to protect a set of SIs. They collaborate

according to a certain redundancy model. There are five different redundancy models: 2N, N+M, NWay, NWayActive, and No-Redundancy. These redundancy models differ in the number of active and standby state assignments each SI may have and the distribution of these assignments among the SUs. The five redundancy models can be summarized as follows:

- The *2N* redundancy model specifies that in an SG at most one SU will have the active HA (High Availability) state for all SIs protected by this SG and it is referred to as the active SU, and at most one SU will have the standby HA state for all SIs and it is called the standby SU. For example SG2 shown in Figure 2-1 (b) has this redundancy model, where we have one active database instance and one standby on behalf of DB-SI.

- An SG with the *N+M* redundancy model is similar to 2N, but has N active SUs and M standby. An SU can be either active or standby for all SIs assigned to it. That is to say, no SU can be simultaneously active for some SIs and standby for some other SIs. An SI can be assigned to at most one SU in the HA active state and to at most one SU in the HA standby state.

- In the *No-Redundancy* redundancy model we have no standby SUs, but we can have spare SUs, i.e. available for assignment. An SI can be assigned to only one SU at a time. An SU is assigned the active HA state for at most one SI.

- An SG with the *NWay* redundancy model contains N SUs that protect multiple SIs. An SU can simultaneously be assigned the active HA state for some SIs and

the standby HA state for some other SIs. At most, one SU may have the active HA state for an SI, but one, or multiple SUs may have the standby HA state for the same SI.

- An SG with the *NwayActive* redundancy model contains N SUs. An SU has to be active for all SIs assigned to it. An SU is never assigned the standby HA state for any SI. From the service side, for each SI, one, or multiple SUs can be assigned the active HA state according to the preferred number of assignments, *numberOfActiveAssignments,* configured for the SI. The *numberOfActiveAssignments* should always be less or equal to the number of SUs in the SG. For example SG2 shown in Figure 2-1 (b) has this redundancy model, and therefore the Wed-SI is assigned active to both SUs, i.e. an HTTP request can be processed by any of the SUs.

An AMF *application* is composed of one or more SGs and the SIs they provide. There are two additional AMF logical entities used for deployment purpose: The cluster and the node. The cluster consists of a collection of nodes under the control of AMF.

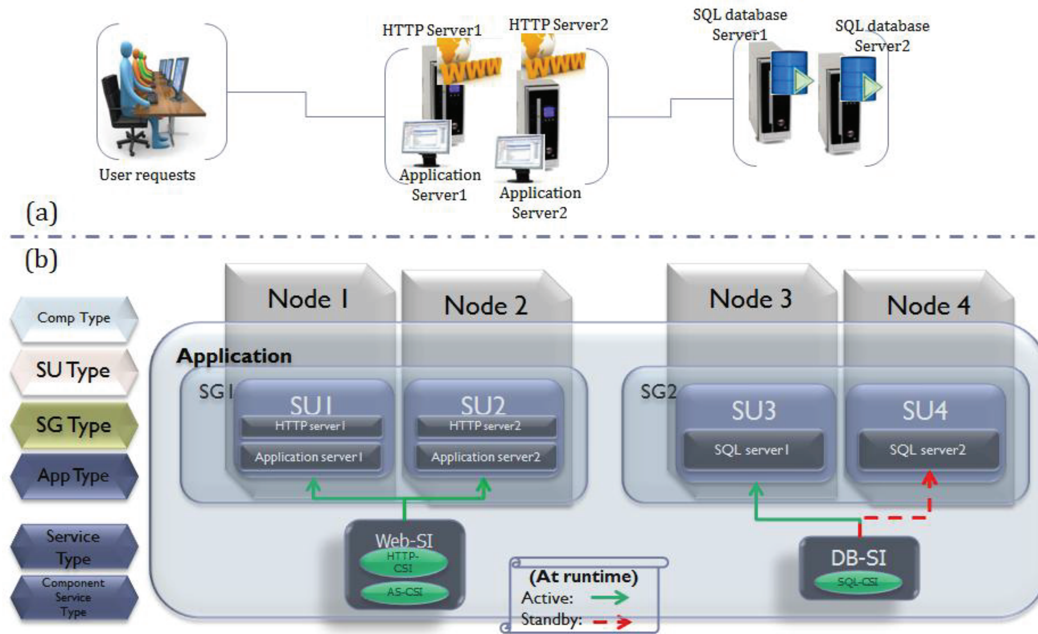


Figure 2-1 An AMF configuration example

All AMF entities, except the cluster and nodes, are typed. The entity types provide AMF with the information about the shared characteristics of their corresponding entities. For example, the component type determines the component service types any component of this type shall provide. AMF uses this information to determine to which component within an SU to assign a particular CSI. Therefore, the types constitute an integral portion of the AMF configuration. In the AMF configuration if a service provider type supports a service type then an entity referring to this type can support the services of the specified service type as shown in Figure 2-2.

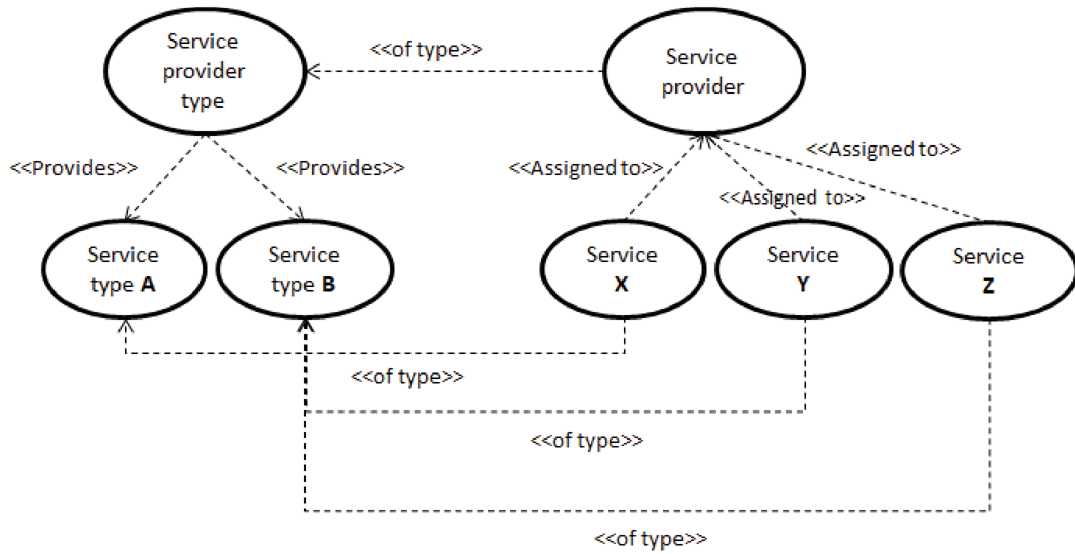


Figure 2-2 The types versus the entities referring to these types

The AMF types are defined as follows:

Component Type: A component type describes a particular version of a software implementation designed to be managed by AMF. The component type specifies the component service types that the components of this type can provide. It defines for each component service type the component capability model and any required dependency on other component service types. The component capability model is defined as triplet `(x, y, b)`, where `x` represents the maximum number of active CSI assignments and `y` the maximum number of standby CSI assignments a component can handle for a particular component service type. While `b` indicates whether active and standby assignments can be handled simultaneously. The component type also specifies the component category, e.g. proxy, container, etc. For example a container component could be a virtual machine

that acts as an execution environment for other components, referred to as contained components.

Component Service Type (CS type): It describes the set of attributes that characterizes the workload that can be assigned to a component of a particular type in conjunction of providing some service.

Service Unit Type (SU type): The service unit type specifies the service types an SU of the type can provide, and the set of component types from which an SU of the type can be built. It may limit the maximum number of components of a particular type that can be included. Thus, the SU type defines any limitation on the collaboration and coexistence of component types within its SUs.

Service Type: A service type defines the set of component service types from which its SIs can be built. The service type may limit the number of CSIs of a particular CS type that can exist in a service instance of the service type. It is also used to describe the type of services supported by an SU type.

Service Group Type (SG type): The service group type defines for its SGs the redundancy model. It also specifies the different SU types permitted for its SGs. Thus the SG type plays a key role in determining the availability of services.

Application Type: The application type defines the set of SG types that may be used to build applications of this type.

Table 2-1 summarizes the AMF entities, and shows their acronyms and corresponding types.

Table 2-1 AMF Entities brief description

Entity	Acronym	Brief description	Corresponding type
Component	–	A set of hardware and/or software resources	Component type
Service Unit	SU	A set of collaborating components	SU type
Service Group	SG	A set of SUs protecting a set of SIs	SG type
Application	–	A set of SGs and the SIs they protect	Application type
Service Instance	SI	The workload assigned to the SU	Service type
Component Service Instance	CSI	The workload assigned to the component	CS type
Node	–	A cluster node	–
Cluster	–	The cluster hosting AMF and the applications	–

The AMF configuration is standardized by [4] as a UML [8] class diagram illustrating the various classes representing the AMF concepts and their relations (shown in Figure 2-3).

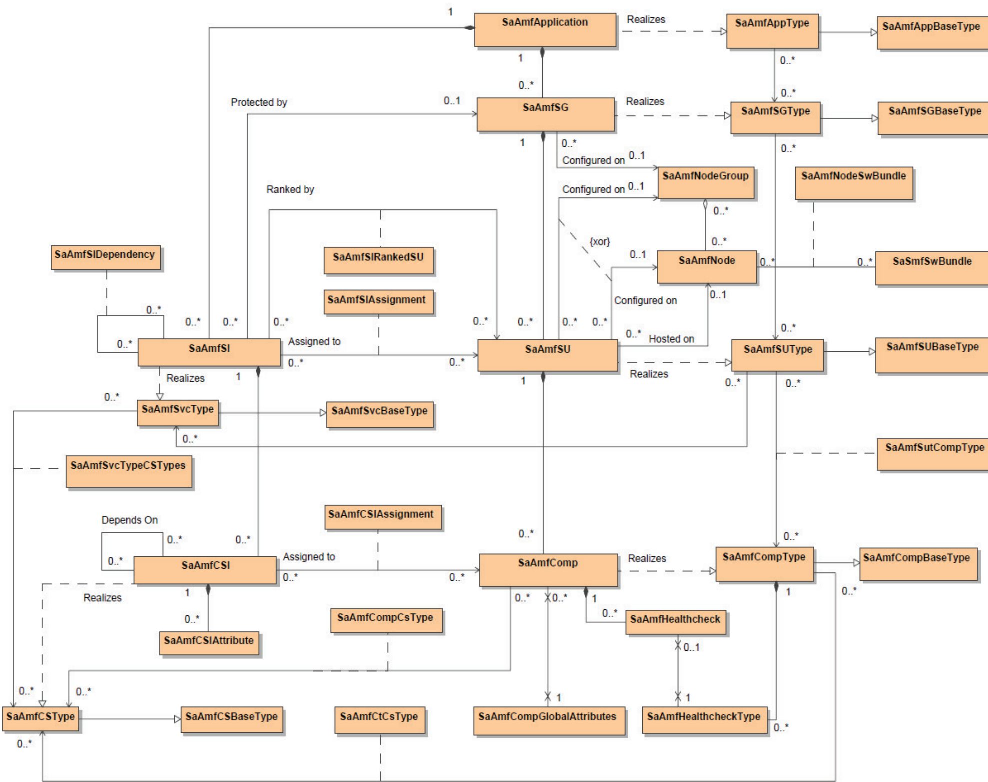


Figure 2-3 The standardized AMF configuration model (taken from [4])

2.3.2 The Dynamic Behavior of AMF

The dynamic (or runtime) behavior of AMF can be summarized by two main activities: (1) managing the life cycle of the components. (2) Maintain the service assignment even in the presence of failures. For this purpose and according to the configuration AMF will instantiate the components, and assign their CSIs (which imply the SI assignment). However this task is not as simple as it seems. Different component categories are managed in a different manner, for example a proxied component can only be instantiated throughout the proxy. The proxy cannot communicate with the proxied without being assigned the proxy-CSI etc. In case of failures, AMF can either detect them by monitoring the health of the components, or another component (or monitoring facility)

can report the error to AMF (through an API call). When this happens, AMF reacts by (1) isolating the error, by cleaning up (abruptly terminating) the faulty component, again according to the component category certain dependencies apply. This cleanup may even be done by restarting the node if necessary. The important issue here is not to reassign the services of the faulty component elsewhere without making sure that the faulty component has been cleaned up. (2) Recovering the service, by re-assigning the CSIs (through re-assigning the SI) to another healthy replica (usually a standby) of the component that can resume the service provisioning. (3) Finally AMF attempts to repair the faulty component (if the configuration allows it) by restarting it. Note that in certain situations the recovery and the repair are combined in one step, e.g. restarting the component without failing-over the services. Such recovery is preferred if it costs less outage than the failover. It is possible that the recovery fails, e.g. a restarted component continues to exhibit a faulty behavior, and in such situations an escalation policy can be enforced. This escalation policy is specified in the configuration, whereby for instance a component restart can escalate to an SU restart that can further escalate to an SU failover and finally a complete node failover. In another scenario, a component may become unresponsive to the AMF commands such as the instantiate or terminate commands; this can result in a node reboot and failing over the services provided by this node to other nodes. The Escalation policy is explained in more details in Chapter 6.

The reader can notice that the service availability is tightly coupled to the AMF behavior which in turn is parameterized and driven by the configuration of the application, and hence the importance of the issues tackled in this thesis: the design and analysis of AMF configurations.

2.3.3 The Entity Types File

The Entity Types File (ETF) [9] is a standardized eXtensible Markup Language (XML) file provided by the software vendor to describe the various deployment options according to which a software can be installed. In addition it describes the software capabilities, limitations and dependencies. It is important here to distinguish the difference between AMF types and ETF prototypes (here forth referred to as ETF types). The ETF types describe a range of deployment options to install the software in a system, whereas the AMF types describe the specific deployment options according to which the software was installed in the system. Therefore we can consider ETF types as meta-types from which AMF types can be derived. The ETF file also describes the software functional dependencies among ETF component types and ETF SU types. However, the AMF configuration only describes the manifestations of these dependencies. For instance, if ETF specifies a certain ETF component type dependency, then this dependency will translate into one or multiple AMF CSI dependencies (and potentially an AMF component instantiation order sequence), but not a dependency at the AMF type level. This is due to the fact that the AMF configuration only includes the information needed for runtime life-cycle and availability management, and therefore only the implications of the dependency affecting this management are specified in the AMF configuration. It is the responsibility of the configuration designer to interpret the ETF dependency and map it to the proper AMF dependency; we will discuss this in more details in Chapter 4 when we describe the dependency handling during the process of AMF configuration generation.

2.4 Availability Analysis

The availability analysis of any system is normally based on analyzing the various states that the system undergoes during its lifespan. This analysis mainly focuses on capturing the failures that cause the system to switch to a faulty state, and the repairs that shift the system back to a healthy state. Since the occurrence of failures is erratic by nature, stochastic models have been used to conduct the availability analysis. Markov models have been extensively used for this purpose e.g. [10][11], mainly for their expressiveness, and their capability of being analytically solved. A two-state Markov Chain is shown in Figure 2-4, the transition from one state to another is governed by a rate.

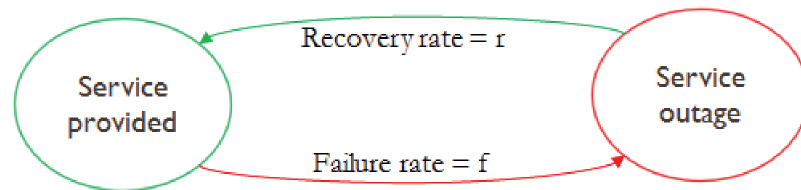


Figure 2-4 A two-states Markov model

A major drawback of using Markov models is the large number of states that are needed in the model in order to describe the system behavior [11]. As an alternative, Stochastic Petri Nets [12][13], an extension of the Petri Net formalism created by Carl Petri [14], are used to capture the complexity of real systems. Such models can either be automatically transformed into Markov Chains (when applicable), or simulated in order to get to required measures (e.g. availability, performance etc.). DSPNs are an extension of Stochastic Petri Nets, they have been introduced in [5] as a continuous-time modeling tool which includes both Stochastic (exponentially distributed) and constant timing events.

2.4.1 Deterministic and Stochastic Petri Nets

In this section we provide a formal description of the DSPN constructs we use in our analysis, followed by an example of using DSPNs for availability analysis. A DSPN is considered as a tuple $\{P; T; I;O;H; g;M_0; \zeta\}$ where:

- P is a finite set of places, which may contain a discrete number of marks called tokens. A marking $M \in \mathbb{N}^{|P|}$(Where $\mathbb{N}$ is the integer set) defines the number of tokens in each place $p \in P$, indicated by $\#(p, M)$.

- T is a finite set of transitions, divided into three disjoint sets, T^{Im}, T^{Exp}, and T^{Det}, of immediate, exponential, and deterministic transitions, respectively.

- $\forall p \in P; \forall t \in T, I_{p,t} : \mathbb{N}^{|P|} \rightarrow \mathbb{N}$, $O_{p,t} : \mathbb{N}^{|P|} \rightarrow \mathbb{N}$, and $H_{p,t} : \mathbb{N}^{|P|} \rightarrow \mathbb{N}$ are the multiplicities of the input arc from p to t, the output arc from t to p, and the inhibitor arc from p to t, respectively. Marking-dependent arc multiplicities can be used to simplify the modeling of complex system behavior.

- $\forall t \in T, g_t : \mathbb{N}^{|P|} \rightarrow \{\text{True; False}\}$ is the guard for transition t. A transition $t \in T$ is enabled in marking M iff $g_t(M)$ = True and $\forall p \in P; (I_{p,t}(M) \leq \#(p, M)) \wedge (H_{p,t}(M) > \#(p, M) \vee H_{p,t}(M) = 0)$. I.e. the guard condition and the arc multiplicity must be satisfied.

- $M_0 \in \mathbb{N}^{|P|}$ is the initial marking.

- $\forall t \in T^{Exp} \cup T^{Det}, \zeta_t: \mathbb{N}^{|P|} \rightarrow (0;+\infty)$ is the mean firing time for transition t, it may be marking-dependent.

In order to illustrate the DSPN constructs we introduce a DSPN example where we are interested in measuring the availability of a software component over a time interval [0, t'], given that the Operating System (OS) where the component is executing exhibits *n* erratic failures during this time interval. When the OS fails, it causes the component to become impaired. When this failure occurs, the associated recovery is an OS restart. The restart duration of the OS is *x*. When the OS is started again, the component is automatically instantiated; the component instantiation duration is *y*. The corresponding DSPN that capture this runtime behavior that we described is shown in Figure 2-5. Most of the DSPN constructs, which we formally introduced earlier in this section, have a graphical syntax. A place *p* is denoted by a hollowed circle, and it is used to abstract a certain state. A token is denoted by a filled circle and it resides in a place. It signifies that we are currently in the place hosting the token (we refer to this as the current "marking" of place *p*). For instance a token in the place "Component_healthy" in Figure 2-5 signifies that the component is healthy. Tokens can leave a state through transitions. The transitions described earlier are graphically represented as follows: (1) stochastic transitions are denoted by a hollowed rectangle. For instance since in our example the OS fails in a stochastic manner, a stochastic transition (namely ST_1) is used to shift the token from the OS_healthy place to the OS_faulty place. A stochastic transition is characterized

by an exponential distributed firing delay according to which it fires[1], for instance in our example the exponential rate of transition would be *n*. (2) Deterministic transitions are denoted by filled rectangles that represent deterministic events, for instance the time needed to restart the OS has a fixed (known) duration, therefore the transition from the OS_faulty to the OS healthy state is a deterministic one. Deterministic transitions are characterized by deterministic firing delays. In our example the delay of TT_2 is x, while the delay for TT_1 is y (the time needed to instantiate the component). (3) Immediate transitions are denoted by lines (or thin filled rectangles) and they represent immediate events, for instance when the OS is faulty, the component immediately becomes impaired, and therefore an immediate transition is used to shift the token from the Component_healthy place to the Component_impaired place. Transitions in general and especially immediate ones can be guarded by an enabling function (which is a Boolean expression referred to as the transition guard or simply guard). When the guard evaluates to false, the transition is disabled. It will become enabled when the guard evaluates to true. For instance transition TT_1 is only enabled when the OS is healthy. I.e. the enabling function or guard of TT_1 should only evaluate to true when there is a token in the OS_healthy state. Finally, arcs are denoted by arrows and they are used to connect places

---

[1]When a transition fires it transfer the token (or in some cases multiple tokens) from the source place(s) to the destination place(s).

to transitions and vice versa. Arcs can have multiplicities to denote the number of tokens they can transport from one place to another when the transition fires. For further details about DSPNs we refer the reader to [15][16].

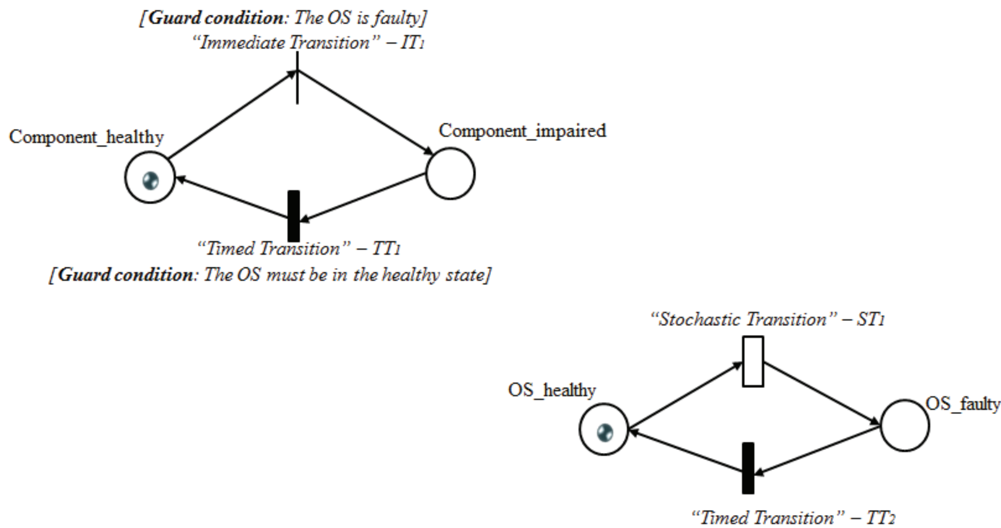


Figure 2-5 A DSPN example

2.4.1.1 Using DSPNs for Availability Analysis

The availability of the component can be measured by solving the DSPN shown in Figure 2-5. This availability can be defined as the probability of having a token in the Component_healthy place between time 0 and t'. To concretize our example, we will solve the DSPN shown in Figure 2-5 with the following parameters: the time interval is 1 year, the OS restart time in 60 seconds (equal the delay of TT_2), and the component instantiation time is 10 seconds (equal the delay of TT_1). We will vary the failure rate (i.e. *n*) of the OS between 1 and 50 failures per year. The results are shown in the graph of Figure 2-6, we can clearly see that when *n* exceeds the 5 failures, the availability drops below the five nines, which is normal since in our example we have no redundancy, i.e.

we have no replica of the component that we can failover to. It is important to note here how the availability of one component (e.g. the OS in our example) can impair other components.

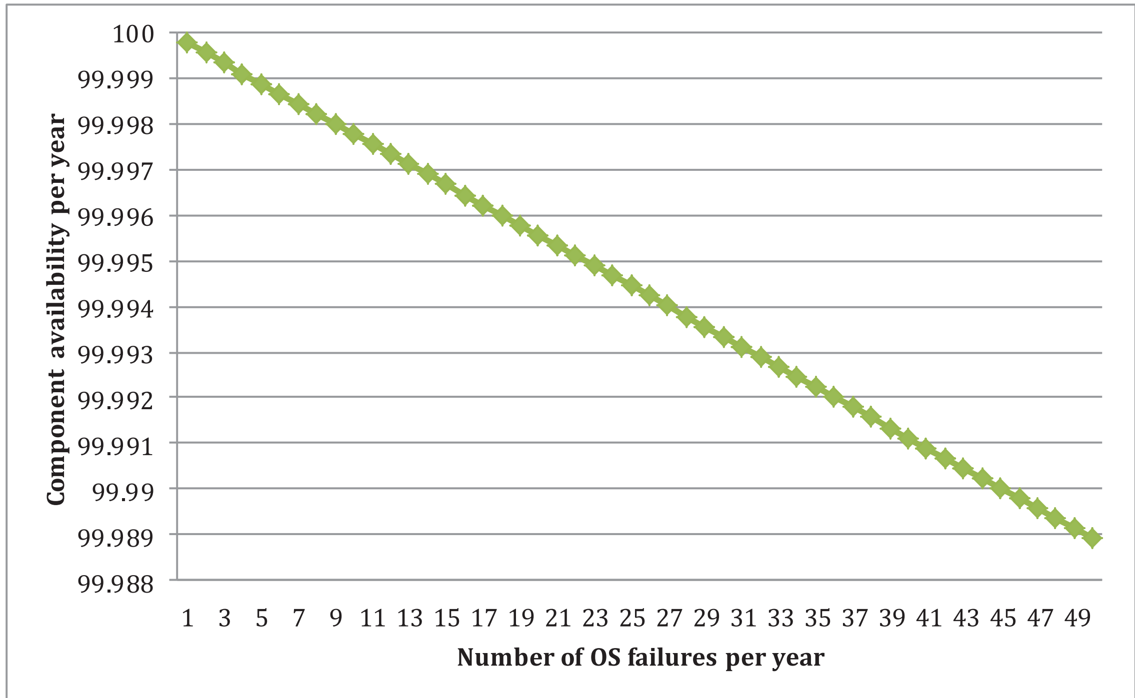


Figure 2-6 Analytical results of solving the DSPN example

3 Related work

Our work involves two main parts, the automatic configuration generation, and the configuration-based service availability evaluation, and therefore we have categorized accordingly the related work.

3.1 Configuration Generation

The SAForum specifications are relatively new and the implementation of SAForum compliant middleware is still an ongoing effort. Existing middleware implementations such as OpenSAF[19], OpenAIS [20] and OpenClovis [21] offer limited if any support for the generation of AMF configurations. Other works described in this section relate to our work in the broader sense of system configuration generation.

The authors in [17] 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. However the authors did not specify any methodology to follow in the process of configuration generation for the PIM. Moreover they did not include ETF in their solution, which is an integral part of the configuration generation

since it provides vital information about the software in terms or grouping, dependency etc. Our view of the automatic configuration generation differs from their view. Their approach is a model transformation that takes an existing configuration form a PIM level to a PSM level. Our work is different from this approach, as we automatically generate this initial configuration or PIM.

In [18] the authors created a modeling framework for the automatic generation of middleware specific deployment descriptors. The implementation of the framework is based on the IBM Rational Software Architect (RSA) modeling tool. They have also implemented configuration generator facility for each different AIS implementation in the form of RSA pluglets. And they used these pluglets to generate AMF configurations for two middleware implementations: (1) for OpenAIS they generate a simple text file from the configuration UML model (2) for OpenSAF they generate an XML file using the Document Object Model (DOM) solution again from the configuration UML model. Again the authors of this work have a different view of what the automatic configuration generation is, and they view it again as a model transformation from an existing configuration at PIM level to a PSM level. Therefore their approach is missing the methodology and concepts of generating AMF configurations.

A closely related work, performed in the context of our project, is presented by the authors in [22]. They define a UML profile for the design of AMF configuration, the domain model of the profile is based on restructuring the standard AMF model and extending it with a set of OCL constraints that are used to verify the compliancy of an AMF configuration. Among the objectives of the work is to (1) enable the validation of

AMF configurations (2) enable a model driven approach for the configuration generation. In this work the authors do not propose a method for the configuration generation, however in [23] they propose a model driven approach for the configuration generation using the Atlas Transformation Language (ATL). Both our approach and the model driven one were carried on in the same project. Our configuration generation method predates the model driven approach which is mostly based on the steps we defined in our method. The model driven approach has a more declarative style, and it generates configurations based on the profile proposed in [22]. Nonetheless it lacks the dependency analysis that we perform at the input level, and does not target the issue multiple configuration generation that we will discuss in the next Chapter.

In [27] the authors present an engine that automatically designs a service infrastructure which will meet the service's availability requirements. The engine explores a design space consisting of multiple combinations of hardware/software and repair options configurations presenting various tradeoffs among cost, availability, and performance. Their approach is rather a combinatorial approach that finds all possible combinations of hardware, operating systems and applications, and filters out the ones that do not satisfy the availability. In this work the configuration is the design of the system stack from hardware to application, rather than configuring the components and the services that constitute the application. They do not present the system recovery behavior or the dependencies in the system.

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[28][29]. The authors in [28], 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, that 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 generating configurations that satisfy the protection level requirements (redundancy model, assignments etc.) or the dependency analysis and handling are not taken into account in [28] or [29].

The work presented in [30] focuses on developing a system that will automatically select which data protection techniques to use, and how to apply them, to meet user-specified dependability. By being selective in picking techniques matters – assigning different levels of protection to different kinds of information – the authors in [30] believe they can save money or free up resources for providing more protection to important data. Their input consists of (1) a description of the user's requirements for data performability and data reliability. (2) A description of the failures to be considered, including their scope and likelihood of occurrence. (3) A description of the data protection techniques. Finally the design engine will come up with the appropriate protection technique. Again their approach is a combinatorial one that filters out the techniques that cannot satisfy the requirements. And they focus on reliability and performability for data, rather than for

applications. So basically they select and configure the protection technique rather than generating system configurations.

In [31] the authors investigate the applicability of MDE to address the middleware QoS configuration challenges by automating the process of mapping the domain-specific QoS requirements onto the right middleware specific configuration options. In particular, their model transformation-based approach begins with domain-specific, platform-independent models (PIMs) of DRE (Distributed-Realtime-Embedded) system. The QoS requirements are then used to iteratively transform the PIM to more refined and detailed middleware platform-specific models (PSMs). The variabilities in the middleware configuration spaces are captured in the form of parameterization of mapping rules of the platform-independent model transformations. Subsequently individual platform-specific transformations are instantiated by specializations, such as by providing exact values of these parameters. Their notion of parameterized transformations and specializations is similar in concept to C++ templates and Java generics. Like most of the automatic configuration generation works we have surveyed, this work also starts from an existing PIM level configuration and automatically generates the PSM level configuration.

3.2 Availability Evaluation

The work presented in [32] is the only work we are aware of that partially tackled the problem of calculating the availability of the services in the context of AMF. In this work the authors define the service availability based on user behavior, and derive formulas to compute service availability starting with the user behavior model. The user behavior can be summarized as follows, when the user issues a request, he waits for the request to be

processed and then the user can enter a thinking state, or issue another request, or decide to end the session. The authors used Stochastic Reward Nets (SRNs are a higher level formalism based on Stochastic Petri Nets (SPN)) to build the user behavior model. As a case study the authors applied their approach to a SAForum compliant media gateway controller (MGC) architecture in VoIP system. In the case study there are two application servers running on the cluster for call processing, each in charge of processing different call features. Process replication is adopted as the mechanism to provide application level software fault tolerance. The structure of the case study AMF configuration is as follows: Each service application has one service instance, and each service instance is assigned to two service units in different cluster nodes, one service unit is active and the other is standby. The systems has three nodes A, B and C. Either node A or node B has one service unit acting as the primary for any of the call processing services. Node C has two service units acting as standby for each of the services.

For the recovery behavior, the authors made the following assumption about the actions performed by AMF. For software faults the AMF tries several levels of recovery actions: 1) component restart, which they assume is fast and has little or no impact on the application; 2) switchover that switches the service to the standby service unit, in the meantime the faulty node is restarted; 3) manual repair of the cluster node if automatic restart of the previous level cannot recover the fault. For hardware faults, they assume recovery actions 2 and 3 are adopted by the AMF. After the fault is detected the AMF tries to recover the fault using from lower level to higher level recovery strategies, each level requiring more time to execute than the previous level.

The authors built a SRN to model this specific configuration; based on the SRN they calculate the probability that both servers are up in steady state. And then they use this information in the user behavior model to calculate the user perceived availability. Finally the authors evaluate various factors such as the user think times, session end probability, and fault detection rate, to see how they affect the calculated availability.

The system modeling part of this work is based on one specific AMF configuration. Even the recovery behavior is based on the assumption that AMF only follows the steps assumed by the authors. The authors did not present any generic method for deriving this model based on an AMF configuration. They also did not present any generic behavior analysis that takes into account the various recovery actions that AMF may engage into. And therefore from the system modeling perspective this work in not reusable since if we change the configuration e.g. by changing the redundancy model, or take into consideration other recovery actions to be performed, or change the escalation sequence assumed by the authors, then the entire system model has to rebuild from scratch. Moreover the authors did not include any components in their configuration. And hence did not consider the different component categories or dependencies which have direct impact on the modeling of the system behavior. In addition the authors make the assumption that the standby service units do not fail, which is not a realistic assumption. So basically the case study, which is the only part of the work related to AMF, did not consider the specificities of the AMF configuration and the AMF behavior. And the system described could as well have been any generic system with one active and one standby component.

Another interesting work is presented in [33]. This work deals with the automatic dependability analysis of systems designed using UML. An automatic transformation is defined for the generation of models to capture systems dependability attributes, like reliability. The main objective aims at the creation of an integrated environment where UML-based design tool sets are augmented with modeling and analysis tools.

Within this work, the authors present an automatic transformation from UML diagrams to Timed Petri Net (TPN) models for model based dependability evaluation. The TPN models output of the transformation can be solved with already available automated tools. In order to achieve this task the authors (1) extended the UML language. Essentially two types of extensions were used: one for identifying redundancy (fault tolerance) structures and the other one for defining the dependability parameters and desired measures. In order to represent redundancy, the authors opted for a “class based” redundancy, which prescribes that components of a redundancy structure must be defined as specific classes. Three basic components are defined: <<redundancy manager>>, <<variant>> and <<adjudicator>>. (2) Transformed the entities and relations of the UML design into an Intermediate Model (IM). The IM is defined as a hypergraph, where each node represents an entity described in the set of UML structural diagrams, and each hyperarc represents a relation between entities. IM nodes have attached a set of attributes, describing the fault activation and the repair processes. (3) Built a TPN dependability model, by generating a set of subnets for each element of the IM.

The drawback of this work is that it does not separate the concept of service from service provider. In other words their work is directed towards the system availability rather than

the service availability. Our analysis is quite different by nature since, for instance in our analysis, the same failure on a service provider entity can have different impacts on different services. And the recovery time and the outage times are not the same for all the services.

The authors in [34] present and analyze a colored stochastic Petri net model of a redundant fault-tolerant system. Their measure of interest is service availability. They defined service availability as the number of successfully completed jobs relative to the total number of arrived jobs. They determine service availability by modeling the system as a simple queuing system processing jobs/requests using a stochastic colored Petri net. More specifically, they model environments with a completely reliable queue and servers that are subject to single points of failure, i.e., the model incorporate failure and repair. They conclude their analysis with the realization that for high utilization one redundant server should be added if possible. This will greatly improve service availability of the system. The positive effect certainly is supported by the fact that an additional server reduces the load on the system. For high utilization adding more than one redundant server certainly has a positive effect but at much lower degree. The interesting part of this work is being able to determine how many extra redundant components are needed to satisfy a certain availability requirements. Of course in our domain several factors must also be considered such as the redundancy model used and the components categories etc.

The authors in [35], present a UML profile called DAMRTS (Dependability Analysis Models for Real-Time Systems) representing an extension to the reference metamodels of

the OMG profile for “Schedulability Performance and Time” (SPT). The objective is to provide the concepts to specify a real-time system with stochastic and probabilistic information allowing the dependability analysis of the system. The authors extend the UML class diagram with the Indicator and Cause classes. The attributes of the indicator class relate to the dynamic aspect of the resources being described. The attributes of the Cause class indicate whether an associated failure became true or not. The authors also propose an extended semantics of UML statecharts related to probabilistic timed automata (timed automata are discrete transition systems extended with a notion of time). The UML statecharts are then easily convertible to probabilistic timed automata. The aim is to verify formally probabilistic temporal properties related to the dependability of real-time systems.

In [25] the authors have proposed a UML Dependability Analysis Modeling (DAM) profile to support the assessment of the dependability of real-time embedded systems (this work builds on the knowledge presented in [36]). The profile conforms to the MARTE profile, issued by the OMG and, in particular, the basic concepts of dependability defined in the DAM profile are expressed in terms of complex non-functional properties (NFP). The DAM profile specifically addresses the quantitative evaluation of dependability and the notions introduced in the profile should complement the ones defined in the QoS&FT sub-profile, which supports instead the specification of FT software architectures. However the authors present the profile as a mean to specify the dependability related specifications of a system, but did not discussed the issues related to the derivation of dependability analysis models. The authors illustrate the use of this profile for the dependability analysis in [26]. The objective of this work is similar

to ours. However in order to reuse their approach we would have to map to the concepts of the AMF into the concepts of their domain model which is not a straight forward task, since we have many concepts in AMF that are not supported in the DAM profile, such as the definition of different component categories, the different redundancy models and their semantics etc. Such nuances affect the runtime behavior of AMF in terms of the availability management and thus must be considered in the availability analysis. In short, in order to reuse their approach we would have to modify their model to fit our requirements, and we would still need to model the runtime behavior of AMF from scratch. Therefore we decided to remain with the standard AMF model and introduce minor extensions instead of shifting to a different model.

The work presented in [37] focuses on the dynamic modeling of degrading and repairable complex systems. The authors propose the extension of Stochastic Petri nets (SPNs) with aging tokens. The reason why aging tokens are introduced is because in existing SPN formulations, memory was associated solely with transitions, which resulted in certain difficulties in modeling the changes in the system configuration while preserving the memory. The concept of aging tokens is introduced to improve the dependability modeling flexibility and clarity of SPNs. Aging tokens can be viewed as a natural extension of colored Petri nets since they are effectively token labels that are allowed to change not only discretely upon the firing of the token, but continuously in the process of enabling a certain transition that has a matching policy. This work proved that aging tokens can be used to replace marking-dependent firing policies.

The authors in [38] analyze the dependability of an electromechanical system (sprinkler system) in the presence of two modes of failures. The on-demand failure mode is when the system fails to start, and the active failure mode is when the system fails during its operational phase. The authors used dynamic fault trees, which extend traditional fault trees by including special constructs to represent sequential relationships between events. A new construct, DDEP (demand dependency), represents the dependencies between the components in the demand phase and its support components in the standby phase, where a component in the demand phase can require the availability of one or more support components in order to commence operation. The system was modeled in dynamic fault trees and then it was analytically solved using Markov chains.

In summary the work on system availability evaluation has been ongoing for decades now; the list of related work in this domain tends to be significantly large in numbers and scope. However the closest related works for our domain are either focused on one hand on defining and solving the mathematical (stochastic) models of the system in order to predict its availability [39][40][41][42], or on the other hand on using/extending UML to support the modeling of the availability features of the system, and then use the UML model to generate a mathematical model that in turn must be solved [43][44][45][46].

4 Generating AMF Compliant Configurations

In a previous work [47], we devised and implemented an approach for the configuration generation of AMF compliant configurations. This work was the first step in tackling the problem of mastering the complexity of creating AMF compliant configuration through automation. In the previous approach several assumptions were made that would burden the configuration designer. In this dissertation we present a different approach that is sounder and more complete for configuration generation.

The contributions described in this chapter can be summarized as follows:

(1) Defining a UML domain model for specifying the configuration requirements

(2) Defining a method for detecting and specifying dependencies at the input level

(3) Defining a top-down configuration generation method based on the configuration requirements and other input.

(4) Defining how to extend our generation in order to generate multiple configurations based on the same input.

In the rest of this chapter we will illustrate the need for each of the contributions and the challenges encountered in achieving them. Note that contributions (1) and (2) are prerequisites for the configuration generation method.

4.1 Overview of Creating AMF Configurations

A typical AMF managed system has the following properties: it is expected to provide a set of services with a certain protection level. These services are provided by components, which are grouped to enable the service provisioning and protection. Finally the software is deployed in a distributed manner on a cluster of nodes. The configuration designer is expected to know the services that the system should provide. For example the designer will know that the system is expected to provide a service of type "web service" for a certain number of users. The designer will then map this information into the proper SIs/CSIs (e.g. an SI can be a grouping of two CSI, HTTP-CSI and APP-Server CSI. In order to further divide the workload, each SI can represent the workload coming from a certain range of source IP addresses that can be specified in the CSI's attributes). Specifying each SI/CSI individually is complicated and time consuming task. The next phase would be to look into the available software, and process the ETF file(s) (the ETF files are provided by the software providers (vendors, developers, etc.)) to find the proper component types and SU types that can support the CSIs and the SIs. In order to figure this out, the designer has to calculate the expected load of CSIs/SIs that each component and SU must be able to handle, and accordingly select the proper types. Certain dependencies may apply according to the selection which may trigger further selections and SI/CSI creation, for this purpose the designer has to thoroughly examine the ETF file and make sure to handle all the dependencies. Moreover, SIs may have different priorities, where for example the SIs representing paying customers should have a higher level of availability then other SIs. For this purpose the designer may want to provide more protection for such SIs by protecting them in different SGs with more redundancy.

After the type selection the designer has to figure out the number of entities needed (e.g. number of components of a certain type per SU etc.), and then proceed by creating them while making sure that the configuration complies to all the AMF specification requirements in terms of how entities can be grouped and how dependencies should be captured etc. This is just an overview of the process of creating AMF configurations without going exhaustively into its details. In short, the process of creating AMF is complicated, time consuming and requires deep domain knowledge. Designing such a configuration in an ad-hoc manner is extremely challenging and error prone, even for relatively small systems consisting of a cluster having few nodes.

4.2 Overview of Automatic Generation of AMF Configurations

The objective of the automation is to overcome the complexity of generating configurations compliant to the AMF specification. We have defined our configuration generation process as follows: on the requirement side the configuration generator specifies a set of services (the SIs and their CSIs) and the corresponding protection level, as well as the cluster description in terms of the available nodes. The description of the (available) software capable of providing these services is provided through one or many ETF files. When this input is specified, the configuration generation engine will process it and subsequently generate the corresponding AMF configuration that satisfies the requirements and enables AMF to manage the provision of these services in a highly available manner. An overall view of the configuration generation process is shown in Figure 4-1.

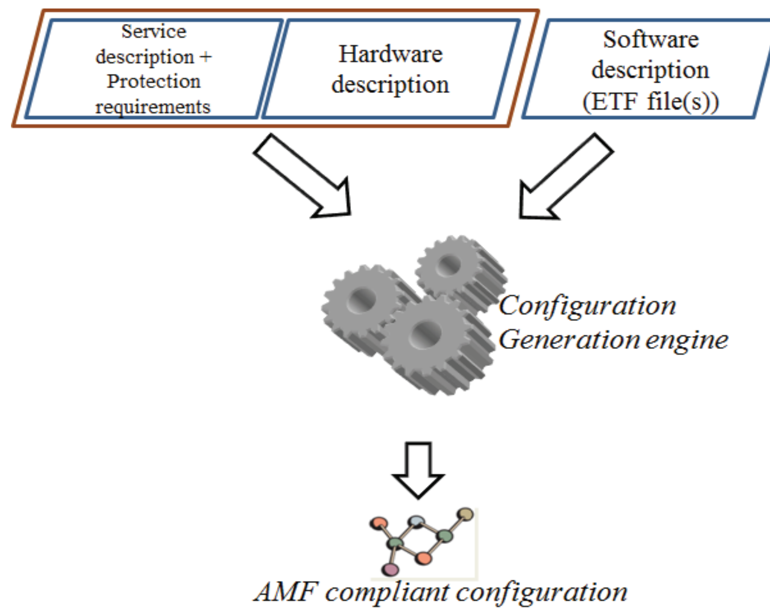


Figure 4-1 Overall view of the AMF configuration generation

Our configuration generation approach consists of three main steps:

(1) Collect the configuration requirements and make sure they are consistent and complete

(2) Select the proper ETF types capable of satisfying the requirements

(3) Create the AMF configuration based on the selected ETF types and the requirements

4.3 The Configuration Requirements (CR)

Collecting the configuration requirements is an interactive stage of our approach where the configuration designer specifies the requirements according to a CR model. If further specifications are deemed necessary based on the dependency analysis, then the input will be augmented with the missing information.

4.3.1 Why a CR Model?

The services to be provided by the AMF managed system are abstracted in the form of SIs and CSIs. In large systems with hundreds of nodes hosting several applications, the number of SIs and respectively CSIs tend to be significantly large. Among these SIs, some share the same characteristics such as their service type, or the level of protection the designer wishes to give them (e.g. number of standby assignments etc.). Similarly, each cluster node must be specified in the AMF configuration, this specification includes information about the node resource capacities as well as repair and recovery information. In clustered systems, the nodes or the subsets of nodes are typically identical, and therefore it is possible to describe these nodes in a generic way. In short specifying each node, SI and CSI individually is a demanding and time consuming task that diminishes the values of the time/effort saving aspect of automating the process of configuration generation. Therefore we need a generic way to specify the nodes, the SIs and their CSIs sharing common features. We defined the CR model to target this issue of enabling the configuration designer to specify the input in a structured and generic manner. This is later on mapped (during the configuration generation process) to the instance level description (the individual SI/CSI level) that must be included in the AMF configuration.

The CR model is expressed as a UML domain model that includes a class diagram that structures the required input and constrained by a set of Object Constraint Language (OCL) [48] constraints that are used to validate the consistency of the input, and make sure that no AMF concepts are violated.

The rationale behind using UML to define the CR model can be summarized as follows: (1) the AMF model is standardized as a UML class diagram and therefore by using the same language we can easily map the concepts defined in the CR model to the AMF concepts in a consistent manner. (2) UML provides the artifacts needed to structurally specify our model and annotate it with the proper constraints. (3) UML is widely accepted standard for object-oriented modeling and therefore it is supported by a wide range of CASE tools, which facilitates the implementation aspect of our approach.

4.3.2 The CR Domain Model

The objective behind defining the CR model is to facilitate the specification of a large number of entities at the input level. For this purpose we have used the notion of templates to specify the entities that share common features. Whereby an instance of the template would include those features and the number of replicas that must be created based on the template. The three main templates of the CR model (shown in Figure 4-3) are the SI template, the CSI template and the node template. For example the SI template holds all the information relative to a single SI generated from the template. Depending on the redundancy model according to which we wish to protect the SI, the number of active and standby assignments varies, and the SI template allows the configuration of those numbers. Within the same SG, if the number of SIs of a certain service type is proportional to the number of other SIs of a different service type, we defined the notion of the proportional SI template, which is a specialization of the SI template. In other situations where the proportionality is not applicable then the regular SI template can be used. In an AMF configuration, an SI groups CSIs, similarly in our CR model, the SI template groups CSI templates. The semantics of our grouping is as follows: Each SI

generated from the SI template will group the number of CSIs specified in each of the CSI templates the SI template groups. For example, if an SI template with 5 SIs groups a CSI template with 3 CSIs, then each of the 5 SIs generated from the SI template will group 3 CSIs generated from the CSI template. The same approach is used with the node template to generate the node entities at a later stage.

Specifying the SIs/CSIs through templates is the first objective of the CR model; the second objective is to be able to specify the protection level required for each set of SI templates. For this purpose we have defined the SG template. The SG template specifies the required properties (e.g. redundancy model) of the SG that is expected to protect the SIs of the SI templates grouped within this SG template. If further information about the protection level is needed (e.g. the number of active assignment in an NwayActive redundancy model), it can be specified per SI template. This is completely aligned with the AMF specifications, where in the same SG different SIs may have different number of assignments (if the redundancy model allows it). The third and final objective of the CR model is to allow a flexible method of specifying the grouping of SIs per SG and application. In other terms, we want to give the configuration designer the flexibility of specifying which SIs can be serviced within the same application, and the distribution of SIs per SG. For this purpose we defined the notion of Administrative Domain with the following semantics: The SIs that are generated from SI templates that belong to different Administrative Domains must not be served by the same application instance. In order not to limit the SIs of the same template to be protected by only one SG, we allow (in the regular SI template) the specification of the minimum and maximum number of SIs per SG. For instance if within SI template SIT_a we specify the number of SIs to be 10, and the

maximum number of SIs (of this SI template) per SG to be 4, then we will generate 3 SGs in the AMF configuration to support the SIs of this SI template. However if within the same SG template of SIT_a we define SIT_b with 10 SIs and a minimum of 5 SIs per SG, then a maximum of 2 SGs can support the SIs of this template. In such case we have a conflict where the maximum of 2 is less than the minimum of 3 SGs (needed to support SIT_a). In other words we must not allow the configuration designer to specify conflicting information. For this purpose, and to make sure that the input does not violate any AMF constraints, we annotated our UML class diagram with various OCL constraints that can detect conflicting information either with the semantics we assigned to the CR model, or the semantics of the AMF specifications. For instance the OCL constraint shown in Figure 4-2 is applied to the SG template to detect the conflict between the minimum and maximum number of SIs across the SI templates of an SG template.

```
context MagicCrSgTemplate
inv CrSgtemp5:
let maxofmin: Integer =
(self. magicCrGroupsSiTemplates
->select(sit|sit. oclIsTypeOf(MagicCrRegularSiTemplate))
->iterate(sit, max:Real = 0|
if sit.magicCrRegSiTempNumberofSis/sit.magicCrRegSiTempMaxSis
> max then
max= (sit.magicCrRegSiTempNumberofSis/sit.magicCrRegSiTempMaxSis).ceil()
endif),
minofmax: Integer =
(self. magicCrGroupsSiTemplates
->select(sit|sit. oclIsTypeOf(MagicCrRegularSiTemplate))
->iterate(sit, min:Real = 0|
if sit.magicCrRegSiTempNumberofSis/sit.magicCrRegSiTempMinSis
> min then
min= (sit.magicCrRegSiTempNumberofSis/sit.magicCrRegSiTempMinSis).ceil()
endif)
in
(self. magicCrGroupsSiTemplates
->select(sit|sit. oclIsTypeOf(MagicCrRegularSiTemplate))
->forAll(sit|  sit. magicCrRegSiTempMaxSis <= maxofmin and
sit.magicCrRegSiTempMinSis >= minofmax)
```

Figure 4-2 Example OCL constraint for the SG template

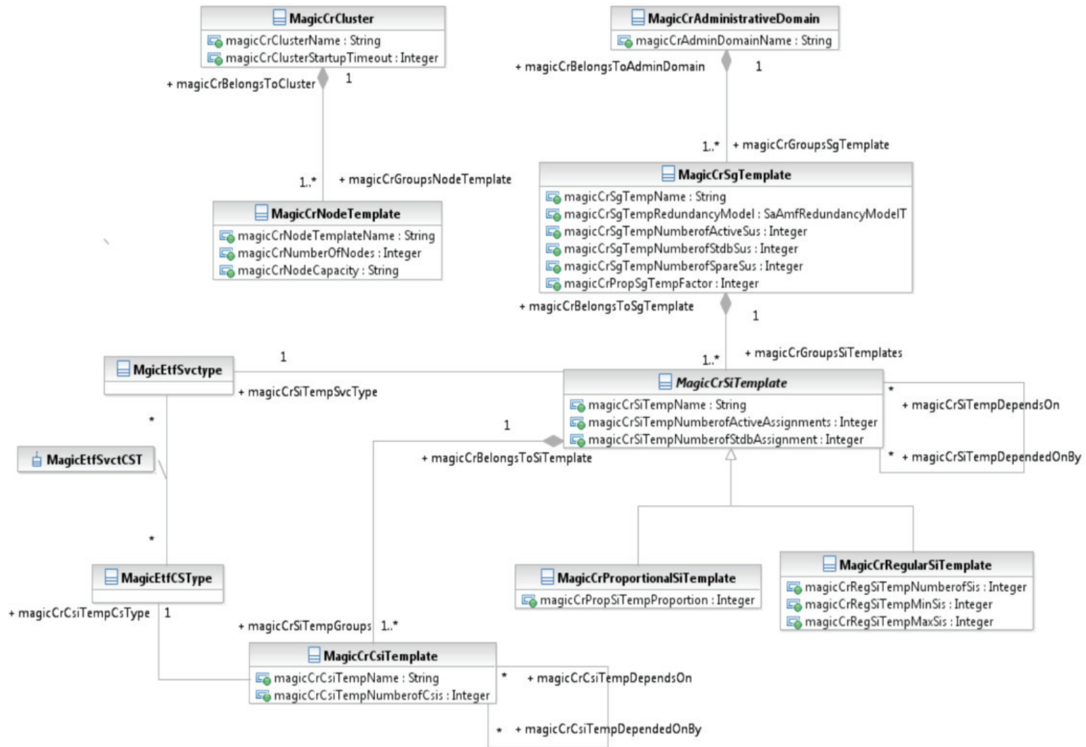


Figure 4-3 The UML class diagram of the CR model

4.3.3 Specifying and Analyzing Dependencies

Before explaining our approach for detecting and specifying the dependency, it is essential that we present and explain the dependency in AMF. In AMF managed systems, service providers are mainly software component instances (i.e. running processes) that collaborate and interact with each other to provide the services that are assigned to the service provider. A service provider that is not assigned any workload is basically idle, and not providing any service. Note that a service provider is either a component or an SU (which is basically a group of components), for simplicity's sake we will remain at the abstraction level of service provider.

The service provider types in AMF are derived from the ETF file service provider types. The ETF service provider type specifies the service types it can provide, and whether it can provide them independently, or it depends on other service provider types that we refer to as sponsoring service provider types. Any service provider must be assigned a workload for each service it is expected to provide. When the service consists of sponsoring another service provider (e.g. proxying a proxied component) then the sponsor service provider must be assigned a sponsoring workload. The type of the sponsoring service is also specified by the software vendor through ETF.

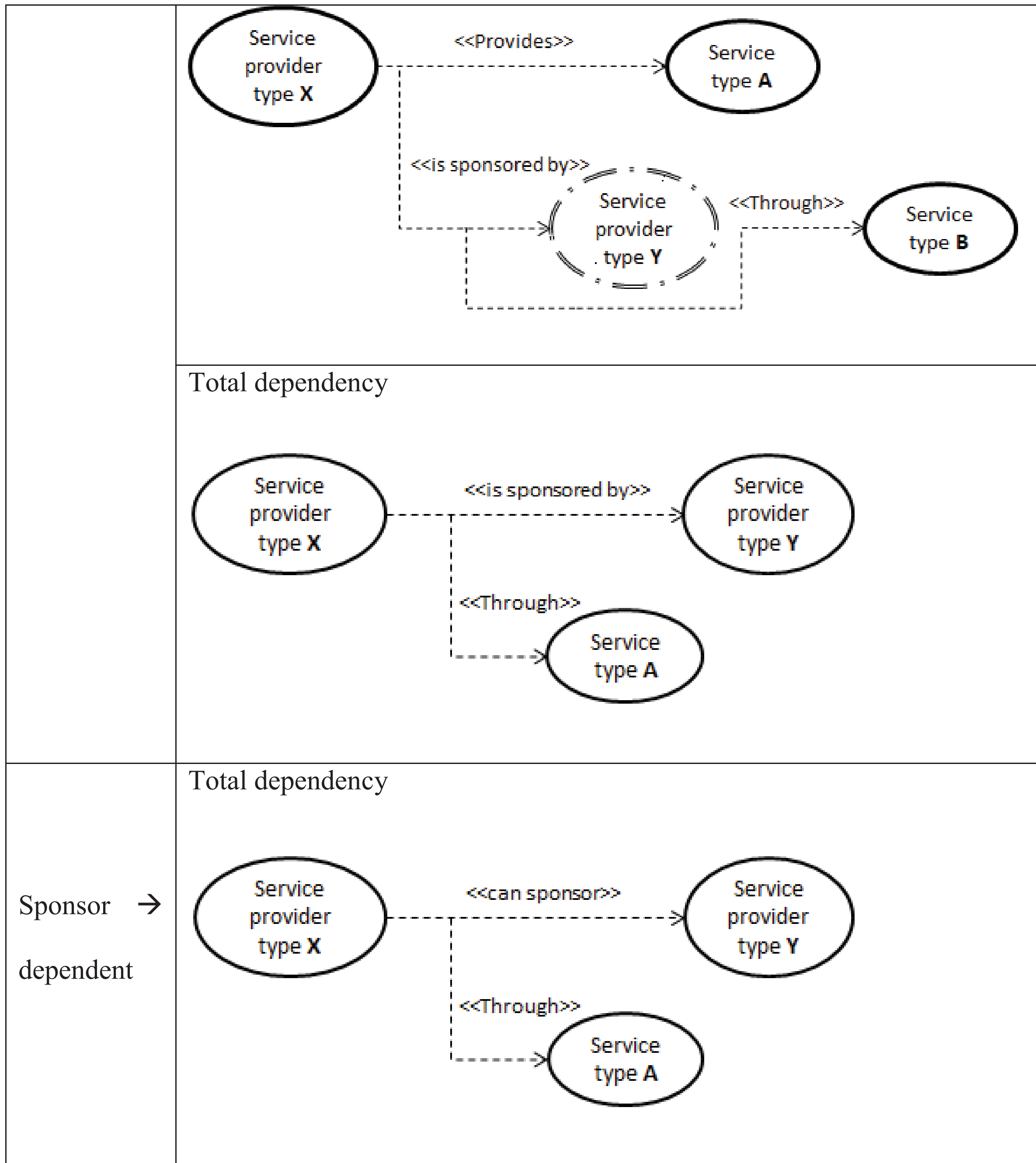
The software dependency can be specified in two different directions, either the dependent specifies the sponsor service provider type, and the sponsoring service type, or the sponsor will specify the dependent service provider type it can sponsor and the corresponding sponsoring service type. The latter direction is typically used in ETF when describing the prototype of a proxy software component that is developed to proxy another software component[2]. We also distinguish between total and partial dependency. In a total dependency the dependent depends on the sponsor in order to provide any service, e.g. if the dependent is contained in a virtual machine, it cannot operate or

[2] For example, when a software component is developed to proxy a legacy software component the latter one would be unaware of the proxy component.

provide any service until the sponsoring virtual machine is instantiated. In partial dependency the dependent only depends on the sponsor when providing specific service types. In the partial dependency, the sponsoring service provider type is not always specified (denoted in dashed borders in Table 4-1). Sometimes, and depending on the nature of the dependency, the sponsor service provider prototype is not specified in ETF, instead the dependent refers only to the sponsoring service type, and any sponsoring type capable of providing the sponsoring service type is a candidate to be selected to satisfy this dependency. Table 4-1illustrates the dependency description provided in ETF by the software vendor. For example the partial dependency specified in the first row is interpreted as follows: In order for a service provider of type *X* to provide any service of type *A*, it depends on another sponsoring service provider of type *Y*, to be assigned a sponsoring service of type *B*. And as aforementioned, sometimes only the sponsoring service type is specifying without referring to the sponsor service provider type.

Table 4-1 Service provider type dependencies

Dependency direction	Dependency order (partial/total)and description
Dependent →sponsor	***Partial dependency***



4.3.3.1 Issues and Challenges

The dependency that is specified in ETF at the service provider type level is actually captured in the system configuration at the instance level. In fact the only dependency

that can be specified in the system configuration is the service dependency and the instantiation dependency of the components. So the challenge here is how to map an ETF type dependency into a dependency at the AMF entity level.

Another issue is that the configuration designer may be unaware of the details of the software description including the existing dependencies, in fact one of the motivations behind the automated configuration generation is to relieve the designer from the burden of exploring the complete software description especially if the system is expected to host a significant number of software components. Therefore the designer may fail to specify the required sponsoring services, and hence even if the automated process was able to locate and include the sponsoring service provider in the configuration, it will not be assigned the sponsoring service simply because it was never defined in the first place.

Finally, it should be noted that the dependency specification consists of two main steps. First the required sponsoring services must be defined. And second the dependency relation among the dependent and sponsors must be specified. In other words it is not enough that the designer specifies the services. The designer must explicitly specify the dependency link between the dependent and the sponsoring service(s). Nonetheless the services are defined at the template level. Once again we face the problem of granularity that the automation is supposed to handle. So the question here is how can the dependency be specified in a generic way without going down to the individual SI/CSI level?

4.3.3.2 The Dependency Specification Approach

Our approach for solving the dependency issue consists of an interactive process that is based on analyzing both: the configuration designer input (in terms of the already specified services and dependencies) and the software description (i.e. available ETF(s)). When a service provider type dependency is detected, the input is examined to check whether the dependency is satisfied, or whether further actions are needed in order to satisfy the dependency. The approach for specifying dependency is illustrated in the activity diagram presented in Figure 4-4. The diagram is annotated on the right with 9 levels, merely to simplify the explanation of the approach.

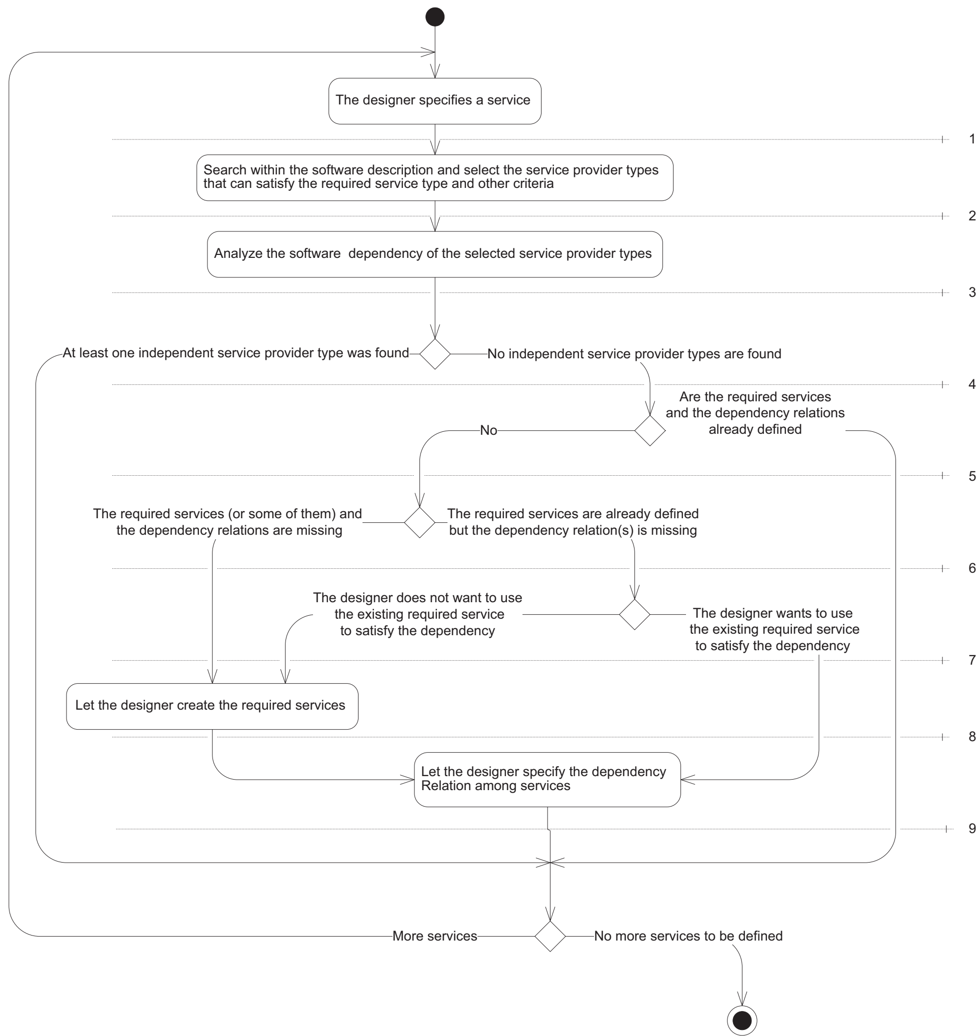


Figure 4-4 The dependency specification approach

Level 1: the designer specifies a service, in practice this specification is performed through a (SI/CSI) template, so in fact it is not a single service. The services of the same template all share the same service type (or CS type depending of the template), and therefore if a dependency is revealed later on, it is applicable to all the SIs/CSIs of this template

Level 2: The software description file, i.e. ETF is analyzed to find all the candidate service provider types that are capable of supporting the service defined in Level 1. Mainly a service provider type is selected if it can provide the required service type, and satisfy other criteria (discussed further in this chapter).

Level 3: The selected service provider types may depend on other service provider types in supporting the required service type. Therefore this dependency must be analyzed. The service provider type's dependency is described in ETF according to Table 4-1. So basically in this analysis step we will extract what are the sponsoring service types that are needed to satisfy the dependency.

Level 4: In fact the approach that will be used to generate the configuration at a later stage (whether it is a single or multiple configuration generation), will affect the decision made at this level. For instance, if among the service provider types selected, we found that several of them are independent and several are dependent, then, if the configuration generation approach is an exhaustive one that considers all possibilities, then the dependency must be satisfied for all the dependent ones. On the other hand, if the configuration generation approach opts for at least one independent service provider type, then there is no need for further dependency analysis. In either case the general dependency specification approach does not change, however, if the decision is made to specify all dependencies than the only modification for the approach would be to keep looping between Level 4 and Level 8 until every dependency is properly captured.

In this section, and for simplicity sake, if at least one independent service provider type is found, we do not consider the need to specify any dependency. Moreover when no independent service provider type is found, and more than one option of the dependent service provider types is found, we opt for specifying the dependency for only one of these options. Again specifying all options is simply a mechanical process of repeating the same approach over and over again until all options are considered.

Level 5: The service dependency does not necessarily always originate from a software functional dependency. And the designer must be given the means to specify dependencies among services regardless of the service provider type's dependency. However it might happen that the dependency that is already specified by the designer satisfies a selected service provider type dependency.

Level 6: As discussed in the previous section, the dependency specification consists of (1) defining the sponsoring services (2) defining the dependency relation that links those services. So basically at this level of our approach, we might find that either the sponsoring services are already fully defined but it is the dependency relation that is missing, or, neither the sponsoring service nor the dependency relation is defined and the designer needs to specify both.

Level 7: Even if all the sponsoring services are already defined. The designer may either choose not to use them and define new sponsoring services specifically for satisfying the dependency. Or use the existing sponsoring services and proceed to defining the dependency relation.

Level 8: the designer will specify the required sponsoring services, the service type of these services is already determined in the step illustrated in Level 3. So basically we will guide the designer by already letting her know the service type to be specified in the template.

Level 9: At this level the designer simply needs to specify the dependency relation among the already defined sponsoring services and the dependent ones.

If there are more services to be defined, the same approach is repeated.

4.3.3.3 The Dependency Relation Specification Mechanism

Once the sponsoring service type is revealed, specifying the dependency relation between a dependent service and a sponsoring service consists of linking the two services through a dependency relation. But what if there are 2 sponsoring services, then a dependency relation must be defined for each. In general, we might have N services of a dependent service type, that depend on M services of the sponsoring service type. Since ETF describes the dependency at the type level, and provides no information about the cardinality of the dependency (i.e. how many of the N service depend on how many of the M service, e.g. should each X of the N services depend on 1 on the M services?), then it is up to the configuration designer to figure out and specify this dependency relation for each service according to the semantics of the service.

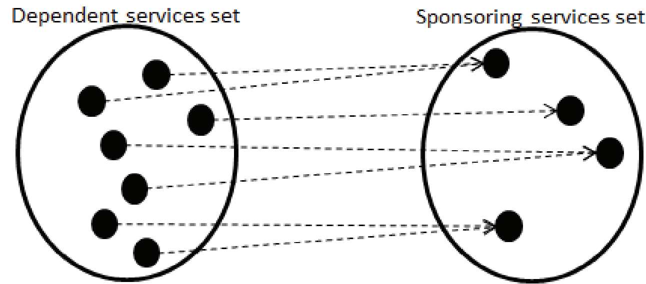


Figure 4-5 A dependency relation between two sets of services

Figure 4-5 illustrates an example of a dependency specification between two sets of services. In the first set, the services are of a dependent service type, and in the second set, the services are of sponsoring service type. In summary it is up to the designer to figure out the dependency relation and link the two sets, we assume that the designer knows the dependency relation according to the application specificities and we offer the support for specifying the dependency relation, i.e. linking the dependent services to the sponsoring ones.

Linking two sets of services is a demanding task if done individually on each of the set's elements. Therefore we need a notation to ease this task and raise the level of abstraction to the set level rather than the elements level. The challenging part is that the semantics of the relation vary from one relation to another; moreover the cardinality of the sets is also a variant.

In order to solve this problem we resort to relational algebra. Relational algebra is a formalism for creating relations among sets [49]. It has a strong formal foundation based on first order logic. A well-established language that is based on first order logic and supports the relations defined in relational algebra is the Structured Query Language

(SQL) [50]. SQL is a query language for relational databases. Our approach to solve the dependency specification is to leverage the capabilities of SQL in specifying the dependency relation by transforming our services into database records. The main objective is to give the designer means by which she can specify the dependency relation in a generic manner.

The dependency specification approach is as follows: first the services created as an instance of the CR model are stored in a database table as data records. Each SI/CSI is characterized by a base name (which is the template name and an index [between 1 and number of SIs/CSIs in the template). When it is time to specify the dependency, the configuration designer will specify three SQL queries specifying (1) the sponsoring set (which will create a database table holding this set), (2) the dependent set (which will also create a database table holding this set), and finally (3) the relationship between the sets representing the dependency. For example the relationship shown in Figure 4-5 can be specified through the below SQL query:

```
SELECT * FROM sponsoringServices, dependentServices
WHERE ([sponsorServiceIndex]*2 =[dependentServiceIndex]
or 2*[sponsorServiceIndex] - 1 = [dependentServiceIndex]);
```

Finally the dependency relation will produce a record set where each record consists of the sponsor and dependent service (SI or CSI, note that the same sponsor or dependent can appear in more than one record). This dependency information captured in the record set is then mapped into a service dependency in the instance of the CR model for which we are specifying the dependency.

4.4 Generating AMF Configurations

The overall process of generating configurations is shown in Figure 4-6. After specifying a complete CR information (with the proper dependencies specified), the following steps consist of (1) select the proper ETF types that can handle the requirements (2) create the AMF configuration based on the selected ETF types and the requirements.

The ETF types may specify capacity limitations on the component types and SU types. Therefore before being able to select the proper ETF types, a prerequisite step is needed. This step consists of determining the expected load of SIs per SU. Knowing the load is crucial (1) for evaluating the candidacy of certain ETF types, and (2) to determine the needed number of components and SGs to be created during the creation of the AMF configuration.

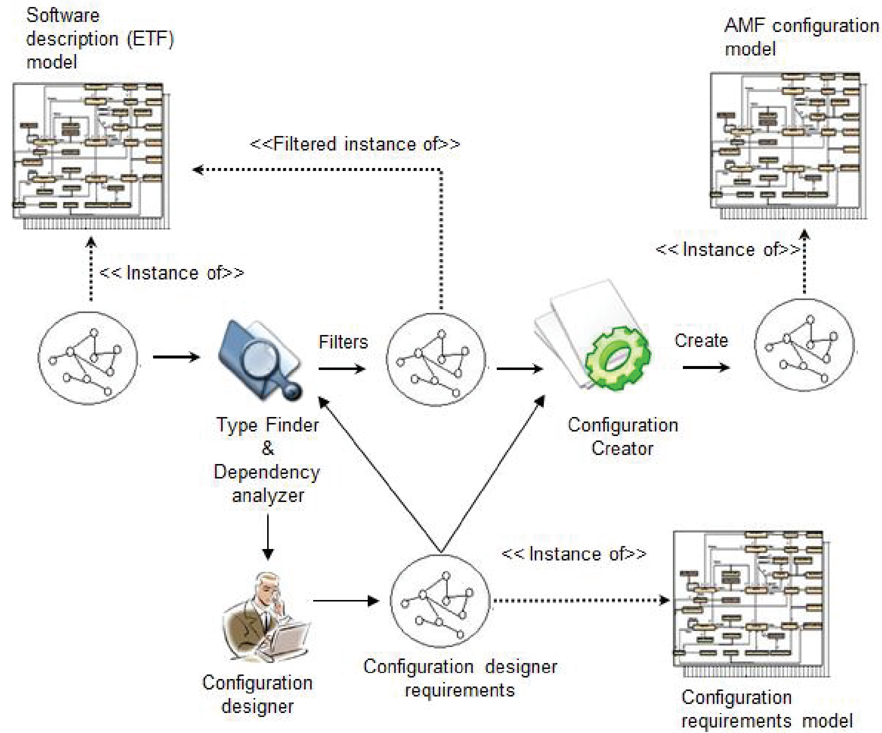


Figure 4-6 Configuration generation process

4.4.1 Determining the SI-Load an SU Is Expected to Support

In our CR model, each SG template can group multiple SI templates. The SIs of these templates are expected to be protected by one or multiple SGs, the number of SUs within each of these SGs is induced from the number of active/standby/spare SUs specified in the SG template. The question we need to answer here is: what is the minimum number of SIs of each SI template an SU of these SGs is supposed to be able to handle? We call this the minimum load of SIs the SUs is expected to handle. As we will see later on, this load is extremely important for the type selection, since ETF types may have limited capacity in terms of handling the SI/CSI load. This load is calculated according to a process of three steps.

Step 1: calculate the maximum number of SGs that are allowed to protect the SIs of a particular SG template

The reason why we need to know the maximum number of SGs is because we are looking at the minimum load of SIs an SU is expected to handle. And therefore by spreading the SIs over greater number of SGs, we decrease the density of SIs per SG, and subsequently the SG's SUs will have less SI load to handle.

The regular SI template specifies the minimum number of SIs that is required to be present within any single SG protecting this SI template SIs.

1. Integer minMAX = Integer.maxValue; // assigning an initial very large value
2. Integer min = 0;
3. **FOR** each $MagicCrRegularSiTemplate_i$ grouped by the MagicCrSgTemplate
    a. min = Ceil ($MagicCrRegularSiTemplate_i$.magicCrRegSiTempNumberofSis / $MagicCrRegularSiTemplate_i$.magicCrRegSiTempMinSis);
    b. **IF** min < minMax **THEN** minMax = min **EndIF**
4. **EndFOR**
5. **IF** minMax > MagicCrSgTemplate.magicCrPropSgTempFactor **THEN**
    a. minMax = MagicCrSgTemplate.magicCrPropSgTempFactor;
6. **EndIF** // the max number of SGs cannot be greater than the factor.

Algorithm 1 Calculating the maximum number of SGs

In Algorithm 1 we calculate the maximum number for SGs for each SI template (at 3.a). However we are interested in the maximum number of SGs allowed for the SG template and not simply a single SI template of the template and that is why we go for the minimum of all the maximum number of SGs (at 3.b). For proportional SI templates the number of SGs should not exceed the factor (at 5, 5.a).

Step 2: for each SI template determine the number of SIs that will be distributed on each SG

Based on the maximum number of SGs calculated in Step 1, in this step we determine the number of SIs from each template that will be assigned to each SG.

1. **FOR** each $\text{MagicCrSiTemplate}_i$ grouped by the MagicCrSgTemplate
    a. **IF** $\text{MagicCrSiTemplate}_i$ is a MagicCrRegularSiTemplate **THEN**
        i. $\text{MagicCrSiTemplate}_i$.expectedSIsperSG = Ceil ($\text{MagicCrRegularSiTemplate}_i$.magicCrRegSiTempNumberofSis / minMax); // the minMax is calculated from Step 1
    b. **ELSE**
        i. $\text{MagicCrSiTemplate}_i$.expectedSIsperSG = $\text{MagicCrSiTemplate}_i$.magicCrPropSiTempProportion * Ceil(MagicCrSgTemplate.magicCrPropSgTempFactor / minMax);
    c. **EndIF**

2. **EndFOR**.

Algorithm 2 Calculating the number of SIs per SGs

In Algorithm 2 we use the ceil value, since we are interested in the maximum number of SIs a certain SG may protect. We make the assumption that all the SGs protecting the SIs of the SI templates of a particular SG template are identical, and we create the configuration accordingly.

Step 3: Distribute the SIs protected by an SG over the SG's SUs

Now that we know what is the minimum number of SIs an SG must handle (from Step 2), we need to determine the load of SIs that each SU of the SG is supposed to support. In this analysis the SIs active and standby assignments are equally distributed among the SG's active and standby SUs. In case of Nway redundancy the SUs we don't distinguish between active and standby SU, however we still assume that the active and standby assignment will be equally distributed among the SG's SUs. That is, the SIs of each template are assumed to be equally distributed among the SG SUs.

We also include in this analysis the number of active and standby assignments the SUs are supposed to have on behalf of the SIs, e.g. if an SI has 2 active assignments than the active load it imposes on the SUs is doubled.

1. **Integer** `isFaultTolerant;` // can assume one of two values, either 0 or 1.

2. **FOR** each $MagicCrSiTemplate_i$ grouped by the MagicCrSgTemplate

a. **IF** MagicCrSgTemplate.magicCrSgTempRedundancyModel = Nway or NwayActive**THEN**

   i. MagicCrSiTemplate$_i$.activeLoadperSU = ceil ((magicCrSiTemplate$_i$.expectedSIsperSG * MagicCrSiTemplate$_i$.magicCrSiTempNumberofActiveAssignments) / (MagicCrSgTemplate.MagicCrSgTempNumberofActiveSus – isFaultTolerant));

   ii. MagicCrSiTemplate$_i$.stdbLoadperSU = ceil ((magicCrSiTemplate$_i$.expectedSIsperSG * MagicCrSiTemplate$_i$.magicCrSiTempNumberofStdbAssignments) / MagicCrSgTemplate.MagicCrSgTempNumberofActiveSus);

b. **ELSEIF** MagicCrSgTemplate.magicCrSgTempRedundancyModel = 2N or N+M **THEN**

   i. MagicCrSiTemplate$_i$.activeLoadperSU = ceil (magicCrSiTemplate$_i$.expectedSIsperSG / MagicCrSgTemplate.MagicCrSgTempNumberofActiveSus);

   ii. MagicCrSiTemplate$_i$.stdbLoadperSU = ceil(magicCrSiTemplate$_i$.expectedSIsperSG) / MagicCrSgTemplate.MagicCrSgTempNumberofStdbSus;

c. **ELSEIF** MagicCrSgTemplate.magicCrSgTempRedundancyModel = No redundancy **THEN**

i. $MagicCrSiTemplate_i$.activeLoadperSU = 1;

ii. $MagicCrSiTemplate_i$.stdbLoadperSU = 0;

d. **EndIF**

3. **EndFOR**

Algorithm 3 Calculating the load of SIs per SU

Each SU within the SG is expected to be able to support the active load and standby load (calculated in Algorithm 3) for each SI template of the SG template.

4.4.2 Top-Down Type Selection Criteria

In the previous approach discussed in [47], the type selection was performed using a bottom up approach that starts by selecting the component types and works its way up to the application type. However this approach would group together component types that belonged to different SU types and therefore they may not be suitable for this grouping. To rectify this issue we propose here a Top-down approach that starts at the application type level and works its way to the component type. Moreover we relaxed some assumptions made in [47] such as having the same service type of SIs of the SG. As a result our search algorithm was modified to accommodate the new requirements. Note that in ETF not all types are mandatory, in other words if there are no restrictions on how a type can be grouped, then a parent type is not needed, we refer to such unrestricted types as orphan types that are not grouped by parent types. In such case our configuration generation method will create the missing types as AMF types.

4.4.2.1 Application Type Selection

The selected application type must simply group a proper SG type, i.e. an SG type that satisfies the SG type selection criteria discussed next.

4.4.2.2 SG Type Selection

The selected SG type must support the proper SU type (by proper we mean that it satisfies the SU type selection criteria discussed next), in addition it must have the redundancy model specified by the SG template.

4.4.2.3 SU Type Selection

Any selected SU type must support all the service types specified by the SI templates of an SG template, in addition the component types of the SU type must support all the CS types specified in all the CSI templates of all the SI templates of a given SG template, with the required capacity.

4.4.2.4 Component Type Selection

In order to determine the proper component type, it needs to satisfy three criteria:

- It needs to provide the required CS type (i.e. the CS type specified in the CSI template).

- It needs to have the appropriate capability model with respect to the CS type for which the component type is selected. For instance, if the components of this component type are to be used in an SG that has a redundancy model of Nway, then the component type's capability model should be *x_Active-and-y_Standby.*
- It needs to have the required capacity in supporting all the CSIs of the required CS type. This capacity is determined by two factors: (1) the component capability model with respect to the CS type. (2) The maximum number of components of this component type in a single SU.This is applicable if the component type has a parent SU type that limits the number of components of this type per SU. Hence the above mentioned capacity is a product of the two factors.

A prerequisite step to determining the eligibility of a component type, is to determine the CSI load of each CS type the components of this component type are expected to provide.

In order to calculate the load of CSIs of each CS type, we first create a list of all the CS types in the SG template, and then we associate with each CS type the load of CSIs that refer to it. Only one component type is used to support a particular CS type.

1. **LIST cstList** // an empty list that will hold the CS types
2. **FOR** each $\text{MagicCrSiTemplate}_i$grouped by the MagicCrSgTemplate
    a. **FOR** each $\text{MagicCrCsiTemplate}_j$grouped by $\text{MagicCrSiTemplate}_i$
    b. **IF** $\text{MagicCrCsiTemplate}_j$.magicCrCsiTempCsType $\notin$ **cstList THEN**
        i. **Add** $\text{MagicCrCsiTemplate}_j$.magicCrCsiTempCsType **TO cstList**

c. **EndIF**

3. **EndFOR**.

Algorithm 4 Creating the list of all the CS types that are referred to by the CSI templates of an SG template

Now that we have a list of all the CS types that the components of any SU within the SGs (The SGs that will be created to protect the SIs of the SI template of a SG template) must support, we will calculate the load of CSIs associated with each CS type.

1. **FOR** each CS $type_i$ that belongs to **cstList**
   a. **FOR** each $MagicCrSiTemplate_j$ grouped by the MagicCrSgTemplate
      i. **FOR** each $MagicCrCsiTemplate_k$ grouped by $MagicCrSiTemplate_j$
         1. **IF** $MagicCrCsiTemplate_k$.magicCrCsiTempCsType = CS $type_i$ **THEN**
            a. CS $type_i$.activeLoadperSU += $MagicCrSiTemplate_j$.activeLoadperSU * $MagicCrCsiTemplate_k$.magicCrCsiTempNumberofCsis;
            b. CS $type_i$.StdbLoadperSU += $MagicCrSiTemplate_j$.stdbLoadperSU * $MagicCrCsiTemplate_k$.magicCrCsiTempNumberofCsis;
         2. **EndIF**

ii. **EndFOR**

b. **EndFOR**

2. **EndFOR**

Algorithm 5 calculating the load of CSIs associated with each CS type

In the component type selection process, any component type selected to support a certain CS type, must also have the capacity to support the load associated with the CS type. It should be noted here that the limited capacity does not apply to orphan component types since they have unlimited capacity because they do not belong to an SU type that would limit the number of components of this component type in an SU.

4.4.3 Creating AMF Types and Entities

After selecting the proper ETF types, our method proceeds by creating the AMF types that are derived from the ETF types. The mapping from the ETF types to the AMF types is performed by creating an AMF type and assigning the attribute values of this type the values specified in the corresponding ETF type. In case of the ETF attributes where a range is specified instead of a specific value, then we go with the upper limit of the range. In case of the orphan ETF types that are selected, we create an AMF type that groups the orphan types.

After creating the AMF types, it is important to determine the number of entities of each type we must create. The number of applications does not have an upper limit. The number of SGs per SG template is bounded by a minimum and a maximum as we have

seen in Section 4.1. The number of SUs per SG is predefined and specified in the SG template. Finally the number of components needs to be determined.

4.4.3.1 Creating Applications

By definition, the SIs of the templates of the same administrative domain can belong to the same application, however if for a single administrative domain more than one application type is needed (this is the case when the proper SG types of the SG templates of the administrative domain belong to different application types). Than we create as many applications as the number of proper application types selected.

4.4.3.2 Creating SGs

It is important here to note, that in Section 4.1 we calculated the maximum number of SGs, but we still don't know what the minimum required number of SGs is. The required number of SGs is not necessarily the maximum. If that is the preference then we can proceed and always create the maximum number of SGs. However this will increase the size of the configuration and thereafter may increase the administrative complexity, e.g. when upgrading, there will be more SGs, SUs and other entities to upgrade. If on the other hand the preference is to minimize the number of SGs of a selected SG type, then we propose the following approach:

(1) If the selected component types have unlimited capacities (e.g. they are orphans), then we can go with the minimum number of SGs. The minimum number of SGs is calculated based on maximizing the number of SIs protected by each SG.

1. Integer maxMin = 0;

2. Integer max = 0;

3. **FOR** each $MagicCrRegularSiTemplate_i$ grouped by the MagicCrSgTemplate

   a. max = Ceil ($MagicCrRegularSiTemplate_i$.magicCrRegSiTempNumberofSis / $MagicCrRegularSiTemplate_i$.magicCrRegSiTempMaxSis);

   b. **IF** max > maxMin **THEN** maxMin = max **EndIF**

4. **EndFOR**

5. **IF** maxMin > MagicCrSgTemplate.magicCrPropSgTempFactor **THEN**

   a. maxMin = MagicCrSgTemplate.magicCrPropSgTempFactor;

6. **EndIF** // the min number of SGs cannot be greater than the factor.

Algorithm 6 calculating the minimum number of SGs

(2) If the component types have limited capacities (i.e. they belong to an SU type). Then we can calculate the maximum component type capacity, and based on that we figure out the maximum capacity of the parent SU type in supporting the SIs of each SI template. However the problem with using this approach is that it cannot solve the complexity of our problem. This problem maps to the knapsack problem, where the SU capacity represents the sack, and the different load of CSIs each SI presents represents the weights. And therefore this problem is NP complete, and cannot be solved in polynomial time.

4.4.3.3 Overcoming the Complexity of Calculating the Required Number of SGs

In order to avoid the complexity of this problem, we use a different approach and instead of starting with the SU capacity to calculate the minimum required number of SGs, we start by looking for the minimum required number of SGs, and then check if the SUs capacity can handle that minimum. In other words, since, we know by now how to calculate the minimum and maximum **allowed** number of SGs, the question is how to determine within this range the minimum **required** number of SGs. The solution of our search problem comes in the form of a binary search algorithm that works as follows:

1. Integer newMinRequired = 0;
2. Integer answer = 0; // holds the final value of the minimum required number of SGs
3. Integer maxMin; // equals the value obtained by running Algorithm 6.
4. Integer minMax; // equals the value obtained by running Algorithm 1, or maxMin+1 if unbounded
5. **IF** the selected SUT **can** handle the load of SIs based on maxMin **THEN**
    a. answer = maxMin; **Exit**;
6. **EndIF**
7. **While** (TRUE)
    a. newMinRequired = Floor((maxMin + minMax)/2);

b. Calculate the load of SIs per SU based on newMinRequired

// this includes repeating the steps in Algorithms 2→5

c. **IF** the selected SUT **can** handle the new load of SIs **THEN**

// decrease the # of SGs

i. newMinRequired --;

ii. Calculate the load of SIs per SU based on newMinRequired

iii. **IF** the selected SUT **cannot** handle the new load of SIs **THEN**

1. answer = newMinRequired +1; **Break**;

iv. **ELSE** // the SUT can still handle the load, so we need to decrease furthermore the # of SGs

1. minMax = newMinRequired;

v. **EndIF**

d. **ELSE** // the selected SU **cannot** handle the load, so we increase the number of SGs

i. newMinRequired ++;

ii. Calculate the load of SIs per SU based on newMinRequired

iii. **IF** the selected SUT **can** handle the new load of SIs **THEN**

1. answer = newMinRequired; **Break**;

iv. **ELSE** // the SUT still cannot handle the load, so we need to increase furthermore the # of SGs

1. maxMin = newMinRequired;

v. **EndIF**

e. **EndIF**

8. **EndWhile**

Algorithm 5 Calculating the minimum required number of SGs

The algorithm above will perform a binary search between the minimum and maximum number of SGs. For each iteration, the load of SIs per SU will be revaluated based on the modified number of SGs. Then the SUT will be re-examined based on its capacity. The process will continue until we find the minimum required number of SGs. The assumption here is the each SI template of the SG template will impose an equal load of its SIs on the SUs of the SG.

4.4.3.4 Creating SUs

The number of SUs to be created within an SG is rather simple; it is simply the sum of the number of active/standby/spare SUs specified by the designer in the SG template.

4.4.3.5 Creating components

Based on the minimum required number of SGs calculated in Section 4.3.2, we repeat the analysis done in Sections 4.4.1 and 4.4.2.4 to determine the exact CS type load of CSIs the selected component type is expected to handle. And based on this number and the max capacity of the component type (active and standby), we can determine the required number of components of the selected component type.

4.4.3.6 Creating the Nodes and the Cluster

The number of nodes to be created is specified in each node template. We have only one cluster that can be created based on the CR information. The SUs are distributed on the nodes in a round robin manner in order to guaranty (as much as possible) hardware redundancy, whereby we try not to distribute two SUs of the same SG on the same node (if the number of nodes is large enough to allow it).

4.4.3.7 Populating the Configuration Attributes with the Proper Values

The attribute values of the AMF configuration come from various sources listed below

- **User specified:** these values are specified by the configuration designer for the services and the cluster, and they are directly mapped to the created entities.

- **ETF mapped:** most of the types' attribute values are specified or bounded by ETF. Most of the values of the AMF type attributes are mapped from their ETF counterparts. Some of the AMF entities attributes get their default values from their types

- **Calculated:** some attributes values must be calculated. Those attributes are configuration specific and must be calculated within the context of each configuration. A good example of such attributes is the attribute that specifies the rank of each SU of the SG with respect to a particular SI. This rank specifies which SI is assigned to which SU at runtime. It is through this attribute that we can guarantee runtime load balancing through the AMF configuration. We will discuss this issue in more details in Chapter 5.

- **Undefined:** a very small portion of the attribute values remain undefined. This is due to the fact that the values of those attributes depend on the specificities of the deployment platform. These attributes must be specified manually after the configuration is generated. A good example of such attributes is the relative path to a Command Line Interface (CLI) command which needs to be adjusted to the execution environment where the component will be deployed. (The current configuration generation method maps the same value defined in ETF, however this value may need adjustments).

4.5 From One to Multiple Configuration Generation

Based on a CR instance specified for the configuration generation, more than one AMF configuration can be generated, for example consider the case where more than one ETF

type supports a required service. Since we do not have any data that favors one type over another (e.g. the type reliability or performance) we decided to explore all the possible valid configurations for a particular input. The goal is to be able thereafter to compare and rank those configurations. Therefore after designing and applying the automatic generation of an AMF configuration, we decided to leverage this process and add to it the capability of generating multiple configurations.

There are four main contributors to the existence of multiple valid configurations for the same input: (1) multiple ETF type selection options (2) multiple AMF types can be created from the same ETF type (3) the number of entities to be created is not fixed (4) multiple deployment options. The multiple configuration generation approach is a combinatorial one that explores all the possible configurations that can be generated based on the same input. And therefore the complexity of the problem grows exponentially with more options to explore. In order to reduce the solution space explosion and reduce the complexity of the problem, we have made several reasonable assumptions among which for instance (1) we do not select multiple component types to provide the same CS type within the same SU. The rationale behind this decision is that at runtime it is up to AMF to decide which component is assigned which CSI. And if a preferable component type was chosen to provide a CS type, then there is no guaranty that AMF will assign the CSIs of this CS type to the components of the preferred component type, if there are other components of other component types that can do the job. (2) We opt for the minimum required entities to be created, as discussed in Section 4.3.3.2. (3) We do not explore all the possible deployment options. The multiple configuration generation approach was designed to identify the selection points for which

optimizing criteria need to be identified. In other words, we can design and configure the same system in many different ways and yet provide the same functionality. Therefore analyzing the non-functional properties of each configuration will provide criteria needed to compare and rank AMF configurations according to certain measurements such as the availability.

4.5.1 Approach for Multiple Configuration Generation

Our approach for multiple configuration generation is based on finding all the types that can satisfy the input requirements. For instance, for a given administrative domain AD1, one or more application types may be found suitable to satisfy the requirements imposed by this administrative domain. In this case for each possible combination of types that satisfy the requirements, we build what we refer to as a type stack. A type stack is a set of types starting from a set of component types in the bottom of the stack, grouped by an SU type that in turn belongs to an SG type that in turn belongs to an application type. For each administrative domain multiple type stacks may be built based on how many suitable candidate types our search algorithms can find at each level. Note that it is only mandatory to have the component types in the type stack. Note that the dependency handling differs in multiple configuration generation, since here we are exhaustively selecting the candidate types, and therefore for the same SI or CSI template all the existing dependencies will have to be specified.

Figure 4-7 illustrates a multiple configuration generation example where for the same configuration requirements comprising multiple administrative domains, the types stacks

(built for each administrative domain) are combined to produce a different configuration corresponding to each combination.

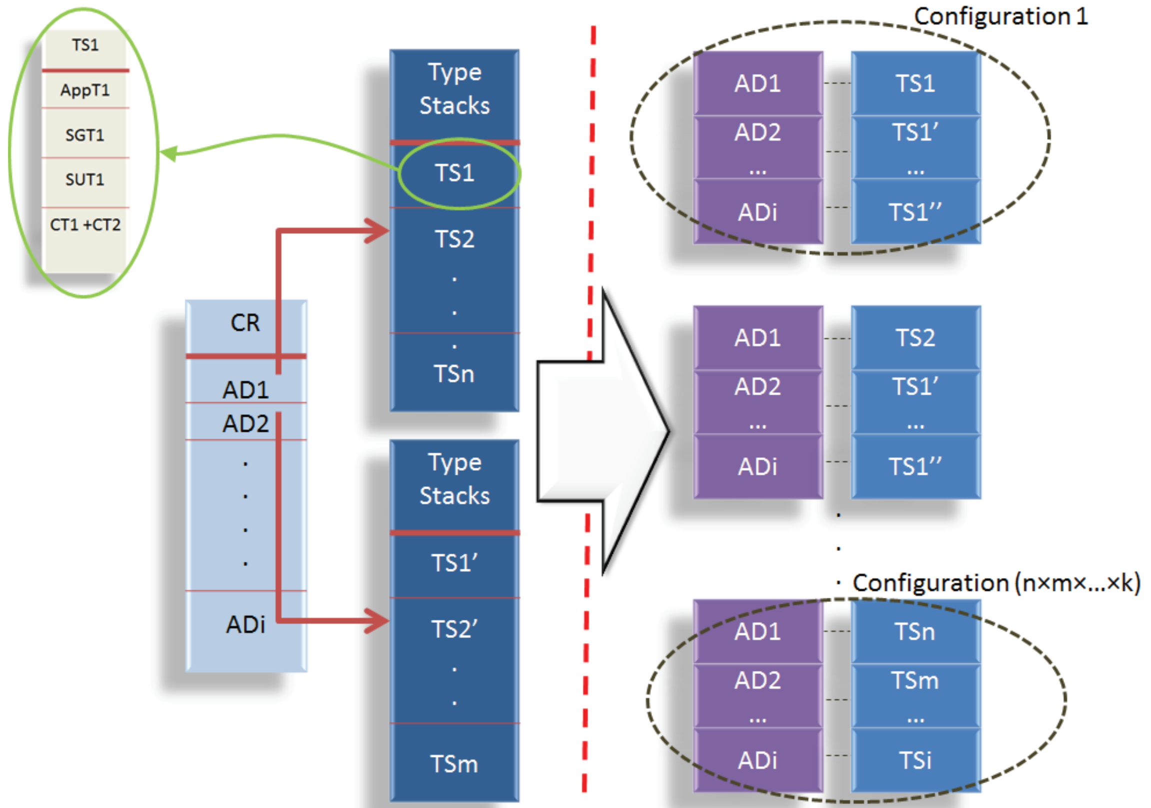


Figure 4-7 Multiple configuration generation illustration

4.6 Summary and Discussion

Generating AMF compliant configurations is a tedious and error prone task. We can mitigate this complexity through automation. Nonetheless we need to make sure the compliancy is captured throughout the automation process. In this section we discuss this issue as well as other challenges that we faced in during the automation of configuration generation.

4.6.1 Compliancy with the AMF Specifications

Beginning with the input, we defined various OCL constraints on the CR model to ensure that the input is consistent and complete. Within the context of our project, the authors in [22] defined a UML domain model that restructures the standard AMF configuration model and annotates it with over 90 OCL constraints that capture and further refine AMF concepts and constraints from the standard specification. This domain model, including all the OCL constraints, has been used as a basis for the design and implementation of a configuration validation tool [24], which was used in different sub-projects in the group to validate generated or third party provided configurations.

For ensuring compliance by construction with the standard, several of the constraints in the domain model were implicitly embedded directly into our configuration generation algorithms (e.g. the constraint specifying the allowed component capability model to be used with a certain redundancy model). However we could not embed directly all of these constraints into our method. This is mainly because of the large scope that certain constraints have. Such a scope can spread over several types and entities. Such constraints can only be invoked after the configuration is generated and not during the generation. For validating our configuration generation technique, we checked the compliance of our generated configurations to the standard using the validator. We defined a test plan in which we generated and checked various configurations that include various corner cases and typical cases.

4.6.2 Selection of Orphan Types

The selection of an orphan type requires the population of the attributes of the created parent type. These attributes may require a good understanding of the software implementation and therefore may not be derived automatically. An example of such a type attribute is the component restart probation period found in an SG type. This attribute applies to components in service units belonging to a service group of this type. If the AMF SG type is created (instead of being derived from ETF) then this attribute needs to be determined without any guidance from the vendor. We need to introduce artificially a value, which will affect the software behavior at runtime and therefore impacts the quality of the generated configuration. It is still unclear how to determine safely the attributes of the different entity types when they are created.

A related observation is that it is necessary to maintain the information whether a type was created by our method or provided by the software vendor in the ETF. Created types do not reflect implementation limitation and therefore should not restrict the use of orphan types. However they may be reused whenever they are appropriate to ease the type creation task.

4.6.3 Selection of Higher Level Types

In the single configuration generation approach, our preference is to only analyze orphan types when the non-orphan ones have been checked and found unsatisfactory with respect to the configuration requirements. However, there might be situations where we have an ETF (or multiple ETFs) where more than one type can be a candidate. This is typically the case for different versions of software or that offers similar services but provided by

different vendors. Some vendors (versions) may constrain the way their components should be configured by having the corresponding component types refer to SU types, while other vendors may provide components that do not need to be constrained from the ETF perspective. In other words, the component types corresponding to these components are orphans. Depending on the required services and the deployment system, there might be situations where orphan component types are a better choice than non-orphan types due to their flexibility. This comes at the price of the difficulties of type creation as aforementioned. Our technique is easily adaptable to select types in any desirable order.

4.6.4 Multiple Configuration Generation

Multiple configuration generation is a more complicated process than the one generating single configurations. This complexity is not only limited to defining the process but also to executing it. For instance while the algorithms for single configuration generations have a polynomial time complexity, in multiple configuration generation this complexity becomes exponential. In addition, having multiple configurations without the proper metrics to evaluate them is impractical. With highly available systems it is extremely important to be able to characterize a system configuration with the level of availability it offers to its services. In Chapter 6 we define an approach to evaluate the service availability of a given AMF configuration which enables us to compare AMF configurations and select the one that best satisfies our requirements.

5 Workload Balancing through AMF Configurations

In this chapter we target the issue of workload balancing through the information specified in the AMF configuration. More specifically we want to make sure that at runtime, AMF will equally distribute the SIs among SUs, and in case a failure occurs, we want to keep this workload assignment balanced.

The contributions discussed in this chapter are as follows:

(1) Defining an approach for workload balancing in NwayActive redundancy model and discuss its applicability to Nway.

(2) Defining an approach for workload balancing in N+M redundancy models

5.1 Introduction

As aforementioned in previous chapters, the assignment of SIs to SUs is performed at runtime, this assignment, and the re-assignment after an SU failure, is performed for each SI according to its ranked list of SUs. The ranking is established at configuration time. In this chapter, we will demonstrate that if the ranked list is determined according to conventional algorithms, like round robin, the shifting of SIs from a failed SU to healthy ones may lead to an unbalanced workload among the SUs, which may lead to overload causing subsequent failures and performance degradation. To ensure a continuous load

balancing in the presence of a failure, we propose an approach that views the issue as a Constraints Satisfaction Problem (CSP) [51].

The AMF specification defines five different redundancy models, in 2N and No-redundancy the workload is balanced by default, for the remaining redundancy models, we define the ranking techniques for NwayActive and N+M, and discuss how the solution can be extended to support Nway. We first introduce the ranking mechanism, and the existing solutions in the literature, and then we present our solutions for both NwayActive and N+M redundancy models.

5.2 The Ranking Mechanism for Runtime SI Assignment

The ranking mechanism defined in [4] indicates in the configuration the preferred SI assignments applicable at runtime. Thus it allows a configuration time solution to be executed at runtime. SIs may have different preferences toward the SUs. This preference can be captured through ranking and used by AMF (for certain redundancy models) to distribute the workload assignments. The ranking is applied as follows:

(1) Each SI has a ranked list of all the SUs in the SG protecting the SI.

(2) At runtime AMF assigns to the SU with the highest rank (lowest integer value) the active state on behalf of the SI. The SU with the next highest rank will be either the standby SU for this SI (e.g. in Nway), or the second active one (in case of NwayActive).

(3) In case of an SU failure, its active assignments will be given to the standby SU(s) with the next highest ranks for each SI. In case of NwayActive the SU with the next higher rank will get the active assignment.

Figure 5-1 shows an example SG with NwayActive redundancy model.

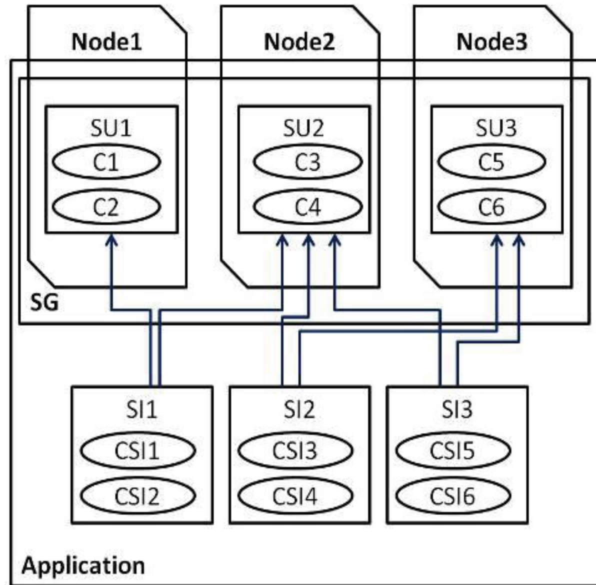


Figure 5-1 An example of NwayActive SG

A corresponding ranking for the assignments shown in Figure 5-1 is presented in

Table 5-1. Note that in this example the *numberOfActiveAssignments* for each SI is configured to two. Therefore for SI1 it is SU1 and SU2 that get the active assignments since they have the highest ranks (lowest integer values). In case SU1 fails, SU3 will get an assignment for SI1 since it is the SU with the next highest rank.

Table 5-1 A ranking example

	SU1	SU2	SU3
SI1	Rank=1	Rank=2	Rank=3
SI2	Rank=3	Rank=1	Rank=2
SI3	Rank=3	Rank=2	Rank=1

5.3 Predefined Workload and Workload Assignment: a Motivating Example

In AMF managed systems, the workload that will be assigned to the SUs is defined as SIs. Each SI in the system configuration is defined based on the semantics and interaction of the service and the distribution of the resources. Considering a telephone billing system as a motivating example, it may be composed of billing software and an in-memory database that stores the customers' accounts. When the database is too large to be deployed on a single node, it is distributed among the nodes of a cluster. Each node will have the same billing software and a different portion of the database[3]. To shorten response time it is preferable that when a customer makes a phone call, the billing software processing this call executes on the node hosting the database portion that includes this customer's account information. The SU is then defined as being the execution of the billing software and the database portion (including the DBMS)

[3] The redundant replicas are also distributed on the standby nodes. Any node will have (remote) access to the complete database, but can only accommodate a portion of the database in-memory.

collocated on the same node. The SI is then the workload of processing the calls of the users whose account information is stored in the database portion of the SU. The standby SU for this SI must also be on a node that stores the replica of the exact portion of the database. In this case, it is important that the workload assignment is predetermined at configuration time, so that the SI assignment and the deployment of the database match. In our approach we assume that the configuration designer defined the SIs such that they impose identical load on the SUs. The process of defining the SIs to meet such a criterion is outside the scope on this dissertation, and is perhaps better addressed using approaches similar to the ones defined in[52][53].

5.4 Existing Workload Balancing Solutions

The peculiarity of our problem comes from the fact that the workload must be specified at configuration time. Load balancing in general has been addressed by both runtime and static algorithms [54]. Existing runtime workload balancing techniques [55][56][57] consider only actual workloads and would select the least busy SU for each incoming call which in our example would then likely to result in remote database access at the call processing time thus increasing the response time. In short, such solutions are agnostic of the assignment preferences. For instance in the work presented in [57], a runtime load-balancing dynamic scheduling (LBDS) algorithm is proposed, where in case of a machine failure it partially re-adjusts the original scheduling solution, with the aim of maximizing the machine utilization ratio, nonetheless the assignment preference is not considered. A relevant conventional configuration time workload balancing algorithm that we can use as a reference is round robin [58][59]. In this case the assignments can be determined and

balanced at configuration time; nonetheless this algorithm has several drawbacks we discuss in the upcoming sections. An interesting work was presented in [60] where an approach for load balancing in the presence of a random node failure is proposed; however several assumptions like knowing the mean time to fail and the mean time to recover are made, as well as nodes knowing the initial workload of other nodes. In [61] the authors propose a load balancing algorithm for distributed systems with N processors. Their aim is to determine the best task-processor assignment given that they are provided with certain parameters such as the probability task i fails on processor x and the time needed to restart this task on this processor. They do not consider standby processors. Our problem is different by nature, and our analysis does not require any failure information.

The work presented in [52][53][62] focuses on finding the optimal static load balancing strategy which determines the optimal load at each host in order to minimize the mean job response time. The objective is to define a static job-scheduling policy to be used at runtime. Such techniques can be useful in defining the SIs and their scope. Nonetheless they do not consider load balancing after failure.

In [63] the authors address the problem of static assignment of non-partitioned files in a distributed storage subsystem, they assume that the file accesses exhibit Poisson arrival rates and that service times have a fixed duration, their solution does not address load balancing after failure. This work is useful in determining the optimal database distribution, which is a prerequisite of our work presented in this chapter.

5.5 Workload Balancing in NwayActive

In this section we explore the issues that may arise when defining the ranks at configuration time, and we present our approach to surmount these issues.

5.5.1 The Problem with Conventional Load Balancing Algorithms at Configuration Time in NwayActive

We examine the problem of load balancing after a failure using the conventional round robin algorithm. As a case study for the *NwayActive* redundancy model we will discuss a scenario with 14 SIs protected with an SG of 6 SUs. The *numberOfActiveAssignments* for each SI is set to 3, i.e. each SI is configured to have three active SUs.

Table 5-2 A ranked list of SUs using a round robin algorithm

	SU1	SU2	SU3	SU4	SU5	SU6
SI1	1	1	1	2	3	4
SI2	2	3	4	1	1	1
SI3	1	1	1	2	3	4
SI4	2	3	4	1	1	1
SI5	1	1	1	2	3	4
SI6	2	3	4	1	1	1
SI7	1	1	1	2	3	4
SI8	2	3	4	1	1	1
SI9	1	1	1	2	3	4
SI10	2	3	4	1	1	1
SI11	1	1	1	2	3	4
SI12	2	3	4	1	1	1
SI13	1	1	1	2	3	4
SI14	2	3	4	1	1	1

Table 5-2 shows a ranked list of SUs generated using a round robin algorithm. According to this ranking, at runtime, AMF will assign the HA active state for each SI in the following manner:

- SU1, SU2 and SU3 will be assigned the HA active state for SI1, SI3, SI5, SI7, SI9, SI11, SI13.
- SU4, SU5 and SU6 will be assigned the HA active state for SI2, SI4, SI6, SI8, SI10, SI12, SI14.

The cells with the bold numbering (of **1**) correspond to the active assignments. Based on this ranking the load will be evenly distributed among the SUs. Each SU will have 7 active assignments. In absence of failure, a round robin algorithm solves the problem of ranking with load balancing.

Now let us assume that SU1 (or SU2 or SU3) fails, all its workload would be shifted to SU4 causing it to bear twice the load of any other SU. All the active assignments of SU1 are shifted to SU4 because it has the next highest rank for all the SIs (i.e. SI1, SI3, SI5, SI7, SI9, SI11, and SI13) assigned to SU1 (and SU2 and SU3). In other words, SU4 can be looked as a "backup" for SU1 with respect to all its assignments.

The shortcoming of the round robin algorithm is that it develops a repetitive pattern as shown in Table 5-2. After SI2, the ranking pattern keeps reappearing after every two SIs. SU1 and SU4 are the only SUs that are backing up the other SUs in case of failure as illustrated in Figure 5-2, which is derived from Table 5-2. SU1 is backing up SU4, SU5 and SU6, while SU4 is backing up SU1, SU2 and SU3.

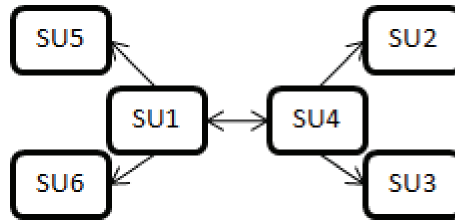


Figure 5-2 The SU back up graph using the round robin algorithm

5.5.2 A Ranking Solution to Ensure Load Balancing Before and After One Failure in NwayActive Redundancy

We have a predefined set of SIs that will be assigned to a set of SUs. The portion of SIs that each SU is supposed to serve can be easily computed, and a corresponding ranking can be produced. However, one main concern as shown in the previous section is what happens after an SU failure. How will the load assigned to the failed SU be distributed among the remaining SUs? We want the SUs to be assigned equal initial load, and in case of failure, we want the load of the failed SU to be evenly assigned among the remaining SUs to maintain a balanced load and avoid cascading failures caused by overload. Our main problem is to capture this at configuration time through the SU ranking for SIs.

As shown in the previous section the problem of the round robin algorithm is in the re-assignment of the SIs to the remaining SUs after an SU failure. Only one SU, e.g. SU4 will be re-assigned all the SIs initially assigned to SU1 when the later fails. Therefore, the solution to this problem of load balancing after single failure must be tackled from the perspective of SUs backing up each other for SIs and make sure that SIs assigned to a particular SU are equally backed up by the remaining SUs.

Each SU can have at most one active assignment with respect to a particular SI. So, if an SU has *X* active assignments, then it is assigned *X* different SIs. We want to ensure that

these *X* SIs are backed up evenly by the other SUs. In order to achieve this, we start by assigning each SU an equal number of SIs to backup, then we determine the SUs that will be active for those SIs, and finally we generate our ranked list of SUs for each SI.

The solution we are presenting in this section is for the *NwayActive* redundancy model, but it applies for the *NWay* redundancy model as we will discuss it briefly at the end of this section. This solution is based on the following assumptions:

- The SUs have the capacity to support the SIs that AMF assigns to them even after an SU fails and its load is shifted to the other SUs.
- The SIs of the same SG have the same protection level, and therefore the *numberOfActiveAssignments* is the same for all of them.
- The SIs impose the same load on the SUs.
- The *numberOfActiveAssignments* is less than the number of SUs.

Systems managed by AMF are intended to have no single point of failure and are expected to tolerate the failure of one SU without causing a service to be dropped. Therefore this work targets a single failure, and load balancing after multiple failures is outside the scope of this thesis.

We first introduce our approach and then apply it to an example.

5.5.2.1 Approach for the NwayActive Redundancy Model

Our approach consists of four steps:

1. Determine the total number of assignments to backup (or simply backup assignment[4]) each SU will have,

2. Distribute the total backup assignments of each SU equally among the other SUs,

3. Balance the total number of active assignments for all the SUs, and

4. Derive the ranked list of SUs for each SI from the assignment table.

In the *NwayActive*redundancy model one or many SUs will be assigned the active state on behalf of an SI. However, only one SU will serve as a backup for this SI if any of its active SUs fails. In this case it is said that the backup SU is backing up the active SUs in terms of this SI. If this particular SI requires x active SUs where x is the *numberOfActiveAssignments*, we say that the backup SU is backing up x active assignments, since one SI can only be backed up by one SU for all its active assignments. It is important here to distinguish that although the backup assignment is in terms of SIs, we bring the *numberOfActiveAssignments* each SI has into the equation because if one SU is backing up an SI this means it is backing up all of its active assignments, while on the other hand being active for an SI means having a maximum of one of its active assignments.

---

[4] The backup assignment is not to be confused with the standby assignment.

Equation 5-1 Backup assignment for each SU

$$Backup = \begin{cases} \left\lfloor \dfrac{numberOfSIs}{numberOfSUs} \right\rfloor \times numberOfActiveAssignments \\ or \\ \left\lceil \dfrac{numberOfSIs}{numberOfSUs} \right\rceil \times numberOfActiveAssignments \end{cases}$$

A prerequisite for ensuring back up balancing is that each SU must back up an equal number of SIs. Therefore, the backup value of an SU is given by Equation 5-1. Since the number of SUs is not always a divisor of the *numberOfSIs,* some SUs will get the floor of this division while others will get the ceiling with the constraint that the sum of the backup assignments for all SUs is equal to *numberOfSIs* * *numberOfActiveAssignments*. Of course, an SU does not back up its own active assignments, but the active assignments of the other SUs as we will see it in Table 5-3.

In order to ensure backup balancing, it is not enough that the SUs backup the same number of active assignments, because if all those active assignments are provided by one SU, and this SU fails, its entire load will go to the backup SU that is already active for its own SIs. We will end up in the same pitfall of the round robin algorithm. Therefore, the number of active assignments backed up by an SU must be the sum of equal contributions ± 1 from all the other SUs. In other words, if the SU is calculated to have *x* back up assignments, and we have *n* SUs, then we make sure that the SU will back up each of the other SUs in *x* ÷ *(n-1)* active assignments. This division may result in a decimal value; some SUs may get an extra back up from the SU in question.

In order to render the solution more concrete, we visualize our assignments in terms of what we defined to be the assignment table shown in Table 5-3, which, when populated, would show simultaneously the active and backup assignments.

Table 5-3 The assignment table blueprint

	SU_1	…	SU_j	…	SU_n	Backing Up
SU_1	N/A					
…		N/A				
SU_i			N/A			
…				N/A		
SU_n					N/A	
Act						–

Table 5-3 represents the blueprint of an assignment table. A value x in cell SU_{ij} represents a number of assignments. However this value can either mean the number of active assignments for the SU at the top of the column, or the number of backup assignments for the SU of the raw. The SU rows represent the number of backup assignments each SU will have for the other SUs. The cells denoted with N/A (Not Allowed) means an SU cannot back up itself. A value x in cell SU_{ij} means that the SU_i will back up SU_j in x of SU_j's active assignments. The "*Baking Up*" column is used to hold the value of the total backup assignments that we calculate with Equation 5-1 for each SU. We assign the SUs cell values of any row i in such a way that (1) their sum is equal to the "*Backing up*" cell value in row i (2) the cell values can only be the floor or the ceil of the baking up value

after it is divided by the number of SUs -1. In other words if SU_i is backing up *x* active assignments, we want those assignments to be equally distributed among other SUs.

The "*Act*" row is used to hold the values of the total active assignments each SU is expected to have. It is the sum of the column cells values. Note that the SUs of the columns and the rows of the table are identical. We simply used different indexing (*i* for row and *j* for column) to avoid ambiguity.

So far we have balanced the backup assignments. However, the total active assignments that each SU is handling from previous calculations may be uneven, i.e. that values of the *'Act'* row may be imbalanced. In order to solve this issue we need to make sure that the SUs have equal active assignments.

If we have a number of SIs each having a *numberOfActiveAssignments* to be assigned to the same number of SUs, the load will be evenly balanced among the SUs if each SU is assigned one of the values defined in Equation 5-2. Again since the number of SUs is not always a divisor of the value of the *numberOfSIs* * *numberOfActiveAssignments*, some SUs will get the floor of this division while others will get the ceiling with the constraints that the sum of the active assignments for all SUs is equal to *numberOfSIs* * *numberOfActiveAssignments*. Some SUs may have an extra load of one active assignment compared to other SUs. The sum of the cells in the "*Act"* row and the sum of the cells in the "*Backing up*" column are both equal to *numberOfSIs* * *numberOfActiveAssignments*. This is due to the fact that this number represents the total active assignments an SG will protect, regardless how we assign the active and backup assignments; this number is always going to be the same.

Equation 5-2 Active assignment for each SU

$$Active = \left\{ \begin{array}{l} \left\lfloor \dfrac{numberOfSIs \times numberOfActiveAssignments}{numberOfSUs} \right\rfloor \\ or \\ \left\lceil \dfrac{numberOfSIs \times numberOfActiveAssignments}{numberOfSUs} \right\rceil \end{array} \right\}$$

Our third step consists of making sure the total active assignments for each SU is one of the values in Equation 5-2. This must be completed without affecting any value in the "*Baking Up*" column. In other words we need to shuffle the values in the row cells in such a way that (1) their sum remains intact and equal to the value of the "*Baking Up*" cell in the row, and (2) achieve a balance so that the cells of the "*Act*" row have values as given by Equation 5-2. This reasoning converts our problem into a constraint satisfaction problem, where we have a set of values within a table, and they are constrained by the fact that the sum of the row cell values and the sum of the column cell values must obey to certain magnitudes.

Algorithm 5-1 solves the constraints satisfaction problem of balancing the active load.

1. balance()
    a. **IF** (each column has a sum equal to *Active*) **THEN**
        i. **RETURN** true
    b. **ELSE**
        i. **RETURN** false

2. **END** balance()

3. solveCSP()

    a. **While** (not balanced())

        i. *minColumn*← the column with minimum sum of cells

        ii. *maxColumn*← column with the maximum sum of cells

        iii. **While** (sum(*maxColumn*) – sum(*minColumn*) > 1)

            1. swap the minimum value in *minColumn* with the maximum value in *maxColunm*

        iv. **EndWhile**

    b. **ENDWhile**

4. 23.**END** solveCSP

Algorithm 5-1 The CSP solution for NwayActive.

This algorithm consists of two main functions, the *balance()* function that test whether the SUs have equal active assignments, and the *solveCSP()* function that keeps swapping row values until the balance is obtained.

Notice that we could have started working with the columns of Table 5-3, and therefore balance the active load first and then work with the backup assignment, or work simultaneously with both as it is the case for such constraints satisfaction problems.

Our final step is to automatically generate the ranked list of SUs for each SI. The rank values of this list are based on the assignment table. The list is generated in the following manner. For each row in the table that corresponds to SU_i, we will calculate the number of different SIs that this SU is backing up by dividing the value in the "*Backing Up*" cell of the row over the *numberOfActiveAssignments*. The assignment table will tell us how many active assignments each of the other SUs will have on behalf of the SIs backed up by SU_i, but we still need to determine to which particular SU each SI will be assigned. Here, again, we face another CSP problem, with the following constraints:

- The number of SUs assigned active to each SI must be equal to the *numberOfActiveAssignments* and
- The number of SIs each SU (excluding SU_i) will be active for is specified by the assignment table and must be respected.

By solving this CSP problem we determine the active SUs and assign them the lower ranks, the SU that was assigned the N/A value (i.e. SU_i) will serve as the backup, and therefore will have the first rank higher than the SUs with the active assignments, the other SUs will have ranks greater than the one assigned to the backup SU. The above process will result in a sub-list of ranked SUs generated for the SIs backed up by SU_i. The same process is repeated for all the rows, and the sub-lists of ranked SUs are joined together repeatedly until we get the ranked list of SUs for all SIs. The process is further illustrated in the next example.

5.5.2.2 Application of the NwayActive Ranking Approach

For illustration purpose, we reuse the example presented in Section 5.5.1. Step 1 would be to calculate backup assignment for each SU. According to Equation 5-1, the backup in terms of SIs is equal to the floor or ceiling of $(14 \div 6) = 2.33$. Some SUs will back up two SIs while others will back up three. We multiply those numbers with the *numberOfActiveAssignments* to get the baking up value each SU will handle in terms of active assignment. Knowing the backup assignments, we can proceed into the second step and populate our table with values as shown in Table 5-4. The values in the cells are assigned by taking the value in the backing up cell of each row and dividing it evenly among the other cells of the same row.

Note that at this step of the table is still not balanced, we simply balanced the backup value among the SUs.

Table 5-4 The assignment table before balancing.

	SU1	SU2	SU3	SU4	SU5	SU6	Backing Up
SU1	N/A	1	2	2	2	2	9
SU2	1	N/A	2	2	2	2	9
SU3	1	1	N/A	1	1	2	6
SU4	1	1	1	N/A	1	2	6
SU5	1	1	1	1	N/A	2	6
SU6	1	1	1	1	2	N/A	6
Act	5	5	7	7	8	10	$\sum = 42$

The third step consists of balancing the active load among SUs, as aforementioned the problem here is a constraint satisfaction problem that involves shuffling the values in the

table cells in such a way that the sum of the columns would be floor or ceil of the value calculated according to Equation 5-2 [$Active = (14 \times 3) \div 6 = 7$].

Solving the CSP by swapping the values in the row cell will result in the balanced Table 5-5 shown below.

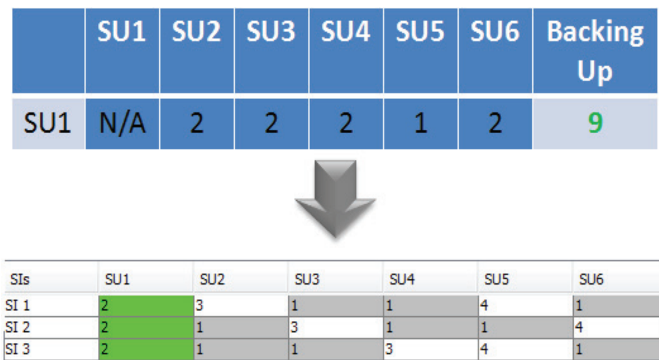
Table 5-5 The assignment table after balancing.

	SU1	SU2	SU3	SU4	SU5	SU6	Backing Up
SU1	N/A	2	2	2	1	2	9
SU2	2	N/A	2	2	2	1	9
SU3	2	1	N/A	1	1	1	6
SU4	1	2	1	N/A	1	1	6
SU5	1	1	1	1	N/A	2	6
SU6	1	1	1	1	2	N/A	6
Act	7	7	7	7	7	7	$\sum = 42$

The final step is to derive from Table 5-5 a ranked list of SUs for the SIs. Each row of our assignment holds the information needed to generate the ranked list of SUs for a subset of SIs. We take a row at a time, extract the information encapsulated in this row, and based on this information, we generate our ranked list of SUs. Figure 5-3 shows the ranked list of SUs generated for the subset of SIs backed up by SU1. The process is carried out as follows:

Based on the first row of Table 5-5 we deduce that SU1 will back up three SIs ($9 \div 3$; where 3 is the *numberOfActiveAssignments*), so for SI1, SI2 and SI3 the backup is SU1, therefore SU1 is not allowed to be active for those SIs and as a result will be assigned a rank higher than the one we will assign to the active SUs of behalf of those three SIs. In

order to determine which SU is active on behalf of which SI, we need to solve the CSP problem defined by assigning 9 active assignments to 5 SUs on behalf of 3 SIs in such a way that each SU will have the exact value of active assignments specified in the assignments table (e.g. the table specifies that SU5 has only one active assignment and therefore will be active for at most one of the SIs backed up by SU1). Figure 5-3 presents one possible solution to this problem. The SU with the active assignment is given the lowest rank value. The rest of the SUs will be given a rank higher than the one assigned to the backup SU (i.e. SU1). This process is repeated successively for all the rows, until we have a complete list of ranked SUs for all SIs.



	SU1	SU2	SU3	SU4	SU5	SU6	Backing Up
SU1	N/A	2	2	2	1	2	9

SIs	SU1	SU2	SU3	SU4	SU5	SU6
SI 1	2	3	1	1	4	1
SI 2	2	1	3	1	1	4
SI 3	2	1	1	3	4	1

Figure 5-3 A mapping from the assignment table row to a ranked sub-list.

Figure 5-4 is a snapshot of our generated ranked list of SUs, the table cells are rendered such that the cells holding the backup assignment value are marked in green, while the cell holding the active assignment value are marked in gray.

SIs	SU1	SU2	SU3	SU4	SU5	SU6
SI 1	2	3	1	1	4	1
SI 2	2	1	3	1	1	4
SI 3	2	1	1	3	4	1
SI 4	3	2	1	1	1	4
SI 5	1	2	3	1	1	4
SI 6	1	2	1	3	4	1
SI 7	1	3	2	1	1	4
SI 8	1	1	2	3	4	1
SI 9	3	1	1	2	1	4
SI 10	1	1	3	2	4	1
SI 11	3	1	1	4	2	1
SI 12	1	3	4	1	2	1
SI 13	3	1	1	4	1	2
SI 14	1	3	4	1	1	2

Figure 5-4 A snapshot of our ranked list of SUs.

5.6 Workload balancing in N+M

N+M differs from NwayActive by having dedicated standby SUs and having the number of active assignments fixed to one. Workload balancing in N+M can be summarized as follows: we have a set of SIs that needs to be assigned to two sets of SUs, an active set and a standby one. The number of SIs assigned to an active SU must be balanced with the one assigned to its active siblings. The same applies to the standby set. From this perspective, again a simple round robin distribution of the SIs among the active SUs and also among the standby SUs seems to achieve the desired result. Nevertheless after a failure, this approach exhibits major drawbacks.

5.6.1 The Problem with Conventional Load Balancing Algorithms at Configuration Time in N+M

Again, we use Round robin as a reference configuration time ranking solution in order to explore the pitfalls of such a solution and how to surmount them. The issue with Round robin is that it does not associate an active SU with a standby one, in other words it does not take into consideration the implicit relation that exists between an active SU for a particular SI and the counterpart standby one. This relation can be defined as follows: *it*

suffices that the standby SU be a standby for any of the SIs assigned to the active SU, so that in case this active SU fails, the standby one will turn into active. The entailment of this relation is the following, if SU_x is assigned the active state on behalf of *x* SIs, and for each of these SIs a different standby SU is assigned the standby state, when SU_x fails, it will be replaced by *x* standby SUs. The consequences of this shortcoming after a failure can be summed up as: (1) we are making unnecessary sacrifices among the standby SUs, where in the worst case scenario, the failure of one active SU may cause the loss of all the standbys by turning them into active ones. (2) If one active is replaced by multiple standbys, then the workload that was once handled by this active SU (that is balanced with the workload handled by its sibling active SUs) is now split among several SUs, causing a misbalance of the active assignments. (3) The number of active SUs after a failure may exceed the configured value set by the system configuration designer.

The second shortcoming of such a simple approach is that it does not take into account what happens with the standby assignments after failure. That is, when a failed active SU is replaced by a standby, the *standby assignments* originally assigned to this standby SU *need to be redistributed among other standby SUs*. Round robin does not cover this issue.

It should be noted that any load balancing solution for the N+M redundancy, must address the above issues, whether it is a runtime or a configuration time solution. We can summarize these issues as requirements that workload balancing approaches should satisfy:

(1) Ensure active workload balancing before a failure.

(2) Maintain the active workload balanced after a failure.

(3) Ensure standby workload balancing before a failure.

(4) Maintain the standby workload balanced after a failure.

(5) Upon failure replace one active SU with one standby.

5.6.2 Can We Find a Solution that Satisfies all the Requirements?

Consider the following example: 5 SUs (3 active and 2 standbys) serving 30 SIs. We balance the active workload of the active SUs by assigning (30/3) 10 SIs in their active state to each of the active SUs, SU1, SU2 and SU3. Similarly, we assign to each standby SU, SU4 and SU5, (30/2) 15 SIs in their standby state.

So far we have achieved the balancing of the workload before failure. In order to make sure that when an active SU fails it is replaced by exactly one standby, all the SIs assigned to the active SU must be re-assigned to at most one standby SU. In other words each standby SU must be standby to an integer number of active SUs. For example, SU4 must be standby for 1, 2, or more active SUs. However, if SU4 is standby for only one SU, then it will be assigned only 10 standby assignments (since each active SU has 10 active assignments). If it is a standby for two SUs, it will be assigned 20 standby assignments (exceeding the desired value of 15). In both cases we violate the balanced number of standby assignment calculated for SU4. Otherwise stated, no combination of 1, 2, or more SUs will have the sum of 15 active assignments for which SU4 can be the standby for. So basically in this given example, when the active and standby workloads are perfectly balanced, we cannot guaranty that when any active SU fails, it will be

replaced by only one standby. In conclusion, having a single approach that satisfies the five requirements defined earlier may not always be possible.

5.6.3 Ranking Solutions Targeting Workload Balancing Before and After One Failure in N+M Redundancy

In this section we present three different solutions for solving workload balancing in N+M. Each solution targets only three requirements of the five based on different priorities. Again we are making the assumption that the SUs have enough active/standby capacity to handle the SIs assigned to them, and that the SIs impose equal loads on the SUs.

5.6.3.1 The One Active for One Standby Solution

This approach for workload balancing in N+M targets the issue of not replacing one active SU with more than one standby, while keeping the active workload balanced before and after a failure. As for the standby workload, while it may not be possible to maintain it in complete balance, we aim at having it "substantially balanced[5]".

[5] Substantially balanced means minimizing the difference between the minimum and maximum number of SIs assigned to the different SUs according to distribution policy of the given approach. See more at each approach.

This approach meets requirements 1, 2 and 5 defined in Section 5.6.1. The consequence of covering these requirements is having an unbalanced standby load even before failure; therefore requirements 3 and 4 are not met, and the approach tries to compensate that as much as possible by keeping the standby load at a minimum misbalance. In case of a failure, the workload originally assigned to the standby SU is then redistributed among its standby siblings in a way that favors standby load balancing.

As a result, in addition to satisfying requirements 1, 2 and 5, this approach has the following properties:

(1) Makes sure that each standby SU is standby for an equal number of active SUs (±1)

    a. With the constraint of keeping the standby workload substantially[6] balanced.

(2) Makes sure that the standby workload of an SU is re-distributed among the other standby SUs again with the constraint of keeping the standby substantially[7] balanced.

---

[6]In this approach some standby SUs may be standby for the workload of one extra active SU compared to their standby siblings. If that is the case, we make sure that the maximum difference in the standby assignments among the standby SUs never exceeds the floor value of Equation (1).

Equation 5-3 and Equation 5-4 below represent our definition of a balanced active and standby workload per SU. We use the floor and ceil notations since the number of SIs may not always be dividable by the number of SUs, and therefore some SUs may get assigned an extra SI. The variable names are self-explanatory, where *stdb* stands for standby.

Equation 5-3 Active load in N+M

$$Active = \left\{ \begin{array}{l} \left\lfloor \dfrac{numberOfSIs}{numberOfActiveSUs} \right\rfloor \\ or \\ \left\lceil \dfrac{numberOfSIs}{numberOfActiveSUs} \right\rceil \end{array} \right\}$$

Equation 5-4 Standby load in N+M

$$Stdb = \left\{ \begin{array}{l} \left\lfloor \dfrac{numberOfSIs}{numberOfStdbSUs} \right\rfloor \\ or \\ \left\lceil \dfrac{numberOfSIs}{numberOfStdbSUs} \right\rceil \end{array} \right\}$$

---

[7]During the redistribution of the standby load, the standby SUs with the least load will get a bigger portion of the standby assignments, so that the overall standby load becomes more balanced.

5.6.3.1.1 The One Active for One Standby Approach

Deriving a configuration-time ranking for load balancing is not a straight forward task. Again here we use the assignment table artifact which visualizes our solution by simultaneously displaying multiple assignments. Since in N+M we need to balance the active and the standby assignment, we defined two assignment tables. We start by introducing what we refer to as the 'standby assignment table'. We define this artifact to visualize our solution in a form of a table that simultaneously displays the active and standby assignments. Table 5-6 represents the blueprint of this table.

Table 5-6 The standby assignment table

	SU*1*	…	SU*j*	…	SU*m*	*#Act*
SU*1*						
…						
SU*i*						
…						
SU*n*						
#Stbd						

In the 'standby assignment table' the rows and columns are indexed by two different sets of SUs. The columns are indexed by the standby SUs and the rows are indexed by the active SUs. The rightmost column holds the value of the balanced active assignments an SU must handle. This value is one of the outcomes (a floor or a ceil) of

Equation 5-3. The bottom row will hold the value of the standby assignments that a standby SU will handle. This value is not necessarily the outcome of Equation 5-4. As we stated in this approach we do not target the issue of balancing the standby assignments, rather we sacrifice it in order to make sure that we do not replace an active SU with multiple standbys. The values in the cells of the standby assignment table are interpreted

as follows: A value x in the cell (SU_i , SU_j) means that SU_j (which belongs to the set of standby SUs) is standby for x SIs assigned active to SU_i (which belongs to the set of active SUs). The steps of this approach are the following:

Step 1: we calculate the active workload of SIs based on

Equation **5-3**.

Step 2: we populate the standby assignment table with the values that we calculated in step one. It does not really matter how we populate the table, since Step 3 will balance the table by reshuffling the cell values.

Step3: in this step we balance the table so that each standby SU is a standby for an equal number of active SUs (±1) compared to its standby siblings. The CSP algorithm used to balance the table takes as input a standby assignment table populated with the active values, and then balances the standby assignments within the table by granting each standby SU the proper number of active SUs that have the proper number of active assignments, with the constraint that the maximum difference in the standby assignment never exceeds the floor value of

Equation **5-3**.

Step 4: this step consists of finding the best possible redistribution of the standby workload among the standby SUs in case we lose one standby SU. We introduce here the notion of the 'backup-standby'. If SU$_x$ has the backup-standby assignment[8] for a certain SI, this means that when the standby SU for this SI fails, SU$_x$ will become a standby for this SI. We define another assignment table, namely the 'backup-standby assignment table' that will give a simultaneous view of the standby and the backup-standby assignments. Table 5-7 represents the blueprint for this table.

Table 5-7 The backup-standby assignment table

	SU$_1$	…	SU$_j$	…	SU$_m$	Backup standby
SU$_1$	N/A					
…		N/A				
SU$_i$			N/A			
…				N/A		
SU$_m$					N/A	
Standby						—

In the 'backup-standby assignment table', the columns and the rows are indexed by the same set of SUs ─ the standby SUs. The rightmost column holds the value of the backup-standby assignments we will assign to SU$_i$. The bottom row will hold the value of standby assignment we will assign to SU$_j$. The values of this bottom row are the same ones that appear in the bottom row of the 'standby assignment table' that was balanced in Step 3.

---

[8] The backup-standby assignment is a virtual assignment that does not imply any actual load on the SU.

We can also notice that the cells of this table have gray margins. These margins will hold certain values that will be used in our calculations. The values in the cells of the 'backup-standby assignment table' are interpreted as follows: a value x in the cell (SU_i, SU_j where $i \neq j$, since the both SUs belong to the same set) means that SU_i is backing up x standby assignments of SUj. The margins are used to reflect the extra standby workload that a certain SU possess compared to the SU with the minimum standby workload. The objective here is that after a standby SU fails, we want to be fair in the way we re-distribute its workload, and consider the original misbalance in the standby workload.

Step 5: The final step is to derive the ranked list of SUs based on the assignment tables balanced in the previous steps. After completing the first 4 steps, deriving the ranked list becomes a mechanical process defined as follows:

(1) Based on the balanced standby assignment table, give each standby SU_j a rank of '2' for each SI it is standby for. For each SU that is supposed to be active for these SIs give a rank of '1'.

(2) Based on the balanced backup-standby assignment table, give each SU that is supposed to back up the standby assignments of SU_j a rank of '3' for the number of backups it is supposed to handle.

5.6.3.1.2 An Example for the One Active for One Standby Approach

As an example of this approach we present a simple example consisting of 23 SIs to be assigned to 7 active SUs and 4 standbys. Step 1 consists of finding the balanced active workload according to

Equation 5-3. In Step 2 we populate the 'standby assignment table' with the values calculated in Step 1 (shown in Table (a) of Figure 5-5. In Step 3 we run the CSP algorithm that balances this table, and we end up with Table (b) of Figure 5-5. At this point we have the standby assignments balanced. We move on to balance the backup assignments in Step 4, where we first populate the margins of the 'backup-standby assignment table'. For example the cells of row SU8 have a value of +3 in their margins because SU8 is standby for two active SUs (Table (c) Figure 5-5) and therefore has 3 extra standby assignments over the SU with the minimum number of standby assignments (SU10). Then we proceed by assigning the backup-standby assignments, where the SU with the least standby assignments (smallest margin value) will get the most backup-standby assignments. The rationale behind this is to exploit the failure of an SU by making the standby distribution more balanced. The values (2,4,1) in column SU8 (Table (d) of Figure 5-5), indicate that when SU8 turns active or fails its 7 standby assignments will be redistributed as 2, 4 and 1 to SU9, SU10 and SU11 respectively based on their margin values. SU9 will have 2 new standby assignments to its original 6, SU10 will have 4 new added to its existing 4, and SU11 will have 1 added to its 6, which renders the standby workload substantially balanced.

The final step is to derive the ranked list of SUs. The runtime assignments will be performed according to this list. This list is defined based on the balanced tables. Figure 5-6 illustrates this process. More specifically it specifies how the ranked list is derived for the SIs for which SU8 is standby. Based on column SU8 of Table (b) of Figure 5-6, we know that SU8 is standby for 7 SIs, and that is why it is given a rank of '2' for these SIs, the same column also indicates that it is SU1 and SU3 that are active SUs for respectively

3 and 4 of these SIs and this is why they are given a rank of ‘1’. The column SU8 of Table (d) of Figure 5-6 specifies the backup-standby SUs for SU8, and accordingly the rest of the standby SUs are given a rank of ‘3’. It should be noted that the empty cell of the ranked list can be filled with any value > 3. The same process is repeated iteratively for each column of our assignment tables until the full list is generated for all the SIs.

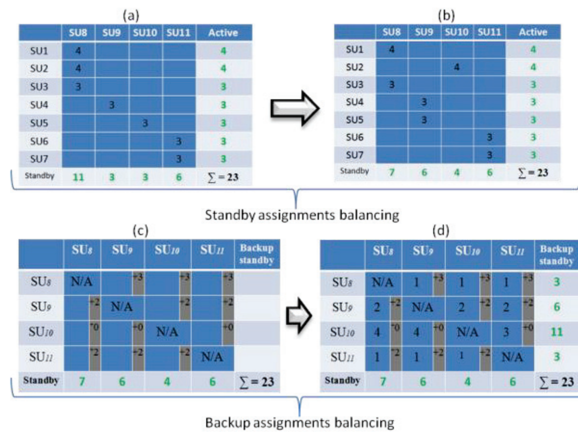


Figure 5-5 The balanced assignment tables

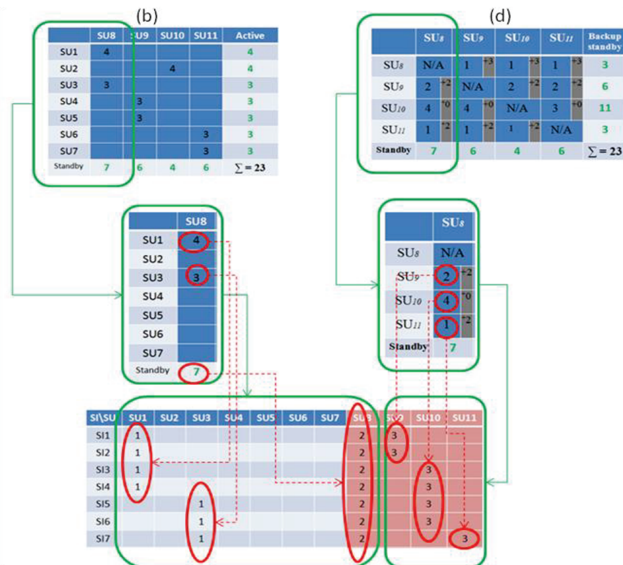


Figure 5-6 The process of deriving the ranked list of SUs

5.6.3.2 *The One Active for One Standby for One Backup Solution*

This solution targets the same requirements as the previous one (1, 2, and 5). In addition it maintains the active load balanced even in the presence of a second failure. The approach for this solution consists of modifying the 'one active for one standby' approach to have only one standby for each active SU after the first failure. For this we simply modify the fourth step and instead of distributing the backup-standby workload in a way that favors standby workload balancing, we simply give the backup-standby assignment to the standby SU with the least standby workload. As a result the active load is balanced not only before and after the first failure, but it also remains balanced even after a second failure.

Table 5-8 illustrates the modifications made in the values of the backup-standby assignments in the presented example. The process of deriving the ranked list remains the same.

Table 5-8 The backup-standby assignments for the 'one active for one standby for one backup' approach

	SU_8	SU_9	SU_{10}	SU_{11}	Backup standby
SU_8	N/A	+3	+3	+3	3
SU_9	+2	N/A	4 +2	+2	6
SU_{10}	7 +0	6 +0	N/A	6 +0	11
SU_{11}	+2	+2	+2	N/A	3
Standby	7	6	4	6	$\sum = 23$

5.6.3.3 *The Complete Balance before Failure Solution*

The objective of this approach is to maintain the workload balanced while the system is in its healthy state and hence it meets requirements 1, 3, and 4. However in order to meet these requirements, we may need to sacrifice the requirement 5 of replacing one failing

active SU with only one standby. After a failure this may result in two drawbacks: first, we may lose more than one standby SU; and second, the active load assigned to these standbys is going to be less than the one assigned to the other active SUs. Thus, the active load is not going to remain balanced after failure.

This approach compensates for these drawbacks by limiting the loss of standbys: It makes sure that an active SU is never replaced with more than two standbys. So in addition to satisfying requirements 1, 3 and 4, this approach has the following property:

- It makes sure that no active SU is replaced by more than two standby SUs after a failure.

5.6.3.3.1 The Complete Balance before Failure Approach

The first three steps of this approach are identical to the ones used in the 'one active for one standby' approach. However after the third step, we proceed towards balancing the standby load per SI rather than balancing per SU workload[9].

Step 4: consists of calculating the balanced standby load (±1) and assign it to each SU. Here we add an extra row to the 'standby assignment table' and call it the 'desired

[9] In the previous approach we balanced per SU. I.e. each standby SU was a standby for an equal number of active SUs (±1), whereas here we balance per SI.

standby' row. The values of this row are calculated based on Equation 5-4 and they represent the desired balanced standby assignments we want to achieve. Note that in this approach the SIs assigned active to one SU are not necessarily assigned standby to exactly one standby SU, in fact this workload can be divided among two standby SUs if this is needed to balance the standby workload among the standby SUs. Next, we will use another CSP algorithm to reshuffle/split the cell values so that the sum of standby assignments of each column matches the desired standby assignment value calculated for this column. In order to do so, we look for the columns with a sum of standby assignments that exceeds the desired standby value for this column, and split one of its values with another column where the sum of the standby assignments is less than the desired standby value of this latter column. The splitting is repeated until the balancing objective is achieved (with the constraint that no cell value is split more than once).

Step 5: The final step is to make sure that after a standby SU turns active or fails, its standby load is evenly re-distributed among the healthy standby SUs. Since the standby load is evenly distributed among SUs, we divide the load of each of the M standby SUs into equal shares, where the number of shares is equal to M - 1. Then we assign each share to be backed up by one of the standby SUs. However we have stated that in this approach 2 standby SUs can turn active simultaneously, this means that the number of standby SUs can become M – 2. The main challenge here is how to come up with a ranking to accommodate both scenarios: the single and the double loss of standby SUs. We solve this problem by introducing another level of backup standby balancing, a level that considers the simultaneous loss of two standby SUs. We also introduce the 'level-two backup standby assignment table'. This table is only used for the standby SUs that

have the implicit relation of both having standby assignments of SIs that are assigned active to the same SU. This relationship among the standby SUs implies that if the common active SU fails, they will both turn active simultaneously.

Step 6: The final step is to derive the ranked list of SUs based on the above analysis. This is carried on as follows:

Based on the balanced standby assignment table, give each standby SU_j a rank of '2' for the number of SIs it is standby for. For each SU that is supposed to be active for these SIs give a rank of '1'.

- Based on the balanced backup-standby assignment table, give each SU that is supposed to back up the standby assignments of SU_j a rank of '3' for the number of backups it is supposed to handle.
  - Based on the 'level-two backup standby assignment table', give each '2nd level standby' SU a rank of 4.

5.6.3.3.2 An Example for the Complete Balance before Failure Approach

We use the example presented previously in Figure 5-7 (c) which illustrates the balanced standby assignment table. We are reusing the first three steps of the 'one active for one standby' approach, and the same CSP algorithm is used to obtain the balanced table shown in Figure 5-7 (b). However a different CSP algorithm is used to balance the standby SI assignments (this is because the constraints for balancing the standby workload have changed in this approach). The objective is to obtain the desired standby values shown in the bottom row by splitting particular cell values among (at most) two cells of the same row. The implication of such splitting is that the active SU will be

replaced by the standby SUs that hold the split values. E.g. in table (c) of Figure 5-7 below, when SU3 fails it will be replaced by both SU8 and SU10. We intentionally split among at most two cells as an optimization decision. By doing this we never replace one active SU with more than two standbys. Figure 5-8 illustrates the balanced backup-standby assignment tables used in this approach. We can see in Figure 5-8 (b) that when SU8 turns active each of the other standby SUs will take over 2 of its standby assignments including SU10. However, since SU10 may turn active simultaneously with SU8 (this is the case when SU3 fails); the 'level-two backup standby assignment' table (Figure 5-8 (c)) is used to capture the redistribution of the load given to SU10 by SU8. The first column of this table indicates that SU9 and SU10 will share equally this workload redistribution. The process of deriving the ranked list of SUs is illustrated in Figure 5-9.

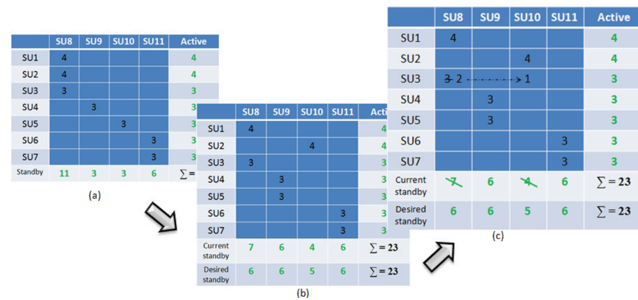


Figure 5-7 The balanced standby assignment table in the 'complete balance before failure' approach

Figure 5-8 The backup-standby assignments for the 'the complete balance before failure' approach

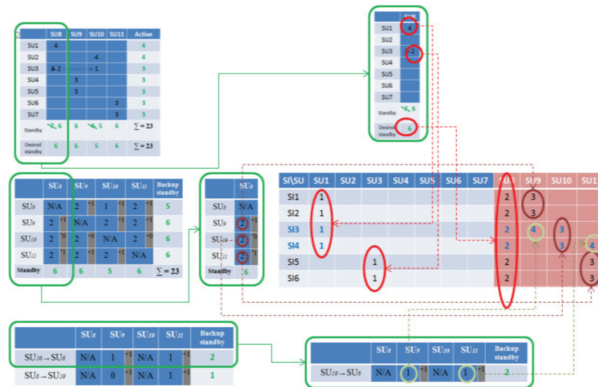


Figure 5-9 The process of deriving the ranked list of SUs in the 'complete balance before failure' approach

5.7 Conclusions and discussion

In this chapter we have introduced ranking solutions to handle workload balancing for the NwayActive and the N+M redundancy models. Most importantly the solutions we presented also satisfy some placement preferences of the workload while maintaining an overall load balance before and after failure. These solutions are provided at configuration time and we used the SU ranking to express the placement preference. The SU ranks express our load distribution preferences that are calculated based on the equations we have defined. AMF will enforce our placement preferences based on the rankings. We define our load balancing solution based on how we wish the load to be distributed before and after a failure. Achieving load balancing before failure is rather straight forward, however in order to make sure the load remains balanced after a failure we had to define the backup notion. We can achieve load balancing after failure if the originally balance load is equally redistributed after failure, I.e. the SUs are backing each other equally. For this we defined the assignment table where two constraints had to be met, (1) the sum of the values in each column had to evaluate to a predefined value (2) the sum of the values in each row had to evaluate to a 'substantially' balanced values. By

solving these two constraints (what we refer to as balancing the table) we make sure that load distribution preference (whether it is active or standby) before and after failure is respected. This preference is thereafter translated into rankings. Based on the balanced tables, defining the ranking is a mechanical process that can be achieved based the mappings we defined.

We were able to satisfy the load balancing in NwayActive, however for N+M, several issues arose and for that we have presented three approaches for N+M. Each approach targets different requirements in workload balancing. Accordingly they may suit some applications and configurations better than others. It is the responsibility of the configuration designer to select the most appropriate one to use. Here we provide some pointers that can guide the selection. When failures in the system are rare (i.e. the system is very reliable), and rapidly fixed, the 'complete balance before failure' is a suitable approach. The same applies for situations where the standby capacity is limited, i.e. a single standby SUs cannot handle the standby assignment of the workload assigned to two active SUs. However if the system is not configured to auto-adjust (i.e. when the faulty active SU is repaired the system does not readjust to its original SI assignments) then this approach is a poor choice because after the system recovers from a failure and the repair is complete, it will still run short of one standby SU, and have an unbalanced active workload with two active SUs having half of the workload that the other active SUs are supporting. The first two approaches are suitable when the standby workload that an SI imposes on an SU is relatively small, and when the number of standby SUs is limited. Nonetheless, when placement preferences is not a priority, the approaches we presented for N+M and implemented through the ranking mechanism, can also be

implemented as a runtime solution using different mechanisms, since they target generic issues that any N+M load balancing solution must handle. A runtime solution, however may bring further challenges such as the algorithm performance, and time/space complexity. At this moment these challenges are not of great significance to our solutions, since the calculation is performed offline, and does not consume resources or cause delays in the live system at runtime.

6 Configuration Based Service Availability Analysis

In this chapter we target the issue of evaluating the availability that, an AMF implementation, can offer to the services specified in the corresponding AMF configuration. We present our approach for defining the prerequisite steps needed to enable this analysis, and thereafter we present the approach for mapping AMF configurations to a stochastic model we defined to enable the quantification of the service availability.

The contributions presented in this chapter can be summarized as follows:

(1) Defining a method to determine how the recommended recoveries are altered into the actual recoveries based on the information specified in the AMF configuration.

(2) Categorizing the dependencies within an AMF configuration and capturing their effect on service availability

(3) Defining a stochastic model that captures the runtime service assignment and recovery behavior of AMF.

(4) Defining the mapping from an AMF configuration to the stochastic model that can be solved to quantify the availability of the services in the AMF configuration.

6.1 Introduction

In Chapter 4 we have established that several configurations can be generated that satisfy the same configuration requirements. It is the non-functional properties of these configurations, such as availability, that will favor one configuration over another. In highly available systems, quantifying the anticipated runtime availability of the services is a crucial task. In this chapter we present an approach for quantifying the service availability based on the configuration.

6.2 The Service Outage

The service availability analysis consists of determining the percentage of the time that the service is provided. From this perspective we calculate the availability by examining the failures that cause the service outage, and analyze the outage duration caused by these failures.

A component failure is defined by [64] as a deviation from providing its intended service. The failure is considered the manifestation of an error in the intended service, whereas the error itself is the manifestation of a fault in the system. In our analysis we consider that whenever an error is reported to or detected by AMF, a failure has occurred, and thereafter a service recovery is triggered.

The actual service outage that is caused by an error spans from the time the error occurred till the time the service was successfully recovered. However in practice errors are not usually instantly detected and reported, therefore there might be a delay from the time the error occurs till the time AMF is aware of the presence of this error. Calculating these “additional” delays is outside the scope of this dissertation. In this document we

analyze and calculate the service outage from the time AMF is aware of this error, till the time AMF recovers the services. Nonetheless if the additional delays are provided they can be incorporated in the analysis process. Figure 6-1 illustrates the discussed timings[10].

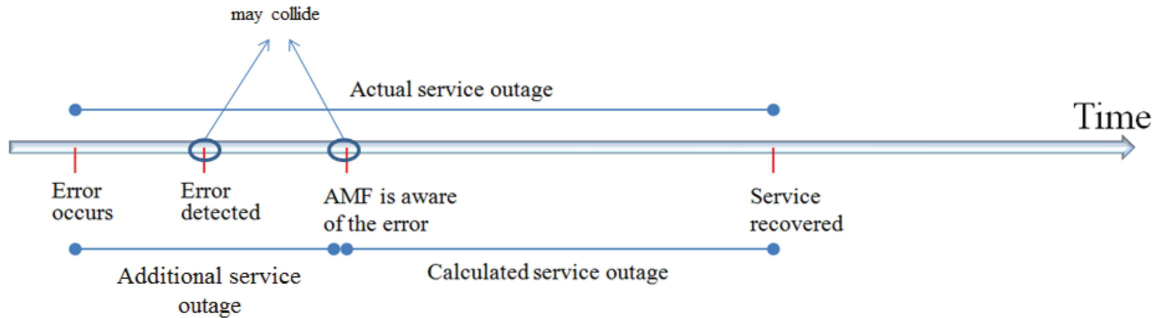


Figure 6-1 Service outage time

6.3 The Service Recovery

AMF is responsible for managing the availability of the SIs and their CSIs. This management is based on the AMF configuration. Error detection is the responsibility of all entities in the AMF managed system. Errors are typically reported to AMF through a component error report API. The error report also specifies the recommended recovery to be performed by AMF. The recommended recovery depends on the scope and type of the error. For example, if the error is at the component level, then the recovery could be a

---

[10] Note that if AMF detects the error, then there is no delay between the time the error is detected and the time AMF becomes aware of it.

component restart. But if the error is related to the execution environment of the component on the current node, then a different recovery may be more suitable like a component failover. If the report contains no recommendation, or if AMF detects the error without receiving of an error report, then AMF resorts to the configuration of the faulty components, and checks its default recommended recovery. After completing the recovery, or if the default recommended recovery does not specify a recovery action, AMF engages in a repair action. If the repair fails, AMF isolates the SU containing the faulty component. It is important here to distinguish between the recovery and the repair. The recovery is applicable at the service level, whereas the repair is applicable at the service provider level. The service may be recovered by shifting it to a healthy redundant component, while the original component is still not repaired. When calculating the service availability we are more interested in the recovery timing then the repair.

There are 9 recoveries defined in the AMF specifications. These recoveries target different scopes of failures and they can be summarized as follows:

- *Component restart* or *failover*: these recoveries are typically recommended when the failure manifests within the faulty component. In case of failover, the component's workload is failed over to another redundant replica that can continue the provision of the service that the workload signifies. Note that even in case of the component failover recovery, after recovering the services, AMF will still attempt to restart the component as a repair measure. When the recovery itself is a component restart, the component is repaired by a restart and the component-services it served before the failure are reassigned to it.

- *Container restart:* a container component provides an execution environment where the contained can be executed; e.g., a component deployed in a virtual machine, which itself is a container component. When the contained component is not achieving its functionality because the container is faulty, restarting the contained will not resolve the problem, a more suitable recovery is restarting container.
- *Node Switchover* or *failover* or *failfast*: when the failure reported on the component indicates that the whole node has been contaminated therefore all its workload needs to be failed over (or switched over) and the node cleaned up. A failfast is a faster version of the failover. Typically in a failover the components are abruptly terminated using the operating system while in failfast they may be terminated by powering-down the hardware for instance, which results in a faster and almost instant termination.
- *Application restart* or *cluster reboot*: these are applied when the failure indicates that a fault is affecting the services provided by the application or the entire cluster. Note that the restart of the application is not equivalent to restarting all its components. Instead all the components must be abruptly terminated first, before any of them is instantiated again. The rationale behind this is to prevent the old incarnation of the components from propagating their state and consequently the fault to their new incarnation by saving or exchanging this information. The same applies to the cluster reboot in terms of nodes.

6.3.1 Recovery Altering Attributes

The recommended recovery is either set as a default for a component or recommended through the API, which means it is embedded in some code. As a result it may not suit all configurations and therefore needs to be tuned to suit better a particular configuration. For instance in the example shown in Figure 2-1 the HTTP server and the Application server may have originated from two different vendors/providers, and grouped together in the same service unit by the system integrator. The software provider may be agnostic of how the software will be used or grouped by the system integrator. Therefore the software provider cannot recommend a recovery that is at the SU level for instance, simply because he/she is not aware of the scope of the SU and how it is formed. It is the system integrator's responsibility to determine the proper scope of recovery and hence adjust the configuration accordingly. For instance, if the HTTP server and the Application server collaborate closely, and a failure in either of them can easily propagate to the other and eventually to the receiver of the service, then the system integrator must force a recovery on both of them when either one fails. In other words whichever recovery that is recommended to either of them must be altered to include both of them. An AMF configuration model includes attributes that allow the mutation of certain recommended recoveries into different recoveries, namely, what we refer to as the "actual recoveries". Hence, with a particular setting of the AMF attributes, a configuration designer can craft more suitable recoveries and force AMF to execute them when needed. We refer to these attributes as "recovery altering attributes" and we have identified the following set of them:

- *Component disable restart* (defined for the component): when the component restart is expected to cause a longer service outage than the failover, then the system integrator can disable it, thus forcing the AM to failover the services even if the recommended recovery is to restart the faulty component.

- *Service-unit failover* (defined for the service-unit): when the components of the service-unit are tightly coupled with service dependency and therefore do not provide fault isolation, the system integrator can use this attribute to specify that all components of the unit must failover together, since the failure may have propagated to all of them.

- *Component restart tolerance* (defined at the service-group level): it defines whether the restart of any component in the service-group should escalate to the restart of the entire service-unit.

- *Service-unit restart tolerance* (defined at the service-group level): it defines whether the restart of any service-unit in the service-group should escalate to the failover of the service-unit.

- *Service-unit failover tolerance* (defined at the node level): it defines whether any service-unit failover at the node level escalates to failing over all the service-units hosted on the node.

- *Enable auto-repair* (defined for the node or service-group): this attribute specifies whether AMF is allowed to engage in the repair of the faulty entity in addition to recovering the service. When it is set to false, AMF is not allowed to perform any

repair and therefore a recovery action that implies a simultaneous repair, e.g. component/application restart and node/cluster reboot.

6.3.2 Issues and Challenges

There are several issues that arise during the availability analysis. The availability analysis is based on analyzing the service outage associated with failures; however the AMF configuration does not include the failure information of the components and nodes. Moreover, the actual recovery that AMF will execute at runtime is not necessarily the same as the recommended one, due to recovery altering attributes. However the availability analysis must be based on the actual recovery. The AMF configuration may specify various dependencies, and these dependencies may impact the service availability and therefore they must be captured by the analysis, hence another challenge is to define the analysis model in such a way that captures the impact of the software dependency on the service outage. The AMF configuration also describes escalation policies that AMF must enforce in case a recovery or a repair fails, and these policies must be reflected in the analysis model. The analysis model must also capture the SI assignment preference, such as the ones specified by the rankings we discussed in Chapter 5.

6.4 The Availability Analysis Framework

Our approach for quantifying the service availability consists of defining an analysis framework based on which we can derive a stochastic model that emulates the runtime system behavior. This model can thereafter be solved to get the availability measures of interest. Nonetheless deriving this stochastic model is not an easy task. It requires

prerequisite steps that we illustrate in this chapter. Our process of defining the analysis framework is as follows:

- Extend the AMF model
- Define the algorithms that determine the actual recovery
- Define the (stochastic) analysis model
- Define the mapping from an AMF configuration instance to an instance of the analysis model
    - Define the dependency mapping
    - Define the recovery mapping
    - Define the service assignment mapping

6.4.1 Extending the Standard AMF Model

The standard AMF model for a configuration does not include all the needed failure related information e.g. failure rates. This is because the model is defined for the purpose of runtime availability management and such information is not needed for this purpose. However, for the availability analysis, this information is crucial, and since we are following a model based approach, we decided to extend the AMF model to accommodate this information. The only two entities in the AMF configuration that are susceptible to failures are the node and the component. Therefore we have associated each one of them with zero or more failure types. Each failure type has a rate and a recommended recovery. This statistical failure information is assumed to be provided by

the software or hardware vendor, or obtained through benchmarks. Other attributes that are needed for the availability analysis but are not specified in the configuration are, for example, the time needed to start or shutdown a node or the probability of successfully instantiating a component. We also added the missing attributes to the relevant classes. The process of obtaining the values of these attributes is outside the scope of this thesis.

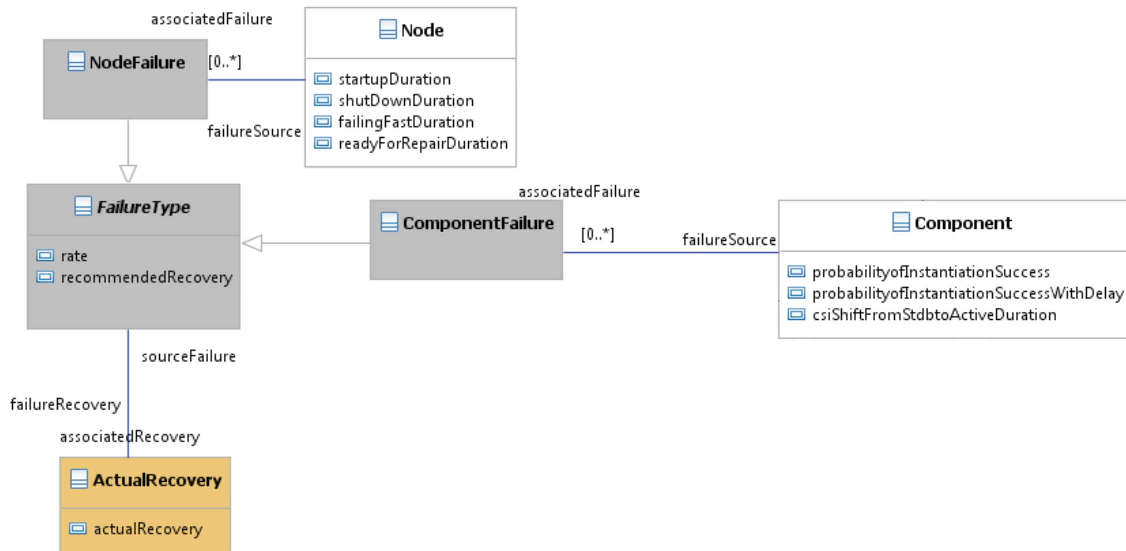


Figure 6-2 The extension to the standard AMF model

Figure 6-2 illustrate the class diagram showing the extensions. Notice that each failure type is associated to an actual recovery. This is because the actual recovery associated with the failure type is not necessarily the recommended one, and hence it needs to be captured in the model. The question remains how to determine the actual recovery?

6.4.2 Actual Recovery Analysis

We have identified two types of recommended recoveries:

- Mutable recommended recoveries: are the ones that can mutate to other recoveries either at the same level or at a higher level based on the recovery altering attributes. These recoveries are shown in Figure 6-3 with the dotted and dashed border.
- Immutable recommended recoveries: are the ones that are immune to the recovery altering attributes and hence cannot mutate to other recoveries. These recoveries are shown in Figure 6-3 with the solid border.

Note that we can consider the SU level recoveries as **mutated** recoveries (shown in Figure 6-3 with the dashed border) since the only way to force their execution is through the proper setting of the configuration attributes. Figure 6-3 illustrate the mutation path that a recommended recovery can follow.

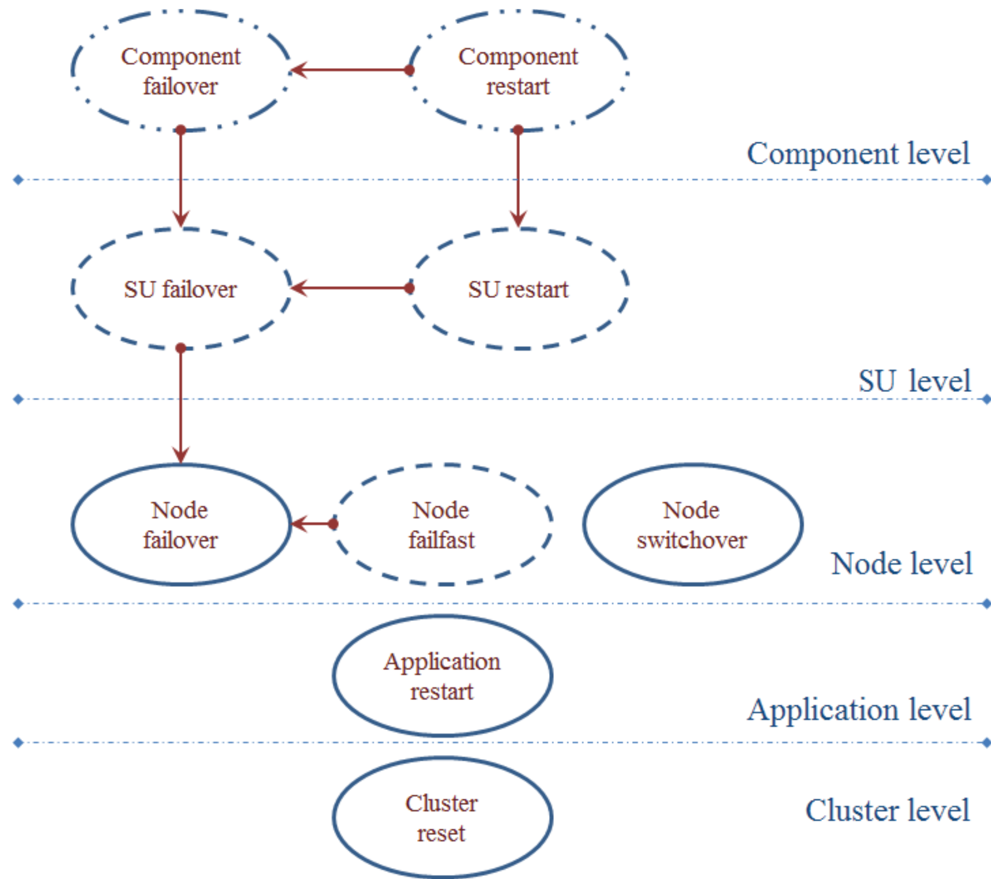


Figure 6-3 Recovery mutation path

In order to determine how the recovery altering attributes mutate a recommended (mutable) recovery, we have devised an actual recovery algorithm that is based on a deep analysis of the AMF specifications. This analysis revealed that the mutation of a recovery

does not solely depend on the recovery altering attributes, but also on other attributes that reflect the properties of certain entities (such as the pre-instantiability[11] of a component).

The actual recovery algorithm is captured through the subsequent flowcharts illustrated in Figure 6-4, Figure 6-5 and Figure 6-6. The main recovery flowchart illustrated in Figure 6-4, will take as input a recommended recovery, and based on the recovery it can branch either to the component restart flowchart illustrated in Figure 6-5, or the component failover flowchart illustrated in Figure 6-6. When we return from either flowchart, the recovery value will be examined again to verify whether or not it will mutate to a node failover. The outputted recovery action in Figure 6-4 is stored in the extended configuration instance that we are analyzing, and it reflects the actual recovery that AMF will perform at runtime in case the failure for which the recommended recovery was defined occurs.

---

[11]A component is said to be non-pre-instantiable if it starts providing its service at the moment of its instantiation, i.e. it cannot remain idle after instantiation waiting for AMF to make the service assignment (which is the case of pre-instantiable components). An SU is considered non-pre-instantiable if it is exclusively composed of non-pre-instantiable components.

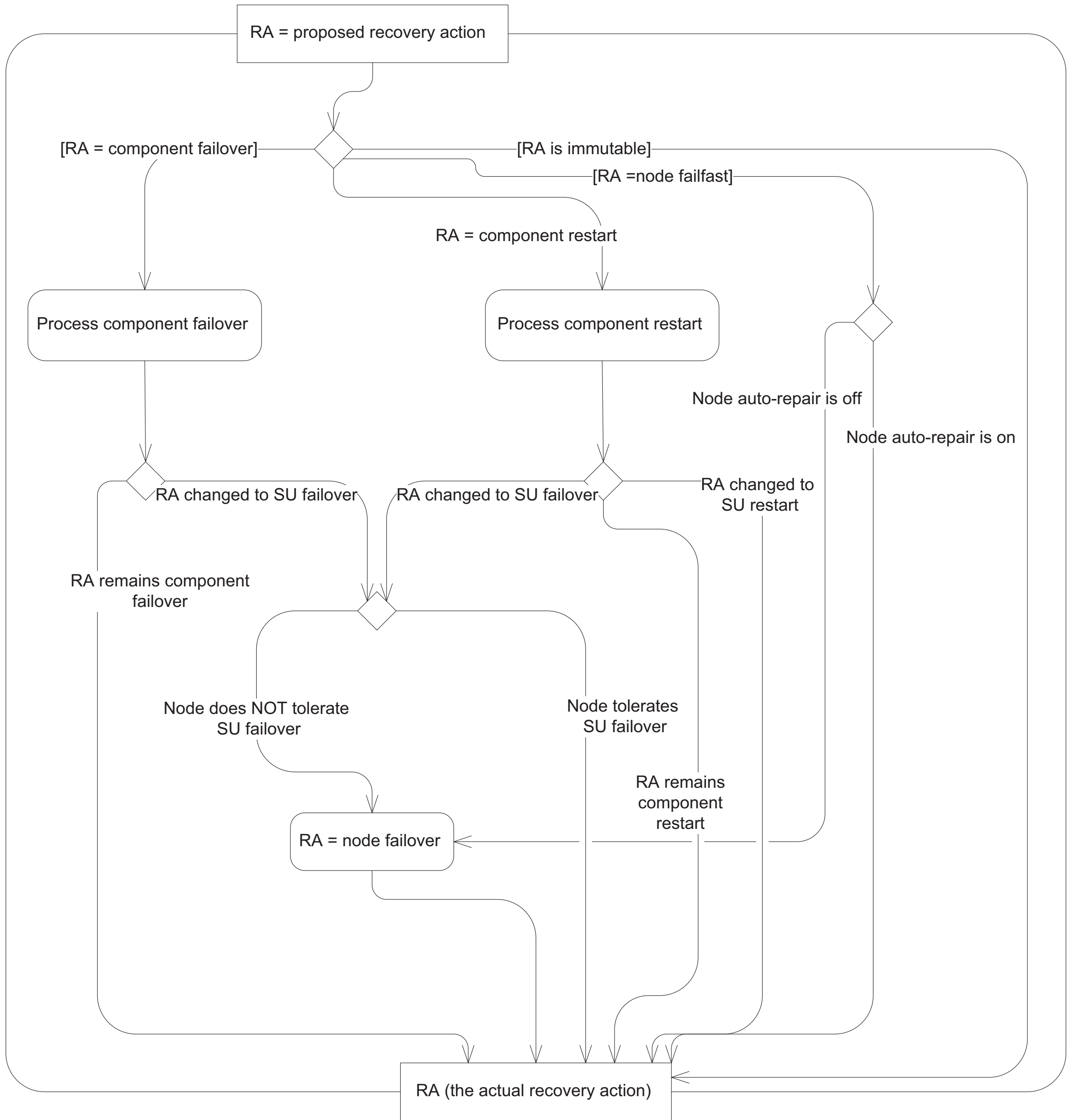


Figure 6-4 The main recovery flowchart

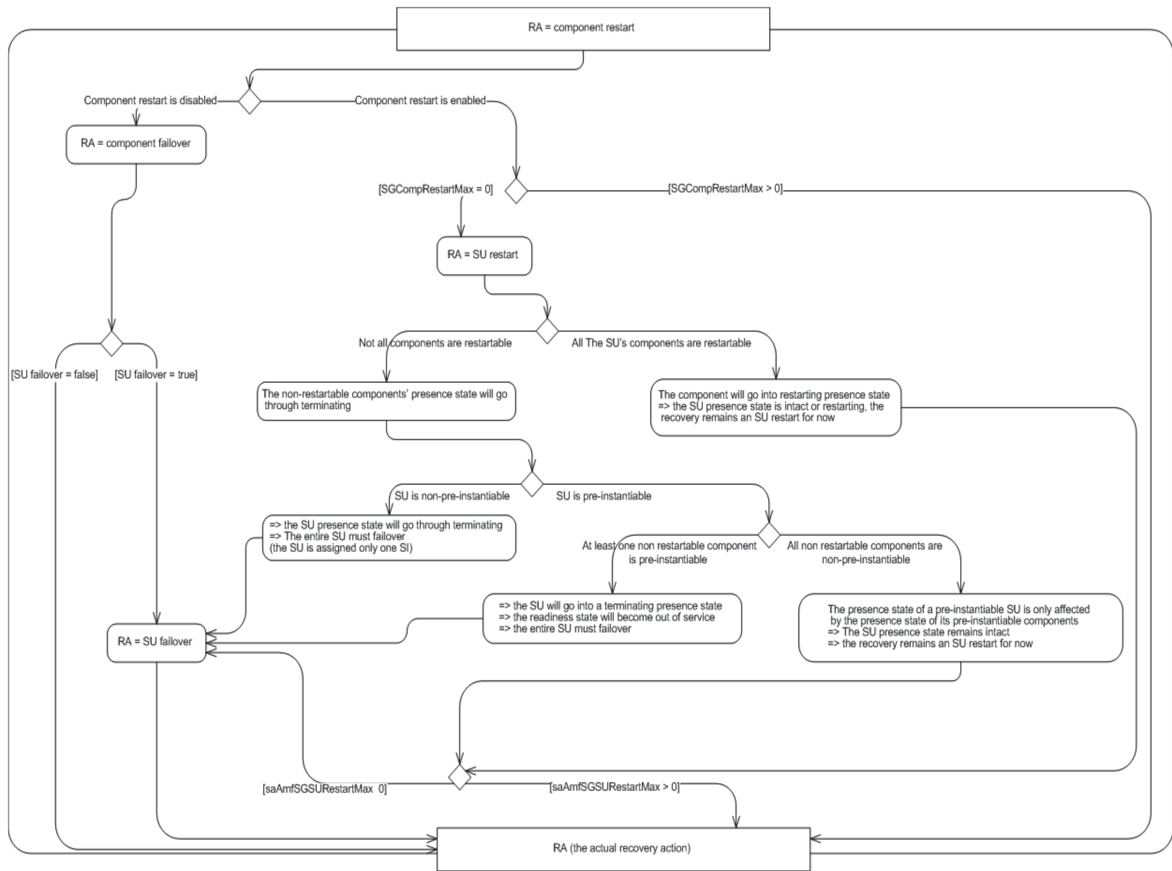


Figure 6-5 The component restart recovery flowchart

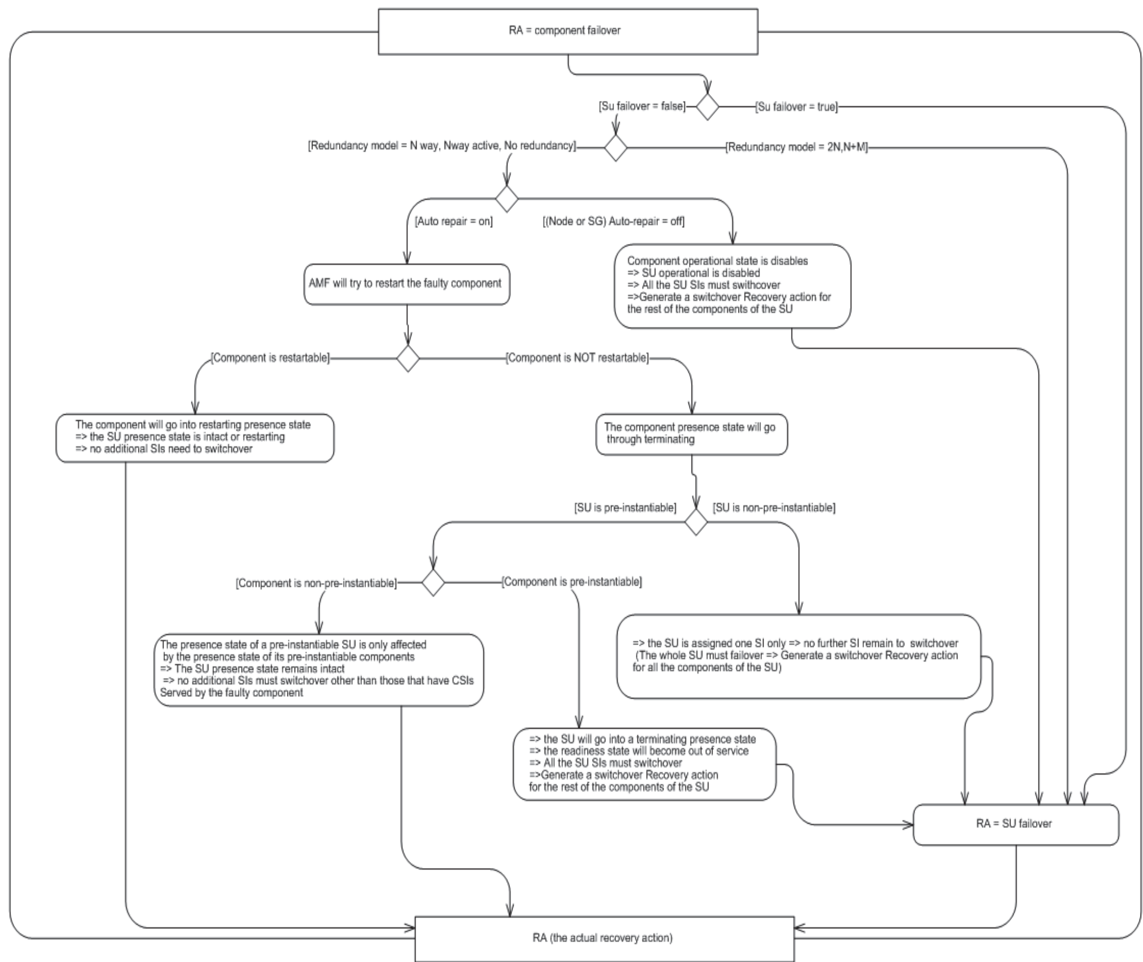


Figure 6-6 The component failover recovery flowchart

6.4.3 Defining the Stochastic Analysis Model

In order to quantify the service availability, we need to formally capture the runtime system behavior in terms of service assignment, and recovery execution. For this purpose we have defined a stochastic model based on the DSPN formalism. The rationale behind using this formalism is the following: the AMF managed systems we are trying to analyze exhibit three types of runtime events, (1) stochastic events like the occurrence of failures. (2) Deterministic events, like the time needed to instantiate a component. (3) Immediate events, such as when a node is abruptly shut down, all the components

running on this node will immediately become un-instantiated. In order to quantify the availability we need a stochastic model where failures/recoveries can be described. Stochastic Petri Nets only support stochastic events, and therefore a more suitable formalism for modeling our system are the Deterministic and Stochastic Petri nets (DSPNs). Another advantage of using DSPNs is their expressiveness in terms of capturing the system behavior by allowing us to specify complex guard conditions (that are marking dependent) which can reflect the complex system behavior, e.g. in terms of capturing the effect of dependencies on the behavior and consequently the service availability.

We defined our stochastic model as DSPN templates. For each AMF entity class on which a recovery can be executed, and also for the SIs/CSIs and the association between component/CSI and SU/SI, we created a DSPN template. The template captures the various states that an entity of these classes can undergo at runtime. The transitions and the guards capture the runtime system behavior in terms of service assignment and recovery execution. We will proceed by introducing each of the templates through the mapping of AMF entities to the corresponding templates. Thereafter we will present how the guard conditions and the transitions rates/times are mapped.

6.4.4 Mapping the Configuration to a DSPN Model ─ Overall Process

We have divided the mapping into two separate phases; the first phase is the structural mapping where, based on a given AMF configuration, we instantiate the DSPN templates (we will present the detailed description of the templates later on in this chapter). The next phase is the annotation, where we annotate the DSPNs with the proper transition

rates and times, and we create the proper guard conditions to capture the dependencies, recoveries, assignments etc. To illustrate how the mapping is performed we consider a very basic example shown in Figure 6-7 where we map an application to an instance of its corresponding template. Assuming that the application has two components, where each component is associated with a failure that has rate, and a recommended recovery which is in both cases an application restart. The first phase of the mapping consists of selecting the predefined templates defined for each entity. For instance the first step is to select the application DSPN template and the component DSPN template and instantiate them. The next step would be to annotate the selected template (we only annotate the application template in this example) with the proper values in terms of rates and guards. As shown in Figure 6-7, the rate of the stochastic transition to the faulty application state is the aggregation of the rates for the component failures where an application restart recovery is recommended. On the other hand we consider the application to be healthy when all the components of the application are terminated, and thus an immediate transition should place a token in the place reflecting the application healthy state. The guard condition of this immediate transition would evaluate to true when all the application components are in a terminated state. I.e. it depends on the marking of the components. Reciprocally (and due to the semantics of the application restart recovery), when the parent application is in a faulty state, a condition would be placed to prohibit the transition of the component to its instantiated state.

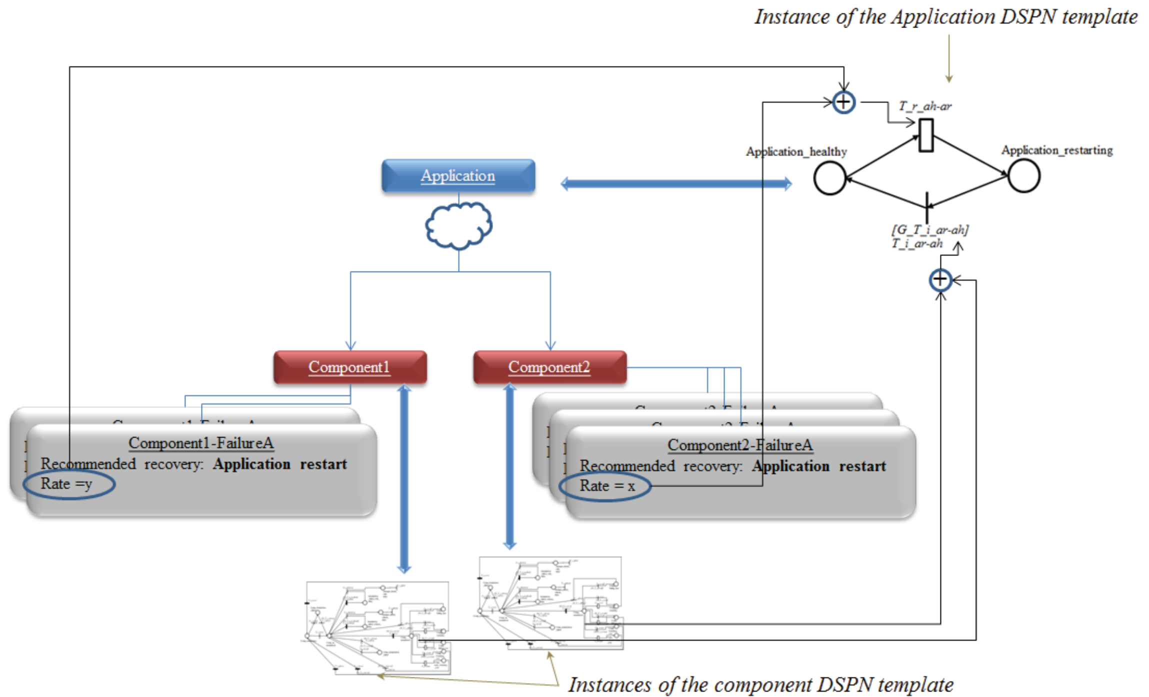


Figure 6-7 Example an application mapping to its corresponding DSPN

We first present an overall description of the structural mapping and then the annotation of the template instances. Since we believe that the dependency mapping is of significant importance we will include it in our overall description. A more detailed description of the mapping (at the attribute level) is described in the definition of the templates themselves.

6.4.4.1 The Structural Mapping

The structural mapping is a relatively straightforward process of creating for each relevant AMF entity (component, SU, application, cluster, node, SI, and CSI) an instance of the corresponding DSPN template, with the proper naming. For each CSI that a component may provide in the SG, a comp-CSI template instance must be created. For each SI in the SG, an SU-SI template instance is created. These association template

instances are used to capture the runtime assignments of the components and SUs on behalf of the CSIs and SIs. This structural mapping, instantiates a DSPN model which can be easily traced back to the AMF configuration model instance in terms of the relevant entities. For instance, based purely on the DSPN model instance, we can determine the structure of the SUs and their components, etc. in the originating AMF model instance.

6.4.4.2 Annotating the Template Instances

The DSPNs created in the structural mapping must be annotated with the proper values. These values are derived from the attribute values of the extended-configuration. For instance, the delay value of the transition that takes the component to the instantiated state is directly mapped from the configuration. However when the AMF configuration specifies that the component category of a component is “contained”, this triggers a specific configuration of several guards of the DSPN instances that capture the dependencies of a component of such a category. This configuration is not directly mapped into a single guard or transition; in fact it is mapped to various guards and transitions in the model in order to capture the way AMF deals with this kind of component category.

6.4.4.2.1 Mapping the transition rates and times

- The times used for the deterministic transitions are mapped from the instance of the extended-configuration model. For instance the time delay of the transition that takes the component from the un-instantiated state to its instantiated state is mapped from the ***saAmfCompInstanteTimeout*** attribute of the component. The

delay for other transition originates from attributes with which we extended the configuration (such as the node startup duration).

- The rates used for the stochastic transitions are collected from the configuration. For each node and component, the failures are analyzed. For the node, the failures that have the same actual recoveries are grouped and their rates are aggregated. E.g. if the node is associated with several failure types from which the actual recovery is a node failfast, then the rate associated with the transition that takes the node into the failing fast state will be the summation of all these rates. Note here that the component as well might have node level recoveries. Therefore all the components residing on the node must be analyzed as well, and the failure rates of the actual recoveries applied on the node must be added to those already specified on the node. For instance the failure types associated with the component that have an actual recovery applied on the node such as a node failover, must have their rates added to the node failover rates of the node itself. Note here the node switchover recovery for the component needs to remain at the component level, and then propagate to the node. This is because the faulty component for which the actual recovery is issued must indeed failed over, and it is the siblings of this component running of the same node that will actually switchover.

As for the component actual recoveries they are grouped into three categories. (1)The ones that remain at the component level and for this again the rates of similar recoveries for the component are aggregated to be used for the relevant transition. (2) For each SU,

all the actual recoveries of its components that evaluate to an SU failover will have their rates aggregated to form the rate of the transition that takes the SU into the failing over state. (3) Similarly for the application, all the actual recoveries of its components that evaluate to an application restart will have their rates aggregated to form the rate of the transition that takes the application into the restarting state.

Finally the cluster reset recovery may be an actual recovery of the node of the component, and therefore the rate of this recovery is the aggregated rate of all the actual recoveries in the configuration that indicate a cluster reset.

6.4.4.2.2 Dependency mapping

In Section 4.2.3 we discussed dependency handling at the configuration requirements level. The goal was to identify the dependencies that would affect the input, and possibly require the designer intervention to complement the input. In this chapter we examine the dependencies from a different perspective. The goal here is to map the dependencies that will affect the service availability. For this purpose we have identified and categorized the dependencies in a somewhat different manner as we explain next. Here we map four different types of dependencies as follows:

- **Instantiation level dependency**: this dependency is specified by the ***saAmfCompInstantiationLevel*** attribute of the component. It is applicable within the scope of the SU.
    - **Meaning**: this dependency states that the components of the lower instantiation level (the dependent ones) cannot be instantiated until all the components with a higher instantiation level (sponsors) within the SU are

instantiated. During an SU restart, the SU with the lower instantiation level are terminated first.

- **Implications**: this dependency does not imply any assignment dependency, i.e. once the components are instantiated they can be assigned independently

- **Impact**: we consider this to be the weakest dependency in terms of its impact on the service availability, because even if at a certain point in time the sponsor is un-instantiated, if the dependent component is already instantiated it can be assigned the service.

- **Mapping**: this dependency is captured in the guard of the transition that takes the component into the instantiated state. This transition is disabled if the sponsor is not already instantiated. During the SU restart, the sponsor cannot go into the terminated state, unless the dependent is already terminated.

- **Proxy/proxied dependency**: this dependency exists between the proxy (sponsor) and the proxied (dependent) component.
    - **Meaning**: this dependency states that the sponsor must be assigned the sponsoring CSI (proxy CSI), before the dependent can be assigned its CSI workload. Moreover the proxy must be assigned the proxy CSI before we can instantiate or terminate the proxied. If the proxied is external, then even the cleanup is performed through the proxy.

- **Implications**: this dependency implies an instantiation dependency, since the proxy must be instantiated and assigned the proxy CSI before the proxied is instantiated.
- **Impact**: we consider this to be stronger than the instantiation dependency in terms of its impact on the service availability, because we must have an active assignment of the sponsoring workload before assigning the CSI workload for the dependent. However even in the absence of the sponsor, if the dependent component is instantiated and assigned the CSIs, it can continue providing service represented by the CSIs until the proxy has recovered.
- **Mapping**: this dependency is captured in the guard of the transition that takes the component into the instantiated state. This transition is disabled if the sponsor is not already instantiated and assigned the proxy CSI. Similarly the proxied cannot be assigned any CSI unless the proxy CSI is assigned active to the proxy. This implies the guard conditions of the transitions to the active or standby states in the Comp-CSI DSPN evaluate to false when the proxy CSI is not assigned active to at least one proxy. During the SU restart, if the proxy resides in the same SU, then it cannot go into the terminated state, unless all the proxied in the SU are already terminated.

- **SI and CSI dependency**: this dependency is an SI-SI or a CSI-CSI dependency. Note that the dependent can have many sponsors and vice versa.

- **Meaning**: this dependency states that the dependent SI/CSI cannot be assigned until all its sponsors are assigned active. Moreover when one of the sponsor SIs is not provided the dependent SI can only survive for a predefined period of time specified in the ***saAmfSIToleranceTime*** attribute of the dependency class.

- **Implications**: this dependency implies that the sponsor SI/CSI must be assigned active before the dependent one; nonetheless it does not imply an instantiation level dependency at the component level. I.e. the component providing the sponsor CSI (or a CSI of a sponsor SI) can be instantiated irrespectively of the component proving the dependent CSI (or a CSI of a dependent SI). If the component is non-pre-instantiable, then its transition to the instantiated state is correlated with the assignment of its CSI.

- **Impact**: we consider this to be a stronger dependency than the proxy-proxied dependency in terms of its impact on the service availability, because when the sponsor CSI is not provided the dependent one is immediately dropped. When the sponsor SI is not provided, the dependent SI can only be provided for a predefined period of time before it is dropped.

- **Mapping**: This dependency is captured by the guards that guard the transition to the active/standby assignment of the CSI, where a CSI cannot be assigned unless its sponsor is assigned active. And by the guards of the SU assignment on behalf of the SI. Moreover due to the tolerance, an

additional deterministic transition is added to the SI DSPN, where for each sponsoring SI, a deterministic transition is added with a time value equal to the corresponding tolerance time. The transition is enabled when the sponsor SI is not provided, when the transition fires, it moves the SI into the dropped state.

- **Container/contained dependency**: this dependency exists between the container (sponsor) and the contained (dependent) component. The container typically acts as an execution environment for the contained; where without the container the contained cannot exist.
    - **Meaning**: this dependency states that the sponsor must be assigned the sponsoring CSI (container CSI), before the dependent can be assigned its CSI workload. Moreover the container must be assigned the container CSI before we can instantiate or terminate the contained. Any recovery applied on the container will affect its contained component(s).
    - **Implications**: this dependency implies all the previous dependencies, in addition the life cycle of the dependent is tightly coupled with the sponsor.
    - **Impact**: we consider this to be the strongest dependency in terms of its impact on the service availability, because we must have an active assignment to the sponsoring workload before instantiating or assigning the CSI workload for the dependent. Moreover in the absence of the sponsor, the dependent cannot exist.

- **Mapping**: this dependency is mapped through not allowing the contained to be instantiated or assigned any service before the container is assigned the container CSI. When the container is not instantiated, the contained will go into the un-instantiated state. When the container CSI is not provided then the workload assigned to the contained is dropped as well.

6.4.5 The DSPN Templates

As mentioned earlier, some attribute values are mapped directly into transition delays or arc multiplicities. In the following tables, attributes in bold text describe the exact name of the configuration attribute from which the value is directly mapped.

6.4.5.1 Naming convention for the template models

In this section we present the naming conventions that we use when creating the DSPN model. Note that the names shown in the templates have to be parameterized for each instance. For example the name of the place Application_healthy of the DSPN template has to be parameterized as Application1_healthy when used in the DSPN instance describing Application1 and the same applies for the transitions. The naming convention is as follows:

- Place: <entity name>_<entity state> e.g. Cluster_healthy
- Transition: *T_t| i|r_<source place reference>-<destination place reference>* e.g. *T_r_ch-cr* stands for a transition with an exponential rate <*r*> going from Cluster_healthy <*ch*> state to cluster resetting <*cr*> place.

- Guard: [G_<transition name>] e.g. [G_T_r_ch-cr] stands for guarding the transition T_r_ch-cr

6.4.5.2 The Cluster DSPN Template

The cluster template (Figure 6-8) is used to capture the states that the cluster may go through. The description of the states, transitions and guards are presented respectively in

Table 6-1, Table 6-2, and Table 6-3.

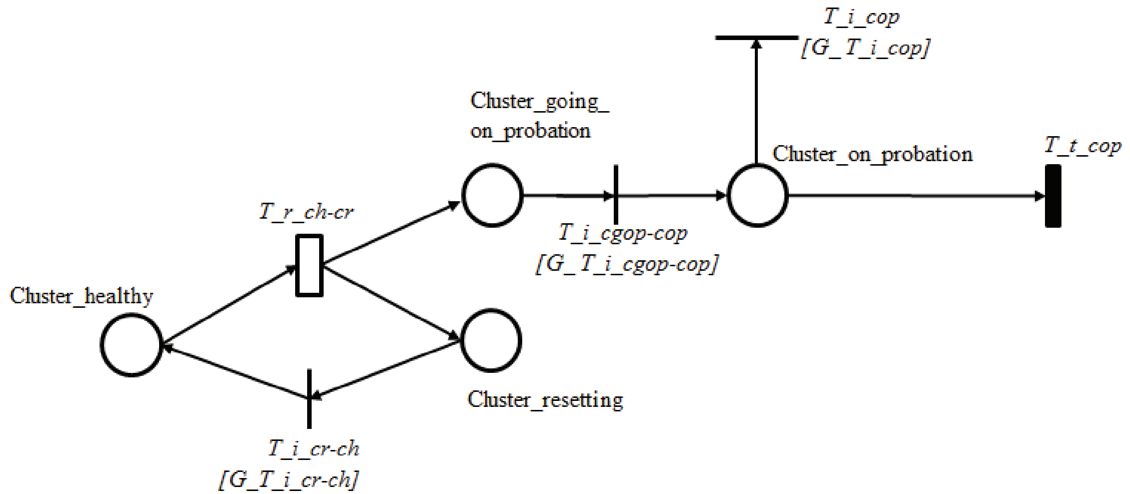


Figure 6-8 The cluster DSPN template

Table 6-1 The cluster states

State[12] (or place)	Description
Cluster_healthy	Initially or after a cluster reset, when all the cluster nodes are shutdown, the cluster is considered to be healthy.
Cluster_resetting	The cluster is resetting as a recovery action.
Cluster_going_on_probation	This state is needed to make sure that when a cluster reset occurs while the cluster is still on probation, then the probation period is reset. The token in this place cannot move on to the Cluster_on_probation place until the latter is empty, i.e. has no tokens.
Cluster_on_probation	A token in this place signifies that no SU-SI assignment is allowed, until either the startup period is over, or all the needed components in the cluster are instantiated.

Table 6-2 Cluster transitions

[12] In this section, we use the term state and place interchangeably, simply because in certain contexts the term state conveys a better meaning.

Transition	Description
T_r_ch-cr	This transition has an exponentially distributed rate. The rate is the summation of all the rates of the cluster reset actual recoveries issued on the cluster's components and nodes. The transition takes a token out of the Cluster_healthy place and positions one token in the Cluster_resetting place, and another in the Cluster_going_on_probation place.
T_i_cr-ch	This is an immediate transition. It takes a token out of the Cluster_resetting place and positions it in the Cluster_healthy place.
T_i_cgop-cop	This is an immediate transition. It takes a token out of the Cluster_going_on_probation place and positions it in the Cluster_on_probation place.
T_i_cop	This is an immediate transition that flashes the token in the Cluster_on_probation place when all the components of the cluster are instantiated.
T_t_cop	This is a deterministic transition; it reflects the time needed for the cluster startup probation period to expire. The time **value** of this transition is specified by the ***saAmfClusterStartupTimeout*** attribute of the *SaAmfCluster* class.

Table 6-3 Cluster guard conditions

Guard	Description
G_ T_i_cr-ch	It guards the transition of the cluster to the Cluster_healthy state, the guard condition evaluates to **true** when: • All the cluster nodes are in the Node_shutdown state. I.e. all the nodes have a token in the Node_shutdown place.
G_ T_i_cgop-cop	It guards the transition of the cluster to the Cluster_on_probation states, the guard condition evaluates to **true** when: • The cluster is not already on probation i.e. there are no tokens in the Cluster_on_probation state. And when the cluster is in a healthy state.
G_ T_i_cop	It guards the flushing of the Cluster_on_probation state. The guard condition evaluates to **true** when either of the below conditions are true: • All the components of the instantiated SUs of the cluster are instantiated i.e. they have a token in the Comp_instantiated state. • When yet another cluster reset recovery occurs while the cluster is still on probation, i.e. the place

	Cluster_going_on_probation has a token.

6.4.5.3 The Node DSPN Template

The node template (Figure 6-9) must capture the various states that the node goes through. The assumption here is that there exists a process monitoring the health of the node, and in case anomalies are detected, the process will report to AMF the recommended recovery. Thus we find in the model the stochastic transition to the failing fast, failing over and switching over states. The description of the states, transitions and guards are presented respectively in Table 6-4, Table 6-5, and Table 6-6.

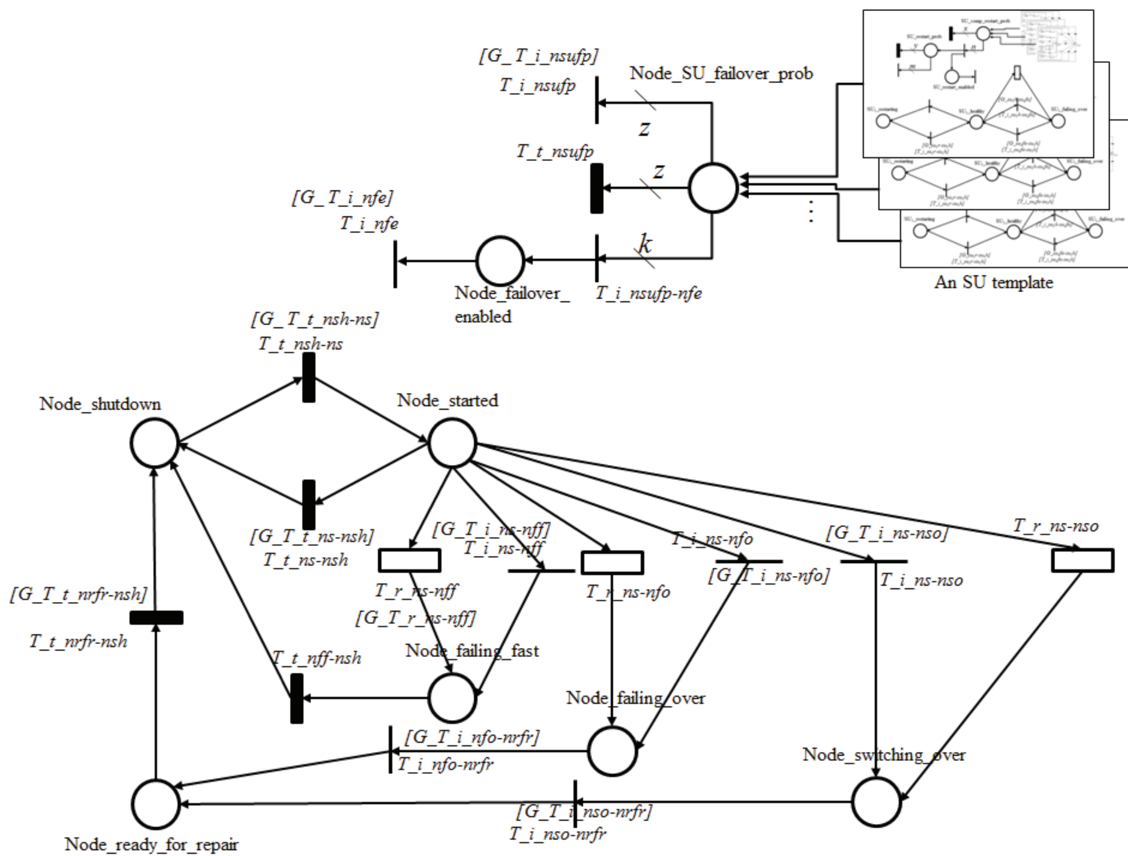


Figure 6-9 The node DSPN template

Table 6-4 The node states

State	Description
Node_shutdown	The AMF node is not started, i.e. the components on this node cannot be instantiated.
Node_started	The AMF node is considered healthy, and ready to host instantiated components.
Node_failing_fast	The node is in a failing fast state as a recovery action, the sojourn time in this state is not very significant.
Node_failing_over	The node is in a failing over state as a recovery action, where all the components are abruptly terminated. When all the components have been cleaned up, the node will become ready for restart.
Node_switching_over	The node is in a switching over state, all the node's components (except for the faulty one(s)) have their workload assignments gracefully removed. When this is completed, and the faulty components have been cleaned up, the node is ready for restart.
Node_ready_for_repair	When the node auto repair is ON, then a node in this state can transition to the shutdown state, as a first step towards being

	restarted.
Node_SU_failover_prob	When any SU of the node fails over, then node enters a failover probation period, this state is used to keep track of the number of SUs failing over within the probation period.
Node_failover_enabled	This state indicates that the node is ready to go into a failing over state as an escalation recovery due to repetitive failovers of the node's SUs

Table 6-5 Node transitions

Transition	Description
T_t_nsh-ns	This is a deterministic transition; it reflects the time needed for a node to go from its shutdown state[13](Node_shutdown) into a

[13] For simplicity's sake we refer to transitions shifting entities from one state to another, the shift of the token is implied.

	started state (Node_started). The time **value** of this transition is specified by the **user**[14]in the extended attributes of the AMF configuration.
T_t_ns-nsh	This is a deterministic transition that reflects the time needed for a node to go from its started state (Node_started) into a shutdown state (Node_shutdown) caused by a cluster reset. The time **value** of this transition is specified by the **user**in the extended attributes of the AMF configuration
T_r_ns-nff	This is a stochastic transition that takes the node from its started state (Node_started) into a failing fast state (Node_failing_fast). This transition has an exponentially distributed rate; the rate is the summation of all the rates of the node failfast actual recoveries issued on the node's components[15]and the node itself.

[14] The user may be the system configurator or administrator who has the knowledge of the hardware and the OS, or has tested the system to get some booting and shutting down timings.

[15]For this, we make the assumption that for each SU, the hosting node is specified in the configuration. Thus we can associate at configuration time the components and nodes.

T_i_ns-nff	This is an immediate transition that transitions a started node (Node_started) to a failing fast state (Node_failing_fast) due to an escalation caused by the node's components failing to instantiate/terminate.
T_r_ns-nfo	This is a stochastic transition that takes the node from its started state (Node_started) into a failing over state (Node_failing_over). This transition has an exponentially distributed rate; the rate is the summation of all the rates of the node failover actual recoveries issued on the node's components or the node itself.
T_i_nsufp	This is an immediate transition that flushes the tokens in the Node_SU_failover_Prob place. It should be triggered after a node reboot to signal that the node is no longer on probation.
T_i_ns_nfo	This is an immediate transition that transitions a started node (Node_started) to a failing over state (Node_failing_over) due to an escalation caused by the node's SUs failing over.
T_i_ns-nso	This is an immediate transition that takes the node into a switching over state (Node_switching_over). It is triggered by any component of the node requesting a node switchover as a recovery action.
T_r_ns-nso	This is a stochastic transition that takes the node from its started state (Node_started) into a switching over state

	(Node_switching_over). This transition has an exponentially distributed rate; the rate is the summation of all the rates of the node switchover actual recoveries issued on the node itself[16].
T_t_nff-nsh	This is a deterministic transition that reflects the time needed for a node to go from failing fast (Node_failing_fast) to shutdown state (Node_shutdown). The time **value** of this transition is specified by the **user** in the extended attributes of the AMF configuration**.**
T_i_nfo-nrfr	This is an immediate transition that transitions a failing over node (Node_failing_over) to a state where it is ready for repair (Node_ready_for_repair) i.e. restart.
T_i_nso-nrfr	This is an immediate transition that transitions a switching over node (Node_switching_over) to a state where it is ready for repair (Node_ready_for_repair) i.e. restart.

[16]We distinguish between the switchover issued on the node and the one issued on the component, because with the node switchover recovery issued on the component the faulty component is in fact failed over, and it is its siblings on the same node that are indeed switched over. When the node switchover is triggered for the node, all the components are switched over.

T_t_nrfr-sd	This is a deterministic transition that reflects the time needed for a node to go from its ready for repair state (Node_ready_for_repair) into a shutdown state (Node_shutdown). The time **value** of this transition is specified by the **user**in the extended attribute (***shutDownDuration***) of the AMF configuration**.**
T_t_nsufp	This is a deterministic transition that reflects the time needed for the SU failover probation period to expire. When this transition fires, it flushes all the tokens in the Node_SU_failover_prob. The time **value** of this transition is specified by the ***saAmfNodeSuFailoverProb*** of the *SaAmfNode* class. **Note** that the arc cardinality '**z**' = the number of tokens in the Node_SU_failover_prob state.
T_i_nsufp-nfe	This is an immediate transition that is triggered when the number of tokens in the Node_SU_failover_prob reaches the threshold. This Threshold (the arc cardinality '**k**') is equal to ***saAmfNodeSuFailoverMax.***
T_i_nfe	This is an immediate transition that is triggered to flush the Node_failover_enabled state, indicating that the node failover escalation was executed.

Table 6-6 Node guard conditions

Guard	Description
G_ T_t_nsh-ns	It guards the transition of the node to the started state. (it is applied to transition *T_t_nsh-ns*included in the name of the guard)The guard condition evaluates to **false** when: • The cluster is in the Cluster_resetting state. I.e. it is false when there is a token in the Cluster_resetting place.
G_T_t_ns-nsh	It guards the transition of the node from the started state to the shutdown state. The guard condition evaluates to **true** when: • The cluster is in a Cluster_resetting state. I.e. it is true when there is a token in the Cluster_resetting place.
G_T_i_ns-nff	It guards the immediate transition of the node to Node_failing_fast state. The guard condition evaluates to **true** when either one of the below conditions are true: • Acomponent of this node is in the Comp_instantiation_failed state (i.e. there is a token in this state) and the ***saAmfNodeFailfastOnInstantiationFailure***attribute of the *SaAmfNode* class is set to **true**. • Acomponent of this nodeis in the Comp_termination_failed state (i.e. there is a token in this

	state) and the ***saAmfNodeFailfastOnTerminationFailure*** is set to **true**
G_T_i_ns-nfo	It guards the transition of the node to Node_failing_over state. The guard condition evaluates to **true** when: • There is a token in the Node_failover_enabled state as a result of the number of SUs failing over reaching the ***saAmfNodefailoverMax***value. I.e. the number of tokens in the Node_SU_failover_prob reaches this value.
G_T_i_ns-nso	It guards the transition of the node to Node_switching_over state. The guard condition evaluates to **true** when: • Any of the node's components has a token in the Comp_node_switchover state.
G_T_i_nfo-nrfr	It guards the transition of the node to being ready for repair state (Node_ready_for_repair) after being in theNode_failing_over state. The guard condition evaluates to **true** when: • All the nodes components have beencleaned up. I.e. there is a token in the Comp_un-instantiated place of all the node's components.
G_T_i_nso-nrfr	It guards the transition of the node to being ready for a restart state after being in a switching over state. The guard condition

	evaluates to **true** when: • All the node's components do not have any HA state on behalf of any CSI. I.e. they all have a token in the Comp_CSI_unassigned for each CSI they can provide.
G_T_t_nrfr-sh	It guards the transition of a node in a Node_ready_for_repair stateto a Node_shut_down state. • This guard has the **same** value asthe node auto repair attribute ***<u>saAmfNodeAutoRepair</u>*** of the *SaAmfNode* class.
G_T_i_nfe	This guard is enabled when there is a token in the Node_failing_over state. It signifies that when the node is already in the failing over state, we can go ahead and flush the token that is in the Node_failover_enabled state.
G_T_i_nsufp	This guard is used to flush the tokens in the Node_SU_failover_prob place. It evaluates to **true** when: • The node has a token is in the Node_shutdown place

6.4.5.4 The Application DSPN template

The application model (Figure 3) is used to capture the state that the application goes through.

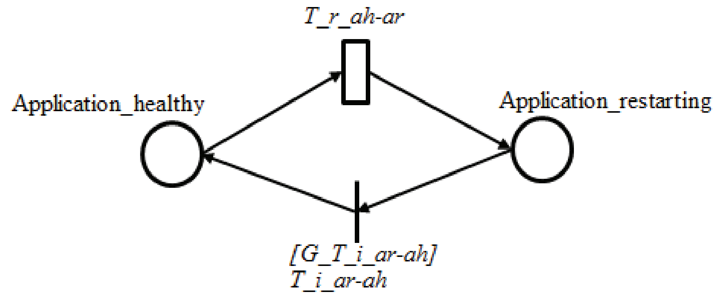


Figure 6-10 The application DSPN template

The application template states, transitions, and guards are explained respectively in Table 6-7, Table 6-8, and Table 6-9.

Table 6-7 Application states

State	Description
Application_healthy	This state simply means that the application is not undergoing a restart in order to overcome a failure.
Application_restarting	The application is restarting as a recovery action.

Table 6-8 Application transitions

Transition	Description
T_r_ah-ar	This is a stochastic transition that takes the application from its healthy state (Application_healthy) into a restarting state (Application_restarting). This transition has an exponentially distributed rate. The rate is the summation of all the rates of the

	application restart actual recoveries issued on the application's components. The transition takes a token out of the Application_healthy place and positions it in the Application_restarting place.
T_i_ar-ah	This is an immediate transition. It takes a token out of the Application_restarting place and positions it in the Application_healthy place.

Table 6-9 Application guard conditions

Guard	Description
G_T_i_ar-ah	It guards the transition of the application to the Application_healthy state, the guard condition evaluates to **true** when: • **All** the application components are in the un-instantiated state. I.e. all the application components have a token in the Comp_un-instantiated place.

6.4.5.5 The SU DSPN Template

The SU model (Figure 6-11) is used to capture the recoveries specified at the SU level.

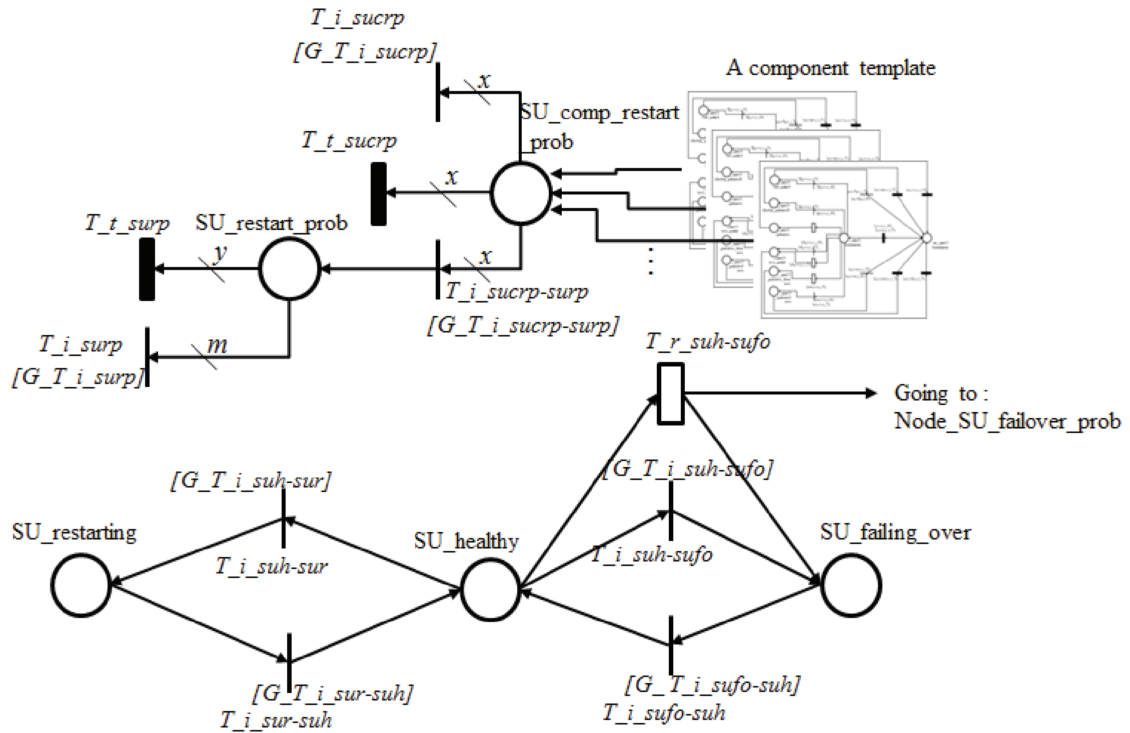


Figure 6-11 The SU DSPN template

The SU can transition from the healthy state into either the failing over state, or the restarting state. The default state of the SU is the SU_healthy state. The SU template states, transitions, and guards are explained respectively in Table 6-10, Table 6-11, and Table 6-12.

Table 6-10 The SU states

State	Description
SU_healthy	This state simply means that the SU is not undergoing a restart or a failover in order to overcome a failure.

SU_restarting	The SU is restarting as an escalation of a component restart recovery action.
SU_failing_over	The SU is failing over as a recovery action or an escalation of a component failover.
SU_comp_restart_prob	The SU is on a component restart probation, caused by a component restart triggered on one of the SU's components
SU_restart_prob	The SU is on an SU restart probation, it escalated to this state after being in the SU component restart probation state.

Table 6-11 The SU transitions

Transition	Description
T_i_suh-sur	This is an immediate transition that takes the SU from a healthy state (SU_healthy) into the restarting state (SU_restarting).
T_i_sur-suh	This is an immediate transition that takes the SU from SU_restarting to SU_healthy.
T_r_suh-sufo	This is a stochastic transition that takes the SU from its healthy state (SU_healthy) into a failing over state (SU_failing_over).This transition has an exponentially distributed rate; the rate is the summation of all the rates of the SU failover actual recoveries

	issued on the SU's components.
T_i_suh-sufo	This is an immediate transition that takes the SU from the SU_healthy to the SU_failing_over state as a result of an escalation.
T_i_sufo-suh	This is an immediate transition that takes the SU back to its SU_healthy state from the SU_failing_over state.
T_t_sucrp	This is a deterministic transition that reflects the duration until the SU component restart probation period expires. When this transition fires it flushes all the tokens from the SU_comp_restart_prob state. The time **value** of this transition is specified by the ***saAmfSgCompRestartProb*** attribute of the *SaAmfSG* class (i.e. the parent SG of this SU). The arc cardinality **x** is equal to the number of tokens in the SU_comp_restart_prob state.
T_i_sucrp-surp	This is an immediate transition that takes the SU to an SU restart probation (SU_restart_prob) state. And flushes the SU_Comp_restart_prob state of its tokens. The arc cardinality **x** is equal to the number of tokens in the SU_comp_restart_prob state.
T_t_surp	This is a deterministic transition that reflects the duration until the SU restart probation period expires. The time **value** of this transition is specified by the ***saAMFSGSuRestartProb*** attribute

	of the *SaAmfSG* class.
T_i_surp	This is an immediate transition that flushes the SU_restart_prob state of its tokens. The arc cardinality **m** is equal to the number of tokens in the SU_restart_prob state.
T_i_sucrp	This is an immediate transition that flushes the SU_Comp_restart_prob state of its tokens. The arc cardinality **x** is equal to the number of tokens in the SU_Comp_restart_prob state.

Table 6-12 The SU guard conditions

Guard	Description
G_T_i_suh-sur	It guards the transition of the SU to its restarting state. The guard condition evaluates to **true** when: • The number of components restarted within the probation period exceeds the threshold. I.e. when the number of tokens in the SU_comp_restart_prob reaches the ***saAMFSGCompRestartMax*** attribute value specified in the parent SG.
G_T_i_sur-suh	It guards the transition of the SU to its healthy state (SU_healthy),

	the guard condition evaluates to **true** when: • All the components are not in a recovering state. I.e. they are either in the Comp_un-instantiated or Comp_instantiated state. In other words none of the components are faulty.
G_T_i_suh-sufo	It guards the transition of the SU to its failing over state. The guard condition evaluates to **true** when: • The number of times the SU is restarted within the probation period exceeds the threshold. I.e. when the number of tokens in the SU_restart_prob reaches the ***saAmfSgSuRestartMax*** attribute value of the parent SG
G_T_i_sufo-suh	It guards the transition of the SU to its healthy state (SU_healthy), the guard condition evaluates to **true** when: • All the components are not in a recovering state. I.e. they are either in the Comp_un-instantiated or Comp_instantiated state. In other words none of the components are faulty.
G_T_i_sucrp	The guard condition evaluates to **true** when any of the below conditions are true:

	• The parent node is shut down • The SU is failing over • The parent application is restarting
G_T_i_surp	The guard condition evaluates to **true** when any of the below conditions are true: • The parent node is shut down • The SU is failing over
G_T_i_sucrp-surp	The guard condition evaluates to **true** when: • The SU is in the SU_restart_prob state.

6.4.5.6 The Component DSPN Template

The component model (Figure 6-12) is used to capture the recoveries that the component undergoes; these recoveries are either intended for this specific component, or they are implicated by other recoveries performed on higher level entities e.g. the SU or the node.

Figure 6-12 The component DSPN template

The component template states, transitions, and guards are explained respectively in Table 6-13, Table 6-14, and Table 6-15.

Table 6-13 The component states

State	Description
Comp_un-instantiated	This state means that the component is still un-instantiated.
Comp_instantiated	The component is instantiated.
Comp_failing_over	The component is failing over
Comp_node_switching_over	The component will be cleaned up and the entire

	node must switchover
Comp_switching_over	The node is switching over, and as a result the component can no longer keep any assignment.
Comp_restarting	The component is restarting as a result of a component restart issued on the component or the SU or the application.
Comp_failing_fast	The component is failing fast as a result of a node failfast.
Instantiation_failed_without_delay	A component in this state means that an attempt to instantiate the component without delay has failed.
Attemps_failed_without_delay	This state is used to keep track of the number of attempts made without delay
Instantiation_failed_with_delay	A component in this state means that an attempt to instantiate the component with delay has failed.
Attempts_failed_with_delay	This state is used to keep track of the number of attempts made with delay
Comp_instantiation_failed	The component reaches this state when all the attempts with/without delay have failed. According to the configuration, when this state is reached, the

	node may go through a failfast state.
Comp_terminating	A component in this state is on its way of being un-instantiated, if the termination fails, it will go into a termination failed state.
Comp_termination_failed	The component reaches this state when an attempt to clean it up fails. According to the configuration, when this state is reached, the node may go through a failfast state.

Table 6-14 The component transitions

Transition	Description
T_t_cu-ci	This is a deterministic transition that shifts the component into the instantiated state. This transition is associated with a ***probability (1-(P_x + P_y)***[17] that the component might instantiate successfully. (P_X and P_y are introduced

[17]These probabilities are statistical information that is assumed to be available to the system administrator.

	subsequently in this table, they are part of the extensions to the configuration). The transition time is equal to the ***saAmfCompInstanteTimeout***attribute value of the component.
T_i_ci-cfo	This is an immediate transition that takes the component to the failing over state (Comp_failing_over) caused by the failover of the SU or the node or the containing container component (if this is a contained component).
T_r_ci-cfo	This is a stochastic transition that takes the component from its instantiated state (Comp_instantiated) into a failing over state (Comp_failing_over). This transition has an exponentially distributed rate equal to the summation of all the actual recoveries of the component that evaluate to a component failover.
T_r_ci-cnso	This is a stochastic transition that takes the component from its instantiated state (Comp_instantiated) into a node level switching over state (Comp_node_switching_over). This transition has an exponentially distributed rate equal to the summation of all the actual recoveries of the component that evaluate to a node switchover.
T_i_ci-cso	This is an immediate transition that takes the component

	from its instantiated state (Comp_instantiated) into a switching over state (Comp_switching_over), state caused by the switchover of the node where the component is running.
T_r_ci-cr	This is a stochastic transition that takes the component from its instantiated state (Comp_instantiated) into a restarting state (Comp_restarting). This transition has an exponentially distributed rate equal to the summation of all the actual recoveries of the component that evaluate to a component restart. If the component is a container, then all the actual recoveries that evaluate to a container restart are added to this rate as well.
T_i_ci-cr	This is an immediate transition that takes the component to the restarting state (Comp_restarting), where the restart is in an enclosing entity, i.e. the SU or the application or the containing component is restarting.
T_i_ci-cff	This is an immediate transition that takes the component into a failing fast state (Comp_failing_fast).
T_i_cff-cu	This is an immediate transition that takes the component from the Comp_failing_fast state to the Comp_terminatingstate.

T_t_cr-ct	This is a deterministic transition that shifts the component from the Comp_restarting to the Comp_terminating state. The time duration of this transition is equal to the ***saAmfCompCleanupTimeout*** attribute value of the component.
T_t_cfo-ct	This is a deterministic transition that shifts the component from the Comp_failing_over to the Comp_terminating state. The time duration of this transition is equal to the ***saAmfCompCleanupTimeout*** attribute value of the component.
T_i_cnso-cu	This is an immediate transition that takes the component from the Comp_node_switching_over state to the Comp_un-instantiated state.
T_i_cso_cu	This is an immediate transition that shifts the component from the Comp_switching_over state to the Comp_un-instantiatedstate.
T_i_cif-cu	This is an immediate transition that takes the component from the Component_instantiation_failed state to the Comp_un-instantiated state.
T_i_cu-cif	This is an immediate transition that takes the component from the Comp_un-instantiated state to the

	Comp_instantiation_failed state
T_t_cu-ifwod	This is a deterministic transition that shifts the component from the Comp_un-instantiated state into the Attemps_failed_without_delay state. It reflects a failed attempt to instantiate the component without delay between attempts. The time duration of this transition is equal to the ***saAmfCompCleanupTimeout***attribute of the component. This transition is associated with a ***probability*** P_x that the component might fail to instantiate without delay.
T_i_ifwod-cu	This is an immediate transition that takes the component into the Comp_un-instantiated state after failing to instantiate without delay. It also places a token in the Attemps_failed_without_delay state to keep track of the number of attempts made.
T_i_afwod	This is an immediate transition that flushes the Attempts_failed_without_delay state. The arc cardinality **y** is equal to the number of tokens in this state.
T_t_cu_ifwd	This is a deterministic transition that shifts the component into the Instantiation_failed_with_delay state. It reflects a failed attempt to instantiate the component with delay

	between attempts. The time duration of this transition is equal to the ***saAmfCompInstantiateTimeout*** attribute of the component + the delay duration specified by ***saAmfCompDelayBetweenInstantiateAttemps***. This transition is associated with a ***probability P_y*** that the component might fail to instantiate with delay.
T_i_ifwd-cu	This is an immediate transition that takes the component into the Comp_un-instantiated state. It also places a token in the Attemps_failed_with_delay state to keep track of the number of attempts made.
T_i_afwod	This is an immediate transition that flushes the Attempts_failed_with_delay state. The arc cardinality **x** is equal to the number of tokens in this state.
T_i_ct-cu	This is an immediate transition that takes the component into the Comp_uninstantiation state. It is associated with a **probability** P_z that the component terminates successfully. P_z is specified by the user in the extended attributes of the AMF configuration.
T_i_ctf-cu	This is an immediate transition that takes the component into the Comp_un-instantiated state.

T_i_ct-ctf	This is an immediate transition that takes the component into the Comp_termination_failed state. It is associated with a **probability (1- P_z)** that the component will not terminate successfully.

Table 6-15 The component guard conditions

Guard	Description
G_T_t_cu-ci	It guards the transition of the component from the Comp_un-instantiated state the Comp_instantiated state, the guard condition evaluates to **true** when all the below conditions are true: • The hosting node is started (i.e. there is a token in the Node_started state of the hosting node) • There is no application restart taking place. (I.e. there is no token in the Application_restarting state of the parent application) • No component with lower instantiation order still un-instantiated. • If the component is a proxied, then the proxy must be assigned the **proxy** CSI (i.e. there is a token in

	the Comp_CSI_active state of the proxy comp with respect to the proxy CSI). • If the component is contained, then the container must be assigned the **container** CSI. (I.e. there is a token in the Comp_CSI_active state of the container comp with respect to the container CSI).
G_T_i_cff-cu	It guards the transition of the component to the un-instantiated state, The guard condition evaluates to **true** when: • The hosting node is in the shutdown state. (I.e. there is a token in the Node_shutdown state of the hosting node).
G_T_i_ci-cso	It guards the transition of the component to a switching over state. The guard condition evaluates to **true** when either of the below conditions are true: • The hosting node is in the switching over state. (I.e. there is a token in the Node_switching_over state of the hosting node). • (If the component is contained), the container component is switching over. (I.e. there is a token

	in the Comp_switching_over state of the container component)
G_T_i_ci-cr	It guards the transition of the component to the restarting state. The guard condition evaluates to **true** when either of the below conditions is true: • The SU goes into a restarting state. (I.e. there is a token in the SU_restarting state of the parent SU) • The application goes into a restarting state. (I.e. there is a token in the Application_restarting state of the parent application) • (If the component is contained), the container component goes into a restarting state. (I.e. there is a token in the Comp_restarting state of the container component) and the ***saAmfCompDisableRestart*** attribute value of the contained is set to false.
G_T_i_ci-cff	It guards the transition of the component to the failing fast state. The guard condition evaluates to **true** when: • The hosting node is in the failing fast state. (I.e. there is a token in the Node_failing_fast state of

	the hosting node)
G_T_i_cso-ct	It guards the transition of the component to the terminating state. The guard condition evaluates to **true** when: • The hosting node is in the shutdown state. (I.e. there is a token in the Node_shutdown state of the hosting node)
G_T_i_ci-cfo	It guards the transition of the component to the failing over state. The guard condition evaluates to **true** when any of the below conditions are true: • The hosting node is failing over. (I.e. there is a token in the Node_failing_over state of the hosting node) • The parent SU is failing over. (I.e. there is a token in the SU_failing_over state of the parent SU) • (If the component is contained), the container component is failing over. (I.e. there is a token in the Comp_failing_over state of the container component). Or if the container component goes into a restarting state. (I.e. there is a token in the

	Comp_restarting state of the container component) and the ***saAmfCompDisableRestart*** attribute value of the contained is set to true.
G_T_i_cnso-cu	It guards the transition of the component to the un-instantiated state. The guard condition evaluates to **true** when: • The hosting node is in shut down. (I.e. there is a token in the Node_shutdown state of the hosting node)
G_T_i_cif-cu	It guards the transition of the component to the un-instantiated state. The guard condition evaluates to **true** when: • The hosting node is shut down. (I.e. there is a token in the Node_shutdown state of the hosting node)
G_T_i_cu-cif	It guards the transition of the component to the instantiation failed state. The guard condition evaluates to **true** when either of the below conditions is true: • The ***CompNumMaxInstantiateWithoutDelay*** is reached (in terms of the number of tokens in the

	Attempts_failed_without_delay state) while the ***CompNumMaxInstantiateWithDelay*** is zero. • The ***CompNumMaxInstantiateWithDelay*** ($\neq$ zero***)*** is reached (in terms of the number of tokens in the Attempts_failed_with_delay state).
G_T_i_ctf-cu	It guards the transition of the component from the termination-failed state to the un-instantiated state. The guard condition evaluates to **true** when: • The node is in a shutdown state. (I.e. there is a token in the Node_shutdown state of the hosting node).

6.4.5.7 The Comp-CSI DSPN Template

For each CSI that the component can serve, it can be assigned the active, standby or unassigned state. Note that there are other states such as quiescing and quiesced that are not of particular interest for our analysis. The Comp-CSI DSPN template (shown in Figure 6-13) is used to capture the runtime assignment state for a component on behalf of a CSI. Note that in the analysis model, an instance of this template must be created for each CSI (within the SG) that the component can provide, i.e. the component supports the provisioning of the CS type of the CSI. The template states, transitions, and guards are explained respectively in Table 6-16, Table 6-17, and Table 6-18.

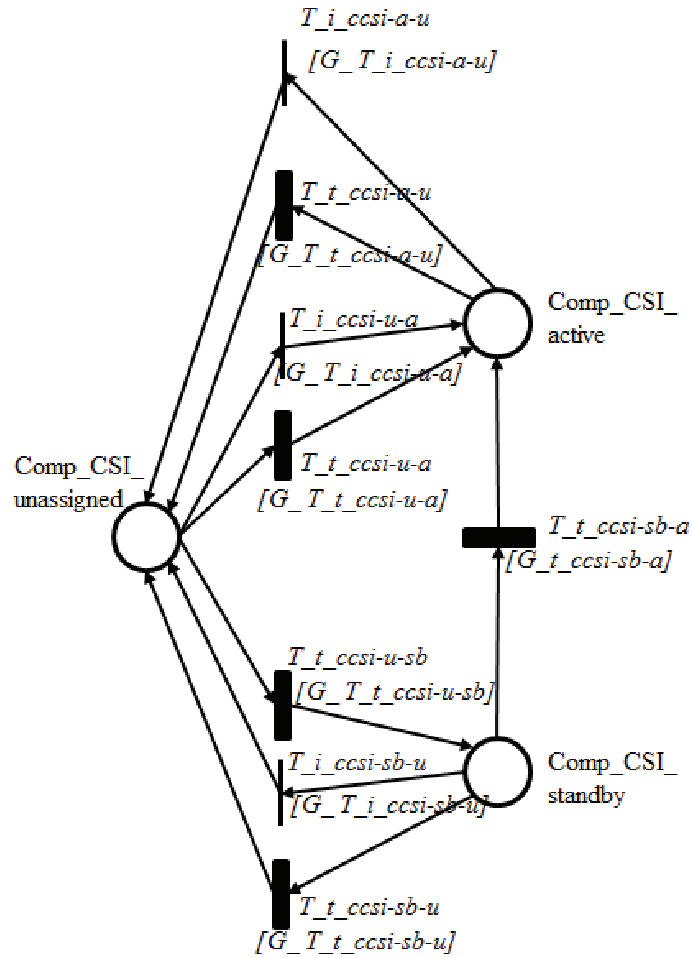


Figure 6-13 The comp-CSI DSPN template

Table 6-16 The Comp-CSI states

State	Description
Comp_CSI_unassigned	This state means that the component is not assigned for this particular CSI.
Comp_CSI_active	The component is assigned active for this CSI
Comp_CSI_standby	The component is assigned standby for this CSI

Table 6-17 The Comp-CSI transitions

Transition	Description
T_t_ccsi-u-a	This is a deterministic transition that reflects the duration it takes the component to go from an unassigned state (Comp_CSI_unassigned) to an active state (Comp_CSI_active) on behalf of a CSI. The time **value** is specified by the ***saAmfCompCSISetCallbackTimeout*** attribute of the component. This transition has a priority that is relevant to the load of CSIs we want this component to handle. For instance if the SI has several CSIs of the same CS type and several components in the SU of the same component type can handle this CS type.Then we would like this load to be distributed equally, thus each component will have a higher priority for a certain set of CSIs it is expected to handle. This load can be obtained based on the algorithms we defined in Chapter 4 for the load distribution.
T_i_ccsi-u-a	This is an immediate transition that takes the component from the unassigned state (Comp_CSI_unassigned) to the active state (Comp_CSI_active) on behalf of the CSI.

T_t_ccsi-a-u	This is a deterministic transition that reflects the duration it takes the component to go from an active state (Comp_CSI_active) to an unassigned state (Comp_CSI_unassigned) on behalf of aCSI. The time **value** is specified by the ***saAmfCompCSIRmvCallbackTimeout*** attribute of the component. If the component is no-pre-instantiable then we use the ***saAmfCompTerminateTimeout*** as the transition delay
T_i_ccsi-a-u	This is an immediate transition that takes the component from the active state (Comp_CSI_active) to the unassigned state (Comp_CSI_unassigned) on behalf of the CSI.
T_t_ccsi-u-sb	This is a deterministic transition that reflects the duration it takes the component to go from an unassigned state (Comp_CSI_unassigned) to a standby state (Comp_CSI_standby) on behalf of a CSI. The time **value** is specified by the ***saAmfCompCSISetCallbackTimeout*** attribute of the component. This transition has a priority that is relevant to the load of CSIs we want this component to handle. For instance if

	the SI has several CSIs of the same CS type and several components in the SU of the same type can handle this CS type. Then we would like this load to be distributed equally, thus each component will have a higher priority for a certain set of CSIs it is expected to handle. Again the load can be calculated based on the load calculation algorithm define in Chapter 4.
T_i_ccsi-sb-u	This is an immediate transition that takes the component from the standby state (Comp_CSI_standby) to the unassigned state (Comp_CSI_unassigned) on behalf of the CSI.
T_t_ccsi-sb-u	This is a deterministic transition that reflects the duration it takes the component to go from a standby state (Comp_CSI_standby) to an unassigned state (Comp_CSI_unassigned) on behalf of a CSI. The time **value** is specified by the ***saAmfCompCSIRmvCallbackTimeout*** attribute of the component.
T_t_ccsi-sb-a	This is a deterministic transition that reflects the duration it takes the component to go from a standby state (Comp_CSI_standby) to anactivestate (Comp_CSI_active)

	on behalf of a CSI. The time **value** is specified by the extended attributes of the component.

Table 6-18 The Comp-CSI guard conditions

Guard	Description
G_ T_i_ccsi-a-u	It guards the immediate transition from the active state (Comp_CSI_active) to the unassigned state (Comp_CSI_unassigned). It evaluates to **true**: • The component is undergoing any recovery (excluding the component switching over). I.e. when there is no token in the Comp_instantiated state, or Component_switching_over state.
G_T_t_ccsi-a-u	It guards the deterministic transition from the active state (Comp_CSI_active) to the unassigned state (Comp_CSI_unassigned). The guard condition evaluates to **true** when: • The component is in a switching over state (Component_switching_over). ○ If the component is a proxied, then the proxy must be assigned the **proxy** CSI (i.e.

	there is a token in the Comp_CSI_active state of a proxy comp with respect to the proxy CSI).
G_ T_i_ccsi-u-a	It guards the immediate transition from the unassigned state (Comp_CSI_unassigned) to the active state (Comp_CSI_active). This guard is applicable for the non-pre-instantiable components. The guard condition evaluates to **true** when: • The parent SI is assigned active to the parent SU and the non-pre-instantiable component is instantiated, i.e. has a token in the instantiated state. o If the component is a proxied, then the proxy must be assigned the **proxy** CSI (i.e. there is a token in the Comp_CSI_active state of the proxy comp with respect to the proxy CSI).
G_ T_t_ccsi-u-a	It guards the deterministic transition from the unassigned state (Comp_CSI_unassigned) to the active state (Comp_CSI_active). The guard condition evaluates to

	true when all of the below conditions are true: • The parent SI is assigned active to the parent SU and the (pre-instantiable) component is instantiated. And the container (or a proxy) CSI is assigned active (if applicable). • If the component capability model allows this transition. I.e. the component has not consumed all of its capability in serving other CSIs and can still handle this assignment.
G_ T_t_ccsi-u-sb	It guards the deterministic transition from the unassigned state (Comp_CSI_unassigned) to the standby state (Comp_CSI_standby). The guard condition evaluates to **true** when all of the below conditions are true: • The parent SI is assigned standby to the parent SU and the (pre-instantiable) component is instantiated, and the container (or a proxy) CSI is assigned active. • If the component capability model allows this transition. I.e. the component has not consumed all of its capability in serving other CSIs and can still

	handle this assignment.
G_ T_i_ccsi-sb-u	It guards the immediate transition from the standby state (Comp_CSI_standby) to the unassigned state (Comp_CSI_unassigned). The guard condition evaluates to **true** when: • The component (or the container) is undergoing any recovery, i.e. it does not have a token in the Comp_instantiated or Comp_un-instantiated state (except the component switching over, in this case we use the deterministic transition).
G_ T_t_ccsi-sb-u	It guards the deterministic transition from the active state (Comp_CSI_active) to the unassigned state (Comp_CSI_unassigned). The guard condition evaluates to **true** when: • The component (or the container) is undergoing a component switching over recovery.
G_T_t_ccsi-sb-a	It guards the deterministic transition from the standby state (Comp_CSI_standby) to the active state (Comp_CSI_active). The guard condition evaluates to **true** when:

	• The parent SI is assigned active to the parent SU and the (pre-instantiable) component is instantiated. And the container (or a proxy) CSI is assigned active (if applicable). • If the component capability model allows this transition. I.e. the component has not consumed all of its capability in serving other CSIs and can still handle this assignment.

6.4.5.8 The SU-SI DSPN Template

For each SI that the SU can serve, it may be assigned the active, standby or unassigned state. The SU-SI DSPN template (shown in Figure 6-14) is used to capture the runtime assignment state for an SU on behalf of the SI. Note that in the analysis model, an instance of this template must be created for each SI within the SG. The template states, transitions, and guards are explained respectively in Table 6-19, Table 6-20, and Table 6-21.Table 6-18

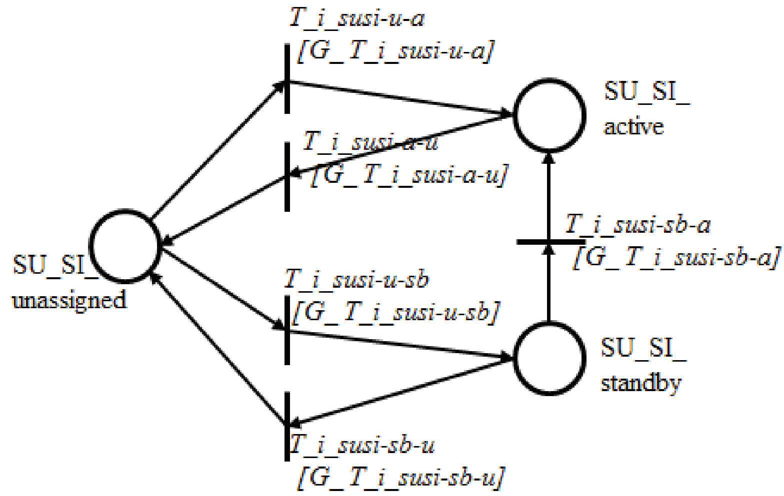


Figure 6-14 The SU-SI DSPN template

Table 6-19 The SU-SI states

State	Description
SU_SI_unassigned	This state means that the SU does not have an assignment for this particular SI.
SU_SI_active	The SU is assigned active for this SI
SU_SI_standby	The SU is assigned standby for this SI

Table 6-20 The SU-SI transitions

Transition	Description
T_i_susi-u-a	This is an immediate transition that takes the SU from the unassigned state (SU_SI_unassigned) to the active state

	(SU_SI_active) on behalf of the SI.
T_i_susi-a-u	This is an immediate transition that takes the SU from the active state (SU_SI_active) to the unassigned state (SU_SI_unassigned) on behalf of the SI.
T_i_susi-u-sb	This is an immediate transition that takes the SU from the unassigned state (SU_SI_unassigned) to the standby state (SU_SI_standby) on behalf of the SI.
T_i_susi-sb-u	This is an immediate transition that takes the SU from the standby state (SU_SI_standby) to the unassigned state (SU_SI_unassigned) on behalf of the SI.
T_i_susi-sb-a	This is an immediate transition that takes the SU from the standby state (SU_SI_standby) to the active state (SU_SI_active) on behalf of the SI.

Table 6-21 The SU-SI guard conditions

Guard	Description
G_ T_i_susi-u-a	It guards the immediate transition from the unassigned state (SU_SI_unassigned) to the active state (SU_SI_active). It evaluates to **true** when all the below

	conditions are true: • This is the most eligible SU for active assignment (i.e. this SU has the highest rank for this SI, or the next highest rank for an additional assignment). • The total number of active SUs for the SI is not satisfied. • If the SI is dependent then the sponsor SI(s) must also be assigned active. • The SU has enough remaining capacity (through its components) to handle all the CSIs of this SI. • The SU active SI assignments have not reached the ***saAmfSGMaxActiveSIsperSU*** attribute value of the parent SG. • If the cluster is on probation, i.e. it has a token in the Cluster_on_probation place, then the assignment should wait until all the needed pre-instantiable components are instantiated.
G_ T_i_susi-a-u	It guards the immediate transition from the active state (SU_SI_active) to the unassigned state

	(SU_SI_unassigned). It evaluates to **true** when: • **At least** one CSI of the SI is **not** assigned active to any of the SU's components
G_ T_i_susi-u-sb	It guards the immediate transition from the unassigned state (SU_SI_unassigned) to the standby state (SU_SI_standby). It evaluates to **true** when: • This is the most eligible SU for standby assignment (i.e. this SU has the highest rank for this SI, or the next highest rank for an additional assignment). • The total number of standby SUs for the SI is not satisfied. • The SU has enough remaining capacity (through its components) to handle all the CSIs of this SI. • The SU active SI assignments have not exceeded the ***saAmfSGMaxStandbySIsperSU*** attribute value of the parent SG. The total number of standby SUs for the SI is **not** satisfied and this is the most eligible SU for standby assignment.

G_ T_i_susi-sb-u	It guards the immediate transition from the standby state (SU_SI_standby) to the unassigned state (SU_SI_unassigned). It evaluates to **true** when: • **At least** one CSI of the SI is **not** assigned standby to any of the SU's components. I.e. there is no token in any the Comp_CSI_standby states relevant to this CSI of the SI.
G_ T_i_susi-sb-a	It guards the deterministic transition from the standby state (SU_SI_standby) to the active state (SU_SI_active). It evaluates to **true** when: • This is the most eligible standby SU for active assignment (i.e. this SU has the highest standby rank for this SI, or the next highest rank for an additional assignment). • The total number of active SUs for the SI is not satisfied. • If the SI is dependent then the sponsor SI(s) must also be assigned active. • The SU has enough remaining capacity (through its components) to handle all the CSIs of this SI.

	• The SU active SI assignments have not exceeded the ***saAmfSGMaxActiveSIsperSU*** attribute value of the parent SG.

6.4.5.9 The SI DSPN Template

The SI DSPN template (shown in Figure 6-15) is the one used to eventually determine the service outage. In other words, the goal of the analysis model is to identify when the SI is in a provided state, and thereafter quantify the sojourn time is this state. By performing this, we will be able to quantify the SI availability, i.e. the service availability. It should be noted here that we evaluate service availability by different criteria than AMF, for instance when the SI is failing over, AMF still considers it to be served, while in our analysis when any CSI of the SI is not assigned active for any reason or period, then we consider this as a service outage. The template states, transitions, and guards are explained respectively in Table 6-22, Table 6-23, and Table 6-24.

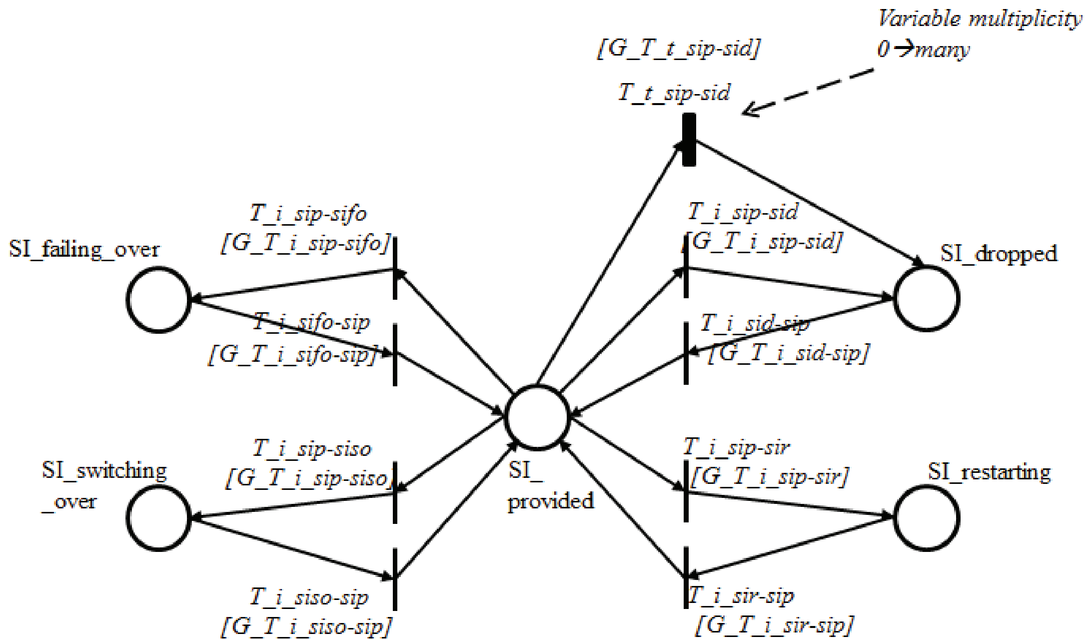


Figure 6-15 The SI DSPN template

Table 6-22 The SI states

State	Description
SI_provided	This state means that the SI is provided, i.e. there is at least one SU assigned active for this SI, and all the CSIs of the SI are assigned active to at least one component, and the component is healthy and instantiated.
SI_failing_over	The SI transitions to this state when at least one of the components assigned active to at least one of its CSIs is in a failing over state, or the node hosting those components is failing over or failing fast.

SI_switching_over	The SI is switching over when the components serving its CSIs are switching over, or the node hosting those components is switching over.
SI_restarting	The SI is restarting when it remains assigned to the same SU but at least one component serving at least one of its CSIs is in a restarting state. An SI in this state must not be reassigned to other SUs.
SI_dropped	The SI is dropped when there is no SU that can be assigned the active state on its behalf, or during a cluster or an application restart.

Table 6-23 The SI transitions

Transition	Description
T_i_sip-sifo	This is an immediate transition that takes the SI from the provided state (SI_provided) to the failing over state (SI_failing_over).
T_i_sifo-sip	This is an immediate transition that takes the SI from the failing over state (SI_failing_over) to the provided state (SI_provided).
T_i_sip-siso	This is an immediate transition that takes the SI from the

	provided state (SI_provided) to the switching over state (SI_switching_over).
T_i_siso-sip	This is an immediate transition that takes the SI from the switching over state (SI_switching_over) to the provided state (SI_provided).
T_i_sir-sip	This is an immediate transition that takes the SI from the restarting state (SI_restarting) to the provided state (SI_provided).
T_i_sip-sir	This is an immediate transition that takes the SI from the provided state (SI_provided) to the restarting state (SI_restarting).
T_i_sid-sip	This is an immediate transition that takes the SI from the dropped state (SI_dropped) to the provided state (SI_provided).
T_i_sip-sid	This is an immediate transition that takes the SI from the provided state (SI_provided) to the dropped state (SI_dropped).
T_t_sip-sid	This is a deterministictransition that takes the SI from the provided state (SI_provided) to the dropped state. It is only used for dependent SIs. This transition is used when

	the sponsoring SI(s) for this (dependent) SI is not provided. The time associated with this transition is the ***saAmfToleranceTime*** attribute of the ***SaAmfSIDependency*** class. This transition has a variable multiplicity, i.e. for each sponsoring SI for the dependent SI; we create an additional transition with the relevant timing and guard.

Table 6-24 The SI conditions

Guard	Description
G_T_i_sip-sifo	It guards the immediate transition from the provided state (SI_provided) to the failing over state (SI_failing_over). It evaluates to true when: • **Any** of the components providing any of the CSIs of the SI (in an active state) are failing over, or failing fast. I.e. they have a token in the Comp_failing_over or Comp_failing_fast states.
G_T_i_sifo-sip	It guards the immediate transition from the failing over state (SI_failing_over) to the provided state (SI_provided). It evaluates to **true** when:

	• All the CSIs of the SIs are provided (assigned active). I.e. they have a token in a Comp_CSI_active state.
G_T_i_sip-siso	It guards the immediate transition from the provided state (SI_provided) to the switching over state (SI_switching_over). It evaluates to true when: • True when **all** of the components providing any of the CSIs of the SI (in an active state) are switching over. I.e. they have a token in the Comp_switching_over or Comp_node_switching_over state.
G_T_i_siso-sip	It guards the immediate transition from the switching over state (SI_switching_over) to the provided state (SI_provided). It evaluates to **true** when: • All the CSIs of the SIs are provided (assigned active). I.e. they have a token in a Comp_CSI_active state.
G_T_i_sir-sip	It guards the immediate transition from the restarting state (SI_restarting) to the provided state (SI_provided). It evaluates to **true** when:

	• All the CSIs of the SIs are provided (assigned active). I.e. they have a token in a Comp_CSI_active state.
G_T_i_sip-sir	It guards the immediate transition from the provided state (SI_provided)to the restarting state (SI_restarting). It evaluates to **true** when: • Any of the components providing any of the CSIs of the SI (in an active state) are restarting (I.e. they have a token in the Comp_restarting state) while the application is **not** restarting
G_T_i_sid-sip	It guards the immediate transition from the dropped state (SI_dropped) to the provided state (SI_provided). It evaluates to **true** when: • All the CSIs of the SIs are provided (assigned active). I.e. they have a token in a Comp_CSI_active state.
G_T_i_sip-sid	It guards the immediate transition from the provided state (SI_provided) to the dropped state (SI_dropped). It evaluates to true when: • The parent application is restarting (i.e. it has a

	token in the Application_restarting state) or the cluster is resetting (i.e. it has a token in the Cluster_resetting state).
G_T_t_sip-sid	It guards the immediate transition from the provided state (SI_provided)to the dropped state. It evaluates to **true** when: • The sponsoring SI is **not** is the SI_ provided state.

6.4.6 Availability Analysis Discussion

In this section we discuss four main issues: measuring availability, the DSPN templates, the mapping, and automating the mapping process.

- Measuring availability: our availability measuring criteria is focused on the SI, whenever any of the CSIs of the SIs are not assigned active for any reason or period of time we consider this to be an outage. This differs from the AMF definition of service outage, where an SI is dropped when no active assignment can be made. However when an SI is in the phase of failing over, i.e. the active assignment is shifting, from an AMF perspective it is not considered an outage but in our analysis it is, since we are interested in the actual service outage. Note that it can be argued that even this is not the user perceived service outage. I.e. the service user might experience a longer outage depending on the delay it takes the service to be delivered after it is restored. In certain related works e.g. [10], the

user behavior is also incorporated in the availability analysis. However this is not the objective of our work. The middleware itself is a distributed software that is susceptible to failure as well, however we do not consider the middleware failure in our analysis. If we were to include the middleware failures on certain nodes, we would have considered it a way that is somehow similar to the one where we consider the node failure.

- The DSPN templates: an important issue here is to make sure that the DSPN templates do capture the runtime behavior of the entities and AMF. Our design of the templates is based on our analysis of the AMF specifications. The templates have been validated by the domain expert to ensure compliancy with the specifications.

- The mapping: our mapping consists of instantiating the templates and annotating the guard conditions and the transitions delays with the proper values (as well as the proper naming for the places, transitions and guards). The question here is how to make sure that the mapping does really capture the information specified in the configuration? For this purpose we defined the templates to remain aligned with the AMF model, whereas each template describes a different entity. When the DSPN model is created, it can be structurally traced back to the AMF configuration structure. The relevant attributes that are directly mapped from the configuration (e.g. the ***saAmfCompInstanteTimeout***). Other configuration attributes such as the component category can be inferred from its guard conditions (e.g. if another component affect its lifecycle then it is a container for this component). In

short the reverse mapping of the DSPN instances in to the AMF configuration is a way of validating to that the mapping is accurate.

- Automating the mapping to the DSPN model: mapping an AMF configuration instance to the DSPN template instances, is not an easy task, especially for large AMF configurations. For this purpose, automation would render the mapping much simpler. Nevertheless the issue here is that there is no standard Petri Net syntax that we can map to, and that can run on any Petri Net tool. The authors in [65] propose the Petri Net Markup Language (PNML) which is an XML-based language for describing Petri Nets. Some tools such as Renew [66] can interpret this format. However, and to the best of our knowledge, none of the tools that can interpret this format support DSPNs or allow the definition of complex guards. A promising tool that we used in our analysis is TimeNet [67]. This tool has the capability of simulating our DSPNs. However, and like most of the well planted Petri Net tools, they predate PNML and thus their Petri Net representation is not based on this language. A major advantage of TimeNet is that it provides a XML schema based on which the Petri Net model is defined. This is extremely useful for a model driven approach to automate the transformation. On one hand we would have the AMF configurations (that are based on a UML model) and on the other hand we would have the Petri Nets that are based on an XML schema. In this dissertation we did not tackle the problem of automating the mapping, and we leave it for future work.

7 Tooling Framework and Case Study

In this chapter we discuss how we implemented our approach for the configuration generation and the availability analysis. We present the tool which implements our dependency specification, configuration generation and load balancing algorithms. And then we report on our experience with the Petri net tool we used to solve our DSPN model. To demonstrate the effectiveness of our approach, we generate the configuration of an online streaming server which allows users to access (watch and listen to etc.) media files either in on-demand or in a broadcast manner. In this chapter, we first introduce our prototype tool. Then, we use it for the case study.

7.1 Tooling Framework

Our configuration generation method relies heavily on various UML models that are used to describe the AMF, ETF, and CR concepts. We defined these models as Ecore models [68]using the Eclipse Modeling Framework (EMF) [69] as our modeling infrastructure. Figure 7-1 illustrates the data flow within our tool, which consists of several modules as follows:

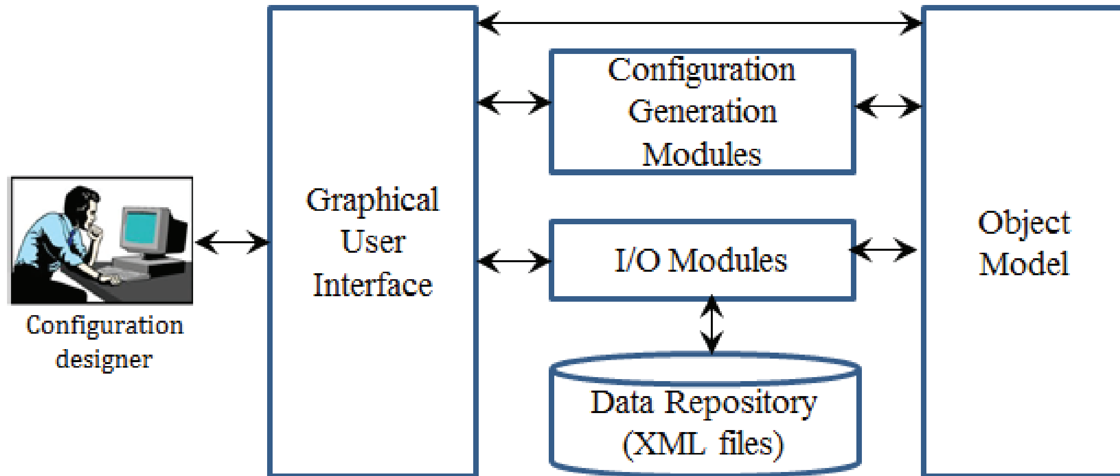


Figure 7-1 The data flow diagram in the configuration generation tool

The Graphical User Interface (GUI): the configuration designer specifies the configuration requirements using the GUI of the tool. It is also through the GUI that the configuration is graphically displayed in a tree like structured where each tree node represents a configuration element. The GUI consists of two main parts. The first part is provided by Eclipse Modeling Tools (Indigo) [70]when we run our packages as an Eclipse application, and the second part is a set of frames that we defined using Java Swing [71] to input the various templates defined in the CR. The two parts are integrated in such a way that from the user perspective they constitute a single GUI. Various aspects of the GUI will be shown in what follows in the chapter. Through the GUI the designer can (1) select the ETF XML files (2) define the templates, (3) specify the dependencies (4) run the configuration generation (5) view, load, edit and save the configuration.

The I/O Modules: these modules are responsible for fetching the designated XML files and parsing them into the proper Object Models. They are also responsible for binding the data in the designated ETF file with the GUI. And therefore presenting to the designer

the various options allowed in ETF (e.g. the various service types that an SI template can specify). The same modules are thereafter used to store the AMF configuration as an XML file based on the standardized IMM XML schema [72].

The Configuration Generation Modules: these modules constitute the bulk on our tool, where we implemented our algorithms for type selection, dependency analysis, configuration generation etc. These modules can instantiate certain objects in the GUI, such as the one representing the dependency specification frame. In addition, these modules will extend the object model created by the I/O modules to include the AMF configuration that is generated.

The Object Model: this model is composed of various instances of the different UML class diagrams discussed earlier (namely the AMF, ETF, and CR).

7.2 The Media Streaming Case Study

In order to demonstrate our approach, we present a media streaming case study. This case study is based on modifying an open source application (namely VLC– the Video LAN Client [73]) so that it can be managed by the OpenSAF middleware which is an open source implementation of the SAForum middleware. The objective of this work was to demonstrate how legacy applications can be tuned to interface with the SAForum middleware and thus rendering the service they provide highly available. VLC is a media streaming server and a media player client at the same time. In this case study the streaming server was modified while the client remained intact. The server was then deployed on a cluster managed by OpenSAF. By regularly checkpointing the stream position on the VLC server, the failover to a standby was performed in a swift manner

and the whole recovery was performed in under a second, and thus the failure was scarcely noticeable by the end user. To illustrate our availability analysis approach we will use the VLC case study. However for the sake of demonstrating our configuration generation process, we extend this case study to include other components that we did not actually incorporate with VLC. The goal is to add more components to the case study with various types of dependencies to better clarify the experience of generating configurations.

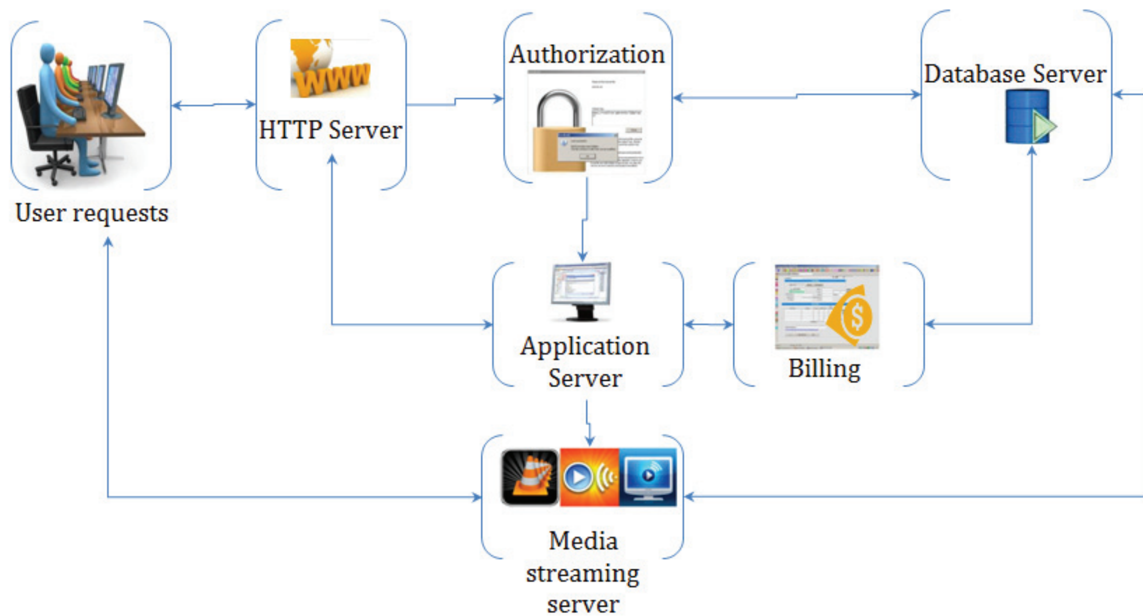


Figure 7-2 Overview of the media streaming example workflow

The workflow in the media streaming application is as follows, the users can request a media, e.g. video (in a pay-per-view fashion) through a web site. On the server side an HTTP server will process the requests. The static ones, i.e. the ones simply browsing the site for existing videos (or other media) will be handled solely by the HTTP server. However the dynamic request that involve logging-in and selecting on-demand movies

will be first forwarded to an Authorization service that will verify the credentials of the user. The user credentials are stored in the Database from which the Authorization service has to fetch them. If the authentication is successful, the request is forwarded to an application server that will dynamically create the customized content of the requested page (e.g. the one showing the account of the user, with the remaining credit etc.). When the user requests a movie for instance, the Application server will consult with the Billing service to verify that this is a valid request. Again, the Billing service will fetch and modify the account information from the Database. Finally, when the request is verified, the Application server will instruct the streaming server to stream the selected media to the user's address. The streaming server will then fetch the media file from the database and stream it in the requested format. The user will then have the right to pause, stop, suspend and resume the stream by interacting with the streaming server until the end of the stream. The streaming server can also broadcast streams to all the listening users. Such service will be used for announcements and advertisement and the users will not be charged for it.

Figure 7-3 illustrates the various dependencies that exist in the streaming application. The HTTP server needs the Authorization and the Application server in order to handle the various types of requests. The Application server needs the Billing software in order to display to the user its remaining credit. The Billing software, the Authentication software and the Streaming server need the database (and its DBMS) in order to function. Finally in our example, the database is assumed to be legacy software that does not interface with the SAForum middleware that is going to manage the availability of the service.

Therefore a proxy component is needed to mediate the interactions between the middleware and the database.

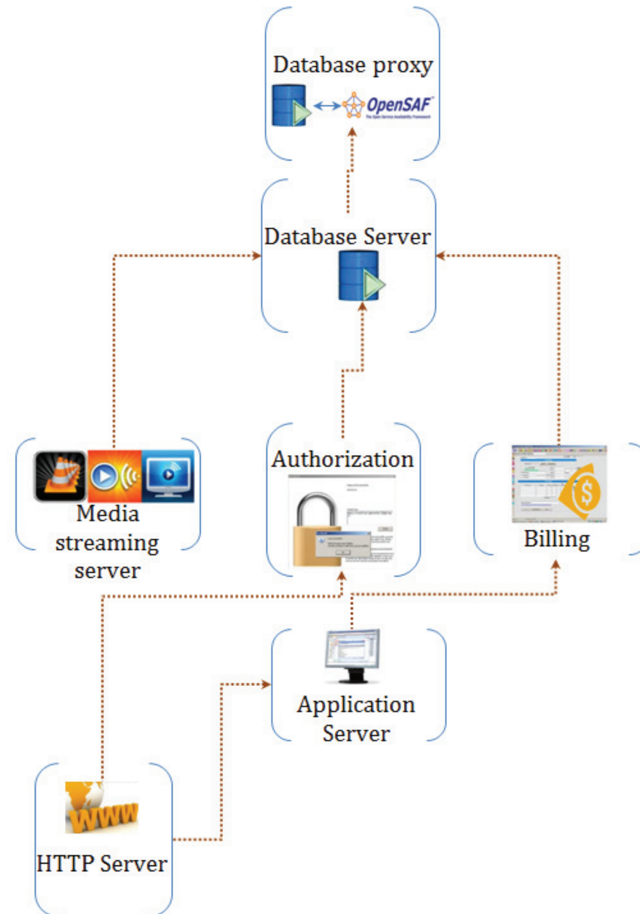


Figure 7-3 Dependency graph in the media streaming example

In our example, the configuration designer is tasked with designing a configuration that allows the streaming to be performed in a highly available manner. The designer will have access to the ETF file(s) that describe the content of a software repository, from which the software can be deployed in the cluster to provide the needed software.

An overview of the ETF file describing the available software is shown in Figure 7-4. We can also see in this figure that the database proxy component type (DB proxy-CT) can

only proxy the Oracle-CT while for MySQL-CT we do not have any proxy, and therefore it is considered non-proxied-non-SA-aware component type. For the VLC component types we differentiate between the SA-aware VLC, the version that was modified to become SA-aware, and the Non-SA-aware one. The component type capability model is shown in the association relating the component type with the CS type it can provide. For instance the Billing-CT can support either five CSIs of the Billing-CS type in an active state, or five standby ones.

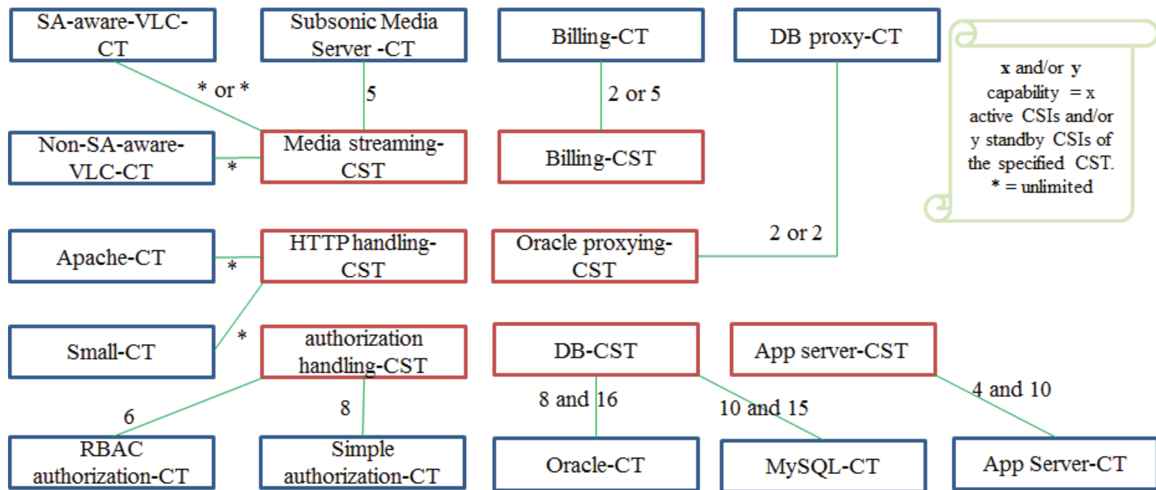


Figure 7-4 Overview of the ETF content (Component type and CS type) for the example application

7.3 Configuration Generation Example

The configuration requirements for the configuration will be specified by the configuration designer using the concepts of the CR model. Figure 7-5 shows the various input frames that can be used by the designer to input the templates.

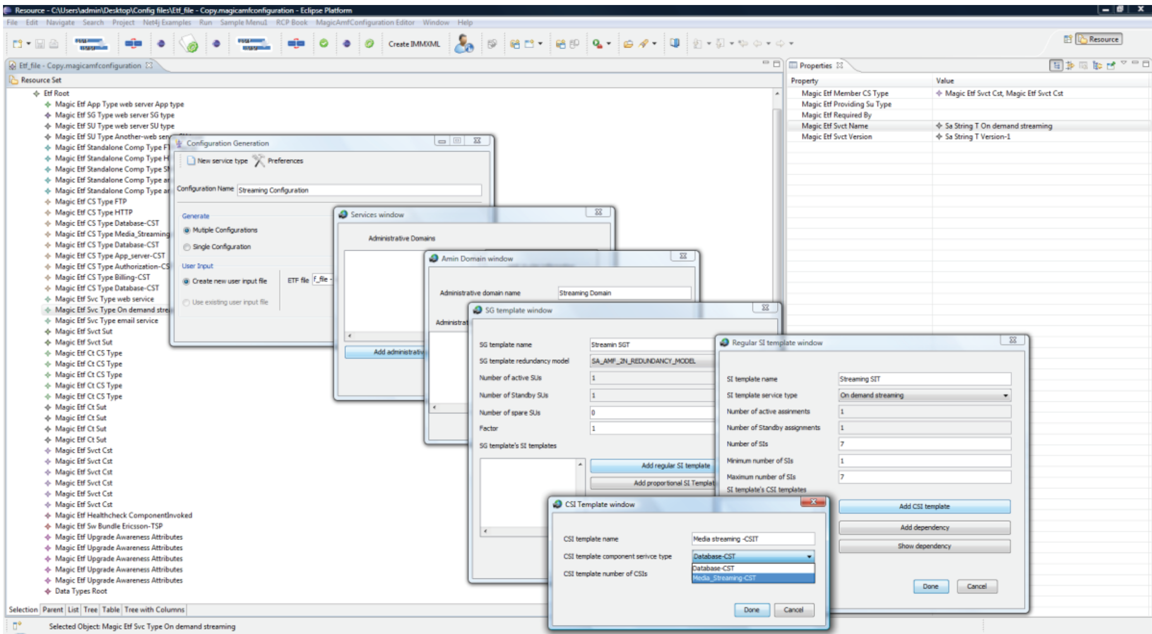


Figure 7-5 A snapshot of the input frames

Figure 7-6 shows a snapshot for the SG, SI, and CSI template frames. The tool will automatically guide the designer through the input, for instance in the SG template below, we can see that when the redundancy model is 2N the number of active/standby SU is locked to ‘1’, which is a requirement of the redundancy model. The same is carried on to the SI template where the number of active/standby assignment is locked to ‘1’ as well. On the CSI template of the same template we can see that the designer has only the option of choosing the CS type of the service type specified for the parent SI template. This is just to show how certain constraints are embedded in the tool to ensure compliancy and consistency.

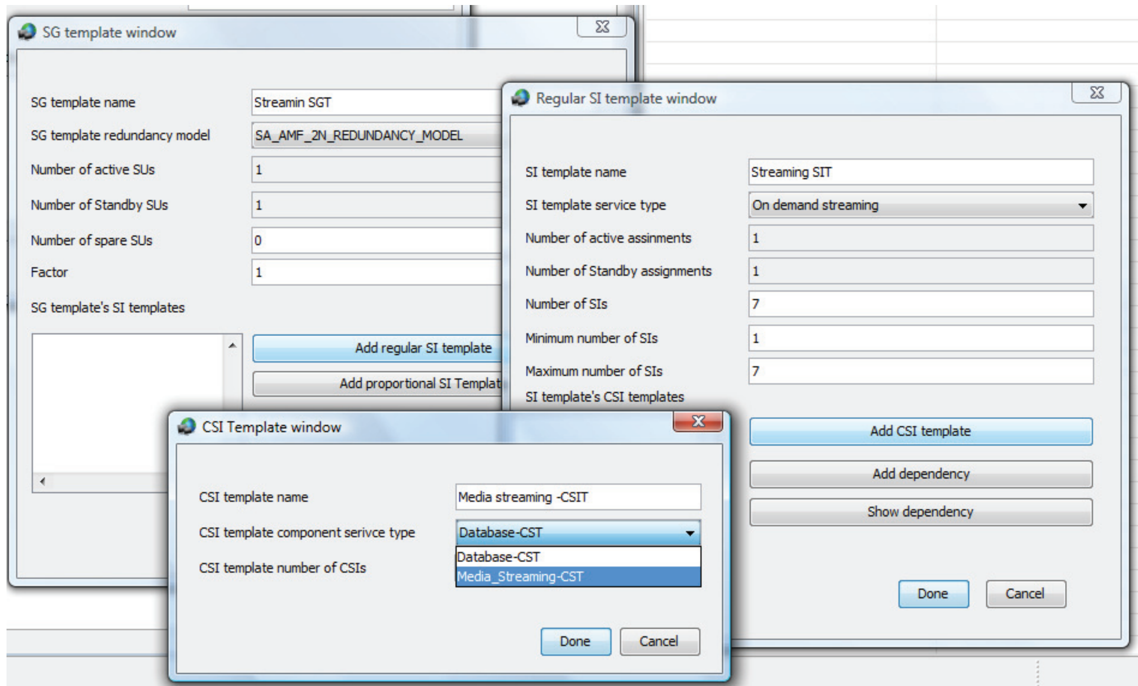


Figure 7-6 A snapshot of the SG, SI, and CSI template

In Figure 7-6 we can see that the service type specified for the SI template is ”On demand streaming” while the ETF file does not specify such a service type (as shown in Figure 7-4). The tool allows the designer to compose service types out of orphan CS types (in our case study all of the CS types are orphan). Figure 7-7 illustrates how a new service type can be created by aggregating a set of orphan CS types. The I/O module of the tool will process the ETF content and bind it with the GUI. E.g. in Figure 7-7 we see how the available orphan CS types are shown to the designer through the value options of the “Component service type” combo box. In short since the definition of the SI template requires the existence of service types, and ETF may not necessarily specify anything besides the component types and the CS types, we give the designer the means to create his/her own service types.

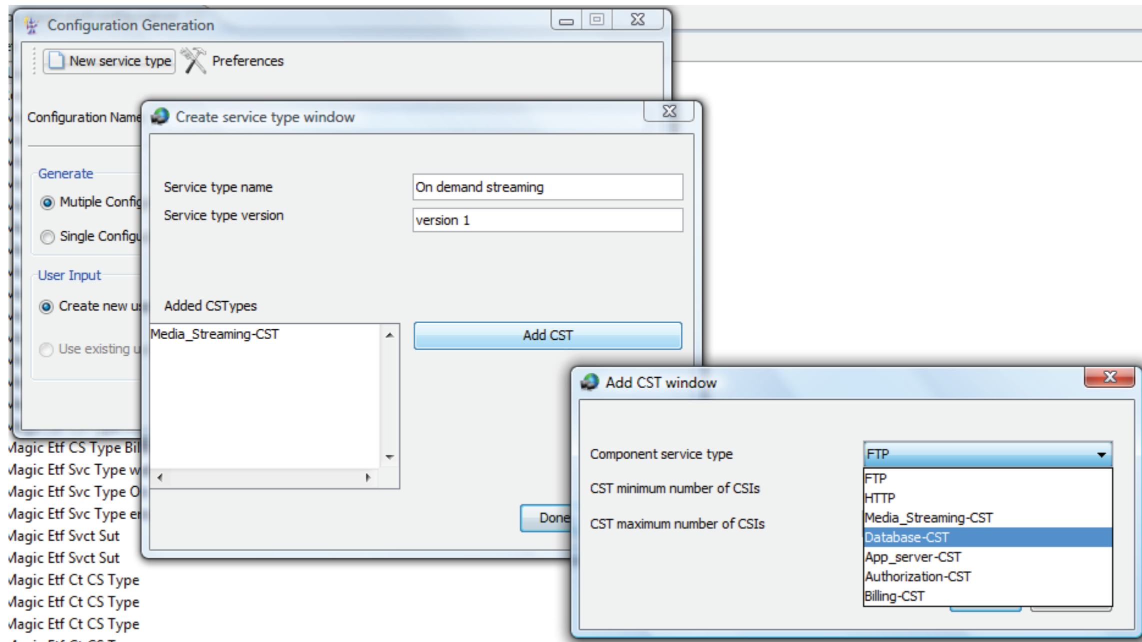


Figure 7-7 A snapshot of creating a service type

The complete user input is as follows. We have a cluster of 6 nodes. On which the designer specified one administrative domain with the following SG templates to be deployed as shown in Table 7-1.

Table 7-1 List of values of certain attributes of the SGTemplates specified for the media streaming application

Attribute \ SGTemplate	Streaming SGT	Web SGT	Security SGT	Billing SGT
magicCrSgTempRedundancyModel	2N	N way active	N+M	N-Way
magicCrSgTempNumberofActiveSus	1	5	2	2
magicCrSgTempNumberofStdbSus	1	0	1	0

Each of the above specified templates groups one SI template as shown below in Table 7-2. Note that the streaming SI template and the Web SI template will each group two CSI templates.

Table 7-2 List of the values of certain attributes of SITemplates and CSITemplates of the media streaming application

Attribute / **SI Template**	Streaming SIT		Web SIT		Authentication SIT	Billing SIT
magicCrSiTempSvcType	On demand streaming		Webservice		Access control	Billing
magicCrSiTempNumberofActiveAssignments	1		3		1	1
magicCrSiTempNumberofStdbAssignment	1		0		1	1
magicCrRegSiTempNumberofSis	7		18		5	4
CSI Template / **Attribute**	Database-CSIT	Media streaming-CSIT	HTTP-CSIT	App server-CSIT	Authentication-CSIT	Billing-CSIT
magicCrCsiTempCsType	Database-CS type	Media streaming-CS type	HTTP-CS type	App server-CS type	Authentication-CS type	Billing-CS type
magicCrCsiTempNumberofCsis	1	1	5	5	2	3

In our example, we have several dependencies that the designer may not be aware of. Our tool incorporates the dependency detection and specification method (explained in Chapter 4) that can detect the dependencies and allow the designer to specify the needed SI and CSI templates and the dependency relation between the sponsor and the dependent (according to Figure 4-4). This relation is specified through SQL queries entered by the designer. The "help" button on the "Dependency window" will open another document

explaining the process of dependency specification, with specific keywords (reflecting predefined database table names) to be used in the SQL queries (as shown in Figure 7-8). For instance the Database-CT in our example is proxied, and needs a proxy component, therefore a proxy CSI and potentially SI needs to be defined by the designer, and the dependency relation set between the proxy and proxied. Note that all the dependencies shown in Figure 7-3 must be specified using this approach.

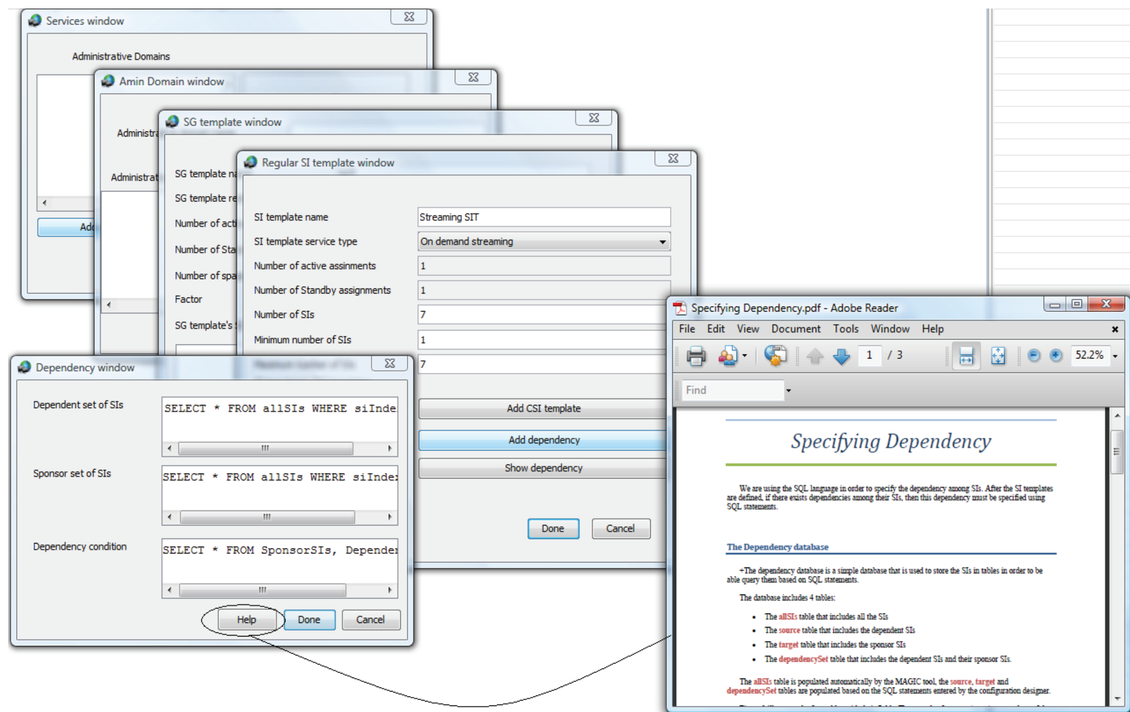


Figure 7-8 A snapshot of the dependency specification

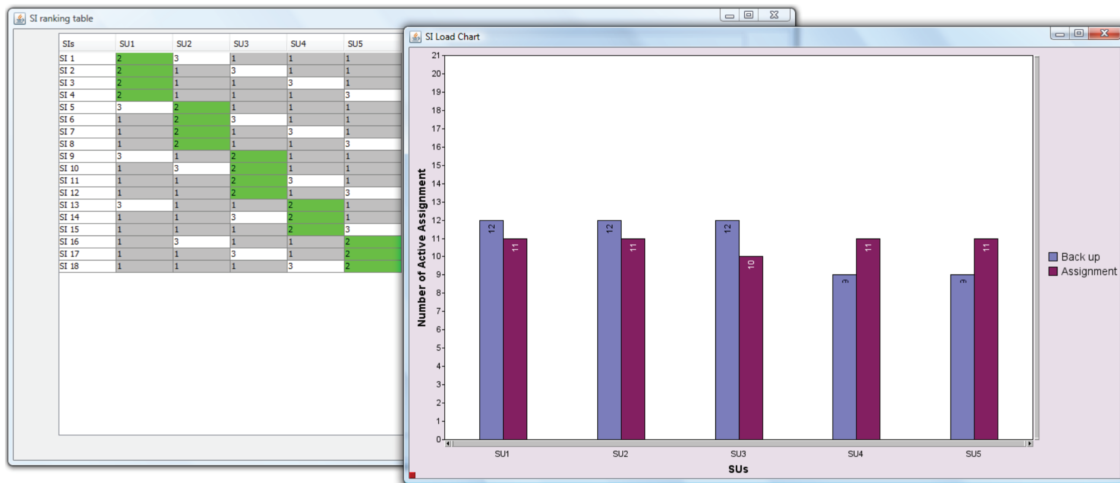
For the sake of simplicity, we kept the input simple e.g. by not specifying a minimum and maximum number of SIs per SG and not overlapping CS types of the different CSI templates belonging to different SI templates of the same SG. This is in order not to burden the reader with additional calculations that the tool will automatically perform.

However we will show how certain calculations are performed, to clarify how certain algorithm defined in Chapter 4 and 5 are used.

When the input is ready, the AMF configuration can be created. It is done by first calculating the expected load of SIs per SU and respectively CSIs per component, then the type selection process will take place. And finally the AMF types and entities will be created. For instance the load of SIs per SU for the Web SIT (according to Algorithm 3) is the ceiling of (18 SIs × 3 active assignments) ÷ 5 SUs = ceiling 10.8 = 11 SIs per SU. The App server CSIT for instance specifies 4 CSIs per SI, i.e. each SU must handle 5×11= 55 CSIs of the App server-CS type. The only eligible component type to be selected for this CS type is the App server-CT. A component of this type can handle at most 4 CSIs of the App server CS type. Hence the needed number of components is the ceiling of 55 ÷ 4 = ceiling 13.75 = 14 components per SU are needed to support the expected load of CSIs. Note that in Algorithm 3, we have a variable (`isFaultTolerant`) that specifies whether we want to include fault tolerance in the equation,If that is the case, then the load of SIs per SU will be calculated with the assumption of having a missing SU in the SG. As a result of using the fault tolerance option[18] the load of SIs per SU will increase to 14, which will increase the number of needed components of the App

[18]This feature can be set in the “preferences” of our tool.

server-CT from 14 to 18 components per SU. The same type of calculations, type selection and entity creation will be carried out for the rest of the templates. We remain with the same SI template to define the ranks of the 5 SUs defined in the SG to support the 18 SIs of the template where each SI has 3 active assignments. A snapshot of the generated rankings for the list of SIs is shown in Figure 7-9. These rankings are derived by applying the method specified in Section 5.5.2.1. An accompanying load chart is generated by the tool simply to visualize the load distribution to the configuration designer. We can see in this chart that (according to the generated rankings) each SU will have either 10 or 11 active assignments, while each SU will have either 9 or 12 backup assignments.



SI ranking table

SIs	SU1	SU2	SU3	SU4	SU5
SI 1	2	3	1	1	1
SI 2	2	1	3	1	1
SI 3	2	1	1	3	1
SI 4	2	1	1	1	3
SI 5	3	2	1	1	1
SI 6	1	2	3	1	1
SI 7	1	2	1	3	1
SI 8	1	2	1	1	3
SI 9	3	1	2	1	1
SI 10	1	3	2	1	1
SI 11	1	1	2	3	1
SI 12	1	1	2	1	3
SI 13	3	1	1	2	1
SI 14	1	1	3	2	1
SI 15	1	1	1	2	3
SI 16	1	3	1	1	2
SI 17	1	1	3	1	2
SI 18	1	1	1	3	2

Figure 7-9 A snapshot of the rankings generated for the SIs of the WebSIT

Finally the configuration generated is shown is Figure 7-10 where we can see on the left hand side panel the entities (surrounded by a rectangle) and the types in a tree like structure. For instance we can see that we have a created application with 4 child SGs corresponding to the templates (plus one SG for proxying the database). Each SG will

have the corresponding child SUs and similarly the SUs will have the child components. On the right hand side panel we see a table showing the attributes of a selected entity and their values. In this snapshot we see the attribute values of a selected SG.

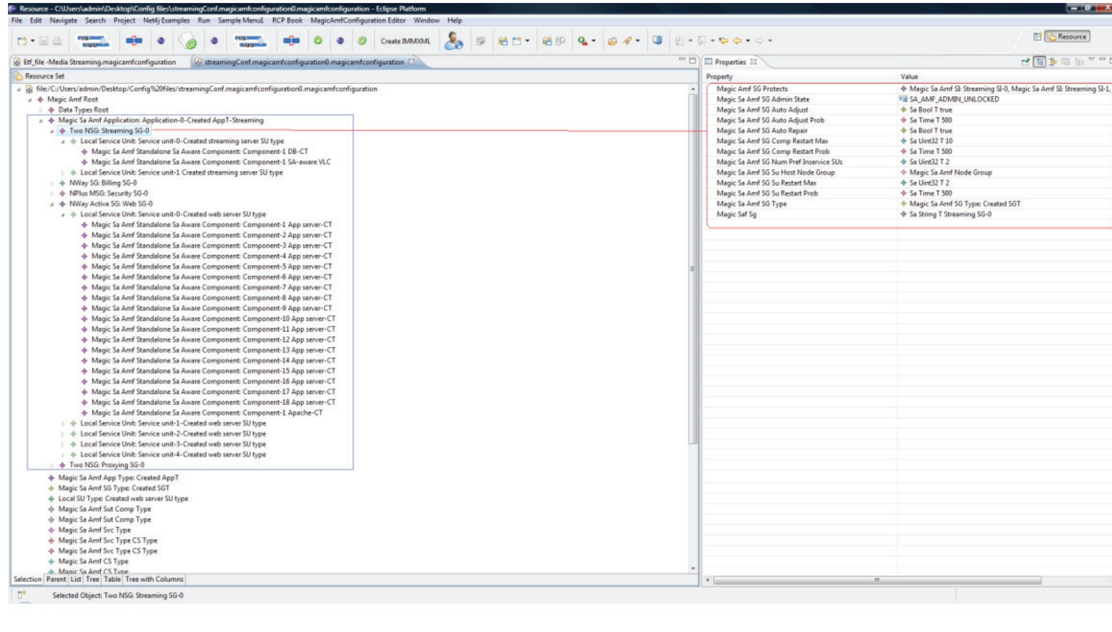


Figure 7-10 A snapshot of the configuration generated

7.4 Availability Analysis Example

In the availability analysis we examine the availability of the streaming SIs (i.e. the seven SIs specified in the "streaming SIT" template). For the sake of simplicity, we consider only the configuration elements that affect the availability of the streaming SIs. Figure 7-11 illustrates the SUs of two SGs of the application that will be considered in the analysis. The Streaming SG has a 2N redundancy model and is protecting seven SIs, while the Proxy SG has the same redundancy model but it protects one SI. We included those elements in the analysis because of the dependency that exists between the streaming server (VLC) and the database and the database with the proxy components. In

the context of our configuration, the failure of other components will not affect the availability of the streaming SIs. Figure 7-11 also shows the failures associated with the relevant entities and their recommended recoveries. The analysis steps are as follows:

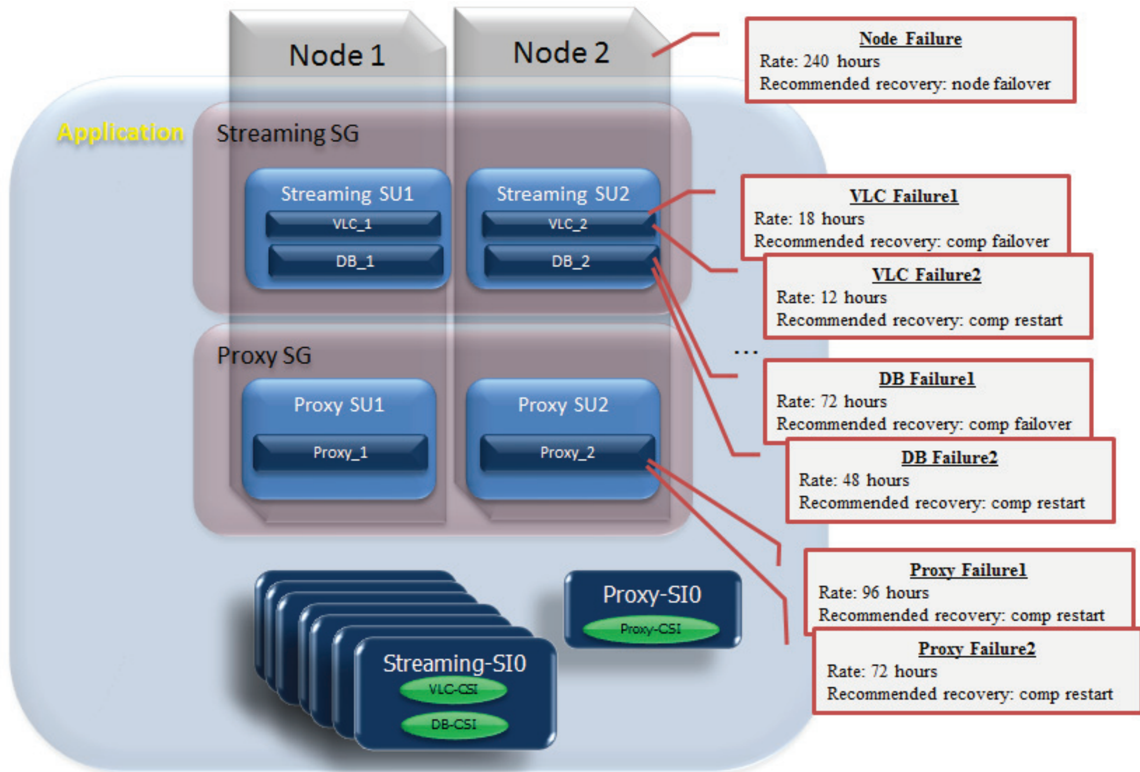


Figure 7-11 The configuration portion under analysis

- The first step is to extend the configuration with the needed information, the failure rates with their respective recommended recoveries, as well as other information such as the node startup time.

- The second step in the analysis is to perform the actual recovery analysis (presented in Section 6.4.2). Based on the values of the disable restart and the SU failover parameters shown in Table 7-3, the restart of the VLC components will

be altered by the actual recovery algorithms (Figure 6-5) to a component failover, and since the SU is configured to fail as a unit it will further be altered to an SU failover. The other recommended recoveries are executed without alteration.

Table 7-3 Parameter values for the configuration under analysis

Parameter Value	Description
0.542^{-1} sec^{-1}	VLC instantiation rate (saAmfCompInstantiateTimeout attribute from the SaAmfComp class)
0.027^{-1} sec^{-1}	VLC cleanup rate (saAmfCompCleanupTimeout attribute from the SaAmfComp class)
0.080^{-1} sec^{-1}	VLC assignment rate (saAmfCompCSIsetCallbackTimeout attribute from the SaAmfComp class)
0.030^{-1} sec^{-1}	VLC switching from standby to active assignment rate
64800^{-1} sec^{-1}	VLC failover rate (MTTF = 18 hours)
43200^{-1} sec^{-1}	VLC restart rate (MTTF = 12 hours)
Disable restart =	The VLC restart is disabled

true	(saAmfCompDisableRestart attribute from the SaAmfComp class)
Parent SU failover = true	Whenever one component fails over, the whole SU must fail over (saAmfSUFailover attribute from the SaAmfSU class)
2^{-1} sec^{-1}	DB instantiation rate (saAmfCompInstantiateTimeout attribute from the SaAmfComp class)
1.5^{-1} sec^{-1}	DB cleanup rate (saAmfCompCleanupTimeout attribute from the SaAmfComp class)
0.5^{-1} sec^{-1}	DB assignment rate (saAmfCompCSIsetCallbackTimeout attribute from the SaAmfComp class)
2^{-1} sec^{-1}	DB switching from standby to active assignment rate
259200^{-1} sec^{-1}	DB failover rate (MTTF = 72 hours)
172800^{-1} sec^{-1}	DB restart rate (MTTF = 48 hours)
Disable restart =	The database restart is enabled

false	(saAmfCompDisableRestart attribute from the SaAmfComp class)
0.7^{-1} sec^{-1}	Proxy instantiation rate (saAmfCompInstantiateTimeout attribute from the SaAmfComp class)
0.8^{-1} sec^{-1}	Proxy cleanup rate (saAmfCompCleanupTimeout attribute from the SaAmfComp class)
0.6^{-1} sec^{-1}	Proxy assignment rate (saAmfCompCSIsetCallbackTimeout attribute from the SaAmfComp class)
0.9^{-1} sec^{-1}	Proxy switching from standby to active assignment rate
345600^{-1} sec^{-1}	Proxy failover rate (MTTF = 96 hours)
259200^{-1} sec^{-1}	Proxy restart rate (MTTF = 72 hours)
Disable restart = false	The proxy restart is enabled (saAmfCompDisableRestart attribute from the SaAmfComp class)

120^{-1} sec^{-1}	Node reboot rate (time needed to reboot a faulty node)
864000^{-1} sec^{-1}	Node failover rate (MTTF = 240 hours)

- The next step is to build the DSPN model corresponding to the configuration shown in Figure 7-11. For each of the entities shown (except the SGs) a DSPN instance is created according to the corresponding DSPN template, For instance for each node, SU, component, SI, CSI the corresponding template is instantiated. For each CSI that a component can provide a Component-CSI template is instantiated. The same apply for each SU and the SI it can support within the SG. Thereafter the template instances are annotated with the proper values (in terms of transition rate values, and guard conditions). The rates are extracted from Table 7-3. The cells with the bold text indicate that those values are do not belong to the attributes of the standard AMF configuration, but the ones that were added (while extending the configuration in the first step). These values can be derived by benchmarking for instance. In the context of our analysis for this case study we examine the availability of a single streaming SI, however the availability of the other streaming SIs is identical, since they are assigned to the same SU and components at any point in time (due to the 2N redundancy model and not having several components of the same type in the same SU).

- The final step is to obtain the availability of the streaming SI by solving the DSPN shown in Figure 7-12, the measure of interest was the probability of having

a token in the Streaming_SI1_provided place at steady state. By using the failure rates shown in Table 7-3, to solve the DSPN, we found that the SI availability measures to 99.999280. I.e. we can achieve the five nines of availability with the given configuration.

Further experimentation: In order to examine the impact of the node failures and the proxy failures on the availability of the streaming SIs, we have multiplied at one experiment the nodes failure rates by a doubling factor ($2^{\lambda-1}$, where λ ranges from 1 to 5). And in the second experiment we used the same node failure rate specified in Table 7-3, but we multiplied the proxy components failure rates by the doubling factor.

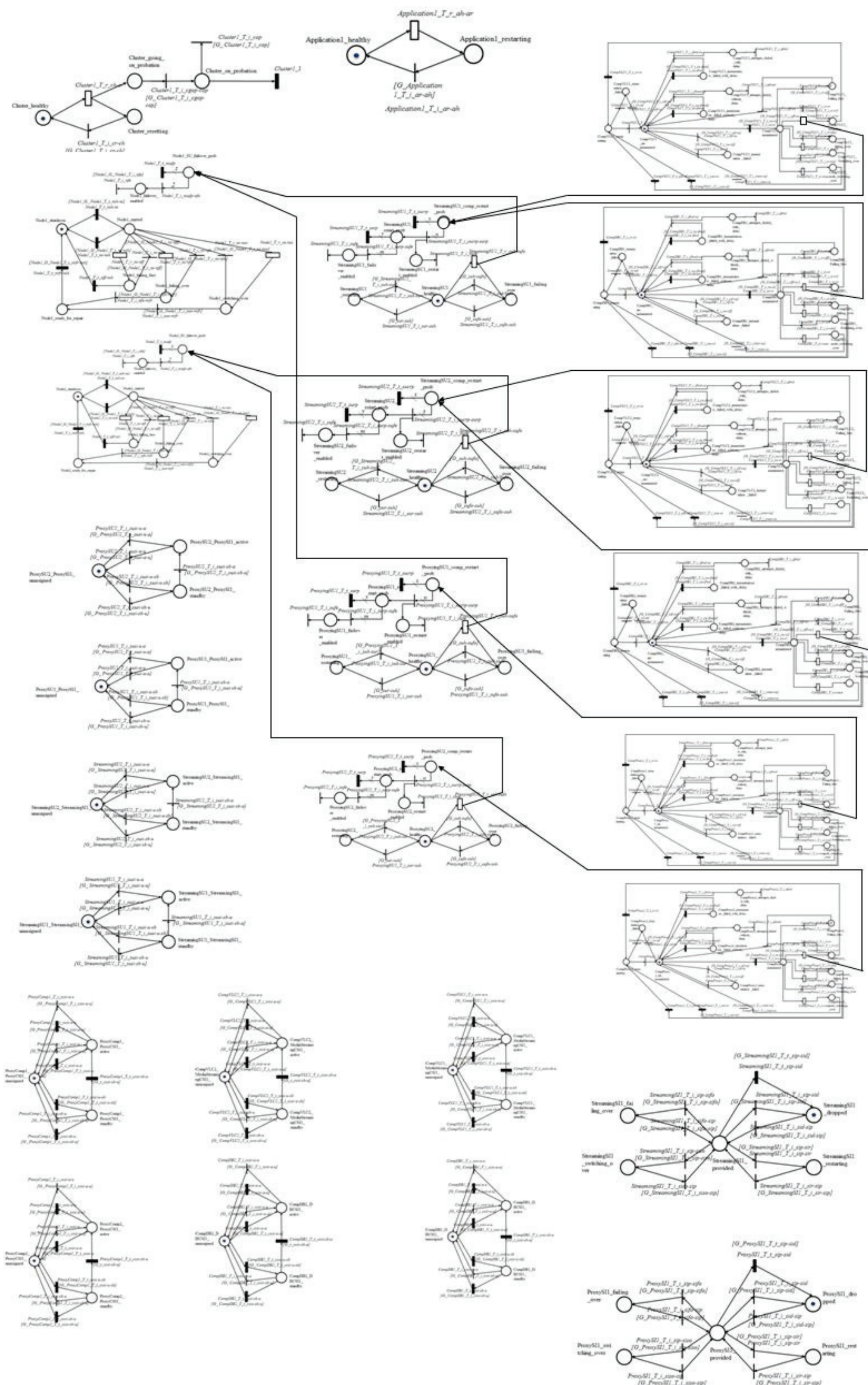


Figure 7-12 The DSPN instance for the availability analysis

Figure 7-13 illustrates the result of our experiment, where the effect of the failure of the proxy on the availability of the streaming SI proved to be much less significant than the effect of the node failure. This is mainly due to the fact the failure of the proxy impacts the database only during the database instantiation and service assignment, otherwise, the failure of the proxy does not prevent the database from performing its intended functionality.

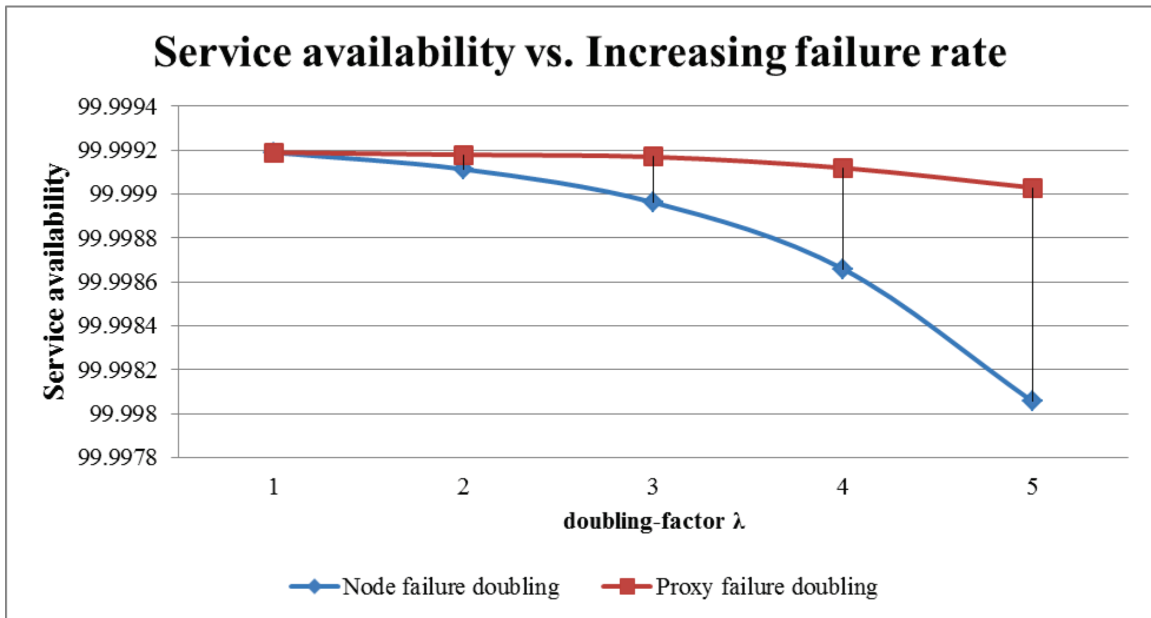


Figure 7-13 The availability chart of the streaming SI

The availability figures shown above are based on the assumption that the active proxy SU and the active streaming SU are on the same node, (i.e. streaming SU1 has higher rank within its SG than streaming SU2 and hence it will be chosen by AMF to be the active SU, and the same for proxy SU1). And it is assumed that the auto-adjust feature for the SG is set to true. Nonetheless this means that during a node failure the proxy SI will have to first be assigned active to the standby SU (proxy SU2) before the any streaming

SI can be assigned active to its respective standby SU (streaming SU2). We can reverse the proxy SI assignment by reversing the ranking of the SUs in the proxy SG. I.e. By having the proxy SU active on Node2 when the streaming SU is active on Node1 (and vice versa), we decouple the assignment of the streaming SI from the assignment of the proxy SI during a node failover, because the proxy SI will already be assigned on a different node. Figure 7-14 illustrates the improvement in the availability of the streaming SI that can be achieved through reversing the assignment of the proxy SI. It should be noted here that we are making the assumption that the delay in proxying when it is performed from a remote node, is generally shorter than the time needed to assign the proxy CSI active to the proxy component, and in this particular example we consider this delay to be negligible.

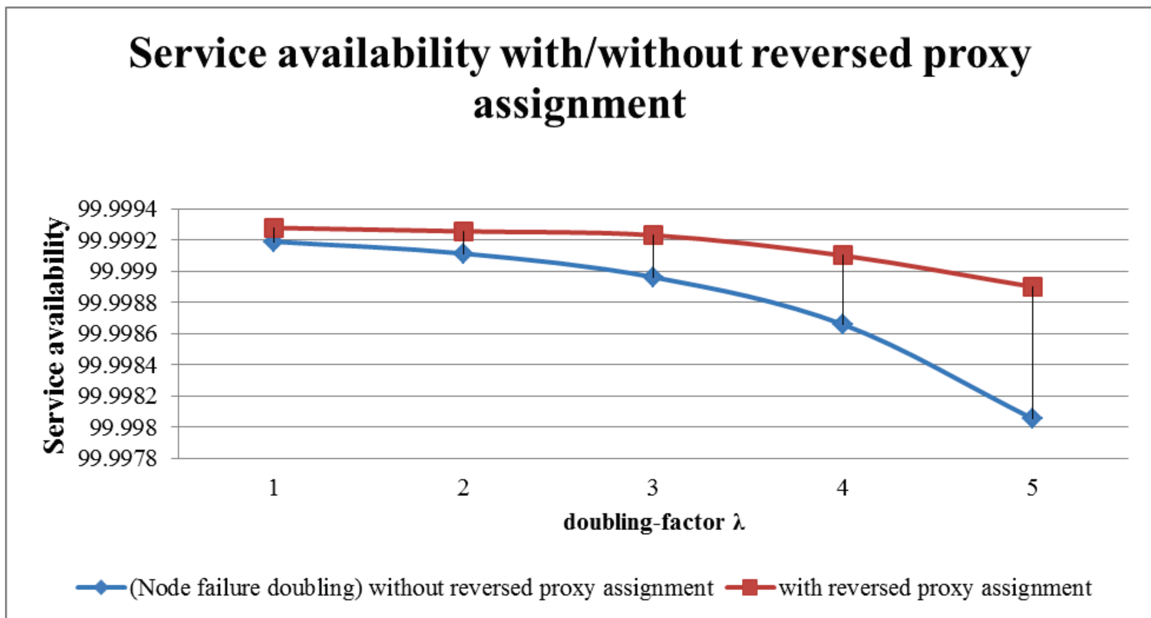


Figure 7-14 The availability chart of the streaming SI with reversed proxy SI assignment

8 Conclusion and Future Work

8.1 Summary of Challenges and Contributions

The SAForum specifications provide open-specifications of APIs based on which highly available applications can be developed. Such applications are portable among any SAForum compliant middleware implementation. The availability management is performed in the context of the AMF configuration. This configuration will define the runtime behavior of the middleware implementation in managing the life-cycle and the service recovery in such applications. Creating AMF configurations is a complex and error prone task. In this dissertation we presented an approach to overcome this complexity by automating the design process of the AMF configurations. The objective behind automation is to enable the creation of AMF configurations in a feasible manner, where the configuration designer is relieved from acquiring detailed domain knowledge and performing complex calculations. To achieve this, we had to surmount several challenges. The first challenge was in defining the starting point of automation, i.e. defining what is the input required for automating this process, and how much of this input should originate from the configuration designer. Depending on the size of the cluster and the number of applications, the input portion that originates from the configuration designer can be tedious to perform, which led us to define the notion of templates, which were implemented in a CR domain model, this allows the specification of the input in a scalable and consistent manner. The consistency is ensured through the

use of OCL constraints. The software description (through ETF(s)) specifies the software dependency at the software type level, but not at the entity level. Though the dependency specification can be partially automated (e.g. specifying the instantiation level), the designer intervention is needed to specify the dependency relation at the service level. The issue here is that the designer may not be aware of such dependencies. In order to solve this issue our approach incorporates dependency analysis and a dependency specification mechanism that notifies the designer of the dependencies and allows specifying the dependency relation in a generic manner. Once the input is complete, the configuration creation can be performed. A key issue here is detecting the ETF attributes that affect the configuration creation. The AMF specification defines various constraints that we identified and embedded in the configuration generation process. Quantifying the workload that the SUs must be capable of handling is an important mandate that is needed at various stages of the process. It is important that this workload is defined in such a way that we do not over-dimension the system and thus underutilize resources, and not fall short and have services dropped. Certain configuration attributes require complex analysis to be populated. The SU ranking for SIs falls under this category. It defines the runtime workload distribution of SUs and it is defined at configuration time. The challenge lies in anticipating the runtime workload redistribution after a failure occurs, and define the proper rankings that maintain the workload balanced before and after a failure. We tackled this problem by defining the workload that each SU must handle, and the workload that each SU must back up. In fact, defining the backup load and making sure it consist of equal contribution from each of the SUs in the SG proved to be a crucial factor in the solution. SU ranking in the N+M redundancy model presented

additional constraints which cannot be addressed in a single solution; therefore we devised three solutions that target different constraints.

In highly available systems, non-functional properties such as the achievable runtime service availability are extremely important. In this dissertation we presented a framework for the availability analysis of the services provided by applications managed by an AMF implementation. In order to achieve this task we had to surmount several challenges. It was clear that the analysis had to be carried based on the information specified in the AMF configuration, since this information includes the recovery actions and essential timings that we are needed. However the configuration is defined for runtime availability management purpose and not for the availability analysis, thus it was missing key information for which we had to extend the configuration model to accommodate. Another issue was that the recommended recovery is not necessarily the actual runtime recovery that AMF will perform. This is due to what we identified to be recovery altering attributes in the configuration. To surmount this challenge we defined actual recovery algorithms capable of analyzing an extended configuration and annotate it with the actual recovery associated with each failure type. AMF has a complex runtime behavior. This behavior varies according to the configuration attributes. For instance the component category affects how AMF handles the recoveries executed on this component. As a result, mapping the AMF configuration to an analysis proved to be a very challenging task, especially when the mapping is affected by the configuration attributes. To master this complexity we defined what we refer to as DSPN templates that capture the generic runtime recovery behavior. Thereafter we defined the mapping that allows the instantiation of these templates and their parameterization with the proper

guards and transitions delays. The parameterization of the instances is based on the configuration attributes. Finally this DSPN instance is solved using off-the-shelve tools to quantify the service availability. In short we are giving the configuration designer the tools to generate and analyze AMF configurations with a minimum understanding of the domain.

8.2 Future work

In this dissertation we introduced the automation to the AMF configuration generation and presented an approach for the service availability analysis. As a result, this opened the door for several research topics that are still not addressed, and they can be summarized as follows:

8.2.1 Mapping Non-Functional User Requirements to the Configuration Requirements

The Configuration Requirement model we introduced in this thesis is meant to be used by the configuration designer who is typically the system integrator with a task to integrate a system that can satisfy the client requirements. The latter requirements do not necessarily come in the form of SIs, CSIs and redundancy models etc. instead they are more likely to be higher level requirements where the client names the services needed and the minimum availability level which must be guaranteed. The mapping from these high level requirements to the Configuration Requirements is still not defined, and it constitutes a very interesting, and challenging research question that remains to be answered.

8.2.2 Designing Configurations that Satisfy the Non-Functional Requirements

The configuration generation approach we presented targets the issue of generating configurations that are compliant to the AMF specifications. We handled functional and certain non-functional requirements such as satisfying dependency, redundancy model, generating configuration that can handle the workload and provide the specified protection for the services etc. However generating configurations that, by construction, satisfy more generic non-functional requirements such as the availability level is still an open research issue. We believe that the availability analysis that we defined paves the way towards achieving this task, but further investigation is still needed. We believe that the way to proceed is to define configuration design patterns to maximize service availability during the design phase of the AMF configuration rather than a brute force solution that generates and analyzes all possible configurations.

8.2.2.1 Configuration Design Patterns

In Chapter 7 we showed that the availability of an SI that is assigned to a proxied component can be improved simply by reversing the assignment of the proxy SI (with certain assumptions). This is just a simple example to show that we can identify design patterns that under certain conditions can improve to service availability. There are several design decisions during the configuration generation (as we have seen in the multiple configuration generation) that can be made based on design patterns, such as the distribution of the SUs of nodes, and how to define node groups for the SGs etc. The

availability analysis approach that we defined can be used to verify these patterns. However the definition of these patterns is still a potential future work.

8.2.3 Defining Ranks that Maintain the Workload Balanced After Multiple Failures

The load balancing solution we presented in this dissertation is based on the ranking mechanism that is specified by the AMF specifications. Our solution maintains the load balanced after a single failure by anticipating how the load should be redistributed. However if the system undergoes several simultaneous failures, maintaining the workload balanced through ranking becomes extremely challenging, since we would have to consider all possible combinations of simultaneous failures. This leaves room for researching alternative techniques for load balancing in the context of AMF managed systems. Another direction would be to build on the lessons we learnt from the configuration time solution with placement preferences and hybrid solution where the preference is defined at configuration time but the assignment is determined at runtime.

8.3 Closing remark

The SA Forum middleware specifications focus on specifying an infrastructure for the provisioning of highly available services. OpenSAF is a robust (open-source) implementation of these specifications that proved the precision and efficacy of the specifications through various case studies. The SA forum did not fall into the pitfall of over-specification and instead it left some maneuver space for the middleware implementers to innovatively distinguish their implementation. This also opened the door for further fundamental research issues that can further enhance/complement the existing

specifications. Our experience of working with the specifications has exposed us to the level of complexity that the industrial companies undertake to implement scalable and robust middleware. Our interactions with the practitioners have opened our eyes to real-world problems which, in order to solve them, require the expertise and knowledge of skilled researchers. We believe that matching the valuable real-world experience of practitioners from the industry with the knowledge and skills of researchers from the academia constitute the perfect setting to achieve successful results that benefit both sides.

RELATED PUBLICATIONS

Patents Granted

1. **Kanso** et al: “Methods And Systems For Generating Availability Management Framework (AMF) Configurations”, Patent No: US8,006,130 B2

Patent Applications Filed by Ericsson Canada Inc.

2. **Kanso** et al: “Automatic generation of AMF compliant configuration top-down approach”.

3. **Kanso** et al: “Method for generating AMF entities composing an AMF application that provides HA services”.

4. **Kanso** et al: “Load and Backup Assignment Balancing in High Availability Systems”.

5. **Kanso** et al: “Ranking Service Units for Providing and Protecting Highly Available Services with Load Balancing in N+M Redundancy Models”.

6. Pietro et al: “Bridging the Gap between High Level User Requirements and Availability Management Framework Configurations”.

7. **Kanso** et al: “Method to Evaluate the Availability of an AMF Configuration”

Journal Papers

8. **A. Kanso**, F. Khendek, M. Toeroe, A. Hamou-Lhadj “Automatic Configuration Generation for Service High Availability with Load Balancing” to appear in the special issue of the “Wiley Concurrency and Computation: Practice and Experience".

Book Chapters

9. A. Mishra, **A. Kanso** “Intregration of the VideoLan Client With OpenSAF: an Example”, in “Service Availability™: Principles and Practice”,Wiley 2012.

Conference Papers

10. **A. Kanso**, M. Toeroe, F. Khendek, “Automatic Annotation of Software Configuration Models with Service Recovery Information” in the “9th IEEE International Conference on Dependable, Autonomic and Secure Computing” (DASC), Sydney, Australia, 2011.

11. **A. Kanso**, M. Toeroe, F. Khendek, “Workload balancing for highly available services: The case of the N+M redundancy model”in the Proc. of the “9th IEEE International Conference on Dependable, Autonomic and Secure Computing” (DASC), Sydney, Australia, 2011.

12. **A. Kanso**, A. Mishra, M. Toeroe, F. Khendek “Integrating Legacy Applications for High Availability: a Case Study” to appear in the Proc. of the “13th IEEE International High Assurance Systems Engineering Symposium” (HASE), Boca Raton, Florida, 2011.

13. **A. Kanso**, F. Khendek, M. Toeroe, A. Hamou-Lhadj, "Ranking Service Units for Providing and Protecting Highly Available Services with Load Balancing", In the Proc. of 10th annual international conference on New Technologies of Distributed Systems (NOTERE). IEEE computer society (Tunisia chapter) pp: 33 - 40 Tozeur, Tunisia, 2010.

14. **A. Kanso**, S. Kohzadi and P. Salehi, "The MAGIC Solution for Generating AMF Configurations and Upgrade Campaigns", In the Proc. of the First International Workshop On Dependable Services and Systems, IWODSS, Montreal, Canada 2010.

15. **A Kanso**, M. Toeroe, A. Hamou-Lhadj, F. Khendek, "Generating AMF Configurations from Software Vendor Constraints and User Requirements", In Proc. of the 4th International Conference on Availability, Reliability and Security (ARES'09), IEEE CS, Fukuoka, Japan, 2009.

16. A. Gherbi, **A. Kanso**, F. Khendek, M. Toeroe and A. Hamou-Lhadj, "A Tool Suite for the Generation and Validation of Configurations for Software Availability", In Proc. of the International Conference on Automated Software Engineering (ASE), Tool Demo, 2009.

17. **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.

9 *References*

[1] IBM Tivoli System Automation for z/OS, Data sheet URL: http://public.dhe.ibm.com/common/ssi/ecm/en/tid14071usen/TID14071USEN_HR.PDF

[2] Oracle Maximum Availability Architecture – MAA http://docs.oracle.com/cd/E11882_01/server.112/e10803.pdf

[3] Service Availability Forum™, URL: http://www.saforum.org

[4] Service Availability Forum, Application Interface Specification. Availability Management Framework SAI-AIS-AMF-B.04.01.

[5] M. Ajmone Marsan and G. Chiola. On Petri Nets with deterministic and exponentially distributed firing times. In G. Rozenberg, editor, Adv. In Petri Nets 1987, Lecture Notes in Computer Science 266, pages 132-145. Springer-Verlag, 1987.

[6] N. Wang, D. Schmidt, and C. O'Ryan. Component-based software engineering: putting the pieces together, Addison-Wesley, 2001.

[7] J.C. Laprie, "Dependability - its attributes, impairments and means", in Predictably Dependable Computing Systems, B. Randell, J.C. Laprie, H. Kopetz, B. Littlewood (eds.), Springer, 1995, pp. 3-24

[8] Object Management Group, Unified Modeling Language - Superstructure Version 2.1.1 formal/2007-02-03, 2007, URL: http://www.omg.org/technology/documents/formal/uml.htm.

[9] Service Availability Forum, Application Interface Specification. Software Management Framework SAI-AIS-SMF-A.01.01.

[10] W. Xie Hairong , W. Xie , H. Sun , Y. Cao , K. S. Trivedi "Modeling of user perceived webserver availability" in Proceedings of the IEEE International Conference on Communications (ICC), Anchorage, Alaska, 2003

[11] G. Ambuj, L. Stephen , "Modeling and analysis of computer system availability," IBM Journal of Research and Development , vol.31, no.6, pp.651-664, Nov. 1987

[12] S. Natkin. Les Reseaux de Petri Stochastiques et leur Application a l'Evaluation des Systemes Informatiques. PhD thesis, CNAM, Paris, 1980.

[13] M.K. Molloy. On the Intergration of Delay and Throughput Measures in Distributed Processing Models. PhD thesis, University of California, 1981.

[14] C.A. Petri. Kommunikation mit Automaten. PhD thesis, University at Bonn,1962.

[15] M. Ajmone Marsan, G. Balbo, A. Bobbio, G. Chiola, G. Conte, and A. Cumani. The effect of execution policies on the semantics and analyis of Stochastic Petri Nets. IEEE Transaction on Software Engineering (7):832-846, July 1989.

[16] H. Choi, V. G. Kulkarni, and K. S. Trivedi. Transient analysis of deterministic and stochastic Petri nets. In M. Ajmone Marsan, editor, Application and Theory of Petri Nets 1993, LectureNotes in Computer Science, volume 691, pages 166-185. Springer-Verlag, 1993.

[17] 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.

[18] Z. Szatmári, A. Kövi, M. Reitenspiess "Applying MDA approach for the SA forum platform", In Proc of the 2nd workshop on Middleware-application interaction (MAI) ACM Vol. 306, pp 19-24 Oslo, Norway, 2008.

[19] OPENSAF, URL: http://www.opensaf.org/

[20] OPENAIS, URL: http://freecode.com/projects/openais

[21] OPENCLOVIS, URL: www.openclovis.org/

[22] P. Salehi, A. Hamou-Lhadj, P. Colombo, M. Toeroe, and F. Khendek, "A UML-Based Domain Specific Modeling Language for the Availability Management Framework", in Proc. of the 12th IEEE International High Assurance Systems Engineering Symposium, San Jose, CA, IEEE Computer Society 2010, ISBN 978-1-4244-9091-2, pp. 35-44.

[23] P. Salehi, P. Colombo, A. Hamou-Lhadj, and F. Khendek, "A Model Driven Approach for AMF Configuration Generation", in Proc. of 6th Workshop on System Analysis and Modelling, Oslo, Norway, Lecture Notes in Computer Science 6598 Springer 2011, ISBN 978-3-642-21651-0, pp. 124-143.

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

[25] S.Bernardi, J. Merseguer, D.C. Petriu, "Adding Dependability Analysis capabilities to the MARTE profile", In Proc. of 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS), vol. 5301 of LNCS, pages 736-750, Toulouse (France), Sept-Oct, 2008, Springer.

[26] S. Bernardi, J. Merseguer D.C. Petriu: "A dependability profile within MARTE". Software and Systems Modeling , LNCS, vol. 10, pp. 1–24. Springer, Heidelberg, 2011

[27] G. Janakiraman, J. Santos, Y. Turner:"Automated Multi-Tier System Designfor Service Availability", In Proceedings of the First Workshop on Design of Self-Managing Systems, June 2003.

[28] T. Hinrich, N. Love, C. Petrie, L. Ramshaw, A. Sahai, S. Singhal, "Using Object-Oriented Constraint Satisfaction for automated Configuration Generation", 15th IFIP/IEEE International

Workshop on Distributed Systems Operations and Management (DSOM), LNCS Vol. 3278, Springer, pp.159-170, Davis, CA, November 15-17, 2004.

[29] A. Sahai, S. Singhal, R. Joshi, V. Machiraju, “Automated Generation of Resource Configurations through Policies”, Fifth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'04), Yorktown Heights, New York, June 7-9,2004.

[30] K. Keeton , J. Wilkes “Automating data dependability” In Proceedings of the 10th ACM-SIGOPS European Workshop Saint-Emilion, France, July 1, 2002.

[31] A. Kavimandan and A. Gokhale. A Parameterized Model Transformations Approach for Automating Middleware QoSConfigurations in Distributed Real-time and Embedded Systems. In Proceedings of ASE Workshop on Automating Service Quality, (WRASQ 2007), Atlanta, GA, Nov. 2007.

[32] W. Xie Hairong , W. Xie , H. Sun , Y. Cao , K. S. Trivedi “Modeling of user perceived webserver availability“ in Proceedings of the IEEE International Conference on Communications (ICC), Anchorage, Alaska, 2003

[33] A. Bondavalli, I. Mura, I. Majzik, “Automated dependability analysis of UMLdesigns”, in: Proc. of Second IEEE International Symposium onObject-Oriented Real-Time Distributed Computing, 1998.

[34] F Salfner, KWolter : “A Petri Net Model for Service Availability in Redundant Computing Systems” In proc of the 2009 Winter Simulation Conference (WSC), pp. 819 - 826 Austin, TX, 2009

[35] N. Addouche, C. Antoine, J. Montmain.: UML models for dependability analysis of real-time systems. In: Proc. International Conference on Systems, Man andCybernetics. Volume 6. IEEE CS. (Oct. 2004) 5209-5214

[36] S. Bernardi, J. Merseguer, “A UML profile for dependability analysis of real-time embedded systems”, In proc. of the 6th international workshop on Software and performance, Buenes Aires, Argentina, 2007

[37] V. Volovoi, “Modeling of system reliability using Petri-nets with aging tokens,” Reliability Engineering andSystem Safety, vol. 84, no. 2, pp.149–161, 2004.

[38] L. Meshkat, J. B. Dugan, J. Andrews, Dependability analysis with on-demand and active failure modes, using dynamic fault trees,IEEE Transactions on Reliability 51 (2002) 240–251

[39] S. Gokhale, J. R. Horgan, K. S. Trivedi, “Integration of specification,simulation and dependability analysis”, in: Workshop on ArchitectingDependable Systems, Orlando, FL, 2002.

[40] S. Gokhale, “Cost–constrained reliability maximization of software systems”,in: Proc. of Annual Reliability and Maintainability Symposium (RAMS 04), Los Angeles, CA, 2004.

[41] S. Gokhale, K. S. Trivedi, “Reliability prediction and sensitivity analysisbased on software architecture”, in: Proc. of Intl. Symposium on SoftwareReliability Engineering (ISSRE 02), Annapolis, MD, 2002.

[42] A.D'Ambrogio,., G. Iazeolla,, R. Mirandola.: A method for the prediction of software reliability. In: Proc. of the 6th Software Engineering and Applications Conference (SEA2002), Cambridge, MA, USA, 2002

[43] A. Pataricza, : From the General Resource Model to a General Fault ModelingParadigm Workshop on Critical Systems, held within UML'2000 (2000)

[44] M. Dal Cin, : Extending UML towards a Useful OO-Language for Modeling Dependability Features. In: Proc. of 9th Int'l Workshop on Object-Oriented Real-Time Dependable Systems, IEEE CS (October 2003) pp. 325-330, Capri Island, Italy, 2003

[45] G. Pai, J. Dugan: Automatic Synthesis of Dynamic Fault Trees from UML systemmodels. In: Proc. of 13th Int. Symposium on Software Reliability Engineering, IEEE CS (November 2002) pp.243-256, Annapolis, MD, USA, 2002.

[46] S. Bernardi, J. Merseguer, D. Petriu.: A UML profile for Dependability Analysisand Modeling of Software Systems. Technical Report RR-08-05, Universidad deZaragoza, Spain (2008)

[47] 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.

[48] OMG, Object Constraint Language, Version 2.2 -http://www.omg.org/spec/OCL/2.2/PDF

[49] G. Schmidt and T. Strohlein, Relations and Graphs- Discrete Mathematics for Computer Scientists (Springer, Berlin, 1993)

[50] D. Chamberlin , R.Boyce, SEQUEL: A structured English query language, Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control, p.249-264, May 1974, Michigan

[51] E.P.K Tsang, "Foundations of Constraint Satisfaction",Academic Press, London and San Diego. 1993.

[52] K. Chonggun, H. Kameda, "An Algorithm for Optimal Static Load Balancing in Distributed Computer Systems", IEEE Transactions on Computers, v.41 n.3, p.381-384. 1992.

[53] A. N. Tantawi and D. Towsle "Optimal static load balancing in distributed computer systems," J. ACM, vol. 32, no. 2, pp. 455- 465. 1985.

[54] Y. Zhang, H. Kameda, S.L. Hung, "Comparison of dynamic and static load-balancing strategies in heterogeneous distributed systems," Computers and Digital Techniques, IEEE Proceedings", vol.144, no.2, pp.100-106, Mar 1997

[55] K, Chandra, "Load Balancing Servers, Firewalls, and Caches", Canada, WILEY 2002.

[56] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors", Journal of Parallel and Distributed ComputingVolume 7, Issue 2, Pages 279-301. 1989.

[57] W. Miao, D. Li, W. Zhang, "A Load-Balancing Dynamic Scheduling Algorithm under Machine Failure Conditions," Intelligent Computation Technology and Automation (ICICTA), 2010 International Conference on , vol.1, no., pp.144-147, 11-12 May 2010

[58] S.S. Kanhere, H. Sethu, A.B. Parekh,,"Fair and efficient packet scheduling using Elastic Round Robin," Parallel and Distributed Systems, IEEE Transactions on , vol.13, no.3, pp.324-336. 2002.

[59] R. Tong, X. Zhu, "A Load Balancing Strategy Based on the Combination of Static and Dynamic," Database Technology and Applications (DBTA), 2010 2nd International Workshop on , vol., no., pp.1-4, 27-28 Nov. 2010

[60] S. Dhakal, M.M. Hayat, J.E. Pezoa, C.T. Abdallah, J.D. Birdwell, J. Chiasson, "Load balancing in the presence of random node failure and recovery, Proceedings 20th IEEE International Parallel & Distributed Processing Symposium, pp.36. 2006

[61] T.C.K Chou, J.A. Abraham, "Load Balancing in Distributed Systems," Software Engineering, IEEE Transactions on , vol.SE-8, no.4, pp. 401- 412. 1982

[62] A. N. Tantawi, D. Towsley, “A General Model for Optimal Static Load Balancing in Star Network Configurations”, Proceedings of the Tenth International Symposium on Computer Performance Modelling, Measurement and Evaluation, p.277-291. 1984.

[63] D.K. Madathil, R.B Thota, P. Paul, T. Xie; , "A static data placement strategy towards perfect load-balancing for distributed storage clusters," Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on , vol., no., pp.1-8, 14-18 April 2008

[64] JC Laprie “Dependable Computing and Fault-Tolerance “- Digest of Papers FTCS-15, 1985

[65] J. Billington and et. al. The Petri Net Markup Language: Concepts, Technology, and Tools. InW.M.P. van der Aalst and E. Best, editors, Application and Theory of Petri Nets 2003, volume 2679 of Lecture Notes in Computer Science, pages 483–506. Springer-Verlag, Berlin, 2003.

[66] Renew: The Reference Net Workshop. URL http://www.renew.de. 2002/03/04

[67] A. Zimmermann, J. Freiheit, R. German, G. Hommel, Petri net modelling and performability evaluation with TimeNET 3.0., in: Proceedings of the 11th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Chicago, IL, 2000.

[68] Eclipse Modeling Framework Core (Ecore), URL:http://download.eclipse.org/modeling/emf/emf/javadoc/2.7.0/org/eclipse/emf/ecore/package-summary.html#details

[69] Eclipse Modeling Framework (EMF), URL:http://www.eclipse.org/modeling/emf/

[70] Eclipse, URL: http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/indigosr2

[71] Java Swing URL: http://www.javaswing.org/

[72] IMM Schema, SAI-AIS-IMM-XSD-A.01.01.xsd (available at: http://www.saforum.org)

[73] VideoLAN Client (VLC). URL: http://www.videolan.org/