A Framework for Requirements Decomposition, SLA Management and Dynamic System Reconfiguration

Mahin Abbasipour

A Thesis

In the Department

of

Electrical & Computer Engineering

Presented in Partial Fulfilment of the Requirements

For the Degree of

Doctor of Philosophy (Electrical & Computer Engineering) at

Concordia University

Montreal, Quebec, Canada

August 2018

CONCORDIA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Mahin Abbasipour

Entitled: A Framework for Requirements Decomposition, SLA Management and Dynamic System Reconfiguration

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

Doctor of Philosophy (Electrical & 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:

______________________ Chair
Dr. S. Samuel Li

______________________ External Examiner
Dr. Michel Dagenais

______________________ External to Program
Dr. Roch Glitho

______________________ Examiner
Dr. Abdelwahab Hamou-Lhadj

______________________ Examiner
Dr. Yan Liu

______________________ Thesis Co-Supervisor
Dr. Ferhat Khendek

______________________ Thesis Co-Supervisor
Dr. MariaToeroe

Approved by

Dr. Wei-Ping Zhu, Graduate Program Director

Tuesday, July 10, 2018 ______________________
Dr. Amir Asif, Dean
Faculty of Engineering and Computer Science

ABSTRACT

A Framework for Requirements Decomposition, SLA Management and Dynamic System Reconfiguration

Mahin Abbasipour, Ph.D.

Concordia University, 2018

To meet user requirements, systems can be built from Commercial-Off-The-Shelf (COTS) components, potentially from different vendors. However, the gap between the requirements referring to the overall system and the components to build the system from can be large. To close the gap, it is required to decompose the requirements to a level where they can be mapped to components.

When the designed system is deployed and ready for operations, its services are sold and provided to customers. One important goal for service providers is to optimize system resource utilization while ensuring the quality of service expressed in the Service Level Agreements (SLAs). For this purpose, the system can be reconfigured dynamically according to the current workload to satisfy the SLAs while using only necessary resources. To manage the reconfiguration of the system at runtime, a set of previously defined patterns called elasticity rules can be used. In elasticity rules, the actions that need to be taken to reconfigure the system are specified. An elasticity rule is generally invoked by a trigger, which is generated in reaction to a monitoring event.

In this thesis, we propose a model-driven management framework which aims at user requirements satisfaction, SLA compliance management and enabling dynamic reconfiguration by reusing the design information at runtime.

An approach has been developed to derive automatically a valid configuration starting from low level requirements called service configurations. However, the service configurations are far from requirements a user would express. To generate a system configuration from user requirements and alleviate the work of designer, we generate service configurations by decomposing functional user requirements to the level where components can be selected and put together to satisfy the user requirements. We integrated our service configurations generator with the previous configuration generator.

In our framework, we reuse the information acquired from system configuration and dimensioning to generate elasticity rules offline. We propose a model driven approach to check the compliance of SLAs and generate triggers for invoking applicable elasticity rules when system reconfiguration is required. For handling multiple triggers generated at the same time, we propose a solution to automatically correlate the actions of invoked elasticity rules, when required. The framework consists of a number of metamodels and a set of model transformations. We use the Unified Modeling Language (UML) and its profiling mechanism to describe all the artifacts in the proposed framework. We implement the profiles using Eclipse Modeling Framework (EMF) and Papyrus. To implement the processes, we use the Atlas Transformation Language (ATL). We also use the APIs of the Object Constraint Language (OCL) in the Eclipse environment to develop a tool for checking constraints and generating triggers.

Acknowledgments

I would like to express my sincere thanks and appreciation to those who made this thesis possible through their wisdom and mentorship, my supervisors Dr. Ferhat Khendek and Dr. Maria Toeroe. All these years, they were very supportive. They taught me to be curious and no challenge is insurmountable. Their advice as a mentor in both life and academic will have a lasting impact.

I would like to extend my thanks to the examining committee for their advice and valuable inputs during the various stages of my PhD.

I also would like to thank my friends Azadeh Jahanbanifar, Atena Roshanfekr and Fatemeh Saraylou for their support and friendships as well as my colleagues in the MAGIC group. The atmosphere of collaboration and discussion in our group was invaluable.

This work has been partially supported by Natural Sciences and Engineering Research Council of Canada (NSERC), Ericsson and Concordia University. I would like to thank them for their financial support and providing me the opportunity to learn and grow.

I could not have come this far without the love and support of my parents; my sisters Maryam, Marjan and Mina Abbasipour; my brother Jalil Fallah and my little angel Parmiss Fallah. They taught me to be open-minded and appreciate different views. Their unwavering love will never be forgotten.

Dedication

This work is dedicated to my parents, Tayebeh Zandieh and Aliasghar Abbasipour. Their endless love, perseverance and self-sacrifice made my achievements possible. I love you mum and dad; you are the treasure of my life.

Table of Contents

List of Figures

List of Tables

List of Acronyms

AIS	Application Interface Specification
AMF	Availability Management Framework
ATL	Atlas Transformation Language
COTS	Commercial-Off-The-Shelf
CST	Component Service Type
CT	Component Type
DSML	Domain Specific Modeling Language
EMF	Eclipse Modeling Framework
ETF	Entity Types File
FR	Functional Requirements
HPI	Hardware Platform Interface
MDA	Model Driven Approach
MDD	Model Driven Development
NFR	Non-functional Requirements
OCL	Object Constraint Language
SA Forum	Service Availability Forum
SI	Service Instance
SLA	Service Level Agreement
SLO	Service Level Objective
SOA	Service Oriented Architecture
SU	Service Unit
UML	Unified Modeling Language
UR	User Requirements
VM	Virtual Machine

Chapter 1

Introduction

1.1 Thesis Motivation

Commercial off-The-Shelf (COTS) components are reusable software components which are developed independently from each other. The use of COTS components promises a better productivity in software development as well as more maintainable system. A COTS-based system is built by integrating such independently developed components together. The organization and characteristics of the entities composing the system (e.g. the components and resources the components require) and their relationships are described in a system configuration.

User requirements are the main goals to meet with the envisioned system. The user requirements are high level and related to the overall system while the components are at lower level; therefore, the gap between these requirements and the components can be large. To close the gap, it is required to decompose the user requirements to a level where the components can be selected. For the selection of appropriate components, both functional and non-functional requirements need to be taken into account. Thus, the first challenge we tackle in this thesis is to define an automated approach for decomposing functional user requirements and selecting components while both functional and non-functional requirements are satisfied.

Service availability, which is defined as the percentage of time the service is provided to the users [1], is required in several domains such as mission critical and telecom systems. Any service outage in these systems can result in catastrophic damages, financial or reputation loss [2]. As a result, service availability is an important non-functional requirement to consider during design of the system.

When the system is built, deployed and is ready to provide services to customers, the service quality the provider agrees to deliver to the customer as well as the rights and obligations of the service provider and the customer are negotiated. The negotiated terms are included in a contract called Service Level Agreement (SLA) [3]. For example, the level of service availability is included in the SLAs. When an obligation is not met, the responsible party may be subject to penalty to compensate the breach of SLA commitment.

At runtime the workload of the deployed system varies dynamically, which results in variable resource usage. Service providers aim at maximizing system's resource utilization while ensuring that the SLAs are always met. For this purpose, instead of allocating a fixed amount resource, service providers want to allocate only as much as required for the current workload and adapt according to workload variations. This is known as elasticity where a system evolves and adapts dynamically to workload variations by scaling up/down and in/out [4]. Generally, whenever there is a potential SLA violation or the resource utilization is low, one or more triggers are generated to invoke the elasticity rules, which consist of actions to take in the current system's situation. One of the main challenges is the definition of the elasticity rules which are applied at runtime to ensure that the resulting system meets the required qualities of service such as availability and performance while using only the necessary resources.

On the other hand, at runtime, more than one trigger may be generated at a time. Each trigger invokes an elasticity rule consisting of actions to perform. An action applied on a configuration entity may have an impact on other entities of the configuration because of the relations and dependencies between them. As a result, handling each trigger independently or in an ad hoc manner may endanger the stability of the system. Therefore, an approach is required to manage reconfiguration dynamically and protect the system from instability and malfunctioning.

1.2 Contributions of this Thesis

To address the aforementioned issues we define a model-driven configuration management framework. The Model-Driven Development (MDD) [5] paradigm separates the application logic from the platform technology and manipulates platform-independent models; thus models are the primary artifacts in the development process [6]. The major advantage of this paradigm is that the models are at higher level of abstraction than the implementation technology and artifacts. This paradigm is appropriate for our purpose as it allows not only to facilitate the understanding, design and maintenance of the system [7], but also to reuse the models generated during the system design phase. In this framework, we use Domain Specific Modeling Languages (DSMLs) to capture the concepts, their relations as well as their constraints [8]. We use the Unified Modeling Language (UML) profiling mechanism [9] to define the DSMLs. The constraints are expressed using the Object Constraint Language (OCL) [10].

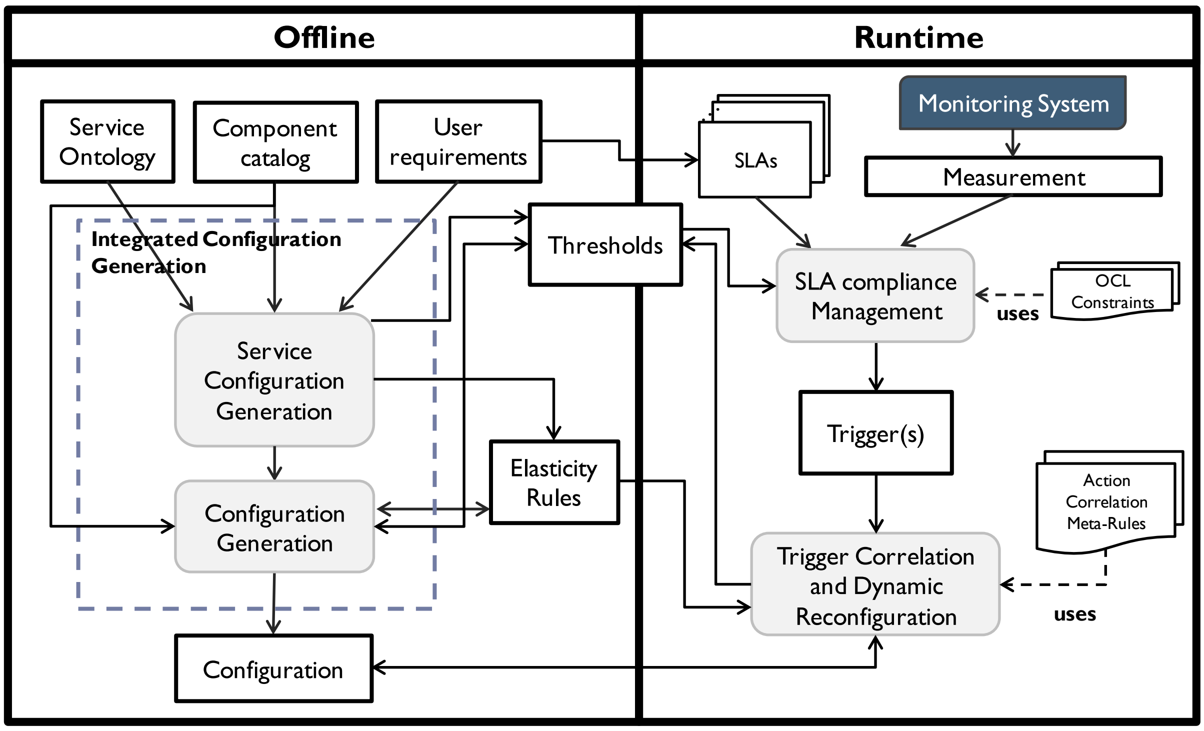


Figure 1.1 The overall picture of the configuration management framework

The main objective of this thesis is to define a model driven solution for designing COTS based systems to ensure user requirements satisfaction, and enabling dynamic reconfiguration by reusing the design information at runtime. As shown in Figure 1.1, our configuration management framework consists of two parts: offline and online. The offline part includes service configurations and elasticity rules generations. The online part consists of SLA compliance management and dynamic system reconfiguration using predefined action correlation meta-rules and the generated elasticity rules.

In [11] an approach has been developed to generate automatically a valid configuration for highly available systems starting from service configurations and software catalog. Service configurations specify the set of services to be provided by a software system. More specifically, they define different characteristics of services such as their types, the number of instances of each type and the relationship between services. Therefore, service configurations are low level and in relation with COTS components and far from the requirements that a user would express. Specifying the service configurations requires deep domain knowledge and expertise. To alleviate the work of the system designer and start configuration generation from higher level requirements, we devised a model driven approach that generates service configurations by decomposing the user requirements automatically with the decomposition knowledge captured in a service ontology and selects the COTS components that satisfy both functional and non-functional requirements.

In this thesis, we generate the elasticity rules at system configuration generation time while taking into account availability of services. During the design of the system, the thresholds related to the capacity of the system are generated.

At runtime, the SLA compliance management watches the workload variations and SLAs. For this purpose, we have defined some OCL constraints which are periodically evaluated at runtime. Violation of the OCL constraints leads to the generation of triggers for dynamic reconfiguration. Triggers initiate the application of the corresponding elasticity rules to reconfigure the system accordingly and avoid SLA violations from the provider and resource wasting.

To handle triggers raised at the same time, we propose a model driven approach to correlate the triggers and the actions of their related elasticity rules. In order to automatically coordinate the actions of invoked elasticity rules, execute them on the fly and also avoid resource oscillation [12] we have defined action correlation meta-rules. The action correlation meta-rules are high level rules that govern the application of the elasticity rules when the triggers are correlated.

As mentioned before, our framework is model-driven; we use DSMLs to capture the concepts, their relations as well as their semantics. To define the DSMLs, we use the UML profiling mechanism. We follow the approach in [8] to define the UML profiles: first, we define the metamodels and then we map the metamodels to the UML metamodel.

The main contributions of this thesis are summarized as follows:

- To design a system that meets user requirements, we propose a model-driven approach to generate service configurations by decomposing functional user requirements to a level where COTS components can be selected while taking into account the non-functional requirements. During the decomposition process, we also generate the traceability links that relate the user requirements and selected components as well as rejected solutions. We integrate our service configurations generator with the configuration generator in [11] to generate automatically configurations for highly available systems from higher level requirements.
- To generate elasticity rules offline, we propose a model-driven approach that reuses the information acquired during configuration generation time. The generated elasticity rules will be used at runtime to reconfigure the system dynamically.
- To check the compliance of SLAs at runtime, we have defined a set of OCL constraints which are evaluated periodically and dynamically. Violation of the OCL constraints leads to the generation of triggers for dynamic system reconfiguration.
- To reconfigure the system dynamically, we propose a model-driven approach that correlates triggers issued on related entities and executes actions of their related elasticity rules on the fly. In order to achieve this, we defined action correlation meta-rules that govern the application of elasticity rules when the triggers are correlated. We also ensure that certain properties of the provided service such as service availability and performance are maintained.

To illustrate our work we use the Service Availability Forum (SA Forum) [13] middleware as an application domain throughout this thesis. However, our work is applicable in more general context where the service and service provider perspectives are described explicitly in the configuration.

As a proof of concept we implemented our approach for user requirement decomposition and service configurations generation using the Atlas Transformation Language (ATL) [14]. To derive the configuration from higher level requirements and generate elasticity rules offline, we extended the current tool for configuration generation [11] and integrated it with the service configurations generator. We implemented a prototype of SLA compliance management in the Eclipse Modeling Framework (EMF) [15] using OCL APIs [16]. We also implemented a prototype of trigger correlation and dynamic reconfiguration using ATL.

1.3 Thesis Organization

The rest of this thesis is organized as follows: In Chapter 2, we provide the background knowledge including the SA Forum services and concepts which are used as an application domain throughout this thesis. We also discuss briefly some concepts from the model-driven paradigm. In Chapter 3 we review the related work in functional user requirements decomposition, elasticity rule generation, trigger correlation and elasticity management. In Chapter 4 we discuss our model-based approach for the decomposition of functional user requirements and selection of COTS components to satisfy user requirements. In Chapter 5 we elaborate on the generation of the elasticity rules at configuration generation time, i.e. offline. In Chapter 6, we discuss our approach for checking the compliance of SLAs and generating triggers for system reconfiguration at runtime. In Chapter 7 we discuss our trigger correlation and dynamic reconfiguration approach to manage the application of elasticity rules at runtime. In

Chapter 8 we conclude this thesis by reviewing its main contributions and potential future work.

Chapter 2

Background

In this chapter, a brief overview of the model driven paradigm [6] and the SA Forum middleware [13] is provided. In the first part of this chapter we introduce the main concepts of model driven development such as Domain Specific Modeling languages (DSML) [8], metamodels, Unified Modeling Language (UML) profiles [9] and traceability in general. In the second part of this chapter, we review the context of service availability in SA Forum and explain how Availability Management Framework (AMF) [17] manages the availability of services.

2.1 Model Driven Development (MDD)

The model driven development is a system development approach which emphasizes abstracting the concepts of the domain by creating and analyzing models. Therefore, models are the primary artifacts in this approach which replace the codes in the software development. These models, which are called platform independent models, capture the information about the system and its behavior rather than the specific implementation and platform details [7]. The model driven approach consists of models which are the abstract representation of the system, metamodels that specify the syntax and semantic of the models, and transformations (e.g. model to model and model to code transformations).

2.1.1 Modeling

A model is an abstract representation of a system at high level. To describe models, various modeling languages as metamodels can be used. A metamodel is a special kind of model that defines the entities, their structures as well as the semantic of instance models. A model which is built according to the syntax and semantics of its metamodel is said to *conform* to its metamodel. The modeling languages can be generic such as UML [18] to formally define models in different domains or they can be customized i.e. Domain Specific Modeling Languages (DSMLs) [8].

2.1.2 Domain Specific Modeling Languages and UML Profiles

A Domain Specific Modeling Language (DSML) is a specialized language for defining the models in a specific domain. DSMLs allow developers to express their application models with specific concepts and properties of the application domain [8]. To benefit UML as a standardized modeling language and also take the advantage of DSML, UML provides profiling mechanisms [18] which allow constraining and customizing the UML for creating DSMLs for specific domains. A UML profile consists of stereotypes, tagged values (i.e. the attributes of the stereotypes), and constraints to restrict and customize the UML. The constraints are side-effect free; therefore, their evaluation does not affect the executing system [10]. The constraints in UML profiles and models can be described by Object Constraint Language (OCL) [10]. OCL expressions can also be used to specify the operations/actions that when executed, the state of the system changes. In our work we define UML profiles to formally define the concepts, relations and constraints of the configuration domains.

2.1.1 Model Transformation

Various operations as model transformations are defined to manipulate models and generate other artifacts (such as source codes, configurations, etc.). Therefore, model transformations are mapping functions from the source model(s) to the target model(s) which facilitate the automation of the development process. Model-to-model and model-to-code transformations are the most common model transformation. Model transformations are mainly unidirectional. In the case that a transformation works in both directions, the input and output sides can be generated from each other.

2.1.2 Traceability

During the development process, many models are created, refined or transformed [19]. In order to keep consistency between the models, it is useful to keep trace links between the models. There are several studies related to traceability. In [20], traceability is classified into two categories: requirement traceability and traceability in MDD. In the first category, traceability is defined as the ability to follow the life of a requirement during development process in both directions [21]. The second category defines the trace as a link between a group of elements in input models and a group of elements in output models [20]. In this approach, trace links are generated via model transformations. The second category can have the same purpose as the first category because MDD is used for the automation of software development process.

2.2 The Service Availability Forum (SA Forum) Middleware

In this section we introduce service availability as well as the SA Forum concepts.

2.2.1 Service Availability

Many critical applications are needed to provide a service with no or minimum outage. The service availability is defined as a probability that a service is provided during a time interval. Therefore, it is defined based on the time that the service is up. It is measured using the following formula:

$$Service\ Availability = \frac{service\ uptime}{service\ uptime + service\ outage\ time}$$

High Availability (HA) is achieved when the service is available at least 99.999% of the time, i.e. five minutes and 15 seconds of downtime per year [1].

2.2.2 The Service Availability Forum

Several leading telecommunications and computing companies have formed the Service Availability Forum (SA Forum) [13]. The solution offered by SA Forum facilitates service availability. The SA Forum middleware consists of two sets of interfaces: the Hardware Platform Interface (HPI) [22] and the Application Interface Specification (AIS) [17]. The SA Forum HPI specifies a generic mechanism to discover the hardware entities present in systems as well as to monitor and control the hardware platform through a consistent, platform independent set of programmatic interfaces. AIS defines the application programming interfaces for functionalities commonly used by application developers to develop highly available applications therefore allowing for portable solutions. AIS defines several services to support the development of HA applications. Among these services, Availability Management Framework (AMF) [17] manages the availability of services that are provided by the constructed system. In the next section we briefly introduce AMF.

2.2.3 The Availability Management Framework (AMF)

AMF maintains service availability by managing redundant entities that together compose the applications deployed within a cluster [1]. To manage an application by AMF, the application needs to be described according to the AMF configuration model. In this configuration, the entities composing the system, their types (i.e. the common characteristics of a set of entities) and their relations are given. In AMF, the entities in the configuration are classified into two categories: service entities to represent the provided services and service provider entities to describe the entities that can provide those services. Thus in AMF, a system then can be viewed from two perspectives: service side and service provider side. On the service side, a Component Service Instance (CSI) is a service entity that represents a chunk of the workload. A Service Instance (SI) is the aggregation of CSIs. CSIs and SIs are logical entities. On the provider side, service provider entities are pieces of application software, physical or virtual computing nodes, which are tangible resources. The application software entities are components capable of supporting/providing CSIs. A set of components that collaborate to combine their services forms a Service Unit (SU) capable of supporting/providing SIs. Each SU is deployed on a computing node. In [23], AMF UML profile has been defined by mapping the AMF configuration metamodel to the UML metamodel.

The reason for the distinction between service entities and service provider entities is that the relation between them is not always one-to-one, and service entities can be assigned and reassigned dynamically to different service provider entities even to multiple of them simultaneously. Indeed, to provide highly available services the provider side contains entities to provide and also to protect the service side entities in case of failure [1]. A set of redundant service units can be grouped into a Service Group (SG), which can be organized with different redundancy models, namely 2N, N+M, N-way active, N-way and No-redundancy [1]. An SG is deployed on a node group eligible hosting its SUs. An application may consist of multiple different SGs. Depending on the redundancy model of the SG, at runtime an SI may have one or more active and zero or more standby assignments; each of which is assigned to a different service unit of the service group. A service unit with an active assignment provides the service. A service unit with a standby assignment does not provide the service, but is ready to become active in a minimal amount of time [1].

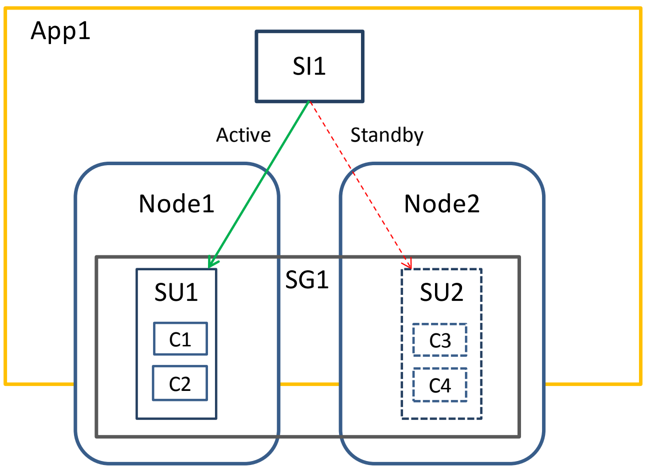


Figure 2.1 An AMF configuration

In Figure 2.1, App1 consists of one SG (SG1) and one SI (SI1). This SG has two SUs (SU1, SU2) and protects the service represented by SI1. Each of the SU1 and SU2 is hosted on a separate node: Node1 and Node2. AMF assigns the active and standby roles on behalf of SI1 to the SUs at runtime. SUs in active and standby roles are shown with solid and dashed lines in this figure. When SU1 fails, AMF changes the standby SU2's assignment to active to continue service provisioning.

2.2.4 Entity Types File (ETF)

In order to design an AMF configuration for a given software system, it is necessary to have a description of the software's components, their capabilities, the services they support, as well as the constraints on any of the parameters and their combination options. This description is provided by the software developer which is compliant to another SA Forum standard, known as the Entity Types File (ETF). Using ETF, software developers can specify the characteristics of their software, capabilities, and limitations in a way that can guide the generation of an AMF configuration. Moreover, ETF types describe how an application's components can be combined by providing information regarding their dependencies and compatibility options.

Chapter 3

Related Work

In this chapter we review the work related to our management framework in five aspects: functional user requirements decomposition, elasticity rule generation, SLA compliance management, trigger correlation and dynamic system reconfiguration. As most of the related work focuses only on one aspect of our framework, we organize this section into five sub-sections, one for each aspect.

3.1. Functional User Requirement Decomposition

The composition/decomposition of functionalities resembles feature models [24]. A feature model is used to describe a software product line, i.e. all the products that can be built from a given set of features. This is done using a feature diagram where features can be marked as mandatory or optional and alternatives can be described. The focus is to define the valid combinations (also called configurations) of a given set of features in bottom-up manner while in our approach the goal is to decompose functionalities and determine components that can be used to compose the system in a top down manner. We decided to use UML for the modeling of service ontology and user requirements instead of feature diagrams to capture properly non-functional requirements when associated with functional requirements, potential communications and interactions between components and/or environment.

The work done in the CARE project [25] is closely related to this research. In [25] software components are evaluated using models of software components and models of the component-based application. Requirements are expressed as logical queries that can be composed of sub-queries using logical operators. The queries and sub-queries are matched against the functionalities of the components. The components can be ranked and selected using different searching/evaluation techniques. The focus in our work is on automating the requirement decomposition – which remains a manual and incremental task through interactions in [25] – and only afterwards choosing appropriate components.

A lot of work has been done in the domain of web service composition, either at runtime or offline during design. The closest works related to our work are the approaches that start from an overall goal or functionality, decompose it into sub-requirements/goals, search for services that meet sub-requirements, compose them, and verify for the satisfaction of the overall goal. Such approaches like [26, 27] are generally based on formal methods, focus on the behavioral description of the services, their behavioral properties and verification techniques. Our work is targeting COTS components described at a higher level of granularity. In addition, our work is completely model driven and enables traceability.

In [28], an ontology is used to store the knowledge about requirements, their relations, components and their non-functional characteristics. The purpose of the ontology is to capture design knowledge. The goal is to propose a generic requirement management process to handle consistency during concurrent engineering activities, completeness, traceability and change management among others.

In [29], component service replacement within a composite service is investigated. An IT service process is given, which consists of an orchestration of service steps (like basic services). One of these service steps may be underperforming, for instance, and the question is how to replace it from the service catalog. To do so, a dependency network is generated using different ontology related technologies to associate a service process with the existing services that can be used as replacements and one is chosen according to certain non-functional criteria. This is more of a replacement problem in the domain of IT services using ontologies in contrast to user requirements decomposition with ontologies.

A lot of work has been done to model Non-Functional Requirements (NFRs) related to the functional requirements. In [30] the modeling is based on software quality standards. To consider NFRs at software development lifecycle, they define ontology for NFRs. In their ontology model, NFRs are modeled as a hierarchy of different NFR types that the parent type has sub-types. For example, accuracy is a sub-type of integrity which is the sub-type of reliability. In the ontology, the knowledge about the decomposition of an NFR type into other NFR types is also kept. Therefore, an NFR like security can be refined into more detailed NFRs such as confidentiality, integrity and availability. They use the ontology to express NFRs, trace them [31] and estimate software effort. In [31], a traceability metamodel is represented and it is explained how this metamodel is used to analyze the impact of a change. In their approach, the decomposition relations within non-functional requirements, the decomposition relations within functional requirements and the relations between functional and non-functional requirements are considered. In our approach, besides traceability links, we also keep track of the non-optimal solutions so the expert later can see for what reason a requirement is not satisfied. In [32], authors propose an approach to analyze the Software Requirements Specification (SRS) documents for extracting different NFR types automatically. The approach is based on natural language processing, machine learning and the ontology proposed in [30]. Their approach consists of two major phases: NFR classification and ontology population. In NFR classification, all sentences of the documents are split and later, the candidate sentences are detected and classified into functional requirements, nonfunctional requirements or design constraints using a machine learning-based classifier. After classifying all the sentences in the SRS, the sentences are linked with the classes of the ontology proposed in [30] to provide reasoning and queries.

The work in [33] discusses the modeling of quality (non-functional characteristics) of COTS components as well as the specification of quality (non-functional) requirements. The work is based on software quality standards and the goal is to provide a taxonomy to enable the selection of appropriate COTS components. In their approach they assume that the requirements are decomposed.

There are some works like in [34, 35, 36] that define models to represent non-functional requirements along with functional requirements but these do not include how the non-functional requirements are handled during the development process.

Traceability relationships help stakeholders understand the many associations and dependencies that exist among software artifacts created during a software development. In order to keep consistency between the models, it is useful to keep the trace links between the models. There are some studies about the traceability approaches. The approach in [37, 19] is similar to our approach. Authors use model transformations for creating traceability. They aim at generating some performance models to study the performance characteristics of the designed system in early stages. The generated traceability links are used to analyze the impact of changes in terms of performance [19].

In [38], similar to us, the authors create traceability model during model transformation. However, the metamodel for the generated traceability link is the extension of the Atlas Model Weaver (AMW) metamodel [39]. AMW is used for establishing links between the elements of models. Our approach also decomposes requirements and includes traceability for non-sufficient solutions as well.

3.2. Elasticity Rules Generation

Elasticity rules can be defined offline or online. In both approaches, the elasticity rules should be precise enough such that by their execution the system is reconfigured properly.

With online approaches [40], [41], [42], [43] and [44] the elasticity rules are learned during the operations of the system using machine learning algorithms. In these studies, the elasticity rules are defined for the states in which the system can be. Each elasticity rule is modeled as: alternative actions that can be taken when the system is in a specific state; the probability of transitioning to a target state when a specific action is taken as well as a reward which indicates the expected benefit of the action. These approaches depend on the learning through the dynamic interaction between the system and a learning agent [45]. Time is divided into intervals. The learning agent watches the current state of the system within an interval; it takes an action among the allowed ones in that state and then observes the new state and the achieved reward. Depending on the achieved reward and state, the probability of transitioning and the expected reward of the elasticity rule may be updated. In these approaches, the new state of the system depends on the probability distribution, which is learned during the operation of the system. Since these elasticity rules are generated online and independently from the configuration, tracing and fixing the problems in the elasticity rules is not easy.

With offline approaches the elasticity rules are predefined before the operation of the system. In these approaches, the elasticity rules are not modified at runtime; therefore, the generated elasticity rules should be precise to reconfigure the system properly. To overcome this challenge, instead of defining elasticity rules with quantitative values, in [46] the elasticity rules are qualitative. For example, an elasticity rule is defined by an expert as follows: “if the workload is high and the response time is low, add VMs”. They use time-series runtime data to predict the future workload as well as a fuzzy controller to specify if qualitative thresholds are reached. The fuzzy logic facilitates the use of qualitative elasticity rules for auto-scaling. In our approach, we generate elasticity rules based on the knowledge used for dimensioning the system at configuration design time. As a result, the defined elasticity rules are more precise than the qualitative elasticity rules or the elasticity rules of online learning approaches which are based on probability.

To express elasticity rules, different languages have been defined/used. Most of these languages use the condition-action syntax for the rules [47], [48], [49] and [50]. The condition is an event such as threshold violation, when it occurs the corresponding action is triggered. The languages differ in the level of abstraction and expressiveness. Some languages like [50] use temporal logic to express the rules. Using temporal logic, time related conditions can be expressed. For example, one can specify a rule like: if event e_1 happens x seconds after event e_2, action A should be taken. [47] and [48] define Domain Specific Modeling Languages (DSMLs) named as SRL and SYBL to specify elasticity rules. SRL [47] has a better expressiveness for events than the languages defined in [48], [49] and [50]. In SRL, events can be behavior related events (e.g. failure of a component) or non-functional related events (violation of a particular quality of service like service availability). Similar to most languages, the definition of actions remains very abstract in SRL and it is not clear how an action makes changes in the system. In SYBL [49], elasticity rules are classified as applications, application components and application component code rules. At the application level, the defined rules are about the whole application (e.g. when the incoming workload hits a threshold). The rules about the components of the system are specified at the component level (e.g. when the resource usage of a component reaches a threshold). At the programming level, the strategies are about the resource usage of a specific code. Someone who is familiar with the system needs to define the triggers and the actions of the rules. The Planning Domain Definition Language (PDDL) [51] is a language which is used extensively to specify the reconfiguration plans including elasticity rules. With PDDL the goal (e.g. a measurement should not cross a threshold), the action (e.g. adding or removing a VM), the impact of the action on the configuration (e.g. the increase in the capacity), the duration of the action as well as the condition under which the action is applicable can be specified. Similar to [51] in our proposed elasticity rule metamodel, the condition checks for the applicability of an action. We also define prerequisites and follow-ups in association with actions to check if prerequisite or follow-up actions are required. This structure enables action correlation at runtime to avoid conflicting actions.

3.3. SLA Compliance Management

Monitoring the system, collecting measurements and assessing them against the SLAs are necessary to ensure the compliance of SLAs.

In [52] and [53], authors propose the rSLA framework to monitor SLAs during their life cycle. The framework consists of three main components: rSLA language, rSLA Service and Xlets. The rSLA language is defined to describe SLAs and service metrics as well as evaluation conditions. In rSLA, it is described how the metrics should be measured and composed to define Service Level Objectives (SLOs). It can also be specified what actions should be taken when the SLAs are violated. The authors of SLAs are the providers. As the SLAs are agreements among the providers and customers and need common understanding of the terms, considering all of the above terms may be cumbersome or not important for the customers as they may not care how the measurements are obtained. The rSLA Service checks for SLA compliance. Xlets provide standard interfaces for monitoring the system and reporting measurements.

In [54], to detect violations with respect to response time, a timed automata is used. The work in [55] is closely related to the SLA compliance management of our framework. The goal in [55] is to detect individual SLA violations only, while in our case we want to avoid the potential SLA violations and achieve this goal with the minimum amount of resources needed according to the workload variations. To check the compliance of SLAs in [55], for each SLA parameter an OCL constraint is defined to check if the measurement has reached the threshold. However, when a new parameter is added to an SLA a new OCL constraint for the violation detection has to be defined as well, which is not the case in our framework.

In [56], authors demonstrate their architecture for SLA violation detection at the application level. In [56], thresholds for SLA parameters such as response time and throughput are defined. According to their architecture, when a user requests for a service, first the request is checked with the SLA to see if the request is coming from the right customer. In the next step, based on the requests, the tasks are generated and executed. There are multiple monitoring agents that collect application level measurements. The measurements are passed to the SLA management framework where it is checked if the value of a threshold has been reached or not.

In [57], a framework for monitoring SLAs is proposed. It consists of three components: a monitoring system for providing measurements, LoM2HiS for mapping monitored data to parameters in the SLAs, and a knowledge database which uses past experience to solve current SLA related issues. This framework is suitable for the infrastructure layer of the cloud. In our framework both, infrastructure and application, levels are handled. On the other hand, [58, 59] for instance do not take SLAs into account.

3.4. Trigger Correlation

In the studies which are threshold based, trigger correlation and the coordination of the related actions which are important for the dynamic reconfiguration of systems are hardly considered. In the current literature, trigger correlation is discussed extensively for fault management of distributed systems and networks where an error caused by a fault is propagated through many related objects and potentially large volume of triggers are generated for the same fault. In these studies, a reported fault is an event which triggers an action and correlation is used as a reduction technique to filter the symptoms and identify the root cause fault; while in our approach triggers are not necessarily symptoms and should not be simply eliminated because the allocation of resources to one entity does not necessarily mean the allocation of resources to another entity even if they are related.

In [60] and [61], authors come up with different correlation graphs based on which the triggers are correlated. The proposed correlation graphs only capture the paths where a fault can propagate, while the edges of our proposed relation graph are of different types and used for correlating the triggered actions. To build a correlation graph in [60], for each entity of the system, the faults that can originate from the entity, the relationships that the entity has with other entities and the faults that propagate along these relationships are specified by an expert. From these elements, a causality graph is inferred. A node in a causality graph is an event which can be a symptom or a root cause fault. A causality graph may have information which may not contribute to the correlation analysis like a cycle. A correlation graph is deduced from a causality graph by eliminating the cycles and aggregating each into a single event or by pruning indirect symptoms (i.e. the symptoms that are not caused by a root cause fault directly). The correlation process in [60] is based on an encoding technique where the events are represented by a code. To code a correlation graph, the root faults in the graph contain bits where each bit corresponds to a single symptom in the correlation graph. For example, for a graph with three symptoms, the code length will be three. The value of 1 for a symptom for a root fault indicates that the root fault causes the corresponding symptom. Therefore, the event correlation process becomes finding problems in the correlation graph whose codes optimally match the observed symptoms. In the case that similar symptoms are caused by different root faults, the root cause is not distinguishable. In [61], the correlation graph is obtained from the dependencies between the functionalities of the managed system. Therefore, the nodes of the correlation graph in [61] are the functionalities and the edges are the dependencies. When a fault occurs in a component, the components which communicate with that component are also affected and similar faults will be reported for them. In [61] they use their proposed correlation graph to identify the component whose failure caused a large number of symptoms. In [60, 61], trigger correlation is the aggregation of similar triggers that report the same fault.

3.5. Dynamic System Reconfiguration

There are many papers which focus on elasticity management. They reconfigure the system by adding/removing VM instances or by scaling up/downsizing the VM instances at runtime [59, 62, 63, 64, 65, 58]. Among them, the study in [65] considers elasticity rule correlation to some extent. In this thesis, we propose a finer grain approach which not only adds or removes resources when it is required but also reorganizes them (e.g. by changing the active and standby assignment roles) for better resource utilization while taking into account the service availability.

In [65], when the load on VMs increases, the VMs are scaled up. In their approach, they considered action correlation for a specific case where hosted VMs with the same supporting physical node need to be scaled up and the supporting node does not have enough resources for all the requests. To handle such conflicts, authors use VM migration. To choose the candidate VM for migration, they select the one for which the cost and time of migration as well as the release of resources resulted from migration is optimal. The accuracy of their approach depends on the weight they set for different types of resources like CPU, memory, etc.

There has been some research study in which they do not use thresholds as the points for allocating or deallocating resources. Instead, they use prediction techniques which take the previous workload and utilized resources as input and forecast the future workload and resource requirements. In these studies, the previous workload at different time intervals with a fixed window size is analyzed to identify any repeating patterns in the workload [45]. Based on the found pattern, the future workload is predicted. Next, the system is scaled according to the defined policies if it is necessary. The size of the window and the accuracy of the prediction have significant impact on the efficiency of the scaling. These approaches try to solve the problem of when and how much to scale; but they do not specify how the system is scaled or the elasticity rules based on which the system is scaled are general.

In [66], [62] and [63] workload variations are predicted by machine learning. In [62] based on the predicted workload, penalty of violating SLAs and the cost of adding VM instances, it is decided if VM instances should be added or removed. In [63], online machine learning is used as a decision maker to add or remove virtualized network functions. Similar to other online learning approaches, the initial performance when the system is learning can be low. The performance can be worse especially when the workload is not even. Others such as [67] used queuing theory to model the cloud system based on the arrival rate of requests and other parameters such as mean service time (i.e. response time) and CPU load. They use this model to predict the response time or the load in the next time interval.

Chapter 4

Ontology-Based User Requirements Decomposition and COTS Component Selection

In this chapter we introduce a part of our model-driven framework which decomposes high level functionalities and selects COTS components that match the decomposed functionalities in order to meet user requirements.

4.1 Introduction

Service providers often offer a limited set of services that can be customized according to the customer's needs. Let us consider for instance the provisioning of a web server service. One client may require a high bandwidth; while another one may want secure interactions and a third one may prefer COTS components from a specific vendor. Instead of having a service configurator going through the process of decomposing the required service until it can be mapped to COTS components for each customer separately, possible decompositions can be stored in a service ontology. Experience with service decomposition and component selection can be reused.

According to [68], an ontology represents some knowledge about a domain and it consists of classes, attributes and relationships. We are interested in a service ontology to represent the information about the different functionalities that compose a service and their known decompositions/compositions. The important classes in a service ontology are therefore functionalities, compositions and interactions; the relationships define how functionalities can be decomposed/composed and where interactions are required. The ontology is constructed in such a way that the elements at the lowest level of decomposition can be mapped to COTS components. With the help of a service ontology, requirement decomposition and component mapping can be automated. Additionally, alternative component sets can be generated so that later compliance with non-functional requirements can be evaluated for different solutions. Optional functionalities that may not be requested by all customers can be easily included. Note that in our work we only use the ontology to store alternative decompositions to allow addressing a functionality by different names (where model elements usually have unique names used as identifiers), but we do not use the formal semantics of ontologies.

The COTS components that are available for providing the desired functionality are stored in a separate model, as they might change more frequently. For instance, a new version of some software may be released every few months, or available hardware elements may change. By keeping the decomposition separate from the components, it is ensured that the ontology and thus the knowledge about the functionality decomposition can be reused even when the COTS components that are available to provide different functionalities change. In addition, it makes it easier to use different sets of COTS components for different customers.

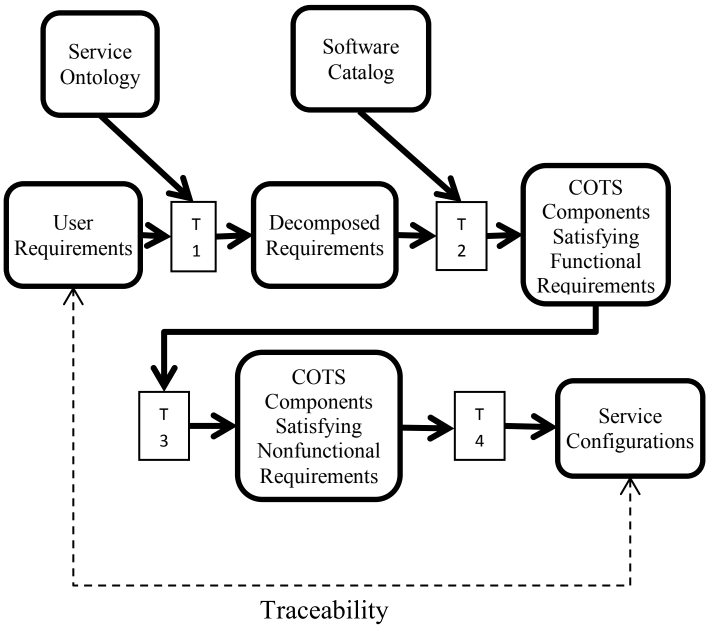


Figure 4.1 Selection of COTS components using service ontology

The overall process of decomposing user requirements using a service ontology and then selecting COTS components is shown in Figure 4.1. UML models are used to describe all the artifacts in Figure 4.1 and the Atlas Transformation Language (ATL) [14] is used for the transformations *T1*, *T2*, *T3* and *T4*.

In [11], a model-driven approach has been developed to derive automatically a valid AMF configuration starting from service configurations model. However, the service configurations model is specific to the AMF domain and far from requirements the user would express. Rather than specifying the requirements in terms of AMF concepts, the user is interested in the service functionality the system will provide and its non-functional properties, like performance and level of availability. The service as perceived by the user may consist of several functionalities. A method to derive service configurations from user requirements (UR) has been proposed in [69]; however, the first step of decomposing the user requirements to a level from where they can be mapped to AMF concepts is left to the service configurator, an expert in the service domain. Moreover, data rate is the only non-functional requirement that is con-

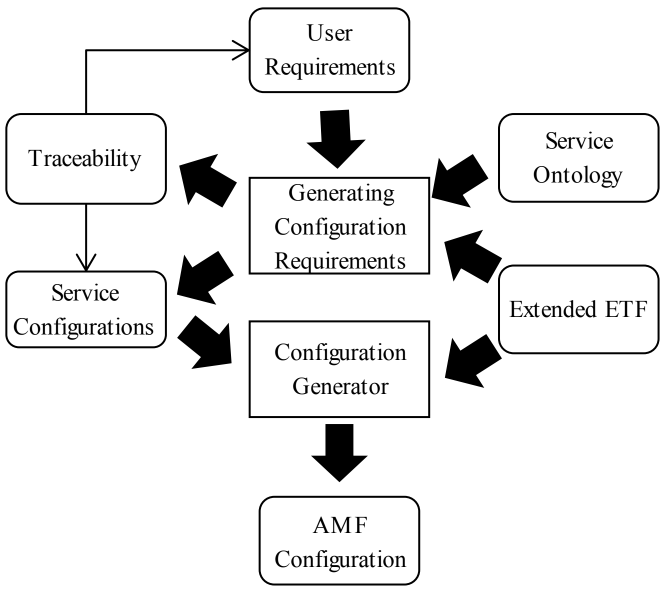


Figure 4.2 Generation of AMF configurations from user requirements

sidered in component selection. As the services requested by different customers are often the same or at least similar, the configurator's work could be simplified and automated by providing a way to store and reuse decompositions of services and functionalities. Therefore, in this thesis, we aim to generate service configurations from available components to satisfy high level user requirements. Figure 4.2 shows the overall picture of the configuration generation.

4.2 Modeling of User Requirements, Service Ontology and COTS components

In this section, the UML model of user requirements, COTS components and service ontology are presented.

4.2.1 User Requirements Metamodel

We model the functional and non-functional user requirements in UML as shown in the metamodel in Figure 4.3.

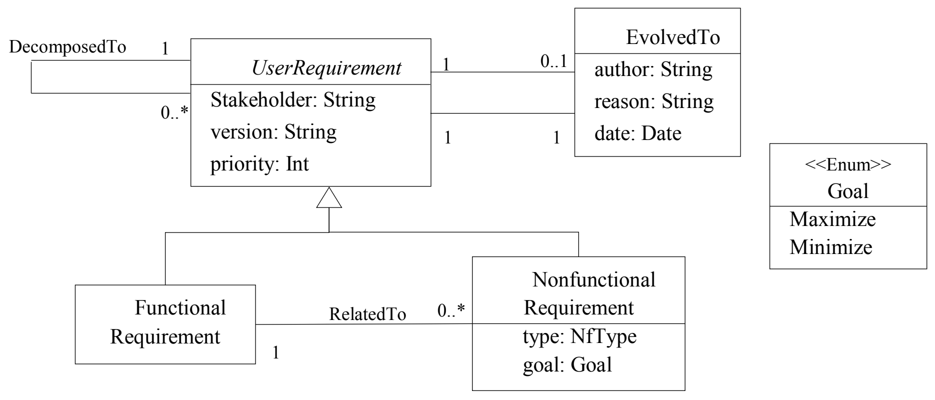


Figure 4.3 User requirements metamodel

Requirements may evolve over time. The evolution is captured by the *EvolvedTo* metaclass. In this relation, it is specified when, by whom and for what reason a specific requirement has changed.

User may decompose functional or non-functional requirements to some extent. It is usually done when for a functional requirement, there are alternative decompositions and user wants to force (a) specific one(s) or to introduce a new decomposition. Non-functional requirements

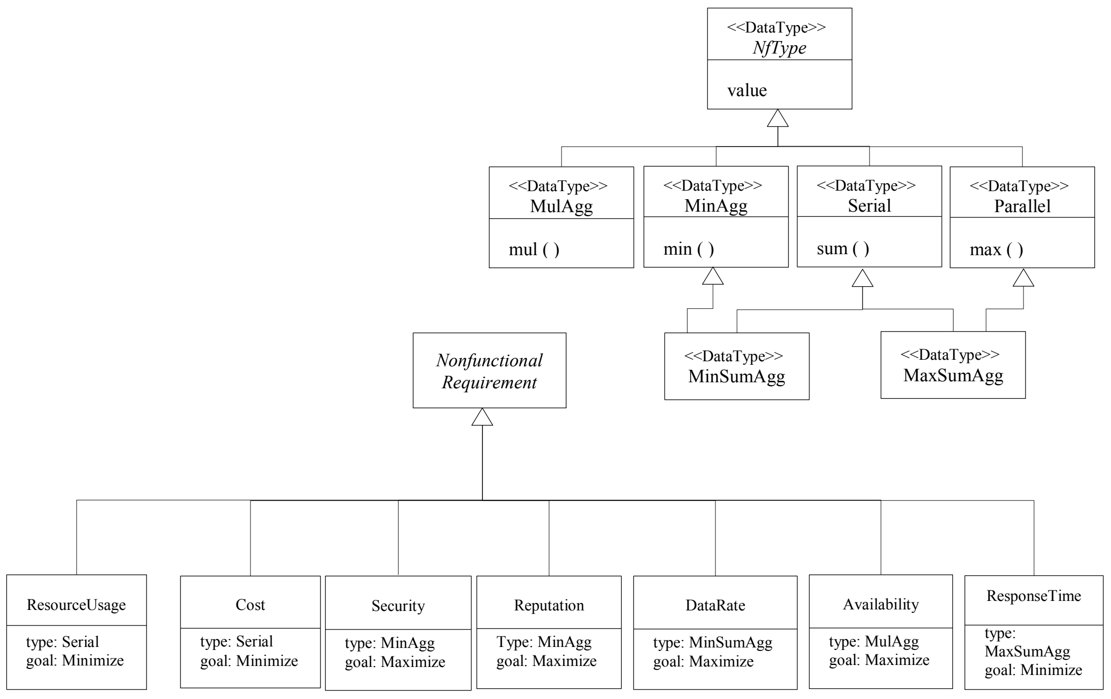


Figure 4.4 Classification of non-functional aspects based on their composition operation

are applied to functional requirements at any level of granularity. Non-functional requirements can be of different types. For example, a customer does not ask for interoperability while a Software-as-a-Service (SaaS) provider who owns the components may request it.

Each non-functional requirement has an attribute named *goal*. This attribute categorizes the non-functional requirements based on their optimization goal. For example for security, a higher value is better and for cost, the lower value is preferable. In Figure 4.4, similar to [70] the non-functional requirements are categorized further based on their appropriate composition operation. For instance, the total cost is the summation of the costs of the composed functionalities. These categorizations are used when components are selected based on the non-functional requirements.

Each non-functional requirement is of a specific data type. A data type can inherit from another data type. In the UML metamodel [18], a data type is a kind of a classifier. Thus the inheritance/generalization relation that is defined between classifiers is defined between data types too. For instance, the type *MaxSumAgg* inherits *max()* and *sum()* operations. The non-functional requirement *ResponseTime* is of type *MaxSumAgg* because depending on the architecture of the system, we have to take the maximum of response times of the sub-systems (if the sub-systems operate in parallel) or sum them up (if the sub-systems co-operate in a serial way, one after the other) or we have to use a combination of the two to obtain the response time for the whole system.

The attribute value can be either a qualitative or a quantitative value. We use Object Constraint Language (OCL) [10] expressions to specify *NFtypes'* operations. For example, the operations for taking the minimum in the *MinAgg* data type and the summation in the *Serial* data type are defined as follows:

```
Context MinAgg :: min (x: MinAgg) : MinAgg
body: If self.value > x.value then x.value else self.value endif

Context Serial :: sum (x: Serial) : Serial
body:  self.value + x.value
```

According to the second operation, if we want to compose two non-functional aspects that are of type *Serial*, then the resulting operation is the summation of their values which is also of type Serial.

An example for an UR model is given in Figure 4.5. For readability purposes, the rectangles represent the functional requirements and the ones with the rounded corners are for the related non-functional requirements. The dotted lines with arrow show the *RelatedTo* relations and the dashed lines without arrow show the *DecomposedTo* relations. The requested service is a triple play, and the user specified the requested functionalities and some elements of the decomposition. Three non-functional requirements were specified for the overall service, namely the maximum cost, the number of customers and the capacity for each customer. Moreover, for one of the composing functionalities, a different capacity is required.

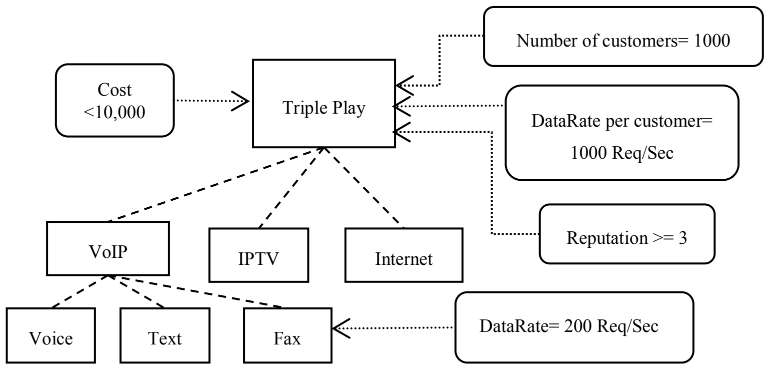


Figure 4.5 Example of user requirements for a triple play service

4.2.2 Extended ETF Metamodel

For describing components in the domain of high availability, ETF is used. The basic entities in ETF are Component Types (CTs) and Component Service Types (CSTs). A CT represents a version of hardware or software and a CST represents the type of the service such component can provide. The CST is different from the service as perceived by a user. A service perceived by a user refers to the functionality that a component provides while a CST defines which attributes of a component have to be configured to provide such functionality [1]. Each service can be provided by different components and the quality of the service that those components can provide may differ. To provide the same service, components might use different types and amount of resources. The components may also provide the same service with different qualities. These characteristics are represented by *Non-functionalCharacteristics* metaclass. The metrics that can be observed by a monitoring system or reported by the component itself are also described in the extended ETF as *Measura-*

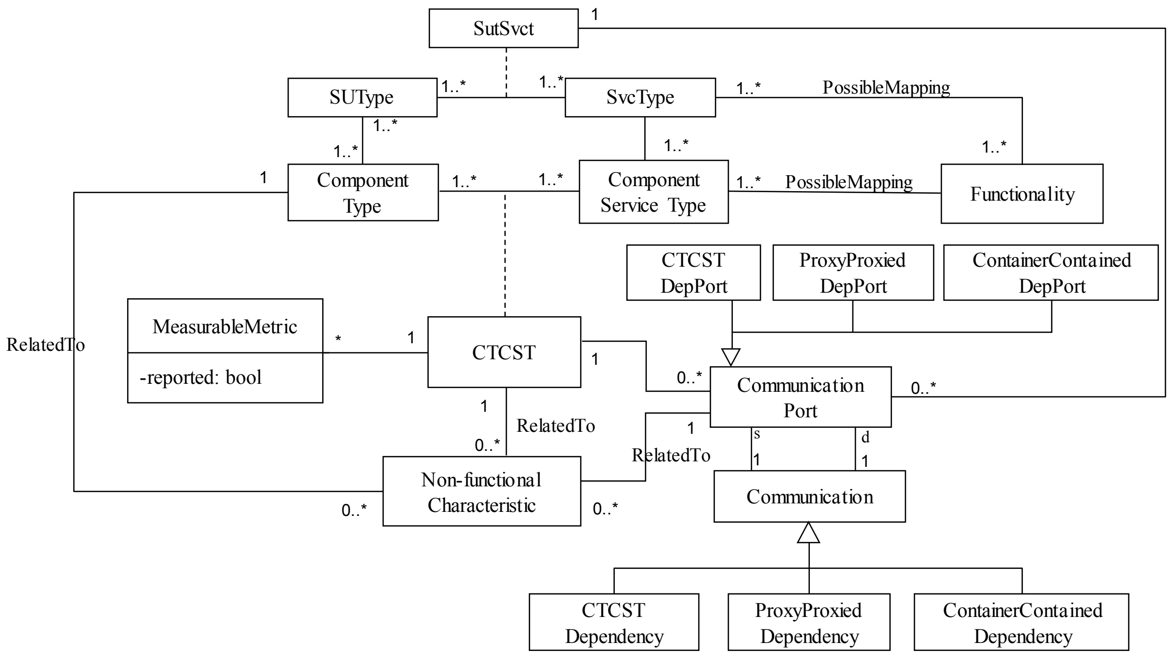


Figure 4.6 A portion of the metamodel for the extended ETF

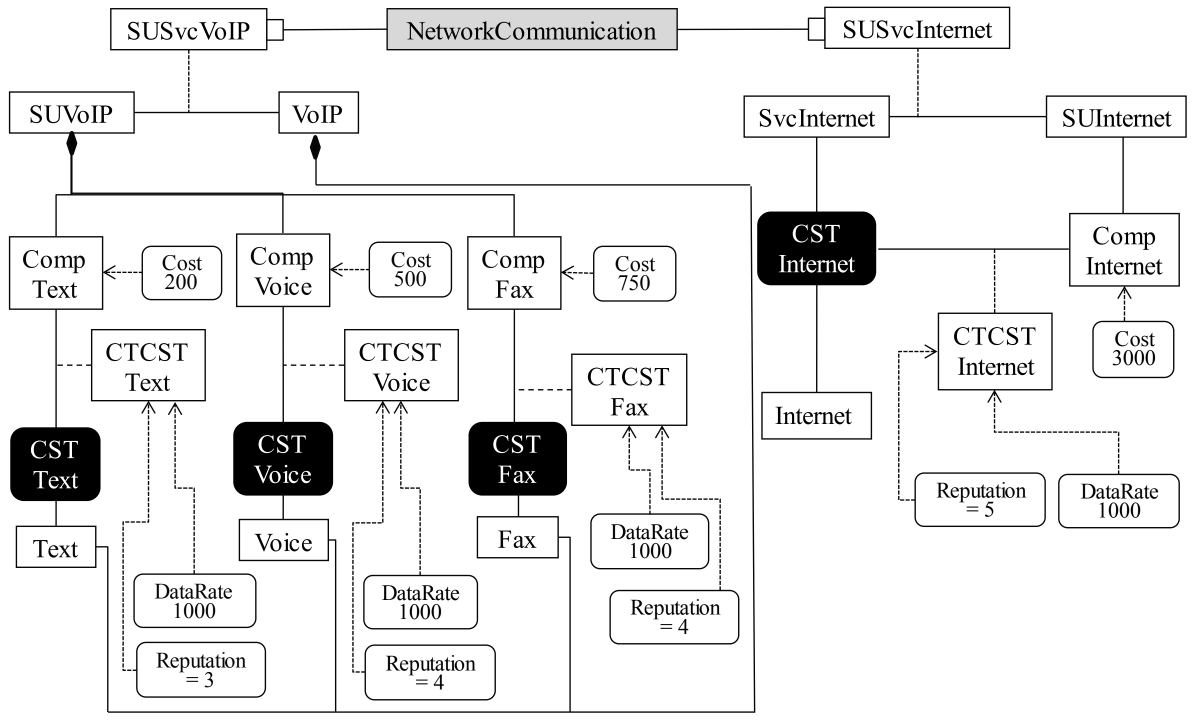


Figure 4.7 A portion of the ETF model for *VoIP*

bleMetric. For the purpose of our work, ETF is extended to include for each CST which functionalities, as perceived by users, it can provide and with what non-functional characteristics.

The relationship between CTs and CSTs is also given in the ETF as CTCST association class [17]. This includes both information on component's capability as active or standby. The *Communication* and *CommunicationPort* elements are used to specify the communication capabilities of a CT when providing a CST [69]. Service provider entities in providing their services can communicate among each other by exchanging data. Besides data exchange, there can also be other dependencies between that a component requires another one to function. Components can be combined with other components as service units to obtain a bigger functionality referred as a service type (shown as *svcType* in the figure). Part of ETF metamodel is shown in Figure 4.6. Figure 4.7 shows an example of ETF model for *VoIP*. For the purpose of transformation, the models of all available ETF files are combined into one ETF model.

4.2.3 Service Ontology Metamodel

The ontology model represents the domain knowledge. Decompositions of functional user requirements are stored in a service ontology. The reason for keeping the knowledge of decomposition is that users may request for same/similar service functionalities over time. For example, one user may prefer to have a component from a specific vendor and the other one wants to have it with more capacity. Moreover, users may request a high level functionality that its decomposition cannot be found in the ETF. Therefore, we keep the knowledge of the decomposition to reuse it for similar requests.

A domain model for ontology is shown in Figure 4.8. Functional requirements can be decomposed in several hierarchies, and for each requirement, several alternative decompositions may exist. Alternative decompositions may have common decomposed functionalities because some sub-functionalities are not strictly needed. The main usage of the ontology is for decomposition. However, it may store additional knowledge about the mapping between

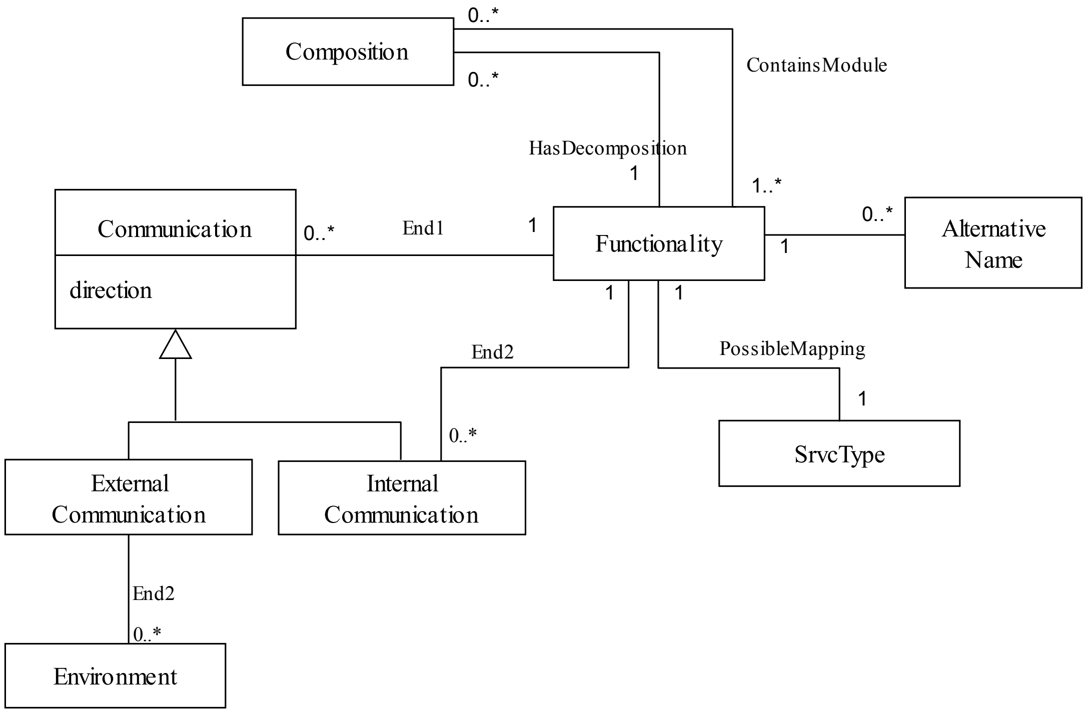


Figure 4.8 Metamodel for the service ontology

functionalities and the service types if it is not specified in the ETF how a specific functionality can be combined with others to make a higher functionality. Functionalities at the lowest level of decomposition (leaf functionality in the ontology model) are mapped to CSTypes. This mapping is not specified in service ontologies as this is already contained in the ETF. A functionality that is not leaf in the ontology can be mapped to either a service type or CST because we may have either a component or a set of components in the software catalog that can provide the composed functionality.

The primary source for constructing a service ontology is ETF because in the ETF, the vendor may specify how CSTs can be grouped together to obtain a higher service as a service type. However in ETF, the groupings of functionalities are specified at most at two levels. We keep further possible grouping of functionalities in the service ontology. The ontology will be enriched during the configuration generation process by storing successful decomposition automatically or by a service configurator by hand. Because similar/same functionalities may be requested over time, we store the knowledge of the decomposition as ontology to reuse this knowledge especially when the decomposition is not specified.

An instance model of ontology metamodel is shown in Figure 4.9. The main classes are functionalities and the compositions between functionalities to identify alternative decompositions. The decomposed functionalities are the contained module of the higher level functionality. To obtain a functionality, it may need to interact with environment or with other functionalities. Whenever a new COTS component becomes available, the software catalog (i.e. ETF) and ontology will be updated as it may add a new functionality or decomposition to the ontology. Similarly, if a component is no longer available, the software catalog will be updated. If by removal of a component a functionality is no longer offered, the service ontology should be updated accordingly. In the model in Figure 4.9, the decomposition for telecom-

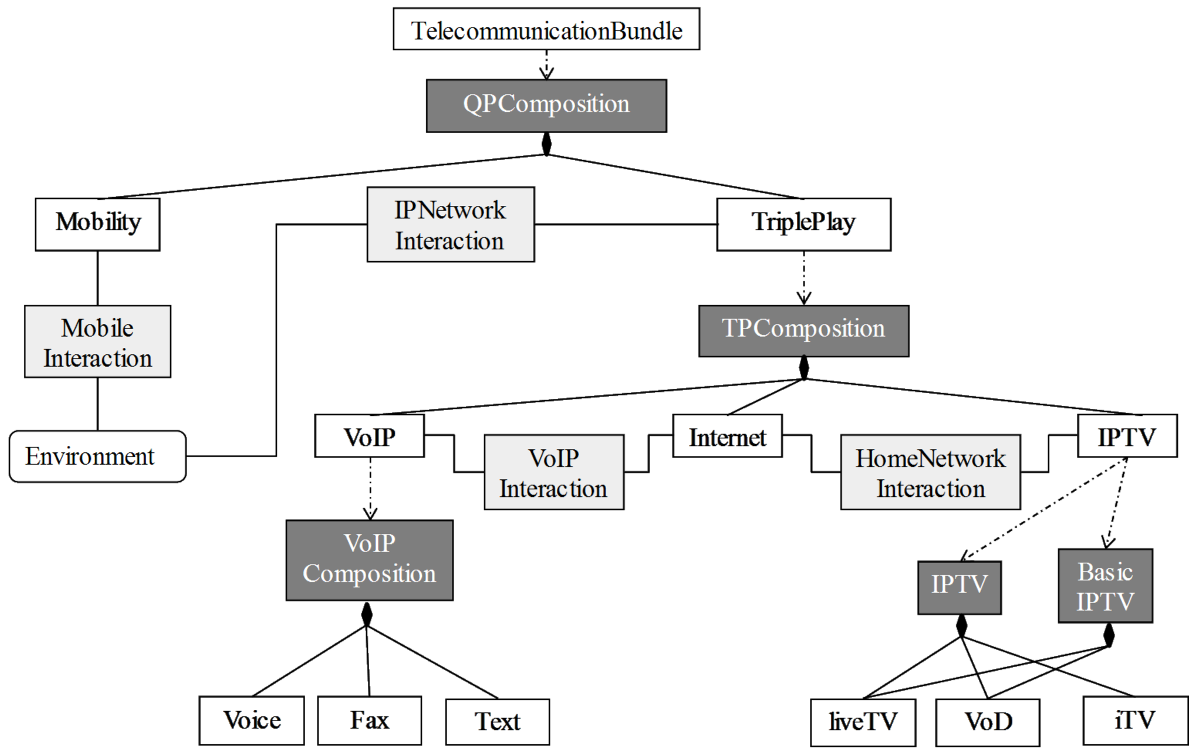


Figure 4.9 An example of service ontology for a telecommunication bundle

munication bundle is shown. The telecommunication bundle consists of the mobility and the triple play services. Both interact with the environment. The functionality *IPTV* which is a decomposed element of *TriplePlay* functionality needs interaction *with* internet service. There are two alternative decompositions for the IPTV functionality: *BasicIPTV* and *CompleteIPTV* with a number of common functionalities but *IPTV* also includes the *iTV* element.

4.3 User Requirements Satisfaction and COTS Components Selection

In this section, the steps toward the selection of components that meet both functional and non-functional requirements are explained. We have divided the process of decomposing the user requirements and generating service configurations into four main model transformations as shown in Figure 4.10. We used ATL [14] to implement the transformations. In this figure, the transformations are shown with dashed rectangles. Additional transformations

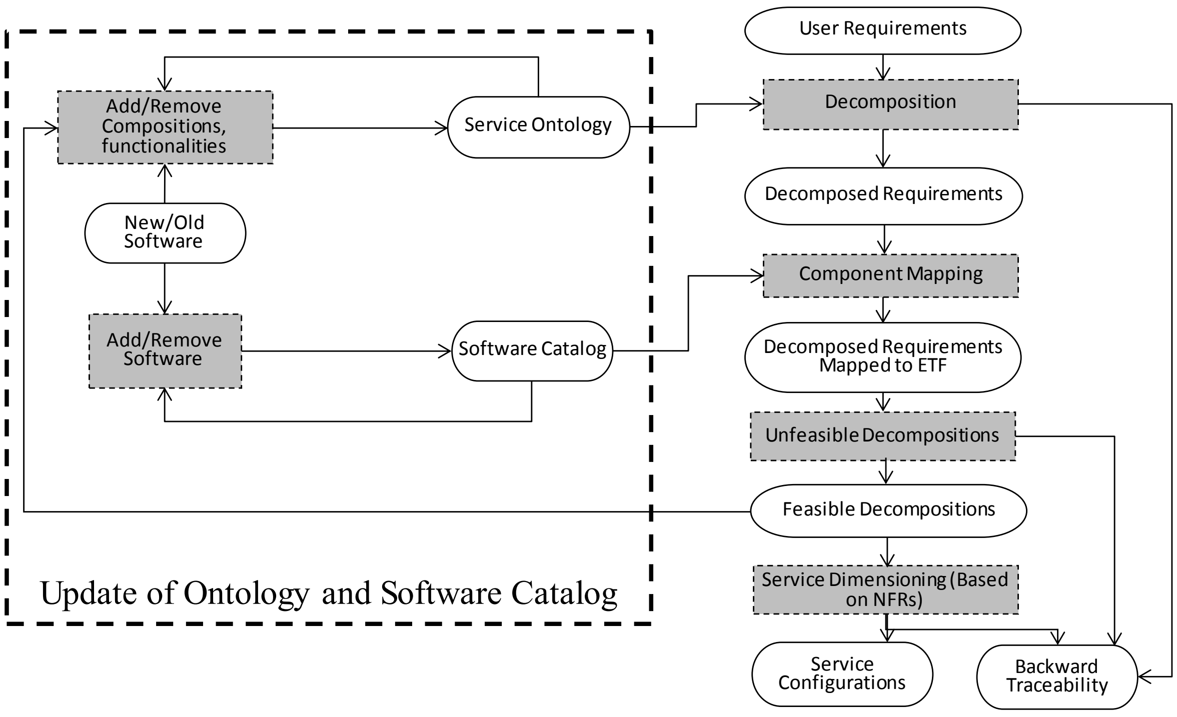


Figure 4.10 Model transformations for decomposing user requirements and mapping to ETF components

were defined to update the ETF model and ontology by new and obsolete ETF files. These updates, shown on the left-hand side, need to be performed first.

4.3.1 Solution Map (SM) Metamodel

To decompose functional requirements, rather than looking at ontology and find the matched functionality and decomposition by hand, we automate it by model transformation. Besides providing automation, it facilitates tracing the decomposed functionalities and selected components. The transformations combine the information of three models: user requirements, ontology and ETF to do the decomposition and find the component mappings. Because the ontology is partially derived from ETF, then it is consistent by construction. The resulting model corresponds to a metamodel, the Solution Map (SM) metamodel, that is a combination of the three input metamodels. The SM metamodel is shown in figure 4.11. In this figure, the functional requirement represents the functionality which is also in the ontology or it is the decomposed functionality that is brought from the ontology.

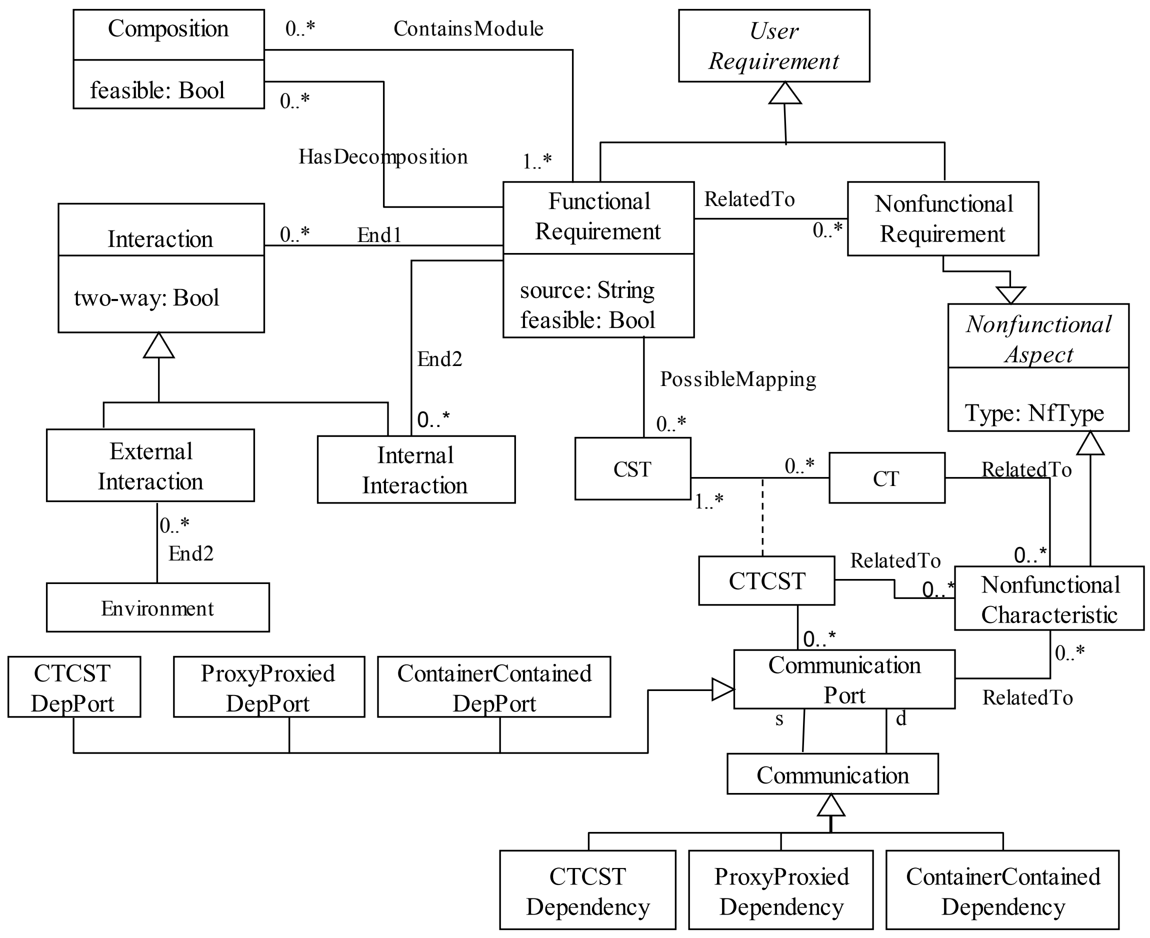


Figure 4.11 The SM metamodel combining the information from the user requirements, ontology and extended ETF

When a functional requirement, either a leaf or higher level functionality, matches the CST that a CT can provide, we have a potential mapping for the functionality which is represented by the *PossibleMapping* association. This mapping means that in the software catalog there is at least one component type that can support the functionality. The non-functional characteristics of a service (CST) provided by a CT and the non-functional requirements requested by a user derive from the same concept and they represent a non-functional aspect. For example, taking cost as a non-functional aspect, the former represents that a CT can provide a CST with a specific price, while the latter represents the cost requested by a user.

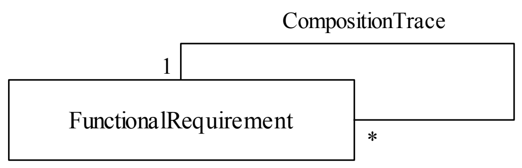


Figure 4.12 Part of the traceability metamodel for traces obtained in the decomposition process

4.3.2 Decomposition of Functional Requirements by Service Ontology

To decompose functional requirements, user requirement model will be combined with ontology model. In order to combine, we only bring the parts into the SM model for which we could find a match in the ontology model. The process of decomposition is done in two steps. In the first step, the user requirements will be transformed into the solution domain which means the *DecomposedTo* relations in the user requirement will be replaced by *Composition* classes. When the decomposition process is being done, backward traceability links between the composed functionality and its decomposed elements are created in a separate model called traceability model. In this model, the roots of the traceability model are the leaf functionalities in the ontology and the leaves are the leaf functionalities in the user requirements model. In addition, a backward trace link between each leaf functionality and its corresponding root (i.e. the leaf functionality in the user requirement model) is created. This traceability model will be refined in next steps. Figure 4.12 shows the metamodel of traceability created in this step.

Considering example in Figure 4.5, for the first step, the composition classes for *TriplePlay*, *VoIP* and will be added. In the second step which is the core transformation for decomposition, the refined user requirement will be combined with the ontology. This means a match for *TriplePlay* functionality will be found in the ontology and it is checked if it is a leaf functionality or not. Although *TriplePlay* and *VoIP* functionalities are decomposed already by the user, we still start the decomposition from the root of user requirements model because the decomposition by user may not be complete. In this case, the decomposition would be completed by the ontology. For example, if the decomposed *VoIP* didn't have Fax functionality as its decomposed element in the user requirements model, then it would be added in the decomposition process. The decomposition process is a recursive process and it will be stopped when all functionalities in the solution are leaf functionalities in the ontology. If for *IPTV* user only had specified *iTV* as its decomposed element, then in the decomposition process, the alternative *BasicIPTV* decomposition would be discarded.

After combining user requirements and ontology, the SM model will be refined by the addition of information from ETF model to see which CTs are able to provide the functionalities. Similar to the decomposition process, all the information from the ETF model will not be brought to the combined model. In this transformation, all functional requirement nodes of the SM model are of interest. For each of these nodes, it is checked whether any CST in the ETF model provide the needed functionality. If it is the case, then all such CSTs are added. They are added as *PossibleMapping* associations from the functionality to the CST. Next, for each CST in the SM model, all the CTs that provide it are added to the SM model together with the appropriate CTCST associations and the non-functional characteristics of the provided service. It is also checked if any CT needs to interact with another CT to provide its CST. If so, the sponsor CT as well as its CST and their CTCST association (if they are not already added because of ontology) together with the communication, communication ports and non-functional characteristics of the ports are added to the SM model. For functionalities which are not leaf, it is checked if the functionality has mapping with service type or CST. For a functionality which is not leaf, it is possible to have mappings to an CST and a service type. This means we have two alternative solutions which one solution is the combination of some component types and the other solution is only one component type that can provide the requested functionality.

A functional requirement can be met by more than one component type providing the same or different CSTs. In this case, they are alternative solutions. It is likely that they have different non-functional characteristics, and the most appropriate one, that provides the best match with the user requirements, needs to be selected. In some cases, no matches can be found in the ETF model. It means that no available component type can provide this functionality which causes the corresponding decompositions not to be valid.

The result of the transformations for triple play is shown in Figure 4.13. As seen in this figure, there are three COTS component types *CompVoD1*, *CompVoD2* and *CompVoD3* that support *VoD* functionality. For functionality *iTV*, no match in the ETF could be found. Therefore, the functionality *iTV* along with the composition *CompleteIPTV* should be marked as infeasible. This is the purpose of next transformation. In Figure 4.13, the non-functional

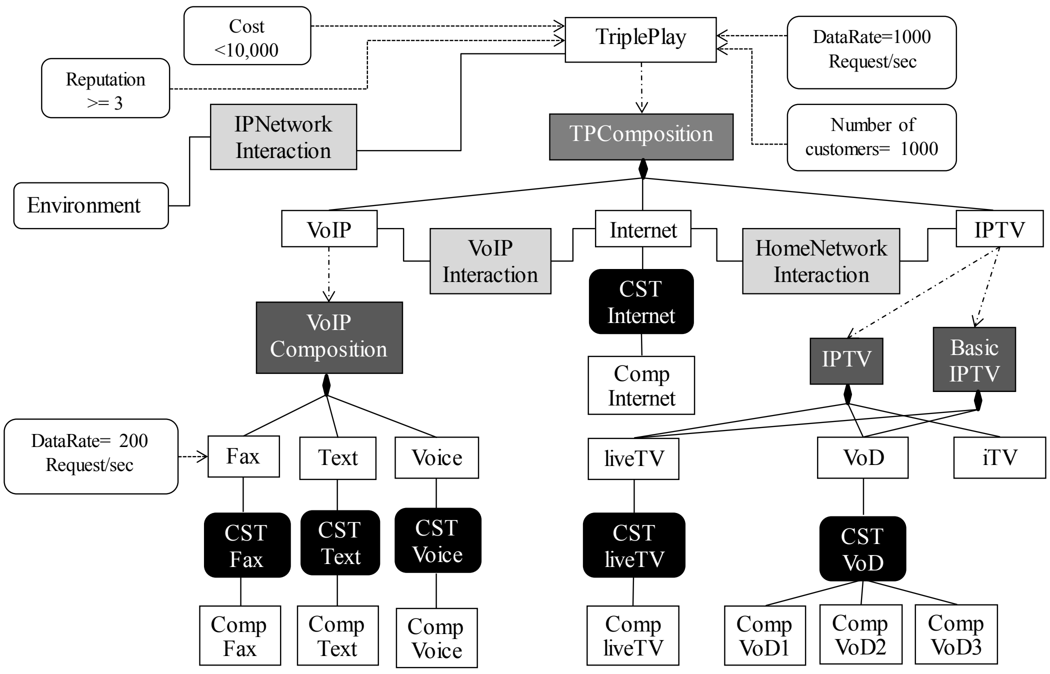


Figure 4.13 The SM model for the *TriplePlay* example of Figure 4.5

characteristics of component types, ports and communications are not shown.

4.3.3 Infeasible Decomposition Marking

In this step, any leaf functionality that no available component type can provide should be marked as infeasible. The compositions that have this functionality must be marked infeasible too as they are the part of the composed functionality. To detect infeasible decompositions, we have defined a helper which checks if a leaf functionality has a mapping. When this helper returns false, the corresponding functionality is infeasible. This helper is defined as follow:

```
helper context MM!FunctionalRequirement def: Boolean =
HasDecomposition.allInstances()
->select (HD|HD.supplier->includes (self))-> isEmpty()
implies
PossibleMapping.allInstances()->exists (PM|PM.endType()->includes (self))
```

This process is recursive. First, all leaf functional requirements without a mapping are marked; then all compositions that contain infeasible functionalities are marked. Functionalities that are not at the leaf level are marked as unfeasible if and only if all of their decompositions are infeasible. For the triple play example in Figure 4.13, only two elements are marked as infeasible. The first one is *iTV* functionality, as it does not have any mapping to a CST. As a consequence, the composition *CompleteIPTV* also has to be marked as unfeasible. The parent of this composition, *IPTV* however, does not have to be marked as unfeasible, as it has an alternative decomposition, *BasicIPTV*, which is feasible.

Once the feasible decompositions are identified they can be used to enrich the ontology. Namely the composition elements that appear in the user requirements model but not in the ontology are added as new compositions to the ontology. This is a transformation similar to the addition of compositions based on a new ETF file. By identifying the infeasible decompositions, the corresponding functionalities from the traceability should be removed because the traceability model keeps the information about the solutions.

4.3.4 Service Dimensioning

4.3.4.1 *Separation of Different Candidate Solution*

Combining user requirements, service ontology and ETF models results in a model with alternative decompositions and where the mapping of decomposed requirements to available CTs is captured. The high level requirement in the user requirements model can be satisfied by different combinations of CTs when there are alternative decompositions, or functionalities with mappings to alternative CTs in the solution map. In the next step, different combinations of CTs that satisfy high level user requirements are identified and separated. We need to separate different solutions (i.e. different sets of components that can be used to provide the required functionality) because later we want to investigate each candidate solution separately based on the non-functional requirements and choose the optimal one that best meets all the requirements.

Candidate solutions that can support the highest level functional user requirement are complete solutions; while the ones that support its constituent functionalities, i.e. they support a part of the functional requirements in the composition tree, are represented by partial solutions (see Figure 4.14).

As seen in the candidate solution metamodel in Figure 4.14, a solution can be a partial or a complete solution and a solution may contain other solutions as well. However, there is an OCL rule that forces this containment relation to be between partial solutions, or between a complete solution and partial solutions as its containments. The solutions in the metamodel represent sets of components that satisfy a functional requirement and the non-functional re-

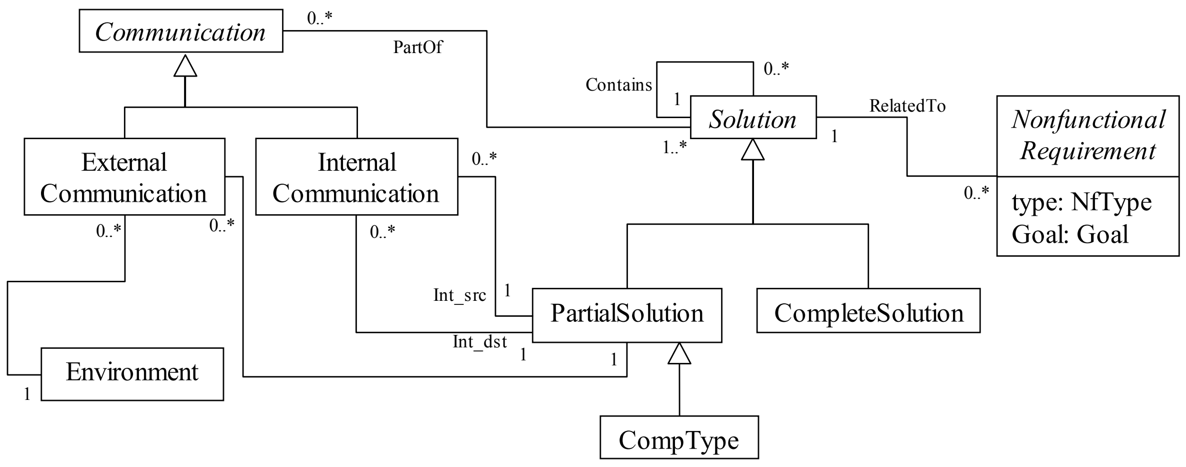


Figure 4.14 The candidate solutions metamodel

quirements that were attached to functional requirements are now attached to the corresponding solutions. In the separated solutions model, the interactions can be either part of a solution or between two partial solutions, for instance communication between two component types.

To calculate the number of complete and partial solutions, the SM model is traversed bottom-up. First, the leaf functionalities that are mapped to CTs are considered. For each functionality at that level, it is checked how many CTs can support it. The number of CTs is the number of solutions for the leaf functionality. The next layer in the decomposition tree consists of composition elements. For a composition element, all possible combinations of the functionalities that it contains have to be considered. Therefore, for a composition element, the number of alternative solutions is the product of the number of partial solutions of its composed functionalities. For a higher level functionality i.e. one that is not a leaf, the number of alternative solutions is equal to the sum of the solutions for its alternative decompositions plus the number of direct mappings to CTs if there is any. The number of complete solutions is determined when the top node in the decomposition tree is reached.

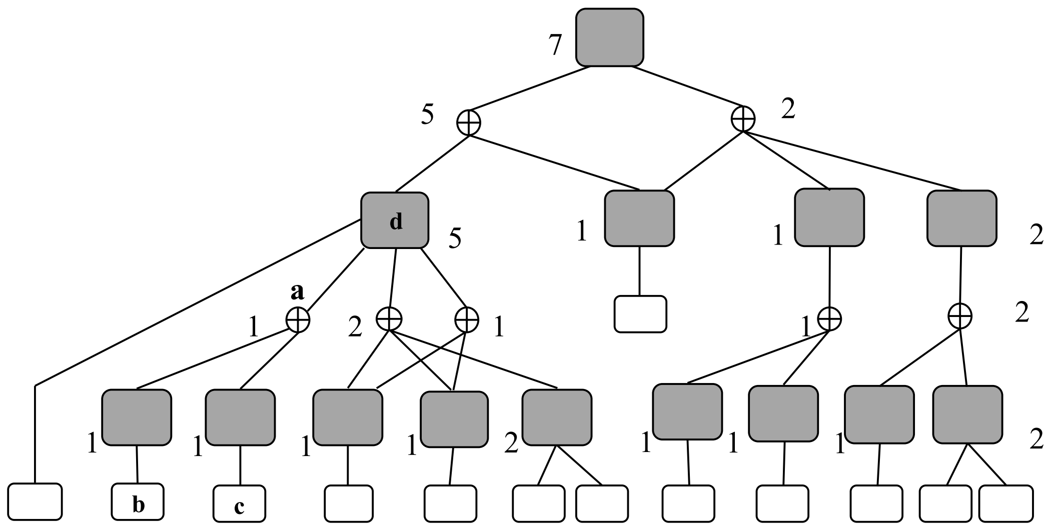


Figure 4.15 Recursive computations of candidate solutions

An example of this calculation is illustrated in Figure 4.15. In this figure, functionalities are shown in gray squares with rounded corners, CTs in white and composition elements are shown with circles with crosses. The numbers next to the functionality and composition elements indicate the number of solutions for them. For instance, for the left-most composition (named *a* in the figure) each child functionality has exactly one solution. Therefore, there is only one solution corresponding to the composition *a* (combining the solutions of both composed functionalities, i.e. the two left-most CTs named *b* and *c*). For the functionality *d* in the next layer, the number of partial solutions is equal to the summation of: partial solutions for its alternative decompositions which is four; plus one solution obtained from a direct mapping to a CT.

To keep track of which solution corresponds to which functionality, trace links between these elements are added to the traceability model. Figure 4.16 shows the metamodel for these traceability links. According to the metamodel, a functionality can have multiple solutions but a solution traces back only to one functionality. Complete solutions trace back to the high level functionality at the root of solution map model and partial solutions to other functionalities at lower levels of the hierarchy.

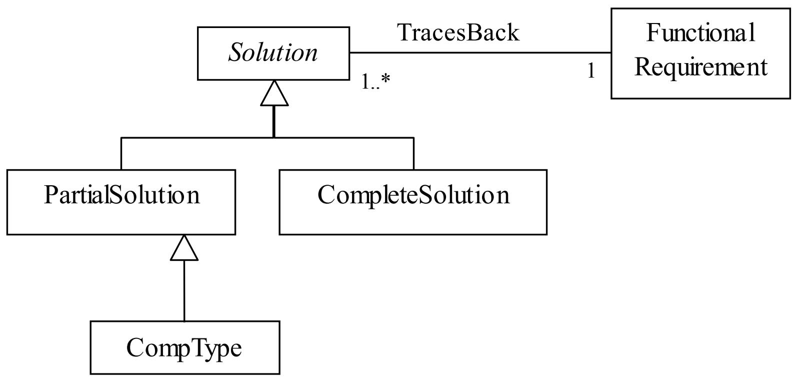


Figure 4.16 Metamodel for tracing between solutions and functional requirements

If we consider the feasible decompositions for the triple play example in Figure 4.13, there are three alternative component types (i.e. three alternative solutions) that support the *VoD* functionality, each with different non-functional characteristics. For instance, the reputation for component types *CompVoD1*, *CompVoD2* and *CompVoD3* is 4, 2 and 3 respectively. Therefore, there are three complete solutions for the *TriplePlay* functionality. The result of this is shown in Figure 4.17. For the purpose of readability, the solutions are represented differently. The complete solutions are represented with dotted gray rectangles, partial solutions with dotted white rectangles and functionalities are shown as white rectangles. The commu-

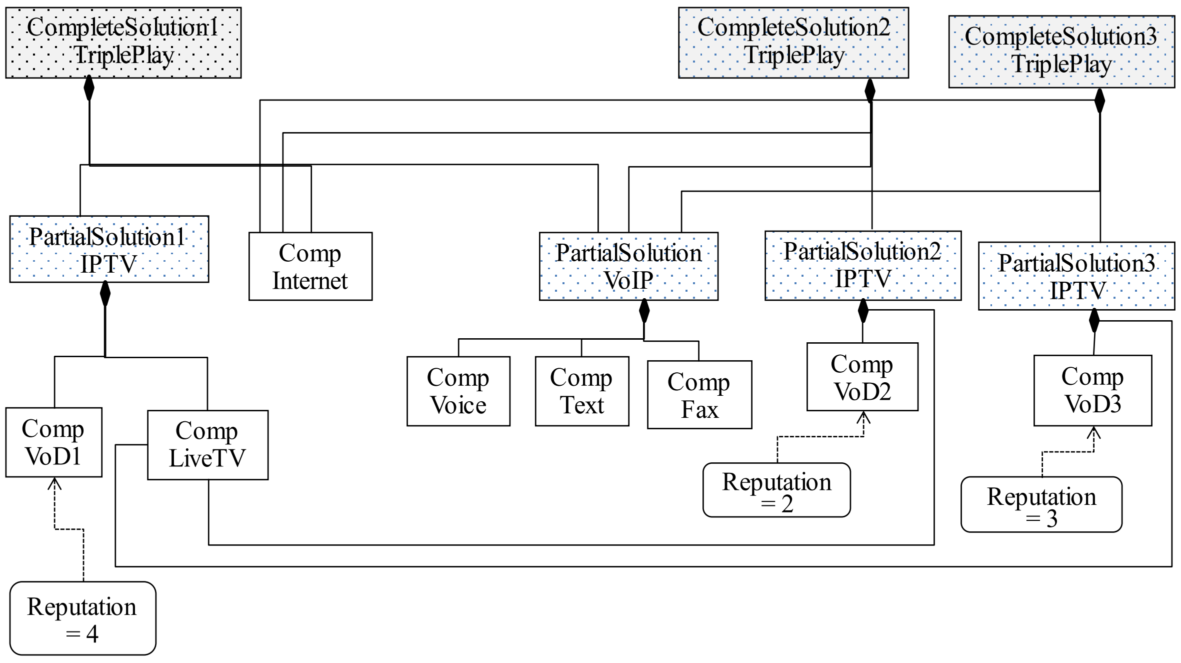


Figure 4.17 The separated solutions for the *TriplePlay* example of Figure 4.5

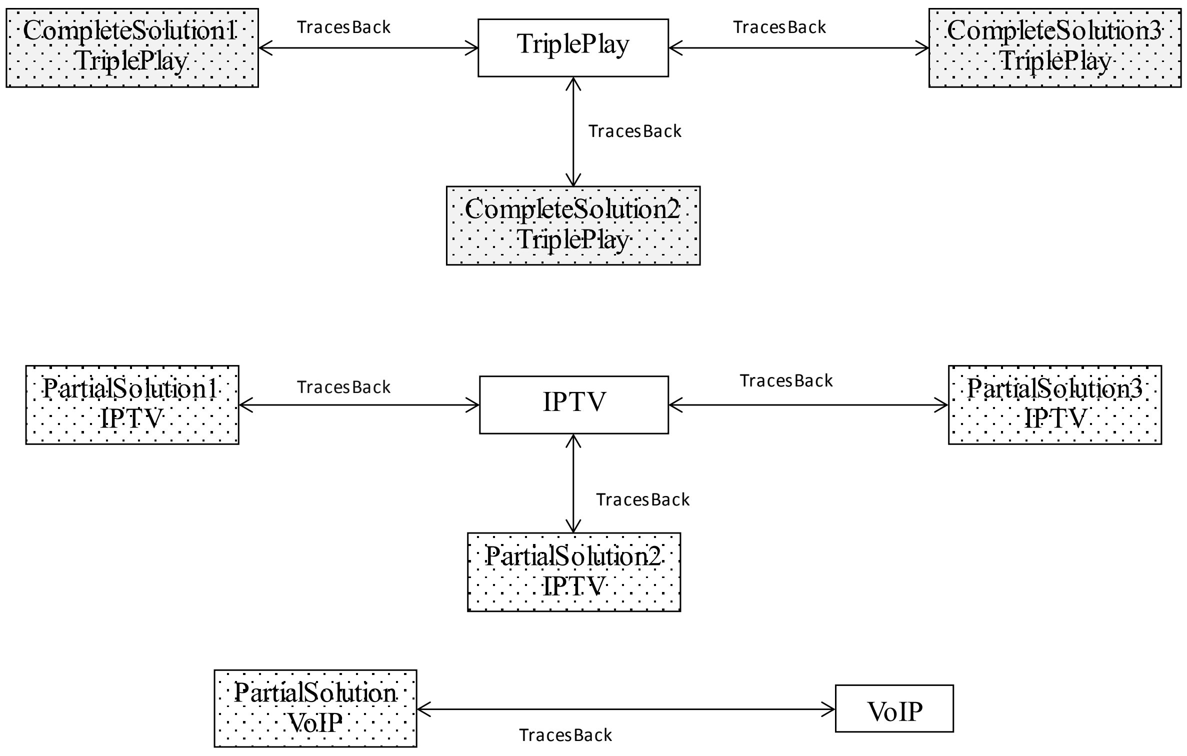


Figure 4.18 Traces between solutions and functionalities for the *TriplePlay* example of Figure 4.5

nications between the solutions (i.e. component types) are not shown in this figure.

The traces between the different solutions and functionalities are shown in Figure 4.18.

4.3.4.2 *Checking Non-functional Requirements*

After separating the different candidate solutions capable of supporting the requested functionality, each solution now has to be checked with respect to the non-functional requirements. Non-functional requirements are attached to solutions of any size, as they can be attached to functional requirements at any level. That means that while some of the non-functional requirements may only apply to a single component type, others apply to a set of component types.

Not all non-functional requirements can be taken into account at this point. Some of them like *Cost* or *Availability* depend on the number of components used in a configuration and we can only do a preliminary check for them at this point. However, for other non-functional requirements like *Security* or *Reputation*, it is enough to know which component types are being used to see if they can be fulfilled. It is worth mentioning that security can also be a functional requirement which in this case the requirement is a function which enforces security such as the login functionality. In this thesis, we consider security as a non-functional requirement.

To proceed we need to determine the non-functional requirements for each solution. Even though a non-functional requirement may be associated with a functional user requirement at the highest level only, all functionalities composing that requested functionality are also constrained by that non-functional requirement. The non-functional requirements therefore have to be distributed over all decompositions of functionalities. For a number of non-functional requirements, the distribution is straightforward: A non-functional requirement associated with a functionality is also associated to all its decomposed functionalities. An example for this is *DataRate*. If a functionality is requested to handle a specific rate of requests, then all the elements that contribute to this functionality are requested to handle that rate as well. This includes the decomposed functionalities and thus the selected component types that are mapped to those decomposed elements. For other non-functional requirements like *Cost*, *ResponseTime* or *Availability*, the distribution is different. If the functionality should not exceed cost *X*, then the sum of all costs for all the decomposed functionalities should not exceed *X*. At this stage, where only the component types are chosen and not the number of components from each type, compliance of a solution with *Cost* cannot be fully determined yet. However, we can already dismiss a solution if the sum of the costs of all component types exceeds *X*.

To check component types against the non-functional requirements, OCL constraints have been defined. Later, these constraints are used to define ATL rules [14] in the implementation. Instead of defining separate rules for each non-functional requirement (i.e. having separate rules for *Reputation*, *Cost*, etc.), the definitions of rules are based on the goal and type attributes of the non-functional requirements. When a new non-functional aspect is introduced and needs to be considered for checking, there is no need to add a new ATL rule based on the new kind as long as it fits into one of the categories we have defined for the non-functional aspects.

For non-functional aspects like *Reputation* that we aim to maximize, a solution is accepted if the non-functional characteristic of each component type in the solution is equal or greater than the requested value. Otherwise, all solutions that contain this component type will be dismissed. The constraint for checking non-functional aspects with the goal of maximize is as follows:

```
Context Solution
RelatedTo->allInstances()->
select(r|r.supplier->includes(self))->collect(c.client)->
forAll(c|c.goal=Goal::Maximize        implies        c.value        <=
(RelatedTo        ->        allInstances()->        select(s|s.supplier->
includes(c.ct))->collect(c.client)->
select(s|s.oclIsTypeOf(c.appliedStereotype()))->
collect(s.value)->at(1)))
```

The rule for non-functional aspects with the goal of minimizing them is similar to the above. The violation of a rule means that all solutions that contain this component type should be marked as insufficient.

If we consider the separated solution in Figure 4.17, the minimum requested reputation is 3, while the reputation for *CompVoD2* is 2. Therefore, *PartialSolution2VoD* and *CompleteSolution2TriplePlay* that contain *CompVoD2* are marked as insufficient solutions.

The maximum number of requests per second that can be handled by a component type is specified in the extended ETF as its *DataRate*. A component of this component type needs to be assigned some workload in order to provide this service. This workload is configured as an instance of the related CST. The component type may allow for multiple workload assignments. This data rate applies for each assignment that is assigned to a component. When a solution is validated against the non-functional requirements like *Security*, *Reputation* and partly against *Cost* and *Availability*, the minimum workload necessary to be assigned to the component types is calculated so the requested data rate can be met. The number of assignments for each component type is calculated as follows:

$$\#assignments_{Component\ type} =$$

$$\text{No of customers} \times \left\lceil \frac{Required\ Data\ Rate \text{ for each customer specified } in\ Requirements}{Provided\ Data\ Rate\ by \text{ the } Component\ Type} \right\rceil \qquad (4.1)$$

For example, if the number of customers is 1000, the number of requests for each customer (i.e. DataRate) is 350 requests per second and by each workload assignment, the corresponding component type can handle 200 requests per second, to be able to support these requests, $1000 \times \left\lceil \frac{350}{200} \right\rceil = 2000$ assignments are required.

The minimum set of assignments and SIs required to satisfy the user requirements form the service configurations used as the starting point for the configuration generation in [11].

4.4 Traceability

4.4.1 Traceability Metamodel

The traceability model is the integration of all traces that are obtained in the different transformation steps. The metamodel for traceability is shown in Figure 4.19.

Bidirectional traceability links between user requirements and configuration solutions allow stakeholders to see why certain component types were chosen and where the requirements are

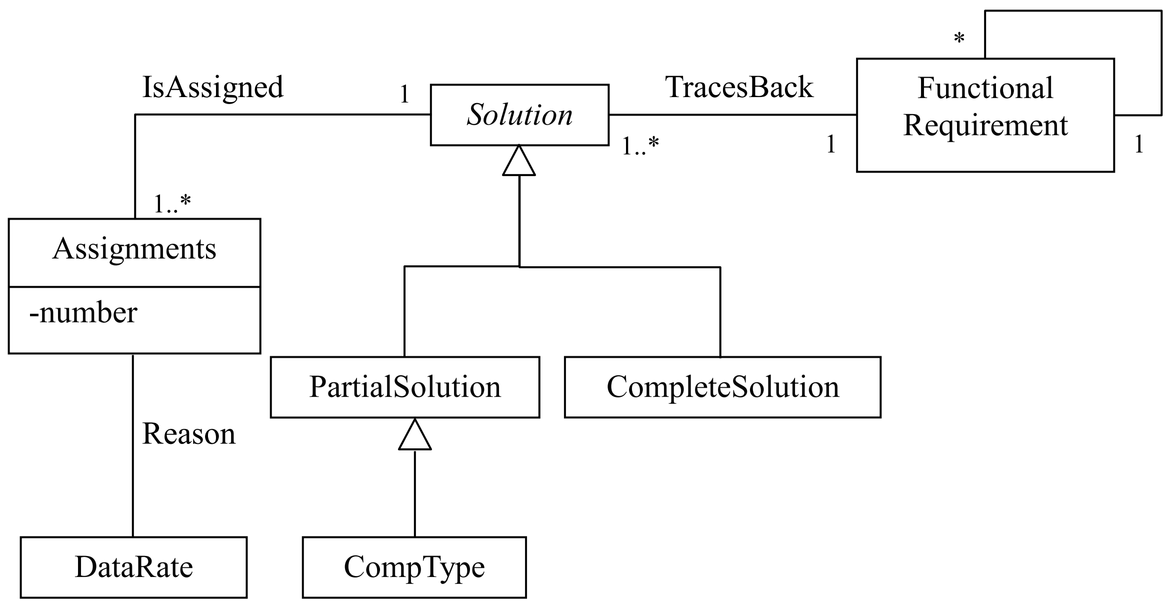


Figure 4.19 The complete traceability metamodel

implemented in the configuration. Traceability links ease the management of requirement evolution as not the entire process of component selection needs to be repeated, but only those parts that trace to the modified requirement(s). For example, suppose the customer decides to increase the data rate to support more requests. With the help of this model, the designer would see what the affected component types are and then the number of workloads that should be assigned to those component types can be recalculated accordingly. In another case, suppose that a component type is no longer supported by a vendor and the configurator wants to see what other possible solutions exist for the affected functionalities. As the traceability model keeps the traces between functionalities and all possible component types, it is easy to find the mapped components and the related data rate that should be considered for workload calculation.

If a non-functional requirement other than rate changes, the previously selected component types may not be acceptable any more. In this case, the affected functionalities will be found by the *RelatedTo* relations in the user requirements model. Then using the traceability model, the component types mapped to the affected functionalities can be determined. The mapped component types can be checked against the new requirement to see if we need to change them or not. If they need to be changed, then new sets of component types can be found for the affected functionalities and the number of their related SIs is calculated. In this case, the traceability model is updated with the new sets of component types and the minimum number of SIs to be assigned to. In this case, the traceability model is useful especially when the affected functionality is big with many hierarchies of decompositions. In this case, there is no need to do the decomposition as traceability has the traces between this functionality and its constituent atomic functionalities. Therefore, the mapping between functionalities and the components can be found by the traceability model without redoing the decomposition and component mapping.

The created traceability model keeps the trace of non-functional requirement *DataRate* and the number of assigned workloads. The non-functional requirements *Security* and *Reputation* will not be brought in this model; because when a component type is in traceability model, it

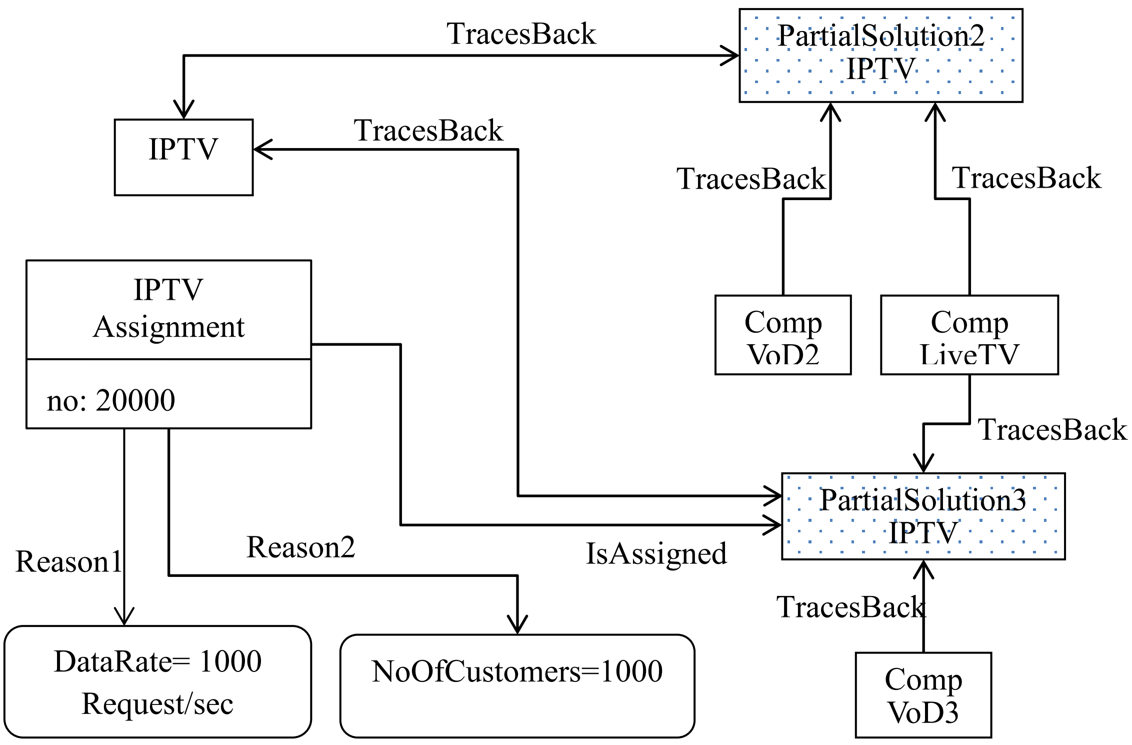


Figure 4.20 Part of the traceability model for the *TriplePlay* Example of Figure 4.5

means that it meets the requested *Reputation* and *Security* otherwise it will be in the insufficient solutions model (see next section). Other non-functional requirements like Cost and Availability that affect the number of instances will be added to the traceability in the next steps of AMF configuration generation.

Figure 4.20 shows a portion of the traceability model for the example in Figure 4.5.

4.4.2 Insufficient Solutions Metamodel

If a solution is dismissed, due to missing component types, non-functional requirements that cannot be satisfied, or other reasons, it is kept in the insufficient solutions model along with the reason for its dismissal to indicate which requirement is not satisfied by the solution. For some solutions, the reason may not be severe and they can be considered as sub-optimal solutions. For example, if a solution has been dismissed because of a requirement with low priority was not met, then we may consider it as a solution if we cannot find any other valid solution. This model may help with the management decisions. Suppose, for instance, that no valid configurations can be found and the reason is the underestimated cost. One may decide to

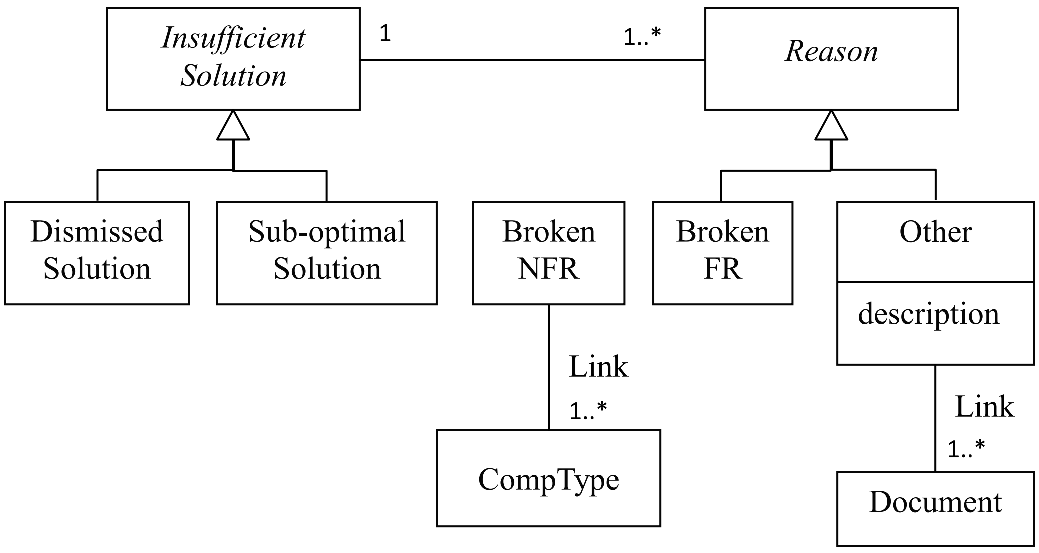


Figure 4.21 The insufficient solutions metamodel

increase the budget to allow for higher cost and find a solution.

The metamodel for insufficient solutions is shown in Figure 4.21. A solution is insufficient because of a non-functional requirement that a component type cannot meet or because of a functional requirement that no available component type can support. If there are other reasons like the functionality will be available on a specific date, then they should be documented.

4.5 Summary

In this chapter, we have presented an approach to determine sets of COTS components that can satisfy both high level functional and non-functional user requirements. We used the extended ETF model to describe COTS components of the domain of high-availability. However, our approach is general and can be applied to other domains using other domain-specific component models. Model-to-model transformations are used to implement the whole approach.

In the first step, the high level functional user requirements are decomposed to the level where they can be matched with functionalities provided by COTS components. The knowledge about the decomposition of different functionalities is specified in a service ontology and the functionalities provided by the available component types are described in a separate model.

While the mapping of functionalities to component service types is an essential step in the component type selection process, it is also necessary to pick based on the non-functional requirements the component types and other entity prototypes combining them. Some component types might not be able to function on their own but require the presence of other component types as indicated in the extended ETF metamodel of Figure 4.6 by the dependencies. These dependent component types also have to be included in the final configuration to be valid and deployable.

After mapping functional requirements to component service types, alternative solutions for providing the overall requirement are separated and evaluated. The evaluation is based on the available component types and whether or not their non-functional properties are sufficient to ensure that the user's non-functional requirements can be met. In the process the service configurations are generated as a set of assignments and service instances that the configuration will need to provide. With determining the service configurations, this work can be integrated with previous work that derives AMF configurations from service configurations [11]. We also aim at generating elasticity rules at the same time as the configuration generation. In the next chapter, we explain the generation of the elasticity rules when the service side and service provider side of the system is dimensioned.

When a valid set of component types that satisfy the user requirements is selected, a traceability model between the selected component types and the requirements is automatically generated. This model can be used to manage the evolution of the user requirements as well as the configuration. However, if no combinations of available component types can satisfy the user requirements, the generated traceability for insufficient solutions can help the designer to find out which requirements and for what reason are not satisfied.

Chapter 5

Design Time Elasticity Rules Generation

In this chapter first we describe the integrated configuration generation process and then, we introduce our approach for automating the generation of elasticity rules during the configuration generation process.

5.1 Introduction

To scale a system dynamically, actions are taken according to a set of defined patterns, called elasticity rules. An elasticity rule may provide different actions that are applicable and can be performed in different situations. An elasticity rule is generally invoked by a trigger, which is generated in reaction to a monitoring event. Elasticity rules could be defined online with the help of agents that watch and learn about the behavior and the usage of the system using machine learning techniques [42], [43] and [44]. The drawback of these approaches is that if the learning agents learn fast, they may learn problems such as Denial of Service (DoS) [71] attacks easily and therefore, they may incorporate the problems in the elasticity rules as well. In this case, the elasticity rules should be updated and the system needs to be brought back in its normal state. Since the relation between the generated elasticity rules and configuration is not clear, identifying the problems in the elasticity rules and updating them is not easy. In other words, such online approaches do not guarantee that the system evolves correctly within the designed boundaries. Moreover depending on the learning method used, it may be difficult to come up with the appropriate reward function. In this thesis, we propose an approach for defining elasticity rules offline, more specifically at system dimensioning time. That is, at the time when the system is dimensioned and configured to provide the required services with the expected highest workload in mind. The elasticity rules of our approach are generated based on the configuration generation process. As a result, the relation between the generated elasticity rules and the configuration is clear and therefore, we can trace the elasticity rules if it is necessary.

The dimensions of the system, i.e. the maximum eligible number of entities in the system are determined according to some formulas based on the characteristics of these entities, their relations and the maximum required workload to service. We propose to capture this information in the form of equations and use them not only for system dimensioning and configuration but also for the definition of the elasticity rules that will govern the system dynamic reconfiguration within the dimensioned scope. Since at runtime several elasticity rules may be invoked simultaneously, it is important to have an elasticity rule structure that allows action correlation to avoid conflicting reconfiguration actions. Offline generation of the elasticity rules is the main contribution proposed in this chapter with the use of a model driven approach.

The remainder of this chapter is organized as follows. The metamodel for describing elasticity rules is presented in Section 5.2. In Section 5.3, the combined process of configuration and the elasticity rule generation is explained. We conclude this chapter in Section 5.4.

5.2 Elasticity Rule Metamodel

The manipulation of configuration entities at runtime to adapt them to the workload changes is achieved by applying elasticity rules.

In our approach, elasticity rules are defined for entity types because instances of the same type share the same features and are subject to the same actions. Figure 5.1 shows the meta-model for the elasticity rule description. The metaclass *EntityType* specifies the type of the configuration entities the elasticity rule applies to.

An elasticity rule may consist of different actions, each applicable and feasible in a specific situation. The applicability of an action is defined with a Boolean expression represented by the *Condition* metaclass in the elasticity rule metamodel. At runtime, for an action to be ap-

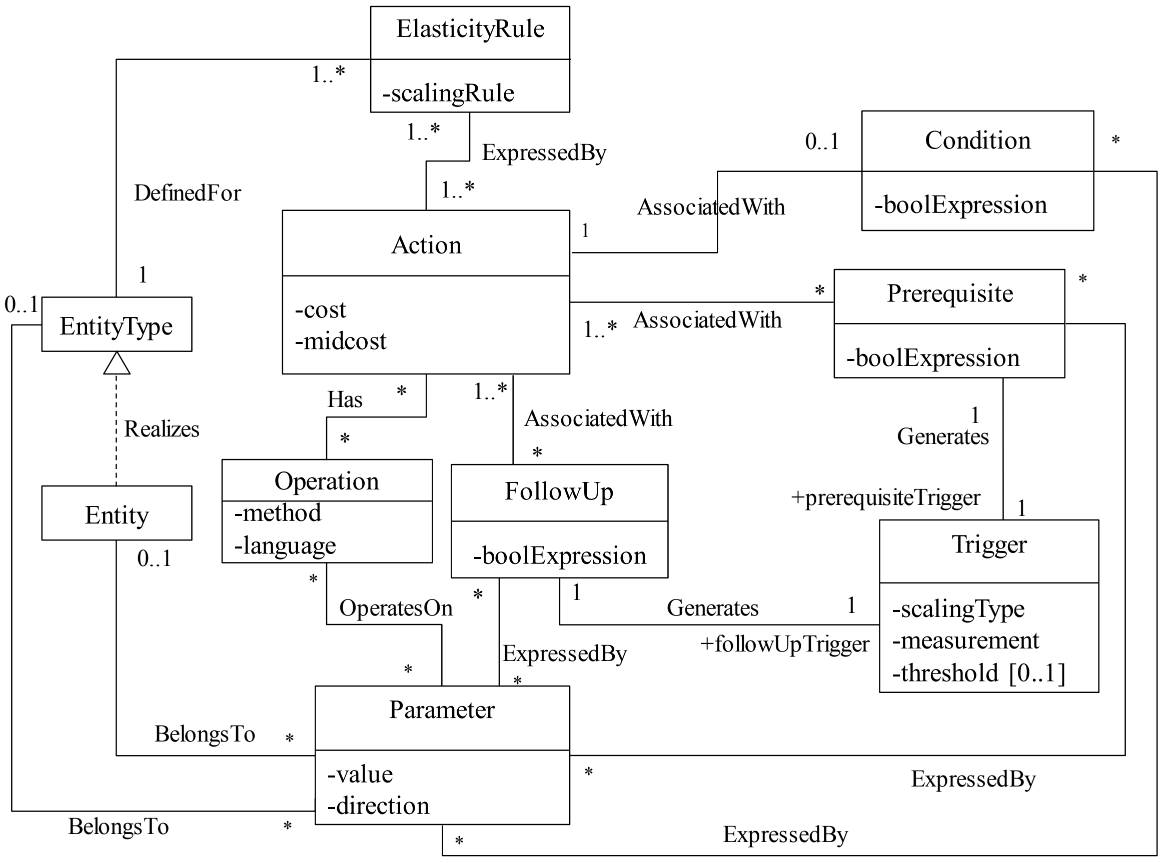


Figure 5.1 The elasticity rule metamodel

plicable and therefore considered for execution, its associated condition must evaluate to true. For example, to scale up a resizable VM, the condition checks if the VM has not reached yet the maximum capacity that it can expand to. If the VM has reached the maximum capacity, the scale up action is not applicable and is not considered for execution.

An applicable action may not always be feasible. For instance, even though the VM has not yet reached its maximum capacity it cannot be expanded for the lack of resources in the hosting node which it depends on. If it is possible to free up some of those resources then the VM can also be resized. Prerequisites are defined to check the feasibility of an action. A prerequisite evaluating to false could be satisfied by taking first actions of other elasticity rules on sponsor entities. In this case, a trigger to invoke the prerequisite elasticity rule is generated for providing the required sponsor resources first. Since prerequisite triggers always initiate the allocation of prerequisite resources, the *scalingType* of these triggers is always *Increase*.

After the execution of an action, a follow-up trigger may be generated to invoke an elasticity rule to execute a follow-up action on the sponsor. For instance, after the removal of SIs or assignments, a follow-up trigger may be generated to initiate an elasticity rule to remove any provider entity without assignments. A follow-up trigger is generated when the *scalingRule* of the executed elasticity rule is *Decrease*.

An action contained in an elasticity rule is an operation, which has a method specified using a language. We use the OCL [10] for expressing the method of the operation and the Boolean expression of the conditions, follow ups and prerequisites in the elasticity rule.

The method of an operation and Boolean expression of a condition, follow-up and prerequisite contain a number of parameters. These parameters belong to the entity type of the elasticity rule or its entities. The values of some of these parameters are set during configuration generation process while others are obtained at runtime from the monitoring system or the configuration.

Each action of an elasticity rule has a cost. The attribute *midCost* represents an approximate cost of the action and its value is the median of the minimum cost (where all the prerequisites are met and it is cost of the given action) and the maximum cost (where none of the prerequisites are met and all prerequisite actions are invoked). The *midCost* for an action is calculated as part of the elasticity rule generation process. Recursively all the prerequisite elasticity rules are generated with their actions to calculate it.

5.3 Simultaneous Generation of Configuration and Elasticity Rules

5.3.1 The Extended Configuration Generation Approach

As mentioned before, elasticity rules are used to reconfigure the system according to the workload variations. Therefore, there is a tight coupling between the elasticity rules and the configuration. Thus, we propose to use the configuration generation process to generate not only the configuration but the elasticity rules as well. We propose to generate the elasticity rules based on the equations used for dimensioning the system that determines the number of configuration entities on the service side as well as on the service provider side.

The extended configuration generation process is shown in Figure 5.2. The user requirements, the service ontology and the software catalog (i.e. ETF) are used as input for the process. First the functional user requirements are decomposed (using the service ontology) to the level they can be matched with component types available in the software catalog that can support these functional requirements. These are the candidate component types and at this stage they are "prototypes" as they allow for different deployment options.

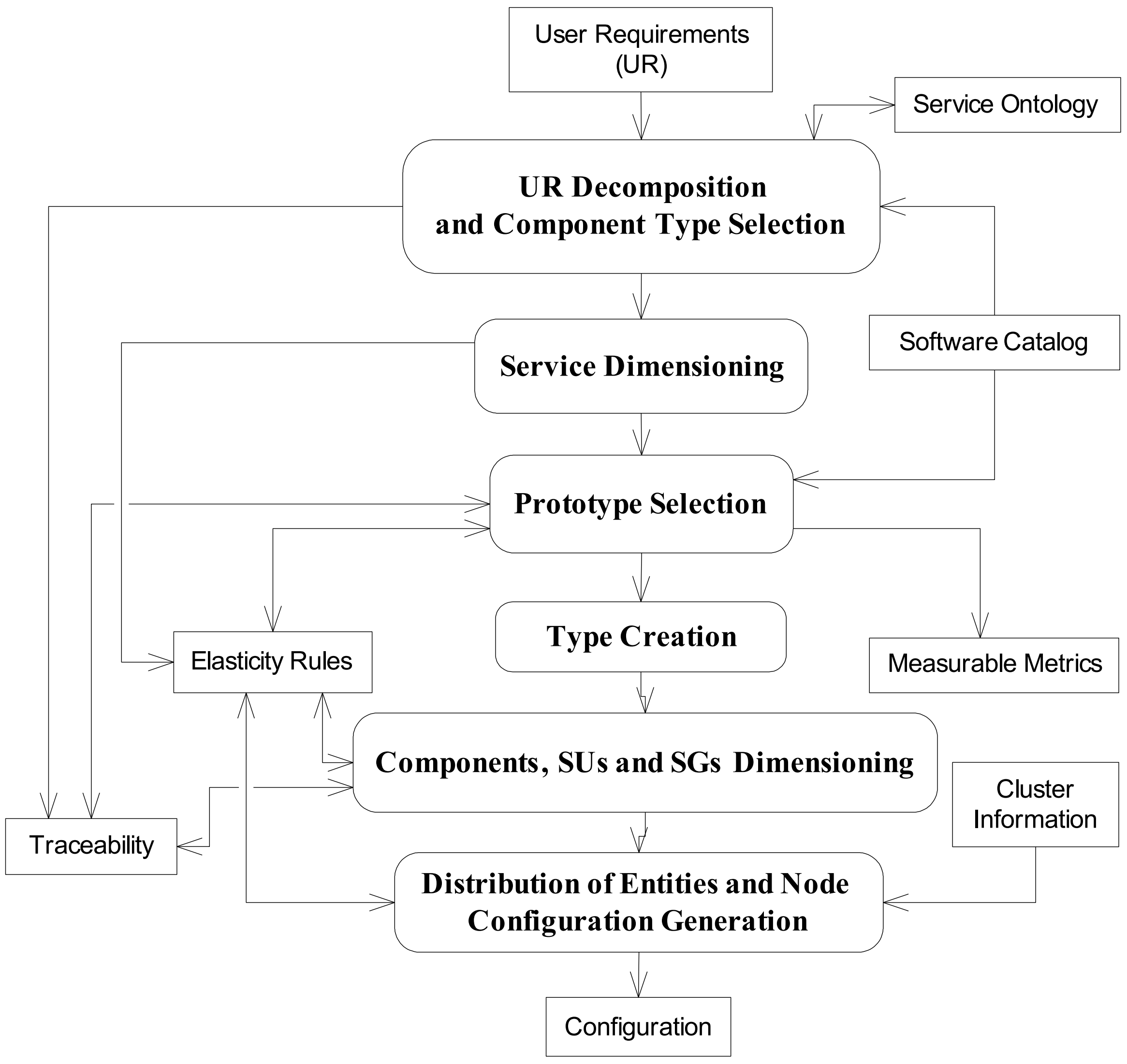


Figure 5.2 The extended configuration generation process

The maximum workload the system should be able to handle is one of the non-functional user requirements. It determines the service side of the system configuration in terms of the active capacity and needs to be expressed as the number of active assignments of the different service types. The *Service Dimensioning* step considers the candidate component types and the maximum requested workload to support, and as part of the service side dimensioning calculates the number of active assignments for each service type using appropriate equations.

In our approach whenever an equation is used for calculating the number of instances of an entity type two elasticity rules are generated for the entity type: one with the scaling type *In-*

crease and one with the scaling type *Decrease*. The equation used for the calculation is transformed into the methods of the actions of the generated elasticity rules while the variables of the equations are transformed into the parameters of those methods. Further details of the elasticity rule generation are explained in the next sub-section. To trigger these elasticity rules, thresholds are also generated.

A threshold represents a point at which some actions should be taken to reconfigure the system. For example, when at runtime the workload for a service instance of service type "A" reaches its maximum threshold, the elasticity rule with scaling type *Increase* needs to be invoked for service type "A". By executing the action of this elasticity rule, the equation based on which this action was defined (and was used to calculate the number of assignments of the service type) is re-applied with the parameters reflecting the current workload. As a result, the required number of active assignments for service type "A" is recalculated for the current workload. By changing the number of active assignments, the service side capacity of the system is reconfigured. Note that to actually perform this reconfiguration prerequisites may need to be satisfied.

The *Prototype Selection* step considers the candidate component types and service unit types and their SG types if defined in the software catalog for selecting those that can provide the requested service availability, another non-functional user requirement. Based on the availability estimation methods of [11], the candidate service unit types that cannot provide the service types with the requested level of availability are removed together with the elasticity rules generated for the service types it supports. It is expected that the prototypes in the software catalog (i.e. ETF) are described by their vendor(s) in terms of their performance, availability and other characteristics as well as their monitoring facilities that can be observed by a monitoring system. In this step, for the remaining candidate service unit types, these metrics are extracted from the software catalog and captured in a measurable metrics model. The generated measureable metrics model can be used later to specify the monitoring agents of a monitoring system.

In the *Type Creation* step, from the candidate prototypes offering different deployment options, fully specified types are created for deployment by parametrization, as well as missing service group types are added with an appropriate redundancy model. This in turn determines whether a service instance of a service type can have one or many active assignments.

In the *Components, SUs and SGs Dimensioning* step, the required numbers of components, service units and service groups are determined based on the previously calculated number of active assignments representing the maximum requested workload. There should be enough service groups and service units of the system to provide and protect all the active assignments. Therefore, this step involves the grouping of active assignments into SIs, adding the standby assignments necessary for the redundancy model and calculating the service provider entities for all. The relation between the service side capacity and the service provider side capacity is often not 1:1 as the latter includes the active capacity as well as the capacity required for the protection of the active assignments by standbys and spare service units. Since this step calculates the number of SGs and SUs using equations we also generate the related elasticity rules as discussed at the service side. It is described later in more details.

In the last step, the required number of nodes (either physical or virtual) is calculated and entities among the nodes are distributed [72]. The distribution guarantees that SUs of the same SG are configured on different virtual and physical nodes and this is maintained also in case of node migration [73]. In this step, the node configuration is generated. Moreover, the initial states of the entities (i.e. locked or unlocked) and their related thresholds are set. In this step, the elasticity rules for the nodes are also generated.

In the user requirements, the workload that the system should support is specified as a range (e.g. minimum and maximum number of requests per second). To generate the configuration entities, the maximum workload that the system should handle is considered. However, generating the configuration for the maximum workload does not mean that all entities of the configuration are instantiated in the system. When a service unit is not assigned any SI, it is removed from the system by "locking" it to reduce the resource/power consumption. The service unit remains in the configuration, but it is said to be in the "locked" state and accordingly terminated or powered off. In contrast, when such a service unit needs to be assigned some workload again it is added back to its SG (i.e. reconfiguring the SG) by "unlocking" it. Thus, its state becomes "unlocked" resulting in instantiation or power up and thus available to provide services. We can also add (remove) an entire SG or a node to (from) the system. In this case, the state of the SG/node changes from "locked" to "unlocked" (from "unlocked" to "locked"). Similarly, on the service side, an SI can be in the "unlocked" or "locked" states depending on whether the chunk of workload it represents needs to be assigned or not. As a result, generating the configuration for the maximum workload means that with all these entities instantiated (i.e. healthy and in the "unlocked" state) the system can handle the maximum workload according to the SLAs. This represents the configuration boundaries therefore the boundary thresholds which represent the maximum limit of resources are set at this point. These do not change.

Figure 5.3 shows some configuration entities at runtime. The availability of the service is maintained by assigning multiple active assignments (shown with green lines in the figure) to different service units of the protecting service group. The "unlocked" and "locked" service units of *ServiceGroup$_1$* are shown with solid and dashed rectangles, respectively. Each service unit is hosted on a separate node. In this example, if an assignment is added to *Ser-*

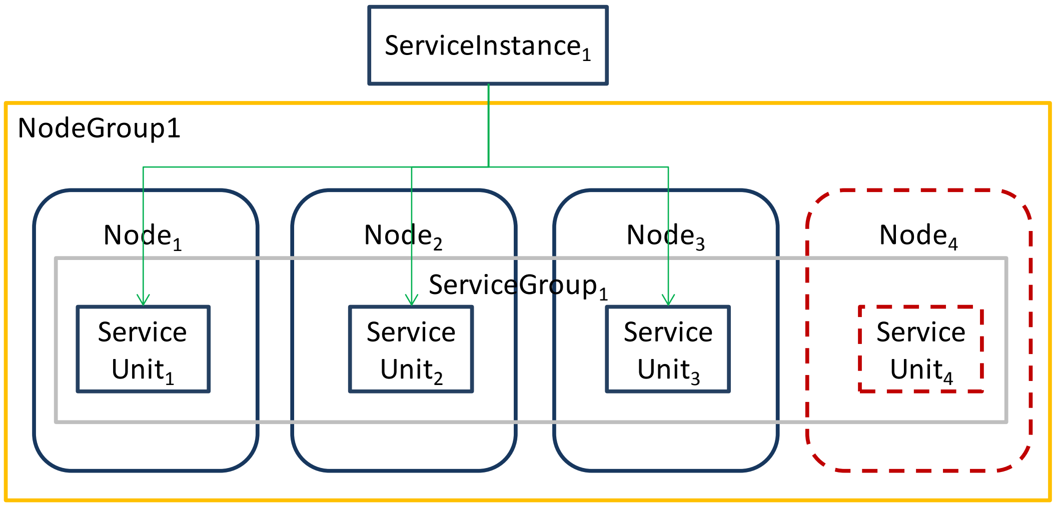


Figure 5.3 An example of configuration at runtime

viceInstance$_1$ due to workload increase, *ServiceUnit*$_4$ and its hosting node are “unlocked” to support the added assignment. Similarly, when one of the active assignments of *ServiceInstance*$_1$ is removed due to workload decrease, the SU and the node that were supporting the assignment are locked to save resources. If any of the service units which provide the service fails, since the workload is shared among other service units with the active role, the service remains available. However, the total capacity is reduced until the failed SU is repaired. If the repair performed by the availability management is not successful, the increased load on the remaining service units is handled by the elasticity management—provided the maximum capacity was not already reached—and the failed SU will be replaced by *ServiceUnit*$_4$. Note that the maximum capacity of the system remains reduced until the failed unit is repaired.

Figure 5.4 shows a portion of the configuration model for the example in Figure 5.3. As shown in Figures 5.3 and 5.4, there is one SI (i.e. *ServiceInstance*$_1$) of *ServiceType*$_1$ which has three assignments and is protected by *ServiceGroup*$_1$. The redundancy model of this service group is N-way active and each service unit of *ServiceGroup*$_1$ can handle at most one active assignment at a time.

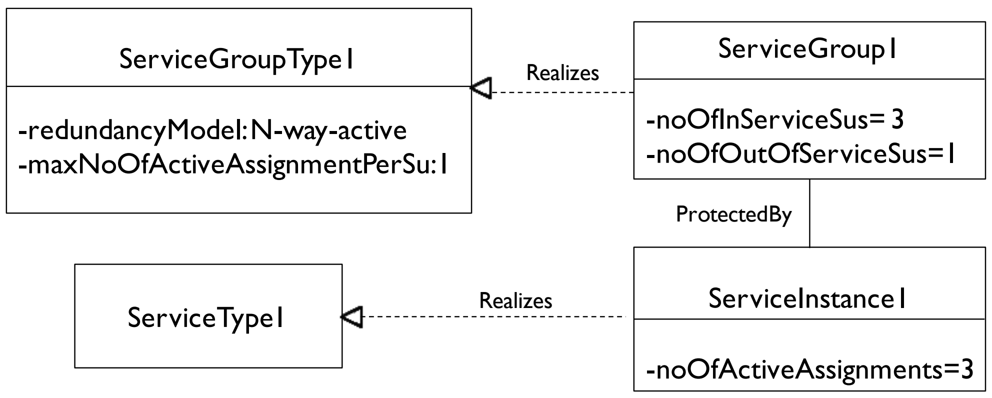


Figure 5.4 Part of the configuration model for the example in Figure 5.3

When the system starts provisioning services, it is unlikely that the workload will be at the maximum. Therefore, we initially dimension the system for the mid workload (i.e. median of minimum and maximum workload specified in the user requirements). Using the same equations, we configure the initial capacity of the system for handling the mid workload by locking configuration entities not needed to support any workload and setting the related attributes. The values of different thresholds of the deployed system are determined at this step to reflect the unlocked capacity of the system.

At runtime then, with the generated elasticity rules that are triggered by the thresholds violations, the system is reconfigured within the configuration boundary as the workload varies within the range of the minimum and the maximum workload specified in the user requirements. Threshold triggers are primarily issued on service entities (i.e. SIs) and computing nodes (either physical or virtual nodes). The threshold triggers on service entities represent variations in the workload coming from users. As explained before, these triggers may lead to the generation of prerequisite/follow-up triggers on provider side entities. In case a threshold trigger is issued on a node while no threshold trigger is generated on its supported SIs, the issued threshold trigger is not directly related to the workload variation. It is related to the distribution of entities among the nodes. Since in our approach we use different estimates to reconfigure the system (e.g. estimation of threshold, of cost and of load) the reconfiguration may not result in an optimal distribution. In this case, complementary actions should be taken to rearrange the entities hosted by the node, which means the rearrangement of assignments or virtual compute nodes. Note that the capacity of an SU in terms of number of assignments is checked as prerequisite or follow-up.

As mentioned earlier, each elasticity rule consists of action(s) and possibly condition, follow-ups, follow-up triggers, prerequisites and prerequisite triggers. These elements are specified in the process of generating the elasticity rules, which we describe in more details in the next sub-sections.

5.3.2 Generating Elasticity Rules for the Service Side

In this section we explain in more details the generation of elasticity rules for the service side. The entity types of the elasticity rules for the service side are the service types realized by SIs.

5.3.2.1 *Action Definition*

Actions *addAssignment* and *addSI* are defined for the case of workload increase and removeAssignment and *removeSI* are the actions defined for the case of workload decrease. They change the number of active assignments of an unlocked SI and/or the number of unlocked SIs in the system. These result in changes of the service side capacity of the system.

In the service dimensioning step of the configuration generation process, to determine the number of active assignments, the assignment rate is calculated first (i.e. the workload capacity represented by one active assignment). It is calculated based on the characteristics of the given service unit type and it remains fixed (up until major changes such as upgrade is performed that would change these characteristics). The number of required active assignments (i.e. the active capacity of service side) is calculated according to equation (5.1):

$$\#ActiveAssignments = ceil\ (Workload/AssignmentRate) \tag{5.1}$$

Equation (5.1) is re-used in the elasticity rule and transformed into the methods of the *addAssignment* and *removeAssignment* actions. The variables of equation (5.1) are transformed into parameters by which the aforementioned actions are defined. The variable *#ActiveAssignments* is transformed into an output parameter calculated by the methods of their operations. The variable W*orkload* is transformed into an input parameter and its value is provided at runtime by the monitoring system. The value of *AssignmentRate* is constant, whose value is determined at the configuration generation.

Depending on the applicable redundancy model a service instance may group some active assignments. If so, equation (5.2) is used in the service dimensioning step to determine the number of required SIs from the calculated active assignments.

$$\#SIs = ceil\ (\#ActiveAssignment\ /\ max\#ActiveAssignments_{SI}) \tag{5.2}$$

The value of *max#ActiveAssignments*$_{SI}$ is determined at service dimensioning time based on the maximum workload of a customer and the number of nodes the customer allowed to use. It remains constant similar to the assignment rate.

Equation (5.2) is transformed into the method of the *addSI* action to add SIs when the workload of a customer exceeds the capacity of the current SIs. This equation is also re-used in the elasticity rules which have their scaling rule set to *Decrease* to define the method of the *removeSI* action. I.e. when fewer SIs are needed for the calculated number of active assignments and some SIs should be locked as a follow-up action. Note that using these equations

(5.1) and (5.2) guarantees that always the minimum required number of active assignments and SIs are in the system.

At runtime, when these operations are executed the service side of the system is re-dimensioned. For example, if the measurement from the monitoring system shows that the current workload represented by an SI with 2 active assignments and with the assignment rate of 400 requests per second has increased to 1100 requests per second, the number of assignments for that SI should change to 3. On the other hand, if a similar increase is detected for a service type where each SI can have only one active assignment then the increase requires three unlocked SIs in the system. If there are two unlocked SIs, a third needs to be added to the system.

As part of such reconfigurations, the threshold values related to the appropriate services, i.e. for the service side of the system, need to be updated. For this purpose, the equations used to determine the boundary thresholds in the service dimensioning step are also re-used to define the method for the *updateThreshold* operation. This operation is part of the *add/removeAssignment* action and it is executed when *add/removeAssignment* operation is executed.

5.3.2.2 *Prerequisite and Follow-up Definition*

Service entities depend on service provider entities for being provided and protected, i.e. the service side relies on the resources of the service provider side. In addition, services may depend on each other within the service side, i.e. to function one service may require another service. Therefore for each dependency, a prerequisite is generated for the case of the addition of an assignment or a service instance, and a follow-up is generated for the removal case.

At the same time both the prerequisites and the follow-ups are generated as they are applicable to the same sponsor entities.

Prerequisite and Follow-Up for Checking the Service Provider Side Capacity

On the one hand side, there should be enough service provider entities to which the added active assignments or the assignments of the added SIs can be assigned. On the other hand, provider entities without assignment should be removed. In the context introduced in Chapter 2, there should be also enough groups of service units (i.e. service groups) to provide and protect the required number of SIs and their assignments. Therefore, the equation used for dimensioning the service groups and service units are reused to define the Boolean expressions of the prerequisite and the follow-up checks. To add an assignment or service instance, inequality (5.3) should be respected:

$$\textit{Current \#SGs} \geq \textit{Required \#SGs for protection} \tag{5.3}$$

To avoid wasting resources however the left-hand side of inequality (5.3) has an upper boundary. That is, the current number of service groups should be equal to their required number. Hence inequality (5.4) is used to check if a service group is extra when an assignment or SI is removed.

$$\textit{Current \#SGs} > \textit{Required \#SGs for protection} \tag{5.4}$$

For the service provider side prerequisite, we start with (5.3) and for the service provider side follow-up, we start with (5.4), and define both sides of the inequalities. The current number of service groups (i.e. the left-hand sides of inequalities (5.3) and (5.4)) is obtained from the system at runtime. The number of service groups which are required for protecting service instances is determined based on the equations used in the service provider dimensioning step that calculates the number of required service groups. The service provider side prerequisite and follow-up are generated at the same time when the number of service groups is determined.

Depending on the redundancy model, the number of service groups is calculated differently [11]. For instance, for the 2N redundancy model where each active assignment requires an SI, Equation (5.5) is used to dimension the SGs.

$$\#SGs = \mathit{ceil}\,(\#SIs \,/\, \min\,(max\#ActiveAssignmentsPerSU,\ max\#StandbyAssignmentsPerSU)) \qquad (5.5)$$

We use (5.5) in the service side elasticity rules to define the right hand side of the Boolean expressions of the prerequisite (5.3) and follow-up (5.4). For this purpose, the variables of (5.5) are transformed into parameters. The number of service instances (#SIs) is transformed into a parameter whose value for the prerequisite is calculated using (5.1) i.e. the required number of service instances; while for the follow-up it is the current number of service instances and comes from the current configuration since it has changed as a result of performing the *removeAssignment/SI* action of the elasticity rule in question. Similarly to the *AssignmentRate*, the variables *max#ActiveAssignmentsPerSU* and *max#StandbyAssignmentsPerSU* are both constant and their values are determined at configuration generation time.

Prerequisites and Follow-Ups for Checking the Service Side Capacity

If a service depends on another service (i.e. the sponsor), then the sponsor entity is dimensioned in terms of active assignments based on the dependent entity according to equation (5.6):

$$\#ActiveAssignments_{Sponsor} = \mathit{ceil}(\#ActiveAssignments_{Dependent} \times AssignmentRate_{Dependent} / AssignmentRate_{Sponsor}) \qquad (5.6)$$

By rewriting (5.6) we can generate right away the prerequisite and the follow-up applicable at runtime to check if the current number of active assignments of the sponsor provides the required capacity for the active assignments of the dependent. From equation (5.6) inequality (5.7) is obtained as the Boolean expression for the prerequisite. The Boolean expression for the follow-up is defined similarly.

$$\textit{Current \#ActiveAssignments}_{Sponsor} \geq \textit{ceil (required \#ActiveAssignments}_{Dependent} \times \textit{AssignmentRate}_{Dependent}/\textit{AssignmentRate}_{Sponsor}) \quad (5.7)$$

For the Boolean expressions of the prerequisite and follow-up, the current number of assignments of the sponsor is transformed into a parameter whose value is obtained at runtime from the system. The required number of assignments of the dependent is transformed into a parameter whose value in the prerequisite is calculated using (5.1). In the follow-up the value is obtained from the current configuration, which just has changed as a result of performing *removeAssignment* action.

In addition, to check if there are enough unlocked SIs currently in the system to group the required number of active assignments, based on (5.2) a prerequisite as well as a follow-up are generated. The inequality (5.8) obtained from (5.2) is used as the Boolean expression of the prerequisite. The Boolean expression of the follow-up is defined similarly.

$$\textit{Current \#SIs} \times \textit{max\#ActiveAssignments}_{SI} \geq \textit{\#RequiredActiveAssignments} \quad (5.8)$$

5.3.2.3 *Prerequisite and Follow-Up Triggers Definitions*

Triggers are normally generated on entities. At the time of the configuration generation, however, we cannot specify on which entity a prerequisite or a follow-up trigger needs to be issued; we can only specify the type of this entity. As a result, the prerequisite and follow-up triggers are also defined for the appropriate type. This means that for a prerequisite/follow-up that checks the capacity of the SGs of a type, the corresponding prerequisite/follow-up trigger is defined on the SG type and for a prerequisite/follow-up that checks the capacity of SIs of a service type, the prerequisite/follow-up trigger is defined on the service type. Then based on this at runtime, the follow-up/prerequisite trigger is issued on the SG or SI sponsoring the SI for which the elasticity rule was invoked. The *scalingType* of a prerequisite trigger is *Increase* and *Decrease* for a follow-up trigger because a prerequisite trigger always initiates the allocation of the prerequisite resources and a follow-up trigger always initiates the release of excess resources of the sponsors. The attribute *measurement* of a prerequisite trigger represents the minimum sponsor capacity which is required to be added to meet the prerequisite Boolean expression. In contrast, the attribute *measurement* of the follow-up trigger represents the minimum sponsor capacity which is required to be removed so that the follow-up Boolean expression is evaluated to false indicating no extra sponsor resource. These prerequisite and follow-up triggers are defined together with their corresponding prerequisite and follow-up Boolean expressions.

5.3.2.4 *Condition Definition*

The *addSI*, *addAssignment*, *removeSI* and *removeAssignment* actions are applicable when by adding/removing SIs or assignments the designed boundaries of the system are not violated. Moreover, the action *addSI* is applicable only if the SI on which the trigger was issued is currently “locked” as the action changes its status to “unlocked”. In contrast, the action *addAssignment* is applicable if the SI for which the trigger is generated is currently “unlocked”. Therefore, the state of the SI is transformed into a parameter by which the Boolean expressions of these conditions are defined. The action *removeAssignment* is applicable if the SI contains some active assignments and *removeSI* is applicable when fewer SIs for grouping active assignments are required. As a result for the *removeAssignment* and *removeSI* actions, the current number of active assignments in an SI is transformed into a parameter by which the Boolean expressions of the conditions are defined. All the aforementioned conditions are generated when their corresponding actions are generated.

5.3.3 Generating Elasticity Rules for the Service Provider Side

In this sub-section, we explain the generation of elasticity rules for SGs and nodes.

5.3.3.1 *Elasticity Rules for SGs*

An elasticity rule for an SG is triggered as a prerequisite when the workload increases and the current SGs cannot provide the added SIs/assignments or as a follow-up action when the workload decreases and SUs and/or SGs become in excess and should be removed.

Action Definition
Depending on the situation different actions are possible:

- *Reconfiguring the Current SGs by Adding or Removing SUs:* The capacity of the system for providing SIs can change by reconfiguring its current SGs. An SG is reconfigured by changing the state of its constituting SUs. That is, the capacity of an SG can increase by “unlocking” some of its “locked” SUs. Similarly, the SG can be reconfigured by “locking” its unassigned SUs when the workload decreases. By taking such actions, the number of “unlocked” SUs in the SG changes. The equation used to dimension the SUs at configuration design time is used in the SG elasticity rule as the method of the *reconfigureSG* operation.
- *Adding New SGs or Removing the Ones in Excess:* When the workload is not at its maximum, some of the SGs may not have any SI to protect. Not to waste resources, the action *removeSG* is taken to lock the excess SGs and their SUs. In contrast when a new SI is required, to increase the capacity of the system, a new SG may be required

as the service provider entity. By performing the action *addSG*, the SG and some of its SUs become “unlocked”. At runtime, when the operation *addSG* or *RemoveSG* is executed based on the required or removed SIs the current number of unlocked SGs in the system changes. The equation used in the service provider dimensioning step to calculate the number of SGs (i.e. equation (5.5)) is used to define the method of *addSG* or *removeSG* operations. The required number of “unlocked” SUs in each SG is determined according to the redundancy model of the SG as described in [11]. For example, if the redundancy model is 2N, the required number of “unlocked” SUs is 2 (i.e. one SU for supporting the active assignments and one SU for supporting the standby assignments).

Prerequisite and Follow-up Definitions

Service units are hosted on nodes; therefore to unlock a service unit, as prerequisite the hosting node should be in the “unlocked” state and it should have enough capacity to host the added service unit. The load that is imposed on the node by requests of a service is estimated by a function at runtime. This function takes into account parameters that characterize the workload as well as the node (e.g. the types of workload the node currently supports, the operating system, etc.). By calculating the estimated load at runtime, we can check if the underlying node will have enough resources to support the required service unit. As a result, in the SG elasticity rule model the Boolean expressions of the prerequisites are defined as (5.9) and (5.10) to check if the node is “unlocked” and if it has enough resources to host the required service unit:

node.state=”unlocked” (5.9)

node.maxNodeThreshold > node.load + su.requiredResource (5.10)

In contrast, by putting a service unit into the “locked” state, the resources of the hosting node may become in excess; thus a follow-up trigger on the node should be generated to initiate the removal of the node or its excess resources (if applicable) by a follow-up action. The resources of the node are in excess if the current load on the node is less than its minimum threshold. Therefore, in the SG elasticity rule model the follow-up is defined as (5.11):

$$\mathit{node.minNodeThreshold} > \mathit{node.load} \tag{5.11}$$

The Boolean expressions (5.9), (5.10) and (5.11) are generated at the last step of the configuration generation process when the nodes for hosting the SGs are determined. At this step, the variables *maxNodeThreshold*, *minNodeThreshold*, *load* and *state* are transformed into parameters that belong to the node. The variable *requiredResource* of (5.10) is transformed into parameter that belongs to the hosted SU.

Prerequisite and Follow-up Triggers Definitions
Since the SUs of an SG may be hosted on different nodes of a node group, it is not possible to specify offline on which node the prerequisite or follow-up trigger should be generated at runtime. However, we can specify to which node group an SU belongs. Therefore, at design time, the prerequisite/follow-up trigger is defined for the node group. At runtime, when the prerequisite for adding an SU is not met or when after the removal of the SU resources become extra, the trigger is generated for the hosting node.

Condition Definition
In case of increase, the action *reconfigureSG* is applicable if the SG on which this action should be taken is unlocked. If the SG is locked, the action *addSG* is applicable. As a result, the state of the SG is transformed into a parameter by which the Boolean expressions of these conditions are defined. In case of decrease, the action *reconfigureSG* is applicable if the SG still protects some SIs. In contrast, the action *removeSG* is applicable when the SG does not

have any SI to protect and as the result of the action it should be locked. Therefore, the current number of protected SIs is transformed into a parameter by which the conditions of *reconfigureSG* and *removeSG* actions are defined. All the aforementioned conditions are generated when their corresponding actions are generated.

5.3.3.2 *Elasticity Rules for Nodes*

The configuration of nodes need to support the SGs, their SUs and if applicable virtual compute nodes, which is guaranteed by the prerequisites and follow-ups of their elasticity rules. However, because of different estimates used at their execution, these may not always guarantee an optimal distribution of assignments to the SUs and SGs hosted on the nodes. To redistribute hosted entities, additional complementary actions may be needed. Therefore, the actions of node elasticity rules are categorized into: Actions to handle prerequisite or follow-up triggers; and actions to redistribute the hosted entities for better resource utilization. Note that in turn actions of the latter category may require actions of the first category as prerequisites/follow-ups. To define the actions of the second type, we define templates based on the distribution principles. We use these templates to generate the different elements of the elasticity rules for the nodes. Since a node can be members of multiple node groups, we do not define the elasticity rules per node group. We define them per node at the last step of the configuration generation process when the node configuration is generated.

In the following, the actions of the node elasticity rules are explained.

Add or Remove a Node

These actions are defined for the cases of adding a node as a prerequisite or removing one as a follow-up action. The action *addNode* is applicable when the state of the node is “locked” and by taking this action, the state will change to “unlocked”. The action *removeNode* is applicable when an "unlocked" node has no services to support and by taking this action, the state of the node changes to "locked".

The prerequisite for the *addNode* action is expressed as inequality (5.12). According to this prerequisite, if the node is hosted by another node (e.g. it is a VM hosted by a physical node), the hosting node should have enough resources for the hosted node (i.e. by unlocking the node, the maximum threshold of its hosting node should not be reached).

node.hostingNode ->notEmpty() ***implies*** *node.hostingNode.load* + *requiredResource* <

node.hostingNode.maxNodeThreshold (5.12)

If the node is hosted by another node, the follow-up as (5.13) is associated with the *removeNode* action to check if by the removal of node, the resources of the hosting node are in excess. Since the nodes can migrate at runtime, we cannot specify at design time on which hosting node the prerequisite and follow-up trigger should be defined. Therefore, at design time, the prerequisite and follow-up triggers are defined on the group of nodes which can host the node.

node.hostingNode ->notEmpty() ***implies*** *node.hostingNode.load* ≥

node.hostingNode.minNodeThreshold (5.13)

Adding or Removing Virtual/Physical Resources to or from the Node

These actions are defined primarily for the cases of prerequisite/follow-up actions. They can be used also to avoid redistribution by adding/removing resources of the current node if it is resizable. A node can be a resizable virtual machine or a hyperscale system like Ericsson HDS 8000 [74]. Resources can be added to a resizable node to decrease the resource utilization, or removed from it to increase. These actions are only included in the elasticity rules of resizable nodes. By these actions, the amount of resources allocated to a node changes. A resizable computing node still has a maximum capacity that it can expand to. If the node has reached its maximum capacity, no more resources can be added to the node and this action is not applicable. Therefore, the condition for *addResources* action is defined as (5.14) to check if the node has not reached its maximum capacity yet.

$$node.maxNodeBoundary > node.currentResource + requiredResource \qquad (5.14)$$

To take the *removeResources* action, as condition, the node should have at least one running process. The prerequisite, prerequisite trigger, follow-up and follow-up trigger of these actions are similar to those of *add/removeNode* actions.

Rearrangement of Workload

These actions are defined to redistribute hosted entities of a node. I.e. trying to resolve the threshold trigger on the node by taking actions on its hosted entities. To decrease the load on a node, the supported services can be moved out to other nodes if as prerequisite there are service provider entities with enough capacity to host them. At runtime, when the node supports multiple services, based on the estimated cost of releasing one unit of resource it is decided which supported service should be moved out. To rearrange the workload, the following actions are defined:

- *Migration of hosted nodes:* If the node is capable of hosting other nodes, some of its hosted nodes can be migrated to other hosting nodes to release the resources of the given node. The prerequisite to migrate a hosted node is expressed as (5.15). Accordingly, in the hosting node group there should be a hosting node with enough capacity to host the hosted node to be migrated. As expressed in (5.16), this action is applicable when the hosting node hosts at least one hosted node. The prerequisite and follow-up triggers are defined on the group of hosting nodes which are eligible to host the hosted node.

nodeGroup.nodes() -> exists (n|n.load + requiredResource ≤ maxNodeThreshold) (5.15)

node.hostedNodes -> size() >0 (5.16)

- *Moving assignments/SIs to other nodes:* Similar to the migration of hosted nodes, one way of releasing a node's resources is to move the assignments or SIs supported by the node to other nodes. The prerequisite, prerequisite trigger, follow-up and follow-up trigger of this action are similar to those of *addSI/Assignment* and *remove-SI/Assignment* actions.
- *Adding SI/assignment to an additional node:* By adding an SI or an assignment, the workload is shared among more nodes and therefore, less load will be imposed on the given node. This action is applicable if the boundary of the system from the service side has not been reached. This action has been explained in Section 5.3.2.
- *Swapping the active and standby assignments:* Standby assignments often impose less load on the nodes than active assignments; therefore, swapping the role may reduce the load on the node having the active assignment. However, for services such as data bases where the load imposed by the active and the standby assignments are quite the same, this action may not be effective. This action has no prerequisite and prerequisite trigger; however, a condition is defined to check if for an active assignment supported by the node, there exists a standby assignment such that the load imposed by its standby will be less than the load imposed by it. Depending on the redundancy model of the protecting SG, other constraints may be needed too. After performing this action, if a failure happens the node may experience high load again as the standby assignment becomes active due to the failure.

5.4 Summary

To reconfigure the system dynamically, a set of elasticity rules are generally used. In this chapter, we propose an approach to generate automatically elasticity rules at configuration design time. While the system's configuration is designed (i.e. generated automatically), the calculations used to dimension the system as well as some computed parameters are reused to define the elasticity rules. We reuse the system dimensioning knowledge instead of learning about the system behavior and usage. Moreover, the elasticity rules are at a finer granularity than what is presented in the related work as we also consider the rearrangement of resources and not only the addition and removal. Since we use the calculations for dimensioning of the system to generate the elasticity rules, the elasticity rules generation requires the configuration generation process.

Chapter 6

SLA Compliance Management

In this chapter we explain the SLA compliance management, a part of our management framework. The SLA compliance management aims at generating triggers whenever there is a potential SLA violation (i.e. an SLA is probable to be violated in the next time interval of monitoring) or the resource utilization is low.

6.1 Introduction

A Service Level Agreement (SLA) is a contract negotiated and agreed on between a service provider and a customer; it defines the expected quality of the services to be provided [75]. For instance, the level of service availability, i.e. the percentage of time the service is provided in a given period of time [1], is part of the SLA. The rights and obligations of each party are also described in the SLA. When any of the parties fails to meet their respective obligations, SLA violations occur and the responsible party may be subject to penalties.

System workload varies over time, which results in variable resource usage. To increase revenue, instead of allocating a fixed amount of resources, service providers try to allocate only as much as needed to support the workload and adapt this allocation according to the workload variations. In the cloud environment, the dynamic resource provisioning according to workload variations is called elasticity. A cloud system evolves and adapts dynamically to workload variations by scaling out/in and up/down [4].

6.2 The Overall Approach for SLA Compliance Management

In the *SLA Compliance Management* process, all the SLAs, their corresponding measurements and the thresholds are combined into an SLA compliance model. The validation of the SLA compliance model against its metamodel is performed periodically. The violation of OCL constraints during this validation will generate automatically triggers for system reconfiguration, to save resources when the workload decreases or avoid SLA violations when the workload increases and the SLAs are about to be violated. The output of this process, i.e. the generated triggers, serves as input for the dynamic reconfiguration process (see Chapter 7).

6.3 Modeling for the SLA Compliance Management

To manage the compliance of the SLAs at runtime, we adopt a model driven approach not

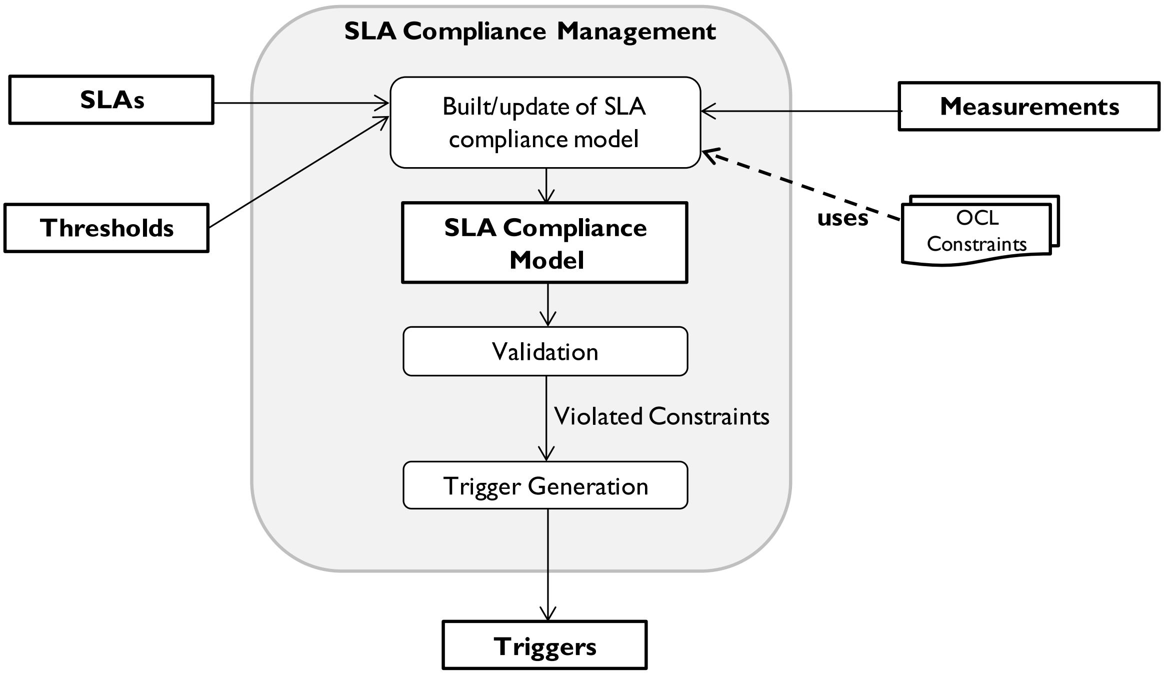


Figure 6.1 The SLA compliance management process

only to facilitate the understanding, design and maintenance of the system [7], but also to reuse the models generated during the system design phase, such as the thresholds, and to build on existing tools. In this section, we introduce the metamodels of SLA, SLA compliance and trigger. We use the Unified Modeling Language (UML) profiling mechanism [9] to customize the UML and design the modeling languages for SLA, SLA compliance, threshold, measurement and trigger. For this purpose, we define the domain model (or domain metamodel) and map it to the UML metamodel [8].

6.3.1 The SLA Metamodel

The SLA metamodel is shown in Figure 6.2. Each SLA has an ID and is an agreement between a provider and a customer. A third party may also participate to verify the agreed terms and play the monitoring role. An SLA includes some service functionalities that the provider agrees to provide with specific Quality of Service (QoS). Abstract metaclass *SlaParameter* captures the different types of QoS included in the SLA. The agreed values are represented by *maxAgreedValue* and *minAgreedValue* in the figure. For example, for the SLA parameter availability, the *minAgreedValue* represents the minimum percentage of the time that the provider guarantees the service is available. For the SLA parameter *DataRate*, the *maxAgreedValue* represents the maximum number of requests per second the customer may send for the specific service, and the *minAgreedValue* represents the minimum amount of service that the provider agreed to provide. If service providers or customers fail to meet the agreed terms, they may be subject to penalties.

The QoS included in the SLAs should be either measurable by the monitoring system or reported by the constituent components of the system; otherwise, it cannot be included in SLAs. Customers may want to specify at which frequency the SLA parameters should be measured. This customization is represented by *SlaMetric* metaclass. However, the frequency

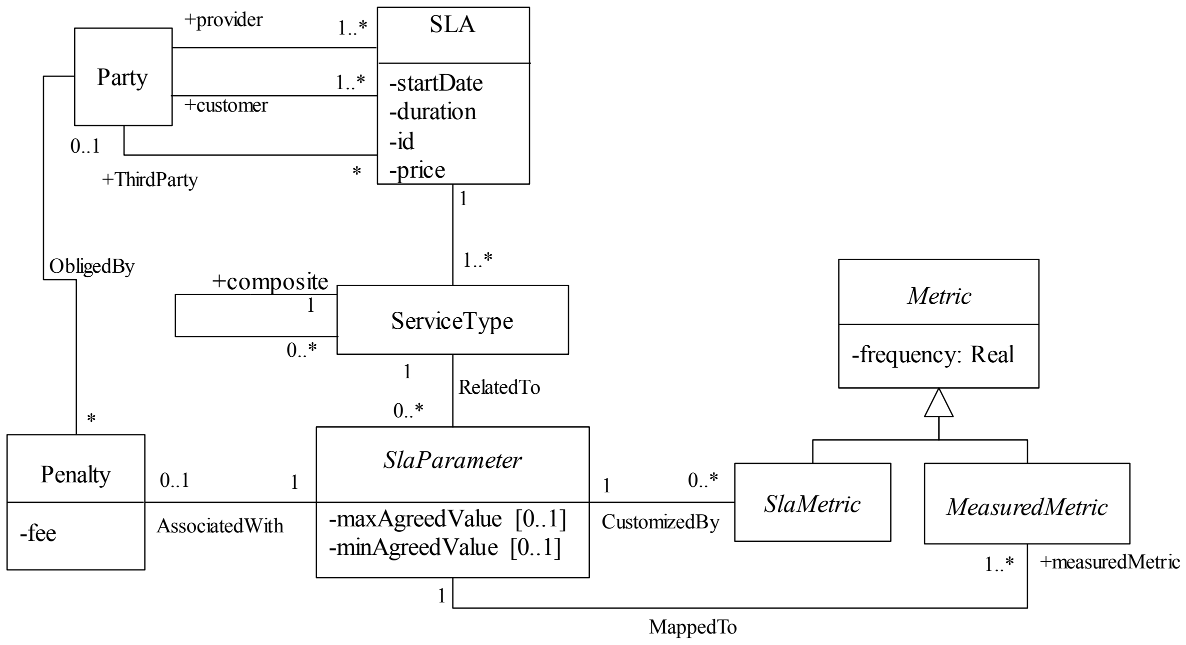


Figure 6.2 The SLA metamodel

specified by the user should be compatible with the capability of the monitoring system. An SLA is applicable for specific time duration and has a cost that customer agrees to pay.

Figure 6.3 shows two SLA models. The *VoIP* functionality is sold to customers C_1 and C_2 with different quality of service. The service functionality is represented with gray rectangles and SLA parameters with rounded square. The dashed lines show *RelatedTo* relations.

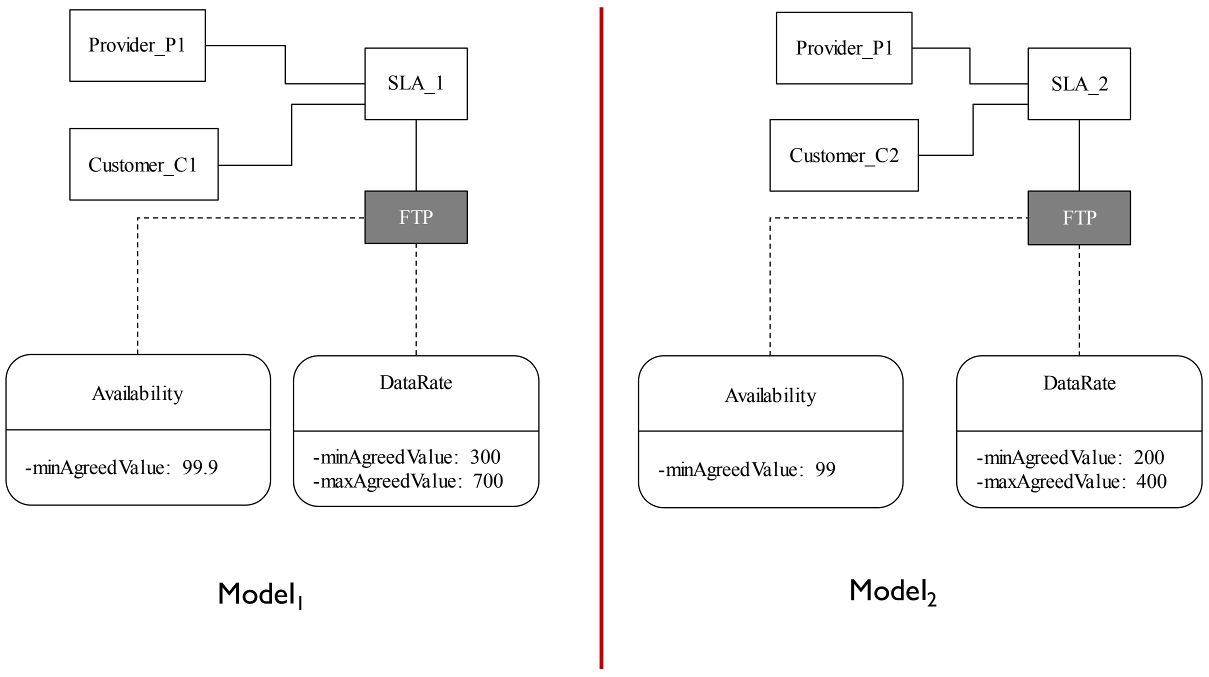


Figure 6.3 Two different SLAs

6.3.2 The Measurement Metamodel

A monitoring system collects the metrics of interest. These measured metrics are related to a computing node or a service. The *Service* metaclass represents instances of a service type, which—in the explained domain in Chapter 2—are represented by service instances. Some of the metrics (e.g. service up/down time) and the SLA parameters perceived by the customers (e.g. availability of service) are not at the same level. To bridge the gap between the measured metrics and the SLA parameters, we have defined mapping rules. Figure 6.4 shows the measurements metamodel. The attribute *mappedValue* represents the value of such mapped measurements. As an example, the mapping rule for mapping service up time and down time to service availability is presented:

```
Context Availability :: mappingRule ( )

self.mappedValue                          =                          self.metric->
select(c|c.oclIsTypeOf(MeasuredUpTime)).measuredValue->at(1)/(self.metric->
select(c|c.oclTypeIsTypeOf(MeasuredUpTime)).measuredValue    ->    at(1)    +
self.metric->select(c|c.oclType(MeasuredDownTime)).measuredValue->
at(1))*100
```

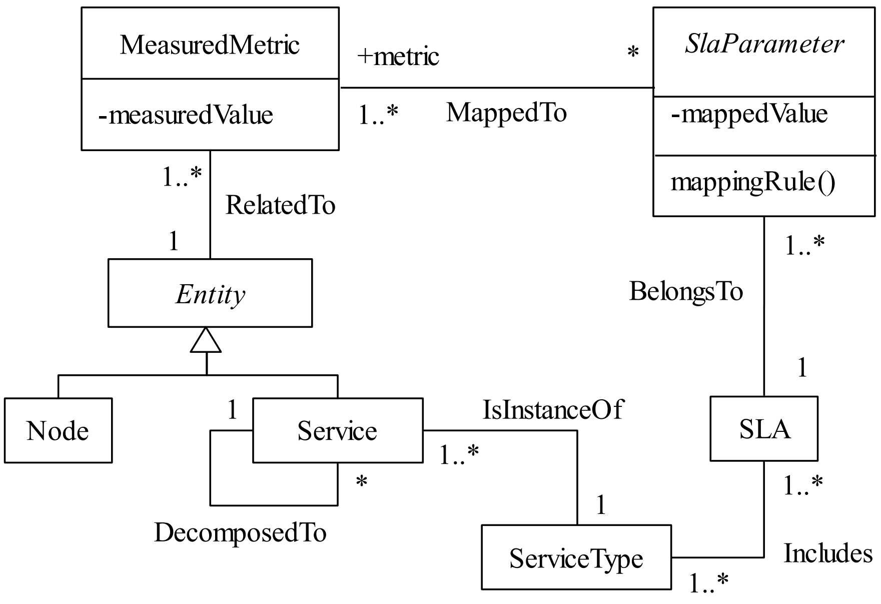


Figure 6.4 The measurement metamodel

Figure 6.5 shows an example of measurement model. In this figure, the measured metrics are represented by rounded squares in light gray. The dotted and dashed lines represent *BelongsTo* and *RelatedTo* relations respectively.

6.3.3 The Threshold Metamodel

We use thresholds as points. When they are reached, actions are required to avoid SLA violations or low resource utilization. Figure 6.6 shows the threshold metamodel. As shown in the

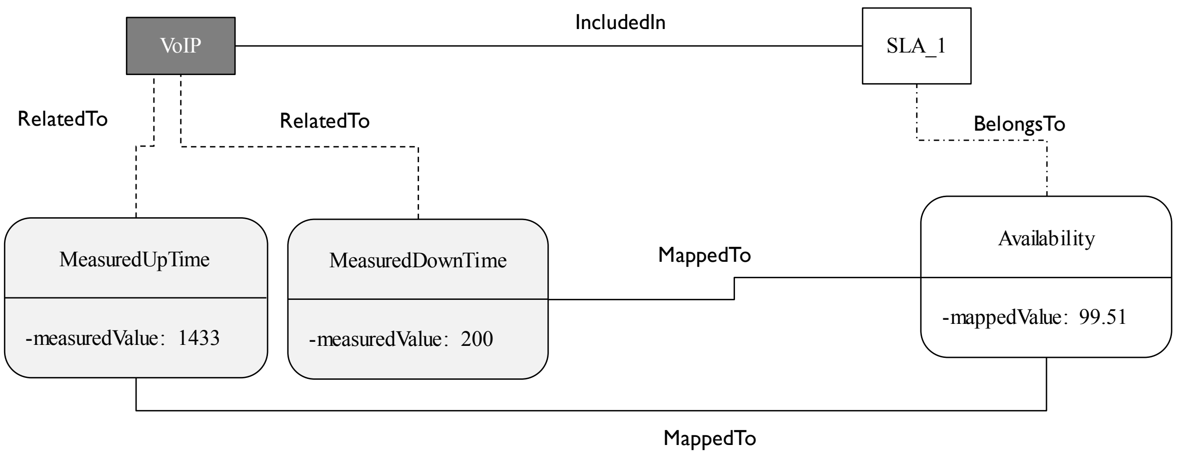


Figure 6.5 An example of measurement model

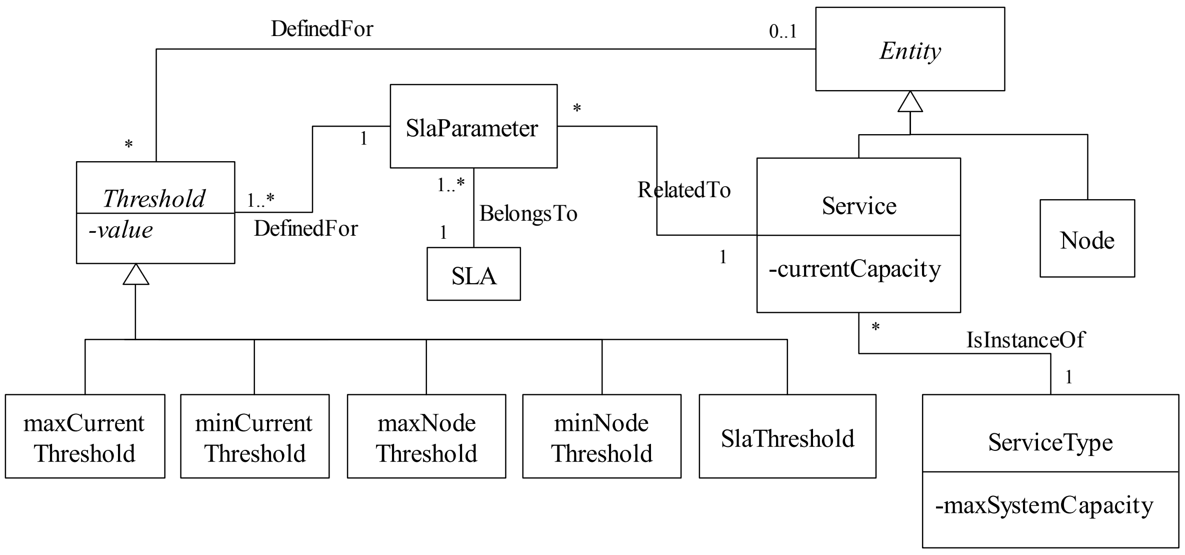


Figure 6.6 The threshold metamodel

figure, thresholds are defined on nodes, service functionalities or individual SLAs. Some of the thresholds are related to all customers' (aggregate) resource usage (i.e. thresholds defined on nodes) while others are related to individual SLAs (e.g. service availability). The attribute *currentCapacity* in the *Service* metaclass specifies the current capacity of a service entity (i.e. a service instance) for handling the workload of a specific customer. The attribute *maxSystemCapacity* is determined at the design phase as the maximum capacity the system can be expanded to for a specific service type without major changes (e.g. upgrade/redesign). For nodes and service entities (i.e. represented as service instances), two thresholds (maximum and minimum thresholds) are defined: The maximum limit represents the load of the node or the service instance without SLAs violation. If no action is taken SLA violation is likely to happen within the next measurement period. The minimum limit represents the load of the node or the service instance for efficient usage of the resources; otherwise they are wasted. In the following the different types of the thresholds are explained:

- *maxCurrentThreshold* and *minCurrentThreshold*: For each service, the system is dimensioned dynamically with a *currentCapacity* to handle the workload of a certain customer. In order to avoid SLA violations, i.e. workload exceeding *currentCapacity*, we define a *maxCurrentThreshold* point with *maxCurrentThreshold* < *currentCapacity*. Not to waste resources, we also define a *minCurrentThreshold*. Unlike resource provisioning, the resources should be released in a reactive manner. The values of *maxCurrentThreshold* and *minCurrentThreshold* are determined by different functions which take into account the current capacity, the measurement period, the average reconfiguration time and the predicted workload.

- *maxNodeThreshold* and *minNodeThreshold*: To avoid SLA violations because of node limitations, e.g. trying to load a node beyond its capacity, we define the *maxNodeThreshold* point at which we may allocate more resources to the node (e.g. virtual machine, hyper scale system [74]), add more nodes to the system or rearrange the assignments (i.e. the relation *load* < *maxNodeThreshold* should be always respected). To avoid wasting resources, the *minNodeThreshold* is defined. The *maxNodeThreshold* and *minNodeThreshold* are vectors that take into account different types of node resources (e.g. CPU, RAM, etc.).
- *slaThreshold*: Some SLA parameters like service availability are set on a per customer basis. Therefore, to avoid SLA violations, we need to watch the SLAs separately using a *slaThreshold* for each QoS of each SLA.

Figure 6.7 shows an example of threshold model. In this figure, different thresholds for the services of type *VoIP* and *$Node_3$* are defined.

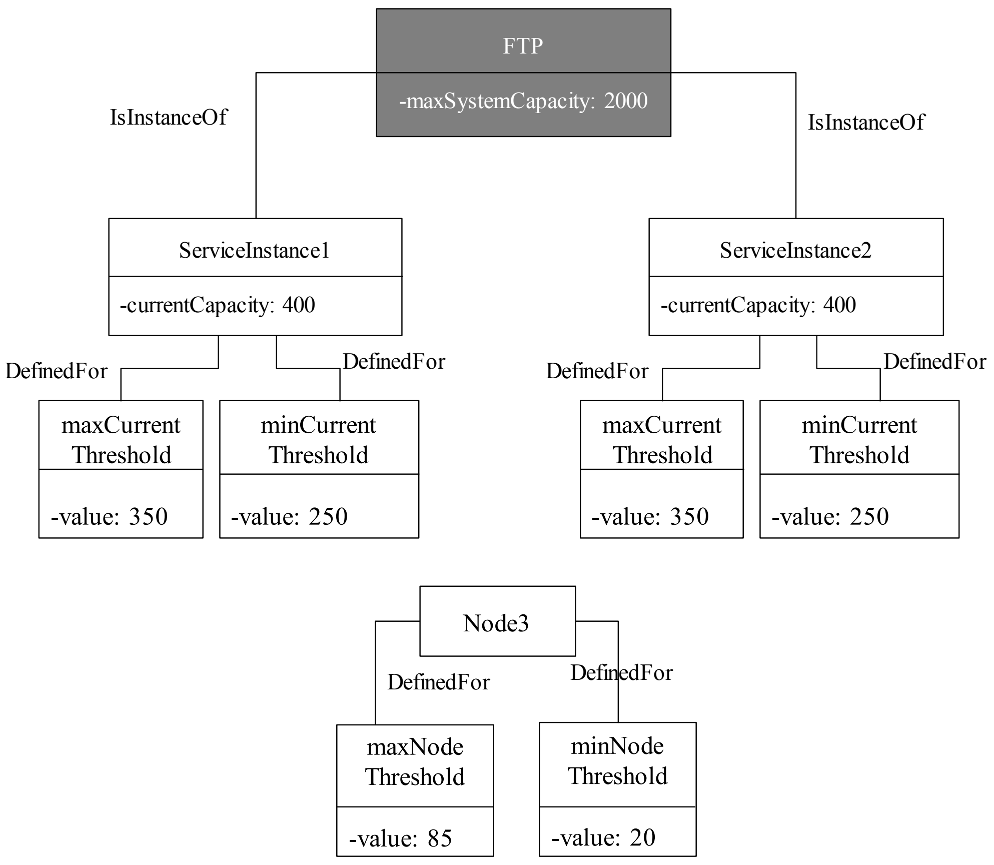


Figure 6.7 An example of threshold model

6.3.4 The SLA Compliance Metamodel

An SLA compliance model is the combination of all SLA models, thresholds model and the measurements obtained from the monitoring system. The main reason for merging all SLA models into one model is that we want to be able not only to avoid violations of each individual SLA but also to trigger elasticity rules which are related to all customers' resource usage.

The SLA compliance metamodel is shown in Figure 6.8 and an instance model of it is shown in Figure 6.9. Different services of the same service type with the same or different QoS (i.e. represented by *SLAParameter*) are generally offered to multiple customers. The *Measured-Metric* metaclass represents the measurements that are collected from the monitoring system per service for each customer or per node of the system. When an SLA parameter is not respected, the *RelatedTo* relation indicates which service of which SLA has been violated.

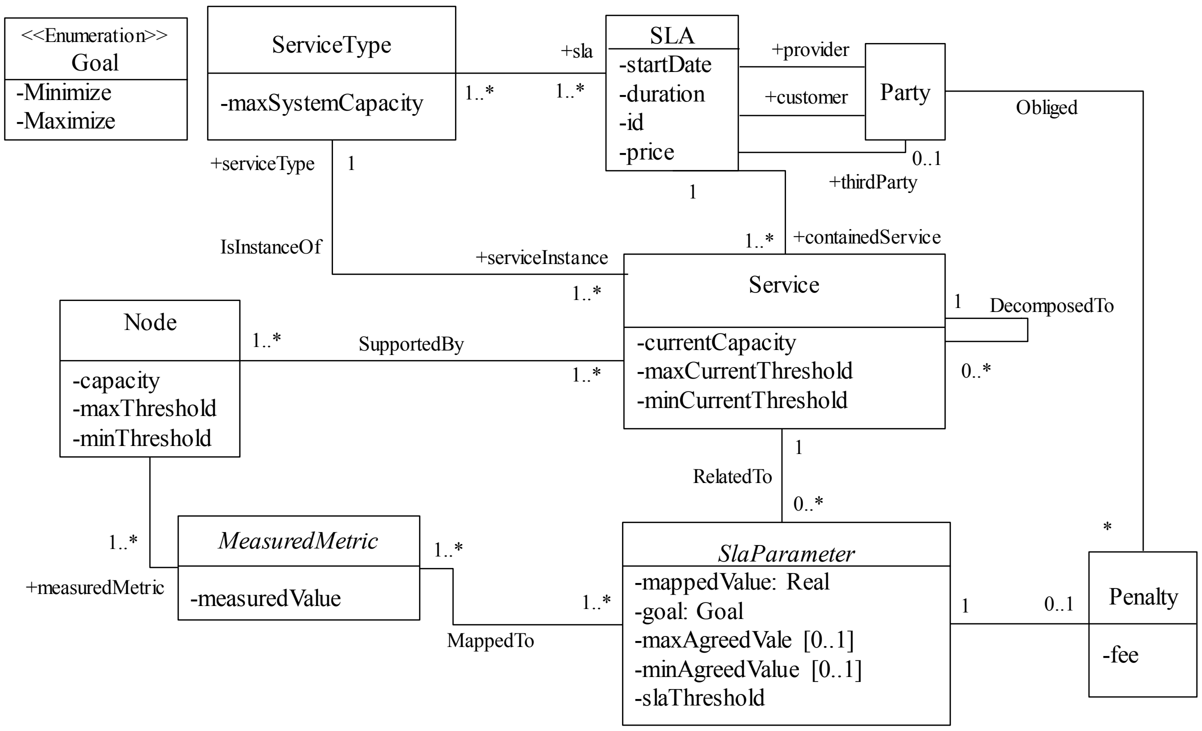


Figure 6.8 The SLA compliance metamodel

The attribute *goal* of an SLA parameter specifies the parameter's optimization goal. For some SLA parameters, like service availability, the optimization goal is maximization while for others like response time, the goal is minimization. We categorize OCL constraints for SLA violation avoidance based on these optimization goals. When a new SLA parameter is introduced and taken into consideration, there is no need to define a new OCL constraint as long as its optimization goal fits into one of the aforementioned categories.

According to UML [9], a constraint is a model element that can have a name (it is optional) and consists of an invariant (i.e. a Boolean expression that must be evaluated to true for the constraint to be satisfied), constrained elements (i.e. a set of elements required to evaluate the constraint) and a context (i.e. the model element on which the constraint is defined). Therefore, an OCL constraint is defined as a tuple of *(name, context, ConstrainedElements, invariant)*. To define an OCL constraint, its different elements should be specified.

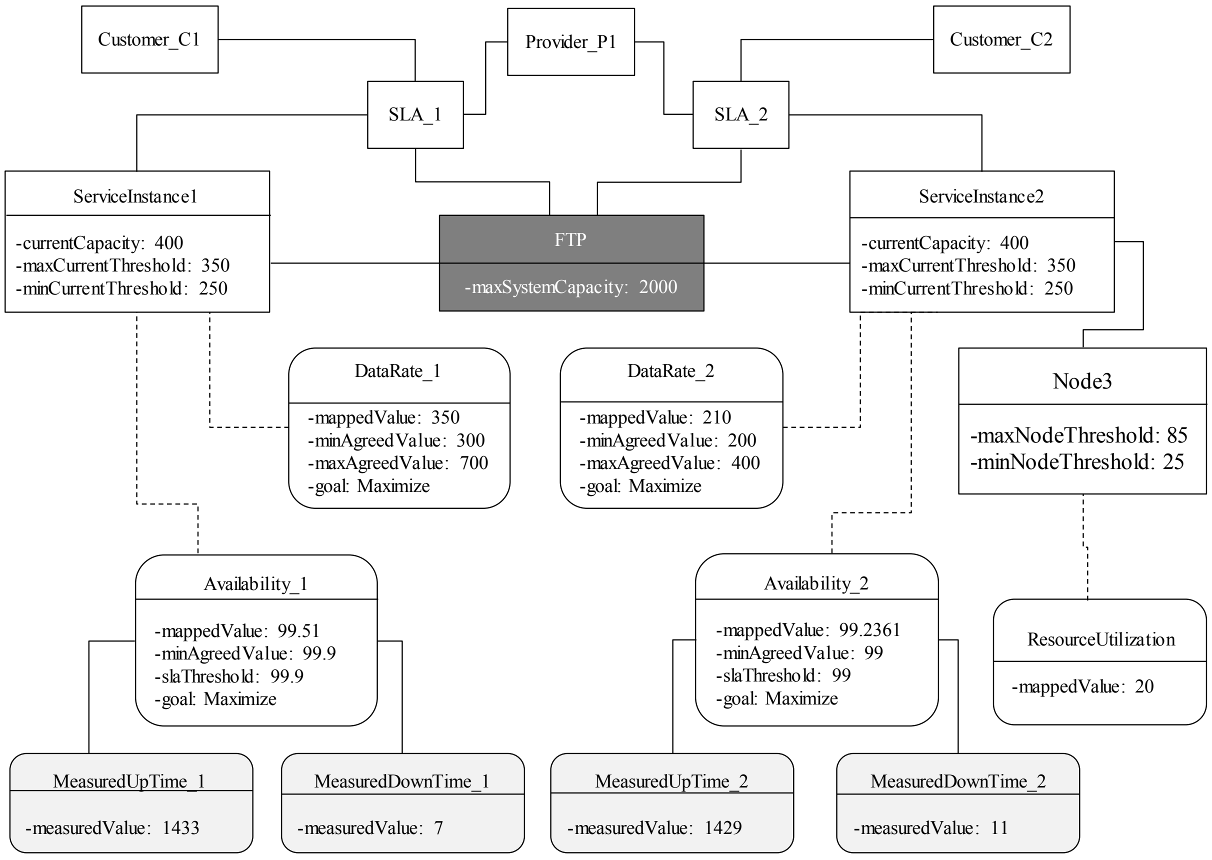


Figure 6.9 An example of SLA compliance model

Depending on the type of SLA violation, different OCL constraints are defined in the SLA compliance metamodel. As shown in Figure 6.10, an SLA can be violated by the provider or by the customer. Violations by providers are categorized into: violations issued on system resources, which can be violations on service entities (i.e. represented as service instances in our domain) or nodes and lead to the generation of triggers for dynamic reconfiguration; individual SLA violations can be related to the design of the system and system boundary related violations. The focus of this thesis is on the triggers, which lead to dynamic reconfigurations (i.e. violations on system resources). When the trigger is of this type, the generated elasticity rules are applied to reconfigure the system dynamically. The different types of violations are defined as follows:

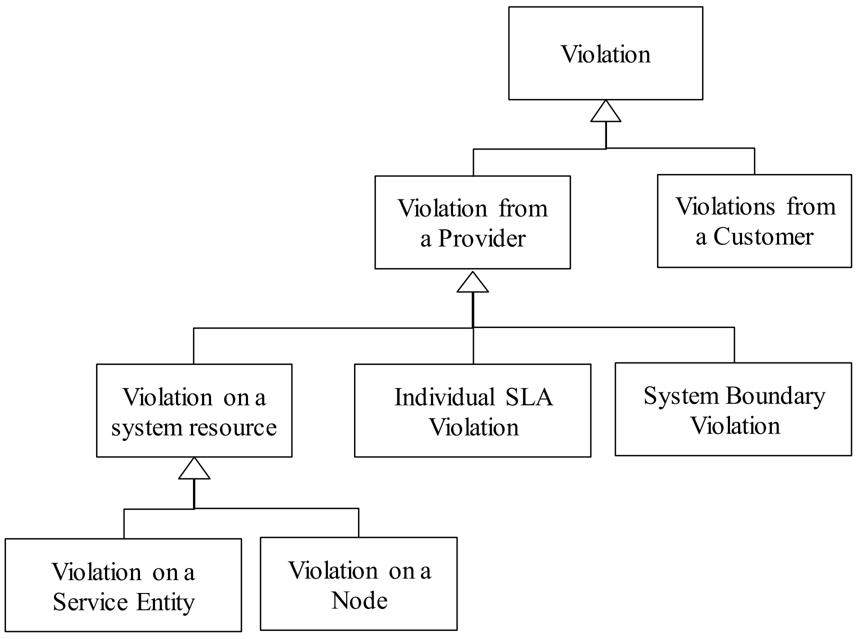

Violation
Violation from a Provider
Violations from a Customer
Violation on a system resource
Individual SLA Violation
System Boundary Violation
Violation on a Service Entity
Violation on a Node

Figure 6.10 The different types of SLA violations

The focus of this thesis is reconfiguration because of generation of triggers for scaling the system. When the trigger is of this type, the generated elasticity rules are used to reconfigure the system dynamically. The different types of violations are defined as follows:

6.3.4.1 *SLA Violations from a Provider*

- *Service Entities Violations:* For each service, the system is configured with a capacity to handle the workload of a specific customer. In order to avoid SLA violations, the relation *workload* < *maxCurrentThreshold* must be respected. If the workload exceeds the threshold, a potential violation is detected and a trigger should be generated to increase the system capacity to a new *currentCapacity* for which a new *maxCurrentThreshold* is defined. To check if the system needs to be scaled due to workload increase, the following OCL constraint is defined. This OCL constraint is named as *Increase*. Later we use this name as the scaling type of the generated trigger to see the violation was due to increase in the workload.

```
Context Service
```

```
Inv Increase: maxAgreedValue > Service.allInstances() ->
select(s|s.sla = self.sla and s.serviceType = self.servicetype) ->
collect(currentCapacity) -> sum() implies
self.maxCurrentThreshold     >     (self.slaParameter     ->     select
(p|p.oclIsTypeOf(DataRate)).mappedValue -> at(1)
```

Not to waste resources we define an OCL constraint to check if the relation *workload* ≥ *minCurrentThreshold* is respected. This OCL constraint is named as *Decrease* because its violation indicates the resources of the system are excess and the system should be shrunk.

```
Context Service
Inv Decrease: self.minCurrentThreshold ≤ (self.slaParameter -> select
(p|p.oclIsTypeOf (DataRate)).mappedValue -> at(1)
```

Considering the SLA compliance model in Figure 6.9, the workload of the customer C_1 represented by *ServiceInstance*$_1$, i.e. 350 requests per second, reaches the *maxCurrentThreshold*. Therefore, its corresponding OCL constraint named as *Increase* is violated which indicates more resources for handling the workload of this customer should be allocated.

- *Node Violations:* Although services are supported by nodes and service side violations (i.e. increase in the workload) usually lead to the violations on the underlying nodes, we still need to distinguish between violations on the services and on the nodes. The reason is that when the node hosts multiple services and the workload increase of an individual service does not reach its threshold, the workload increases of the hosted services will accumulate on the hosting node and the total load may cause violation. This happens when the distribution of entities among the nodes is not optimal. Therefore, to detect SLA violations because of node limitations, the relation *load*

< *maxNodeThreshold* should be respected. Similar to the *maxNodeThreshold*, the *load* represents the load on the different types of resources which is measured by the monitoring system. In addition, the relation *load* ≥ *minNodeThreshold* should be respected in order to not to waste resources. The load that is imposed on the node by requests of a service is estimated by a function at runtime. This function takes into account parameters that characterize the workload as well as the node (e.g. the types of workload the node currently supports, the operating system, etc.).

```
Context Node
Inv Increase: self.maxNodeThreshold > (self.measuredMetric-> select
(p|p.oclIsTypeOf(ResourceUsage))->at(1).measuredValue)


Context Node
Inv Decrease: self.minNodeThreshold ≤ (self.measuredMetric->select
(p|p.oclIsTypeOf(ResourceUsage))->at(1).measuredValue)
```

Similar to the OCL constraints defined on the services, these OCL constraints are also named as *Increase* and *Decrease* as their violations indicate if the load on the node has to be decreased or if the node or some resources of the node are in excess. Considering the SLA compliance model in Figure 6.9, the current load on Node_3 is 20 which is less than the *minNodeThreshold* which is 25. Therefore, for Node_3 the OCL constraint with the name of *Decrease* is violated.

- *Individual SLA violations:* Some SLA parameters behave similarly with respect to violation. Some of them like availability and throughput for which a higher value is preferable (i.e. the attribute *goal* is equal to *Maximize*) will be violated by a service provider when in the SLA compliance model, the experienced quality is less than their defined *slaThreshold* (i.e. the relation *mappedValue* > *slaThreshold* must be respected all the time if *goal=Maximize*); while for others like response time, the violation happens from the provider side when the measured response time is greater than the *slaThreshold* (i.e. the relation *mappedValue* < *slaThreshold* must be respected if *goal=Minimize*). We use OCL constraints as follows to define these restrictions:

```
Context SlaParameter
Inv Maximize: Self.goal=Goal::Maximize implies self.mappedValue >
self.slaThreshold


Context SlaParameter
Inv Minimize: Self.goal=Goal::Minimize implies self.mappedValue <
self.slaThreshold
```

Considering the SLA compliance model in Figure 6.9, for customer C_1 the measured availability of VoIP is 99.51 which is less than its corresponding *slaThreshold* of 99.51; therefore in this example, the OCL constraint of availability which has the goal of *Maximize* is violated.

- *System Boundary Violation:* Customers have periods of activity and inactivity; therefore, the customers may not use resources all at the same time. To make the most profit, providers sell the same resource to multiple customers. This is known as overbooking technique [76]. In this thesis, we assume that the provider sells the services to the maximum number of customers such that minimum or no SLA violation occurs and the revenue is the most. With overbooking, there is a risk that the customers want to use resources all at the same time [77, 78]. In this case, the system reaches its maximum capacity. When the value of *maxCurrentThreshold* is reached, system cannot be expanded further; thus the admission control/overload protection needs to be engaged to protect the system from overload. In addition, the provider may decide to redesign the system with new user requirements using the traceability model generated during configuration generation process (explained in Chapter 4). The following OCL constraint detects the potential SLA violations when the system reaches its maximum capacity:

```
Context ServiceType
Inv SystemBoundary: maxSystemCapacity = self.serviceInstance -> col-
lect(currentCapacity)->sum() implies self.sla -> forAll(sla:SLA| let
services: sla.containedService -> select (s|s.serviceType = self) in
services -> collect(currentCapacity) -> sum () < services ->
at(1).slaParameter -> select (p|p.oclIsTypeOf(DataRate))->
at(1).maxAgreedValue implies services ->
forAll(srvc:Service|srvc.maxCurrentThreshold > srvc.slaParameter ->
select (p|p.oclIsTypeOf(DataRate)) -> at(1).mappedValue))
```

6.3.4.2 *Customer Side SLA Violation*

Unlike provider side violations, the violations from customers cannot be avoided. However, it is important to detect any service overuse by a customer to take the appropriate actions (e.g. charging or dropping the extra workload). By the following OCL constraint it is detected if a customer has violated an SLA:

```
Context DataRate
Inv CustomerViolation: self.mappedValue ≤ self.maxAgreedValue
```

6.3.5 The Trigger Metamodel

Whenever a system needs to be scaled because of potential SLA violation or low resource utilization, one or more triggers are generated to invoke the elasticity rules. To scale the system at runtime, a trigger is issued on a configuration entity to reconfigure the entity or on an entity type to add or remove instances of a type. Figure 6.11 shows the metamodel of triggers for scaling of the system. In this metamodel, the attribute *scalingType* can have the value of either *Increase* or *Decrease* and specifies whether an action to increase or decrease the resources is needed. The attributes *measurement* and *threshold* represent the measurements from the monitoring system and the current threshold value that has been violated. The values of threshold and measurement are used to determine the amount of resources that should be given to or released from the entity to resolve the violation of the received trigger. For example, if the current load on a node is 85% and the threshold on the node is 80%, the load of the node should be decreased by at least 5% to resolve the issued trigger.

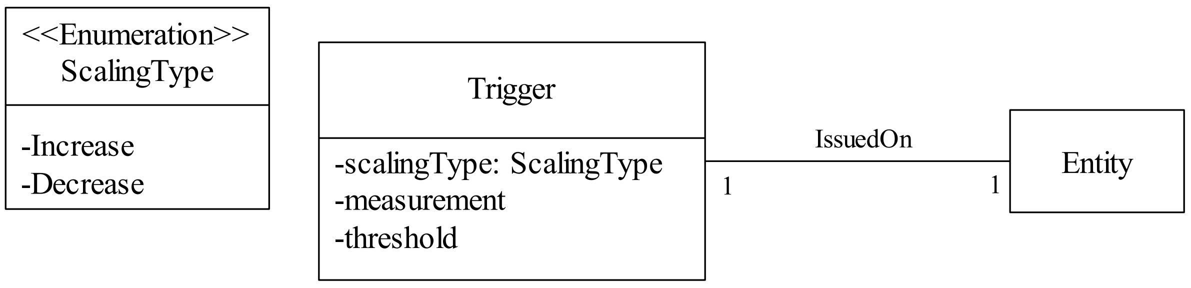


Figure 6.11 The trigger metamodel

6.4 Building/and Update the SLA Compliance Model

To build the SLA compliance model all SLA, measurement and the threshold models are combined at runtime. We use the Atlas Transformation Language (ATL) [14] transformation to implement this process. When any of the measurement or threshold models are updated or new/old SLAs are added/terminated, the SLA compliance model is updated too. New measurements arrive at the end of each measurement period. The measurement period should be long enough to process the previous measurements and reconfigure the system as necessary before the arrival of new measurements. After each reconfiguration, the thresholds model may also be updated before the new measurements arrive. Although new SLAs arrive or existing ones can be terminated at any time, we update the SLA compliance model at the end of each time interval. For illustration purposes, assume a new SLA (representing a new customer) arrives when a reconfiguration is being performed. If we update the SLA compliance model as soon as the new SLA arrives, when the previously generated triggers have not been resolved yet, at the validation of the updated SLA compliance model the same triggers may be regenerated. Handling the same triggers may cause instability in the system.

When a new SLA for a new customer arrives, all the SLA elements are added to the current SLA compliance model. Since no service instance is yet assigned to represent the workload of the new customer, for each service type contained in the new SLA, a *Service* model element with the current capacity of 0 is created in the SLA compliance model. This added element represents a service instance which needs to be added to represent the workload of the new customer. The validation of the updated SLA compliance model leads to the generation of a trigger to add that service instance. Similarly, when an SLA is removed, the elements related to only this SLA should be removed from the SLA compliance model together with their measurements and thresholds. This is achieved with a different transformation that takes the SLA to be removed and the SLA compliance model as input and generates a new SLA compliance model. In ATL language, the number of input and output models cannot be arbitrary; therefore, to add or remove multiple SLAs, we execute the corresponding transformation multiple times as required. In the prototype implementation the addition and removal of SLAs are done offline.

6.5 Trigger Generation

As mentioned earlier, the validation of SLA compliance model may lead to the generation of triggers. An SLA compliance model is valid when all the constraints of its metamodel are satisfied.

To generate a trigger, its different elements should be specified based on the constituent elements of the violated OCL constraint. In the SLA compliance metamodel, the constraints for the scaling of the system are defined on nodes and services. These OCL constraints are violated when there are not enough resources for such entities or their resources are in excess. Therefore, the entity on which a trigger is generated is the node or the service for which the respective OCL constraint is violated.

The constrained elements based on which the invariant is defined are the measurement and the threshold. The constraint checks if the measurement has reached the value of the current threshold. If the value of the threshold reaches (i.e. the constraint is violated), the constrained elements of the violated OCL constraints are extracted to specify the measurement and threshold elements of the generated trigger.

We use the names of the constraints as the *scalingType* of the generated triggers to initiate resource allocation (when the name is *Increase*) or release of surplus resources (when the name is *Decrease*).

6.6 Prototype Implementation and Preliminary Evaluation

In this section we present a preliminary evaluation of our SLA compliance management using a prototype implementation and discuss the results. We aim at analyzing the growth of the execution time with respect to the size of the SLA compliance model. Since the SLA compliance model contains SLAs model and a part of the configuration model, size of the system as well as size of the SLAs model (i.e. depends on the number of the SLAs and the number of included SLA parameters in each SLA) are reflected by the size of the SLA compliance model too.

The experiments were performed on a machine with an Intel® Core™ i7 with 2.7 GHz and 8 Gigabytes RAM and a Windows 7 operating system. Each test was performed five times and the average is reported here as the execution time.

6.6.1 Validation of the SLA Compliance Model and Trigger Generation

To generate triggers from violated OCL constraints, we used OCL APIs [16] in a standalone java application. The OCL constraints of SLA compliance profile are extracted and validated given an SLA compliance model. If a constraint is evaluated to false, a trigger is generated and takes the name of the violated constraint (i.e. either *Increase* or *Decrease*); corresponding entity is the context of the violated constraint (i.e. either a node or a service) and the measurement and threshold are from the constrained elements (i.e. the measurement and the threshold) of the violated constraint.

Table 1 presents the results for SLA compliance model validation and trigger generation given different SLA compliance models and measurements. The first column of the table is the number of elements in the SLA compliance model. The SLA compliance models differ in the number of nodes, the number of SLAs and the number of services of the same or different service types. Therefore, the size of the SLA compliance models reflects somehow the size of the system as well as the size of the SLAs model. It is worth mentioning that the size of the SLAs model depends on the number of the SLAs and the number of included parameters in

Table 6.1 SLA compliance model validation and trigger generation performance evaluation

	Number of Model Elements	Number of Constraint Checks performed	Number of Generated Triggers	Execution Time (ms)
CASE 1	13	7	1	694
CASE 2	16	11	2	1189
CASE 3	24	14	3	1377
CASE 4	26	18	4	1845
CASE 5	42	28	6	2844
CASE 6	87	77	8	7573

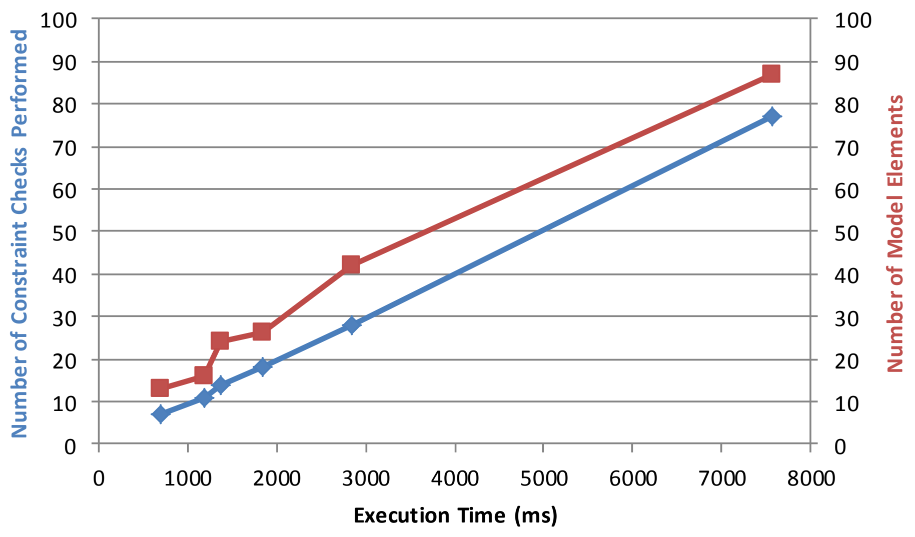


Figure 6.12 Performance evaluation for SLA compliance model validation and trigger generation given different SLA compliance models

each SLA. These SLA compliance models were built offline. For each case, the input measurements were compiled also offline in such a way that some would violate their corresponding thresholds. The second column is the total number of constraints to check. As explained in Section 6.3.4, constraints are defined on SLA parameters, service entities (i.e. SIs) and nodes where for service entities and nodes (i.e. configuration entities) more than one constraint is defined. As a result, as the size of the SLA compliance model increases with the increase of the number of nodes, service entities or SLAs, the number of constraints to check increases too. The third and fourth columns show the number of the generated triggers and the total execution time of the SLA compliance model validation and the trigger generation, respectively. The result of this evaluation is represented by the chart in Figure 6.12. As the number of elements in the SLA compliance model increases, more constraints are checked, and therefore the validation time increases. From the validation of the SLA compliance models we can conclude that the execution time grows linearly with respect to the numbers of constraints to check (in each case, the proportion of execution time to the number of constraints to check is almost 100).

6.7 Summary

Service providers aim at increasing their revenue by operating a system with the minimum amount of resources necessary to avoid SLA violation penalties. For this purpose, there is a need for an SLA management and dynamic reconfiguration that scales the system (up/down and in/out) according to the workload changes while avoiding SLA violations. In this chapter we described a part of our model-driven framework that checks the compliance of SLAs at runtime and generates triggers when dynamic reconfiguration is required. It is model driven; thus it is at the right level of abstraction. OCL constraints are written for categories of SLA parameters and are not specific for each parameter, which eases future extension. The SLA compliance management reuses models developed at the system design stage (e.g. Thresholds model). In this chapter, we discussed the usage of models to check the compliance of SLAs and generate triggers for dynamic reconfiguration. In the current implementation, the evolution of the SLA compliance models is done offline.

Chapter 7

Trigger Correlation for Dynamic System Reconfiguration

In this chapter we describe our trigger correlation and dynamic reconfiguration as part of the framework. At the end of this chapter, we evaluate the complexity of our dynamic reconfiguration approach and conduct experiments to evaluate its efficiency.

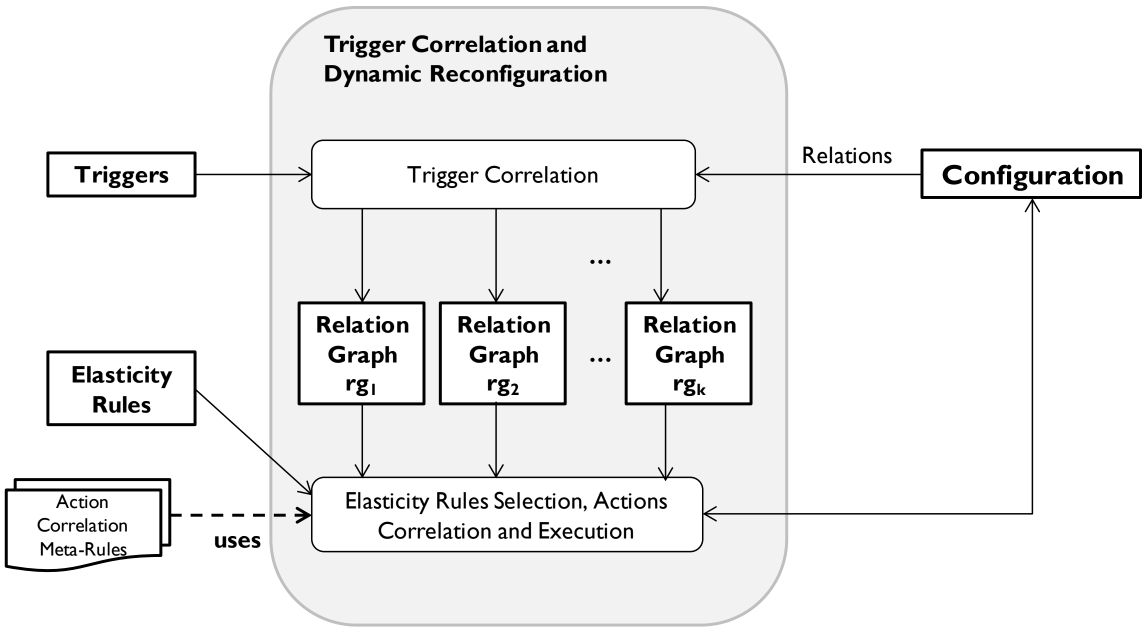
7.1 Introduction

Triggers generated from the SLA compliance management invoke elasticity rules, which consist of actions to take in the current system's situation. One may be tempted to handle each trigger separately. The issue in this case is that the triggers may be related, and handling them separately may lead to serious problems. For illustration purpose, let us assume two triggers t_1 and t_2 invoke two opposite elasticity rules e_1 and e_2, respectively, where e_1's action is to remove a node and e_2's action is to add a node. If the triggers are handled separately, resource oscillation [12] will certainly occur. In cloud systems due to the existence of multiple layers (infrastructure, platform and application layer), one root cause may generate multiple triggers in the different layers. For example, some workload decrease at the application layer may cause triggers at the application layer as well as triggers at the infrastructure layer. If these triggers are considered separately, the corresponding elasticity rules may remove some critical resources twice and this may jeopardize the availability of the service. As a result, correlation of the triggers and coordination of their actions is necessary.

To correlate triggers different solutions have been proposed in the literature [79, 80, 81, 60, 61, 82]. In these studies, a trigger is issued to signal an error caused by a fault in an entity of the system. As an error may propagate throughout the system, a large number of errors and therefore symptom triggers may be generated. The majority of these correlation solutions aim at eliminating the symptoms and identifying the root cause, i.e. the fault in the system. When the root cause is identified, appropriate actions are taken. In this thesis, a trigger is issued due to workload changes for an entity to invoke elasticity rules for resource allocation or deallocation. When triggers on related entities are issued simultaneously, these are not necessarily only "symptom" triggers because allocation of resources to one entity does not necessarily mean allocation of resources to the other ones and as a result they should not be simply eliminated. Instead, resource allocations/deallocations for related entities should be coordinated. In this chapter, we present an approach to correlate these triggers and also their related elasticity rules and actions (i.e. resource allocation or deallocation) to reconfigure the system. In order to achieve this, we defined meta-rules to coordinate the invoked actions at runtime.

7.2 The Overall Approach for Trigger Correlation and Dynamic Reconfiguration

Figure 7.1 shows the dynamic reconfiguration as part of the management framework in more details. In the *Trigger Correlation and Dynamic Reconfiguration* process, the triggers generated on related entities of the configuration are first correlated and a set of graphs called relation graphs are defined. For each trigger in a relation graph, the applicable elasticity rule is then selected. Since an elasticity rule may contain multiple alternative actions, based on the



Trigger Correlation and Dynamic Reconfiguration
Triggers
Trigger Correlation
Relations
Configuration
...
Relation Graph rg1
Relation Graph rg2
...
Relation Graph rgk
Elasticity Rules
Action Correlation Meta-Rules
uses
Elasticity Rules Selection, Actions Correlation and Execution

Figure 7.1 The overall approach for trigger correlation and dynamic reconfiguration process

current situation an optimal action among these alternatives is selected. The actions of two unrelated triggers do not impact each other. Therefore, actions of elasticity rules invoked by triggers in different relation graphs can be executed in parallel. For each relation graph, based on the relations between the triggers the optimal actions of the selected elasticity rules are correlated using a set of action correlation meta-rules. With the reconfiguration of the system, the values of the thresholds may also be updated. In the next sections we formally define the relation graphs and action paths as used in this thesis before elaborating on these processes in Section 7.4 and Section 7.5.

7.3 Modeling for Trigger Correlation and Dynamic Reconfiguration

7.3.1 The Relation Graphs Metamodel

When the triggers are correlated, a set of relation graphs are generated. Later, we use the generated relation graphs to correlate the actions of applicable elasticity rules (see Section 7.5). Figure 7.2 shows the metamodel of the relation graphs. Each relation graph consists of some triggers and relations between them. As shown in the figure, the relation between the

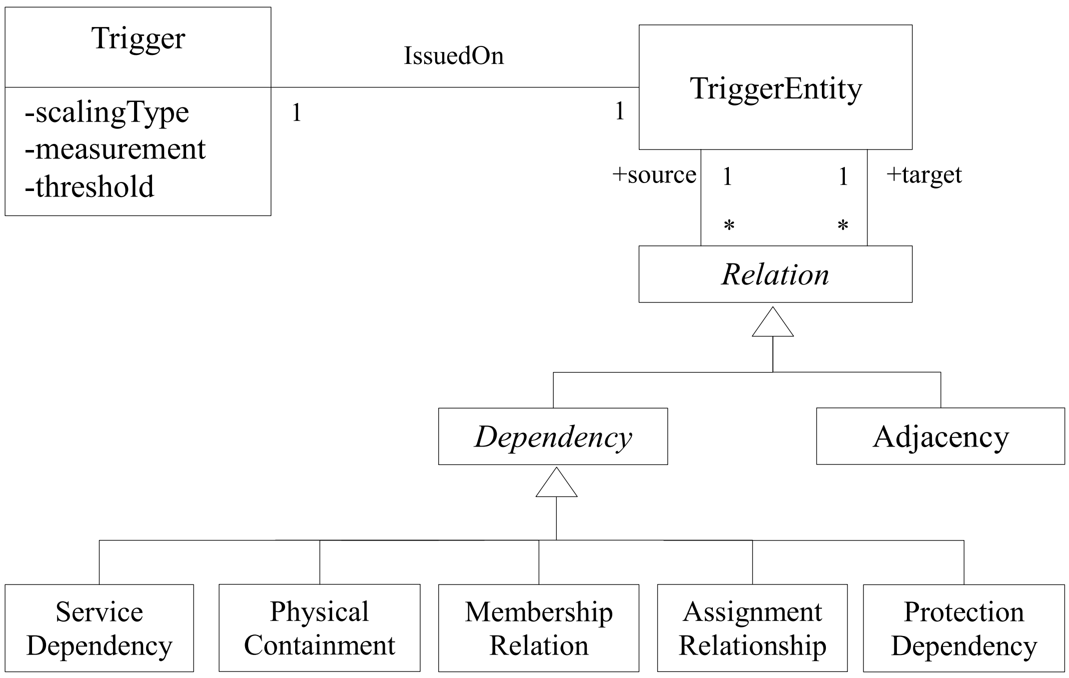


Figure 7.2 The relation graph metamodel

triggers is categorized into adjacency and dependency relations. The dependency relation is categorized further into service or protection dependency, assignment relationship, membership relation and physical containment. The different types of relations are explained in Section 7.4 in more details.

7.4 Trigger Correlation

Since triggers may lead to reconfiguration actions through the invocation of elasticity rules, when two configuration entities are related the actions to be applied may also be related and need to be coordinated. Therefore, we need to put into relation the triggers raised at the same time before correlating their respective actions invoked. To correlate triggers, we need to take into account the relations between the configuration entities they are related to. We first discuss the different types of relations between configuration entities before explaining how the triggers are correlated according to these relations.

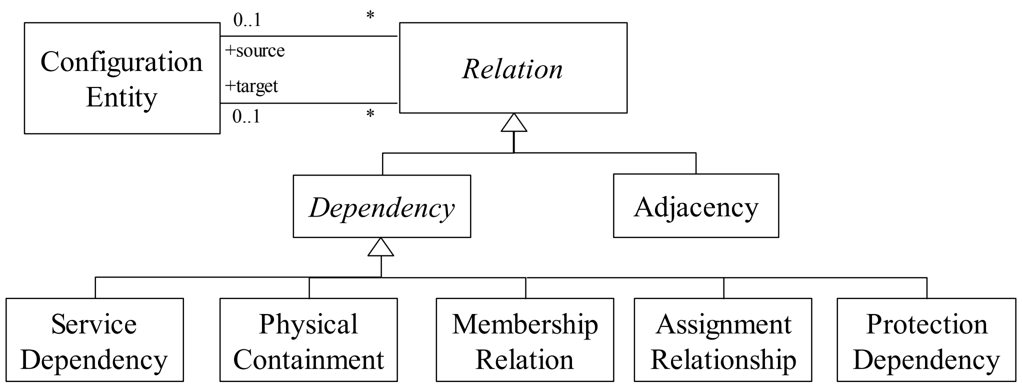


Figure 7.3 Different types of relations between configuration entities

As shown in Figure 7.3, the relations between entities can be of different types and categorized into two categories, dependency relations and adjacency relation. The first group consists of directed relations while the second defines a symmetric relation. The different types of relations are defined as follows:

Service Dependency: Dependency relation exists between services when the provision of one service (i.e. dependent entity) depends on the provision of another service (i.e. sponsor entity); therefore, the sponsor needs to be provided first to support the dependent.

Protection Dependency: This relation is defined between the active and the standby assignments of an SI.

Assignment Relationship: This relation is defined between a service unit (i.e. service provider entity) and the assignment (i.e. service entity) assigned to it.

Membership Relation: A membership relation exists between two entities when an entity is logically a member of a group represented by the other entity. For example, a node is a member of a node group.

Physical Containment: A physical containment relation exists between two entities if one entity is physically part of the other entity. In this case, the container entity (i.e. sponsor) provides resources for the contained entity (i.e. dependent). For example, a physical node is a container entity which provides resources for its hosted VMs as contained entities.

Adjacency: Two entities are adjacent when they are depending on (they are dependent of) the same sponsor. In this case, the common sponsor is called the common entity.

Trigger correlation is defined as a procedure of putting two or more triggers into relation, if any, to handle them together [83]. Correlated triggers are put into a relation graph where nodes are the triggers and edges represent the relations between these triggers. A set of relation graphs is automatically generated based on the triggers and the relations between the entities they correspond to. The algorithm correlating triggers generated for the same measurement period is provided in the appendix (*Algorithm1*). The entities of the triggers are looked up in the current configuration. Any relations between these entities are transferred to their associated triggers.

7.5 Elasticity Rule Selection and Execution

After the correlation of triggers, the generated relation graphs are processed in parallel. For each relation graph, the applicable elasticity rules are selected and their actions are correlated. The correlated actions are executed on the fly. Therefore, in our approach, we do not build or evaluate different action paths before their execution.

In this section, we first introduce our approach of selecting the applicable elasticity rules given the correlated triggers; then we explain the selection of the optimal action among all the alternatives available for execution. We also introduce a set of meta-rules used for the correlation of the optimal actions of the selected elasticity rules.

7.5.1 Selecting Applicable Elasticity Rules

The generated triggers invoke the applicable elasticity rules. On the one hand side, a trigger is issued on a configuration entity when any of its current threshold values is reached. On the other hand, an elasticity rule specifies the actions that can be taken on an instance of a given type to resolve a given type of threshold violation. Therefore, an elasticity rule is considered for invocation if the *entityType* for which the elasticity rule is defined is the same as the type of the entity on which the trigger was generated.

The *scalingType* of a trigger is either *Increase* to initiate resource allocation or *Decrease* to release surplus resources. On the other hand, the *scalingRule* of an elasticity rule is *Increase* if its actions add resources; and it is *Decrease* if its actions remove resources. As a result, for an elasticity rule to be applicable its *scalingRule* should be equal to the *scalingType* of the trigger (see *Algorithm 2* in the appendix). It is worth noting that the applicability of an elasticity rule is different from the applicability of its contained actions. Once the applicable elasticity rule is selected, the applicability and feasibility of its contained actions should be determined to select the optimal action (see Section 7.5.2).

7.5.2 Selecting the Optimal Action

In an elasticity rule, multiple actions may be specified. When such an applicable elasticity rule is invoked by a trigger, among these alternatives an optimal action needs to be selected for execution depending on the condition and the prerequisite(s) met (see *Algorithm 3* in the appendix).

To be considered as optimal, the contained action of the invoked elasticity rule should at least be applicable in the current situation (i.e. its corresponding condition must be evaluated to true). It also needs to be feasible, so among the applicable alternatives, the feasible action with the least cost is selected if there is such. In the case that none of the applicable alternatives are feasible, the infeasible action with least *midCost* is selected for invocation and an appropriate prerequisite trigger is generated. If the cost of an infeasible action is less than the cost of a feasible action, we still select the feasible one because according to the current situation no prerequisite action is required, which more likely results in an efficient reconfiguration.

For illustration purpose, let us assume that due to workload increase, a trigger (T_1) for scaling the system is received from the SLA compliance management (see Figure 7.4). Based on the *scalingType* of the trigger (increase) and the related entity, the applicable elasticity rule is selected and invoked. As shown in the figure, the actions of invoked elasticity rules have a cost. In the elasticity rule invoked by T_1, the defined action is the addition of an assignment. To add an assignment, the prerequisite is that there should be an SU in the SG to which the new assignment can be assigned. If there is no such SU in the SG (i.e. the prerequisite is not met), a prerequisite trigger is generated to initiate another elasticity rule for reconfiguring the SG by adding to it a new service unit. However, a service unit requires a node to host it. If there is no such node, a prerequisite trigger is generated again to invoke the corresponding elasticity rule. The actions contained in the node elasticity rule are: moving out some workload to other nodes with approximate cost of 500 which is feasible if there are enough providers for them; scaling up the node with approximate cost of 300, which is applicable if the node has not reached the maximum capacity yet and feasible if the physical node hosting the node has enough resources. Considering the current situation, assume that all of the contained actions are applicable, but only the first action which is moving out some workload is feasible. In this example, the first action is chosen as the optimal

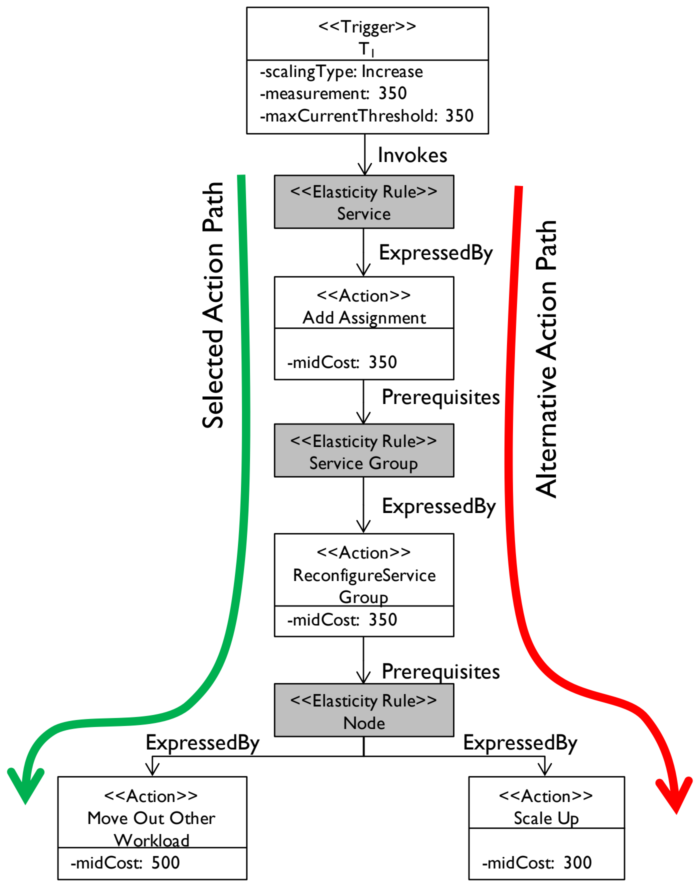


Figure 7.4 An example of invoked action path

action as all the action's prerequisites are met and most likely it will result in an efficient reconfiguration in terms of cost.

In this example, trigger T_1 leads to the invocation of multiple elasticity rules where the invocation of one elasticity rule is a prerequisite for another one. The path resulted from the execution of an elasticity rule is called an action path.

7.5.3 Action Correlation Meta-Rules

A meta-rule is a higher level rule that governs the application of other rules by indicating how to apply them [84]. In this thesis, we use higher level rules to govern the application of elasticity rules and execute their actions on the fly. We refer to these rules as action correlation meta-rules and their applicability is governed by the relations between the triggers. *Al-*

gorithm 4, provided in the appendix, is used to apply the action correlation meta-rules at runtime. They have been implemented as Atlas Transformation Language (ATL) [29] lazy rules in our framework.

7.5.3.1 *Meta-Rules for Dependency Relation*

Triggers on a sponsor entity can be due to the violation of one its thresholds and because of its dependent(s) as to take an action on a dependent entity, first the capacity of its sponsor is checked as a prerequisite. If both cases apply and a prerequisite action is taken to provide a sponsor first, it may also resolve the sponsor's trigger. To illustrate, let us assume that the workload for a service represented by an SI has more than one active assignment. Suppose at some point in time, the workload increases and two triggers are generated: One on the service instance (dependent) and one on the node (sponsor) which supports one of the assignments of the service instance. In this example, the least costly action of the elasticity rule invoked by the dependent trigger is executed first, which is adding an assignment on another node (i.e. the system is scaled out). Once the action path of the dependent entity is executed, the workload is shared between more nodes and therefore less workload will be imposed on the original sponsor node for which the sponsor trigger was received. As a result, the sponsor trigger may be resolved and to determine that the sponsor trigger needs to be updated. As a result, the first meta-rule for the dependency relation is defined to handle horizontal scaling (i.e. scaling out). It is as follows:

- ***Meta-Rule 1:*** If the relation between triggers is of type physical containment or assignment relationship and the optimal action for resolving the dependent trigger is scale-out, the action path for the dependent entity is executed before the path for the sponsor entity.

Meta-Rule 1 handles the cases where the relation between the triggers is of type physical containment or assignment relationship and the execution of the action path for the dependent provides solution for the sponsor through adding a new sponsor (i.e. scaling out). Note that it is possible that adding an assignment was not the least costly action or it was not an option at all and therefore the first meta-rule is not always applicable.

- ***Meta-Rule 2:*** If multiple triggers have physical containment relations with the same container trigger and the optimal action for resolving each contained trigger is scale-up, some of the corresponding entities of the contained triggers may be migrated base on the cost of the migration. The corresponding entities of the contained triggers are sorted in ascending order using the metric *m = (migrationCost/releasedResource)*, where *migrationCost* is the approximate cost of migrating the contained entity to another container and *releasedResource* is the amount of resources released by migration. The contained entities with smaller *m* are migrated until the container trigger is resolved.

Unlike *Mata-Rule 1*, *Meta-Rule 2* handles vertical scaling (i.e. scale up). According to *Meta-Rule 2*, if multiple contained entities (i.e. dependents such as VMs) depending on the same container (i.e. sponsor such as a physical host) need to be scaled up but the container does not have enough resources for all of them, one or more contained entities (i.e. dependents) whose migration release more critical resources with less cost are migrated to other containers first to release resources of the container. The released resources of the sponsor can then be given to the remaining dependent entities to scale up.

If the relation between triggers is of type dependency, but neither *Meta-Rule 1* nor *Meta-Rule 2* can be applied, still we need to make sure that the action paths of dependent and sponsor are not executed simultaneously. For that purpose, we execute the action path of the sponsor before the action path of dependent. Therefore, the third meta-rule is defined as follows:

- ***Meta-Rule 3:*** If the relation between triggers is of type dependency but none of *Meta-Rule 1* and *Meta-Rule 2* can be applied, the action path for the sponsor entity is executed before the path for the dependent entity.

7.5.3.2 *Meta-Rules for Adjacency Relation*

When triggers invoke elasticity rules on adjacent entities, it is possible that the actions of the elasticity rules would like to manipulate the common sponsor entity of the adjacent entities (i.e. their container or sponsor) simultaneously. These actions may be conflicting or interfering. To prevent such conflicts, only one action at a time is taken on the common entity, i.e. the actions are ordered. The order of actions on the common entity affects the efficiency of reconfiguration. To optimize it, the following meta-rules are defined:

- ***Meta-Rule 4:*** The actions releasing resources of the common entity are taken first.
- ***Meta-Rule 5:*** Any action that would remove a common resource/entity (e.g. removing a node) is considered only after executing all the action paths of all adjacent triggers.

When executing the action paths, triggers which release resources are given higher priority than triggers which allocate resources to enable reallocation. However, the actions releasing resources of the common entity are delayed until all the adjacent triggers have been considered. Thus, the resources of the common entity are released at the end only if they have not been reallocated by corresponding actions of other adjacent triggers. When all the resources of a common entity can be removed, the common entity is removed as well (e.g. a service group is removed when its entire member service units can be removed or when a common entity such as node has no process to run).

7.6 An Example for Trigger Correlation and System Reconfiguration

Suppose at some point in time, the configuration is as shown in Figure 7.5 (a). In this configuration, there are two service groups (Service Group$_1$ and Service Group$_2$) which are protecting three SIs (SI$_1$, SI$_2$ and SI$_3$). The service units of Service Group$_1$ can be hosted only on the nodes of Node Group$_1$ (Node$_1$, Node$_2$, Node$_3$ and Node$_4$) and the service units of Service Group$_2$ can be hosted only on nodes of Node Group$_2$. As shown in the figure with the service dependency relation, the provisioning of the service represented by SI$_2$ depends on the provisioning of the service represented by SI$_3$. In this example, each assignment of SI$_2$ requires one assignment of SI$_3$. At this point of time, two triggers (t_1 and t_2) are generated by the SLA compliance management framework for SI$_1$ and SI$_2$, respectively. Assume Trigger t_1 is generated due to the decrease in the workload represented by SI$_1$ to the point that two assignments should be removed, and Trigger t_2 is generated due to the increase of the workload represented by SI$_2$.

To reconfigure the system, first the triggers issued on related entities are put into relation. SI$_1$ and SI$_2$ are protected by the same service group (having the same logical container). Therefore, their corresponding triggers are put in the adjacency relation. In this relation, the common entity is Service Group$_1$. Figure 7.5 (b) shows the relation graph resulted from the trigger correlation process.

Next, the applicable elasticity rules are selected and based on the defined action correlation meta-rules, the triggers of the relation graph are ordered for the invocation of the applicable elasticity rules. Based on *Meta-Rule 4* for the adjacency relation, the action path resulting from t_1 should be executed before the action path resulting from t_2 because the *scalingType* of Trigger t_1 is *Decrease*. Therefore, Trigger t_1 is considered first and its corresponding elasticity rule is executed. According to the elasticity rule for SI_1, two assignments should be removed to reconfigure the system. Figure 7.5 (c) shows the configuration resulting from the removal of assignments. As shown in the figure, by the removal of assignments, service units hosted on $Node_3$ and $Node_4$ become unassigned (without assignments). Considering *Meta-Rule 5*, the issue of follow-up trigger on Service $Group_1$ as common entity is delayed till the adjacent Trigger t_2 manipulates the common entity.

According to the elasticity rule initiated by Trigger t_2, one assignment should be added to handle the workload increase represented by SI_3. To take this action, two prerequisites should be met: There should be a service unit in a service group to which the added assignment can be assigned and also its sponsor should have enough capacity to support the increase. The first prerequisite can be met by Service $Unit_3$ or Service $Unit_4$. Since according to the service dependency each assignment of SI_2 needs one assignment of SI_3, the increase in the workload represented by SI_2 cannot be sponsored by the current number of SI_3's assignments. Therefore, the second prerequisite is not met by the current configuration. To make the action feasible, a prerequisite trigger on SI_3 is generated to increase the sponsor's capacity. The generated prerequisite trigger invokes the elasticity rule for SI_3. According to SI_2's elasticity rule, one assignment of SI_3 should be added to resolve the prerequisite trigger; however, the action cannot be taken until Service $Group_2$ is reconfigured in a way that the added assignment can be assigned. Therefore as a prerequisite, the required service unit should be added to Service $Group_2$ first. To add a service unit, there should be a node to provide the required resources for the added service unit. Although $Node_4$ has enough resources, it cannot host the service units of Service $Group_2$ because $Node_4$ is not a member of Node Group$_2$ on which Service Group$_2$ can be configured. Since this perquisite is not met, first a node is added so that the service unit can be added to Service Group$_2$. Figure 7.5 (d) shows the configuration resulting from the execution of *Action Path$_2$*. Note that node groups are shown in Figure 7.5 (a) only.

Once all adjacent triggers (i.e. t_1 and t_2) with the same common entity have been processed, the delayed follow-up trigger on Service Group$_1$ can be evaluated and therefore the service unit hosted on Node$_4$ is removed from Service Group$_1$. Node$_4$ does not have any running service units. Therefore, the resource removal action can be taken at this moment. Figure 7.5 (e) shows the configuration resulting from the execution of the delayed follow-up actions in Action Path$_1$. As explained in this example, the action paths are not pre-built and the actions are executed right away.

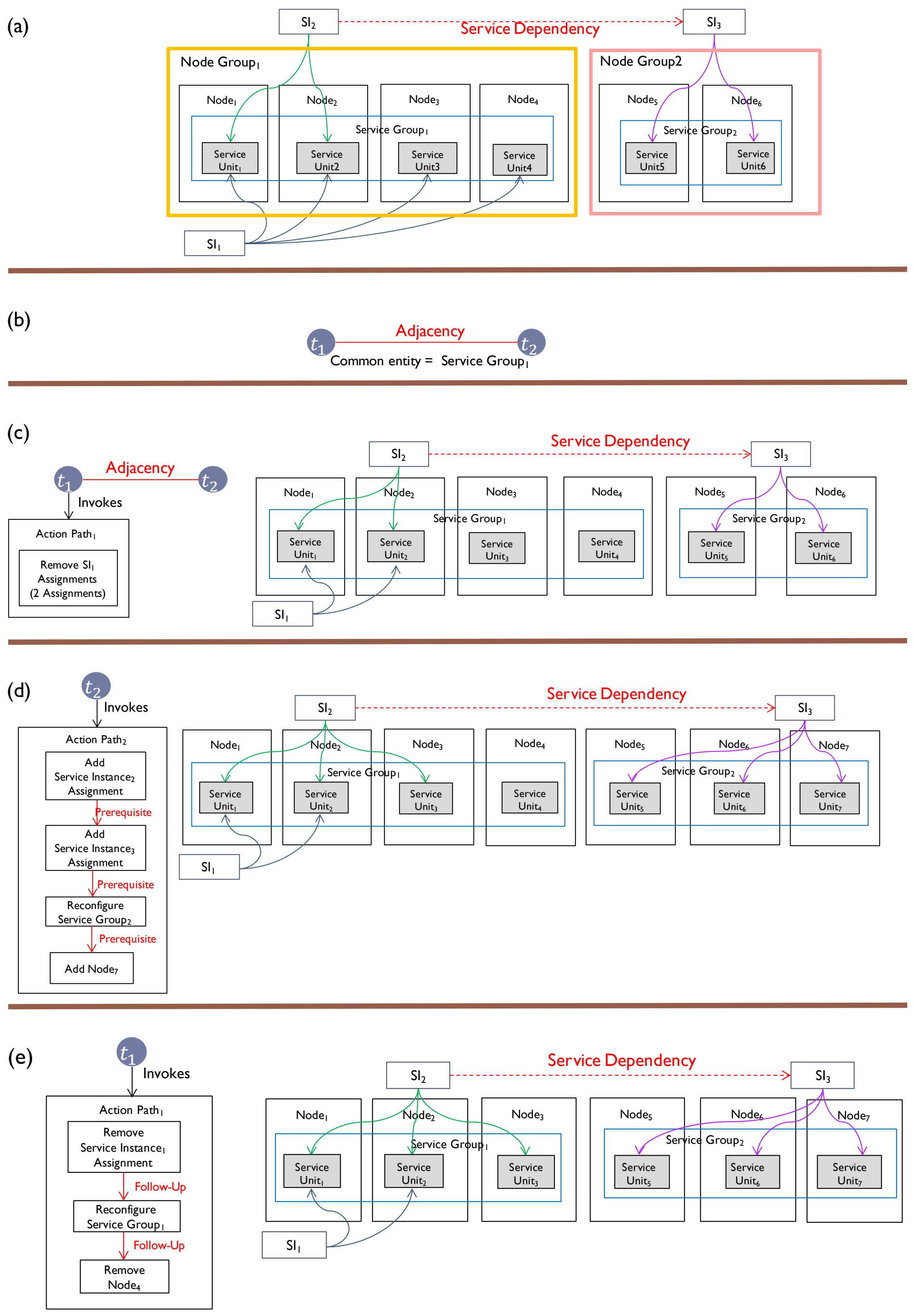
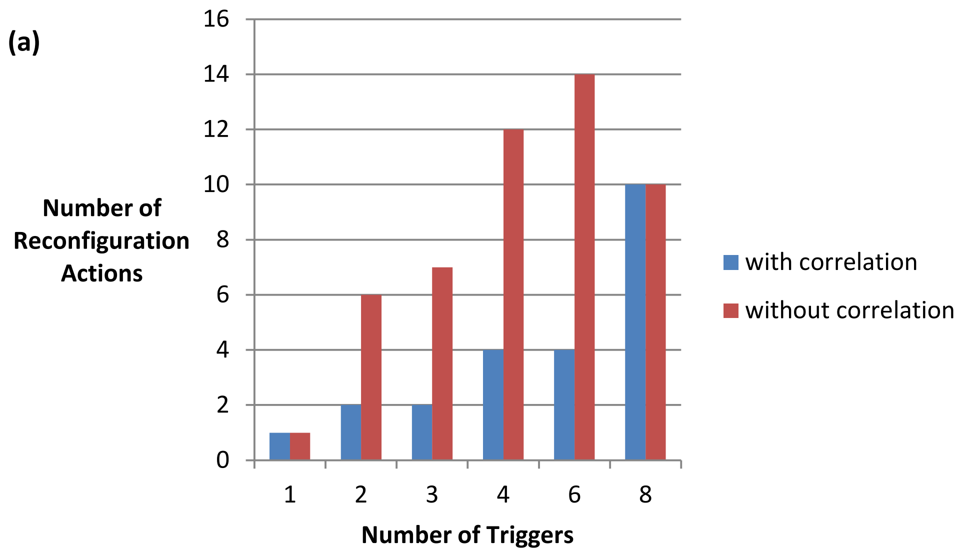


Figure 7.5 System reconfiguration-An example

7.7 Prototype Implementation and Experimental Evaluation

We implemented a prototype of trigger correlation and dynamic reconfiguration using ATL [14]. To analyze the efficiency of our approach for trigger correlation and dynamic reconfiguration, we consider the triggers generated in the previous experiment (i.e. Section 6.6). To perform this, the same machine with the same specification as the previous experiments in Section 6.6 was used. For each case, we also used different AMF configuration models as the current configurations. The AMF configuration models conform to AMF UML profile defined in [23]. It is worth mentioning that the generated triggers are not redundant and therefore, with the correlation approach the number of triggers remains the same after correlation. Since we manipulate the models, the execution time is the time of making changes in the configuration model and does not include the execution time of the actions. For example, when a node is added, this addition manifests as a change in the number of instantiated nodes in the configuration model; however, in real systems, creation of VM instances may take several minutes [85]. As a result to analyze the efficiency of our approach, we measure the execution time as well as the number of reconfiguration actions with our approach where the triggers as well as the actions of invoked elasticity rules are correlated, and compare them to the execution time and number of actions when the triggers are not correlated. Figure 7.6 shows the result of this comparison. As shown in Figure 7.6 (a), the results demonstrate the reduction in the number of actions by the correlation approach in overall which means less applicable elasticity rules are selected and invoked at runtime. As a result as shown in Figure 7.6 (b), by the correlation approach the execution time which includes the correlation time is reduced in overall as well. As the actions are executed at runtime, reducing the number of reconfiguration action is an important goal for real time and highly available systems. In the case that the triggers are not related (like last case in Figure 7.6),



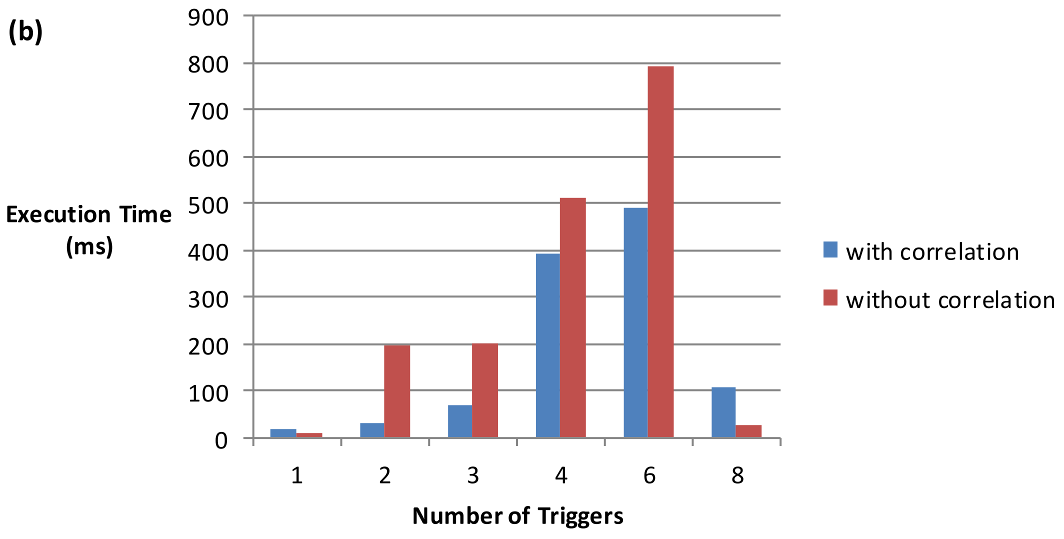


Figure 7.6 Comparison of the execution time and the number of reconfiguration actions for dynamic reconfiguration with correlation and dynamic reconfiguration without correlation

the execution time is more in our approach which is due to the time for checking relations between the triggers to correlate them. It is worth mentioning that the stability of the system is not guaranteed when the triggers are not correlated.

7.8 Summary

Since multiple triggers may be generated simultaneously, handling the triggers independently may jeopardize the stability of the system. In this chapter, we proposed a model driven approach for correlating the triggers and the actions of their related elasticity rules. Triggers are correlated based on the relations existing between their corresponding configuration entities. The result of trigger correlation is represented as a set of relation graphs. For each trigger of a relation graph, the applicable elasticity rule is then selected. In order to correlate the actions of applicable elasticity rules, we defined action correlation meta-rules that govern the application of elasticity rules when the triggers are correlated. The goal is not only to reconfigure the system properly and avoid resource oscillation but also to minimize the number of reconfiguration actions because any change in the configuration needs to be applied at runtime on system entities. Moreover, to reduce overhead and react to workload changes in a timely manner the correlated actions are executed on the fly. I.e. no action path is evaluated or built before the execution. A correlated action is executed right away once its prerequisites are met if there is any.

We performed some experiments that show that our solution reduces the time of the dynamic reconfiguration and the number of reconfiguration actions while avoiding resource oscillation compared to the reconfiguration solution without trigger correlation.

In our approach, to choose an action among alternatives the cost of actions is considered, but this is an approximation and it includes the approximate cost of prerequisites as well. Thus, it is not guaranteed that the resulting reconfiguration is the best solution in terms of cost.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

In this thesis, we presented a model-driven management framework for user requirement decomposition, offline elasticity rule generation, SLA compliance management and dynamic reconfiguration using the generated elasticity rules.

In [11] an approach has been developed to generate automatically a valid configuration starting from service configurations and software catalog. However, specifying the service configurations requires extensive domain knowledge and expertise. To alleviate the work of designer, we devised a model driven approach to generate automatically service configurations from the user requirements. For this purpose, the user requirements are decomposed automatically with the decomposition knowledge captured in a service ontology model and the COTS components that satisfy both functional and non-functional requirements are selected. When a valid set of component types is selected, traceability links between the requirements and the selected component types are automatically generated. After successful decomposition, the service ontology model may be updated with any alternative decomposition for existing functionalities it includes. When a solution is dismissed due to missing component types or non-functional requirements that cannot be satisfied, the trace links between the rejected solutions and the requirements are generated. This model may help with the management decisions to modify a user requirement if it is necessary.

To manage the reconfiguration of the system at runtime, we use a set of patterns called elasticity rules. While the system's configuration is designed (i.e. generated automatically), the calculations used to dimension the system as well as some computed parameters are reused to define the elasticity rules. The elasticity rules considered in this thesis are at a finer granularity than what is presented in the related work as we also consider the rearrangement of resources and not only the addition and removal.

To adapt the system based on the workload fluctuation at runtime, we proposed a model-driven approach which reuses the models developed at the design stage (e.g. configuration model). We defined OCL constraints that are periodically evaluated at runtime to generate triggers automatically from the violated OCL constraints. The defined constraints are general and defined for categories of SLA parameters; therefore, when a new SLA parameter is introduced and needs to be considered for checking, there is no need to add a new constraint based on the new parameter as long as it fits into one of the categories we have defined.

The generated triggers initiate the application of corresponding elasticity rules to reconfigure the system and avoid SLA violations by the provider and resource wasting. Since multiple triggers may be generated simultaneously, they may invoke elasticity rules that contain actions which impact each other. As a result, handling the triggers independently may jeopardize the stability of the system. In this thesis, we proposed a model based approach for trigger correlation, elasticity rule selection and the coordination of the related actions. In order to correlate the actions and execute them on the fly, we have defined some action correlation meta-rules that govern the application of elasticity rules when the triggers are correlated. The goal is not only to reconfigure the system properly and avoid resource oscillation but also to minimize the number of reconfiguration actions because any change in the configuration needs to be applied at runtime on system entities. This approach is part of our management framework.

We performed some experiments that show that our solution reduces the time of the dynamic reconfiguration and the number of reconfiguration actions while avoiding resource oscillation compared to the reconfiguration solution without trigger correlation. The work presented in this thesis has been published in [86, 87, 88, 89, 90, 91, 92].

8.2 Future Research

In this section we briefly discuss the issues which are left open in this thesis and that can be considered in the future.

8.2.1 Elasticity Rule Generation

In this thesis, we proposed to use the system dimensioning information to define elasticity rules at design time. Therefore, the elasticity rules generation requires the configuration generation process. We discussed our approach in the context where the service and service provider perspectives are described explicitly in the configuration. The approach can be extended to apply it in more general context where there is no distinction between the service and service provider.

8.2.2 SLA Compliance Management

To provide resources in a proactive manner, the values of current thresholds are less than the current capacity of the system. In this thesis, we assumed the values of the threshold for the capacity of the system is given. As future work, the monitoring period, reconfiguration time and the predicted workload as well as the cost of reconfiguration versus the penalty of SLA violations can be considered to determine the values of the thresholds.

8.2.3 Dynamic Reconfiguration

As mentioned in Chapter 7, the goal of our approach for dynamic reconfiguration is to reconfigure the system properly while reducing the number of reconfiguration action and optimizing resource utilization. For this purpose, some action correlation meta-rules have been introduced. Future work can involve the design of new heuristics as new action correlation meta-rules to improve the performance of the proposed approach. For this purpose, a set of large scale configurations need to be considered as case studies to analyze the performance of different heuristics.

In our approach, we assumed the load that is imposed on a node by requests of a service is estimated by a given function at runtime. As future work, the parameters that characterize the workload as well as the node (e.g. the types of workload the node currently supports, the operating system, etc.) can be considered to define the function.

In this thesis, we only handled the triggers which lead to dynamic reconfiguration. The future work can involve the approach for handling design related triggers by which the system configuration has to be redesigned. For example, the system configuration needs to be modified due to frequent service outage and consider for instance alternate components. The traceability links generated at configuration time can be reused to identify on which parts of the configuration the modifications should be applied.

Bibliography

[1] M. Toeroe and F. Tam, Service Availability: Principles and Practice, John Wiley & Sons, 2012.

[2] A. Kanso, "Automated Configuration Design and Analysis for Service High-Availability," PhD thesis, Concordia University, 2012.

[3] K. D. Larson, "The Role of Service Level Agreements in IT Service," *Information Management & Computer ,* vol. 6, no. 3, pp. 128-132, 1998.

[4] N. R. Herbst, S. Kounev and R. Reussner, "Elasticity in Cloud Computing: What It Is, and What It Is Not," in *International Conference on Autonomic Computing (ICAC13)*, San Jose, 2013.

[5] "MDA," [Online]. Available: http://www.omg.org/mda/. [Accessed 11 July 2018].

[6] B. Selic, "The Pragmatics of Model-Driven Development," *IEEE software,* vol. 20, no. 5, pp. 19-25, 2003.

[7] "MDA User Guide, version 1.0.0," 01 06 2003. [Online]. Available: https://www.omg.org/mda/mda_files/MDA_Guide_Version1-0.pdf. [Accessed 23 01 2015].

[8] B. Selic, "A Systematic Approach to Domain-Specific Language Design Using UML," in *Object and Component-Oriented Real-Time Distributed Computing*, 2007.

[9] "OMG Unified Modeling language (OMG UML), Superstructure," 2011. [Online]. Available: https://www.omg.org/spec/UML/2.4.1/About-UML/. [Accessed 11 July 2018].

[10] "OMG Object Constraint Language (OCL), version 2.4," February 2014. [Online]. Available: http://www.omg.org/spec/OCL/. [Accessed 11 July 2018].

[11] P. Pourali, "Pattern-Based Generation of AMF Configurations," *Master thesis, Concordia University,* 2014.

[12] X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, P. Padala and K. Shin, "What Does Control Theory Bring to Systems Research?," *ACM SIGOPS Operating Systems Review,* vol. 43, no. 1, pp. 62-69, 2009.

[13] "Service Availability Forum," [Online]. Available: http://www.saforum.org. [Accessed 11 July 2018].

[14] "ATL/User Guide-The ATL Language," [Online]. Available: https://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language. [Accessed 11 July 2018].

[15] "Eclipse Modeling Framework (EMF)," [Online]. Available: https://www.eclipse.org/modeling/emf/. [Accessed 1 November 2017].

[16] "OCL - Eclipsepedia - Eclipse Wiki," [Online]. Available: https://wiki.eclipse.org/OCL. [Accessed 01 August 2017].

[17] "Service Availability Forum, “Application Interface Specification, Availability Management Framework”, SAI-AIS-AMF-B.04.01," [Online]. Available: http://www.saforum.org/HOA/assn16627/images/SAI-AIS-AMF-B.04.01.pdf. [Accessed 18 June 2016].

[18] "OMG Unified Modeling Language (OMG UML) Infrastructure, version 2.4.1," 2011. [Online]. Available: https://www.omg.org/spec/UML/2.4.1/About-UML/. [Accessed 11 July 2018].

[19] M. Alhaj and D. C. Petriu, "Traceability Links in Model Transformations between Software and Performance Models," in *SDL*, 2013.

[20] I. Galvao and A. Goknil, "Survey of Traceability Approaches in Model-Driven Engineering," Object Computing Conference, 2007. EDOC 2007. 11th IEEE International, 2007.

[21] O. C. Gotel and A. C. Finkelstein, "An Analysis of the Requirements Traceability Problem," in *Requirements Engineering*, 1994.

[22] "Service Availability Forum, Hardware Platform Interface, Specification,SAI-HPI-B.03.02," [Online]. Available: http://www.saforum.org/HOA/assn16627/images/SAI-HPI-B.03.02.pdf. [Accessed 11 July 2018].

[23] P. Salehi, A. Hamoud-Lhadj, M. Toeroe and F. Khendek, "A UML-Based Domain Specific Modeling Language for the Availability Management Framework," *Computer Standards & Interfaces,* vol. 44, pp. 63-83, 2016.

[24] K. Czarnecki, S. Helsen and U. Eisenecker, "Staged Configuration Using Feature Models," in *Third Software Product-Line Conference (SPLC'04)*, 2004.

[25] L. Chung, W. Ma and K. Cooper, "Requirements Elicitation Through Model-Driven Evaluation of Software Components," in *5th International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems*, 2006.

[26] Z. J. Oster, G. R. Santhanam and S. Basu, "Decomposing the Service Composition Problem," in *8th IEEE European Conference on Web Services*, 2010.

[27] Z. J. Oster, G. R. Santhanam and S. Basu, "Identifying Optimal Composite Services by Decomposing the Service Composition Problem," in *IEEE European Conference on Web Services*, 2011.

[28] J. Lin, M. S. Fox and T. Bilgic, "A Requirement Ontology for Engineering Design," in *Third ISPE International Conference on Concurrent Engineering*, 1996.

[29] C. Bartsch, L. Shwartz, C. Ward, G. Grabarnik and M. J. Buco, "Decomposition of IT Service Processes and Alternative Service Identification using Ontologies," in *Network Operations and Management Symposium (NOMS 2008)*, 2008.

[30] M. Kassab, O. Ormandjieva and M. Daneva, "An Ontology Based Approach to Non-Functional Requirements Conceptualization," in *Software Engineering Advances, 2009. ICSEA '09. Fourth International Conference on*, 2009.

[31] M. Kassab, O. Ormandjieva and M. Daneva, "A Traceability Metamodel for Change Management of Non-Functional Requirements," in *Sixth International Conference on Software Engineering Research, Management and Applications*, 2008.

[32] A. Rashwan, O. Ormandjieva and R. Witte, "Ontology-Based Classification of Non-Functional Requirements in Software Specifications: A New Corpus and SVM-Based Classifier," in *Computer Software and Applications Conference (COMPSAC)*, 2013.

[33] X. Franch and J. P. Carvallo, "Using Quality Models in Software Package Selection," *IEEE Software,* vol. 20, no. 1, pp. 34-41, 2003.

[34] H. Wada, J. Suzuki and K. Oba, "Modeling Non-Functional Aspects in Service Oriented Architecture," in *IEEE International Conference on Services Computing SCC06*, 2006.

[35] R. Heckel, M. Lohmann and S. Thone, "Towards a UML Profile for Service-Oriented Architectures," in *Model Driven Architecture: Foundations and Applications*, 2003.

[36] L. M. Cysneiros and J. C. S. d. P. Leite, "Using UML to Reflect Non-Functional Requirements," in *Proceedings of theconference of the Centre for Advanced Studies on Collaborative research*, 2001.

[37] M. Alhaj and D. C. Petriu, "Approach for Generating Performance Models from UML Models of SOA Systems," in *Proceedings of CASCON'2010*, Toronto, 2010.

[38] A. Mate and J. Trujillo, "A Trace Metamodel Proposal Based on the Model Driven Architecture Framework for the Traceability of User Requirements in Data Warehouses," *Information Systems,* vol. 37, no. 8, pp. 753-766, 2012.

[39] M. D. Del Fabro, J. Bezivin, F. Jouault and P. Valduriez, "Applying Generic Model Management to Data Mapping," in *BDA*, 2005.

[40] E. Barret, E. Howley and J. Dugan, "Applying Reinforcement Learning towards Automating Resource Allocation and Application Scalability in the Cloud," *Concurrency and Computation: Practice and Experience,* vol. 25, no. 12, pp. 1656-1674, 30 May 2012.

[41] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre and I. Truck, "Using Reinforcement Learning for Autonomic Resource Allocation in Clouds: towards a Fully Automated Workflow," in *The Seventh International Conference on Autonomic and Autonomous Systems (ICAS 2011)*, 2011.

[42] G. Tesauro, N. K. Jong, R. Das and M. N. Bennani, "A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation," in *IEEE International Conference on Autonomic Computing (ICAC'06)*, 2006.

[43] A. Galstyan, K. Czajkowski and K. Lerman, "Resource Allocation in the Grid Using Reinforcement Learning," in *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, New York, July 19 - 23, 2004.

[44] J. Rao, X. Bu, C.-Z. Xu, L. Wang and G. Yin, "VCONF: A Reinforcement Learning Approach to Virtual Machines Auto-configuration," in *Proceedings of the 6th international conference on Autonomic computing (ICAC'09)*, Barcelona, Spain, June 15–19, 2009.

[45] Y. Al-Dhuraibi, F. Paraiso and N. Djarallah, "Elasticity in Cloud Computing: State of the Art and Research Challenges," in *IEEE Transactions on Services Computing*, 01 June 2017.

[46] P. Jamshidi, A. Ahmad and C. Pahl, "Autonomic Resource Provisioning for Coud-Based Software," in *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2014)*, Hyderabad, India, June 02 - 03, 2014.

[47] J. Domaschka, K. Kritikos and A. Rossini, "Towards a Generic Language for Scalability Rules," in *European Conference on Service-Oriented and Cloud Computing*, 2014.

[48] C. Chapman, W. Emmerich, F. G. Márquez, S. Clayman and A. Galis, "Software Architecture Definition for On-demand Coud Provisioning," *Cluster Computing,* vol. 15, no. 2, pp. 79-100, 2012.

[49] G. Copil, D. Moldovan, H.-L. Truong and S. Dustdar, "SYBL: An Extensible Language for Controlling Elasticity in Cloud Applications," in *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2013.

[50] C. Giblin, S. Muller and B. Pfitzmann, "From Regulatory Policies to Event Monitoring Rules: Towards Model-Driven Compliance Automation," IBM Research, Zurich, 16 October 2006.

[51] D. Mcdermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld and D. Wilkins, "PDDL- The Planning Domain Definition Language," 1998. [Online]. Available: http://www.cs.cmu.edu/~mmv/planning/readings/98aips-PDDL.pdf. [Accessed 23 September 2016].

[52] H. Ludwig, K. Stamou, M. Mohamed, N. Mandagere, B. Langston, G. Alatorre, H. Nakaruma, O. Anya and A. Keller, "rSLA: Monitoring SLAs in Dynamic Service Environments," in *International Conference on Service-Oriented Computing*, 2015.

[53] S. Tata, M. Mohamed and T. Sakairi, "rSLA: A Service Level agreement Language for Cloud Services," in *Cloud Computing (CLOUD)*, 2016.

[54] F. Raimondi, J. Skene, W. Emmerich and B. Wozna, "A Methodology for Online Monitoring Non-Functional Specifications of Web-Services," in *First International Workshop on Property*

Verification for Software Components and Services (PROVECS), 2007.

[55] J. Skene and W. Emmerich, "Generating a Contract Checker for an SLA Language," 2004. [Online]. Available: http://eprints.ucl.ac.uk/712/1/9.9.1coala.pdf. [Accessed December 2014].

[56] V. C. Emeakaroha, M. A. S. Netto, I. Brandic and C. A. F. De Rose, "Application Level Monitoring and SLA Violation Detection for Multi-tenant Cloud Services," *Emerging Research in Cloud Distributed Computing Systems,* 2015.

[57] V. C. Emeakaroha, M. A. Netto, R. N. Calheiros, I. Brandic, R. Buyya and C. A. De Rose, "Towards Autonomic Detection of SLA Violations in Cloud Infrastructures," *Future Generation Computer Systems 28,* vol. 28, no. 7, pp. 1017-1029, 2012.

[58] B. Konig, J. A. Calero and J. Kirschnick, "Elastic Monitoring Framework for Cloud Infrastructures," *IET Communications,* vol. 6, no. 10, pp. 1306-1315, 2012.

[59] M. Sedaghat, F. Hernandez-Rodriguez and E. Elmroth, "A Virtual Machine Re-packing Approach to the Horizontal vs. Vertical Elasticity Trade-Off for Cloud Autoscaling," in *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, 2013.

[60] S. A. Yemini, S. Kliger, E. Mozes, Y. Yemini and D. Ohsie, "High Speed and Robust Event Correlation," *IEEE communications Magazine,* vol. 34, no. 5, pp. 82-90, 1996.

[61] B. Gruschke, "Integrated Event Management: Event Correlation using Dependency Graphs," in *InProceedings of the 9th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM 98)*, 1998.

[62] R. S. Shariffdeen, D. Munasinghe, H. Bhathiya, U. Bandara and H. Dilum Bandara, "Workload and Resource Aware Proactive Auto-Scaler for PaaS Cloud," in *2016 IEEE 9th International Conference on Cloud Computing*, 2016.

[63] P. Tang, F. Li, W. Zhou, W. Hu and L. Yang, "Efficient Auto-Scaling Approach in the Telco Cloud Using Self-Learning Algorithm," in *IEEE Global Communications Conference (GLOBECOM)*, December 2015.

[64] C. Wang, A. Gupta and B. Urgaonkar, "Fine-Grained Resource Scaling in a Public Cloud: A Tenant’s Perspective," in *2016 IEEE 9th International Conference on Cloud Computing*, 2016.

[65] Z. Shen, S. Subbiah, X. Gu and J. Wilkes, "Cloudscale: Elastic Resource Scaling for Multi-tenant Cloud Systems," in *2nd ACM Symposium on Cloud Computing*, 2011.

[66] Q. Zhang, H. Chen and Z. Yin, "PRMRAP: A Proactive Virtual Resource Management Framework in Cloud," in *1st International Conference on Edge Computing*, 2017.

[67] A. Ali-Eldin, J. Tordsson and E. Elmroth, "An Adaptive Hybrid Elasticity Controller for Cloud Infrastructures," in *Network Operations and Management Symposium (NOMS)*, 2012.

[68] T. Gruber, "Ontology," *Encyclopedia of Database Systems, Springer,* pp. 1963-1965, 2009.

[69] P. Colombo, P. Salehi, F. Khendek and M. Toeroe, "Bridging the Gap between High Level User Requirements and Availability Management Framework Configurations," in *17th International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2012.

[70] D. Mazmanov, C. Curescu, H. Olsson, A. Ton and J. Kempf, "Handling Performance Sensitive Native Cloud Applications with Distributed Cloud Computing and SLA Management," in *6th International Conference on Utility and Cloud Computing (UCC)*, 2013.

[71] S. V. Raghavan and E. Dawson, "An Investigation into the Detection and Mitigation of Denial of Dervice (DoS) Attacks," Springer Science & Business Media, 2011.

[72] P. C. Rangarajan, F. Khendek and M. Toeroe, "Managing the Availability of VNFs with the Availability Management Framework," in *Network and Service Management (CNSM)*, 2017.

[73] A. Jahanbanifar, F. Khendek and M. Toeroe, "Providing Hardware Redundancy for Highly Available Services in Virtualized Environments," in *Software Security and Reliability (SERE)*, 2014.

[74] "Ericsson Introduces A Hyperscale Cloud Solution," [Online]. Available: http://archive.ericsson.net/service/internet/picov/get?DocNo=28701-FGB1010554&Lang=EN&HighestFree=Y. [Accessed July 2016].

[75] J. Skene, D. D. Lamanna and W. Emmerich, "Precise Service Level Agreements," in *Proceedings of the 26th International Conference on Software Engineering*, 2004.

[76] M. Toeroe and F. Tam, Service Availability: Principles and Practice, John Wiley & Sons, 2012.

[77] T. Luis and J. Tordsson, "An Autonomic Approach to Risk-Aware Data Center Overbooking," *Cloud Computing IEEE Transaction,* vol. 2, no. 3, 2014.

[78] C. Vazquez, L. Tomas, G. Moreno and J. Tordsson, "A Fuzzy Approach to Cloud Admission Control for Safe Overbooking," in *Fuzzy Logic and Applications*, 2013.

[79] S. Katker and M. Paterok, "Fault Isolation and Event Correlation for Integrated Fault Management," *Integrated Network Management V, Part of the series IFIP — The International Federation for Information Processing,* pp. 583-596, 1997.

[80] R. D. Gardner and D. A. Harle, "Methods and Systems for Alarm Correlation," in *Global Telecommunications Conference*, 1996.

[81] M. Steinder and A. S. Sethi, "The Present and Future of Event Correlation: A Need for End-to-End Service Fault Localization," in *World Multi-Conf. Systemics, Cybernetics, and Informatics*, Orlando, 2001.

[82] M. K. Agarwal, K. Appleby , M. Gupta, G. Kar, A. Neogi and A. Sailer, "Problem Determination

Using Dependency Graphs and Run-Time Behavior Models," in *International Workshop on Distributed Systems: Operations and Management*, 2004.

[83] G. Jakobson and M. Weissman, "Real-time Telecommunication Network Management: Extending Event Correlation with Temporal Constraints," *InIntegrated Network Management IV,* pp. 290-301, 1995.

[84] R. Davis and B. G. Buchanan, "Meta-Level Knowledge," *Rulebased expert systems, The MYCIN Experiments of the Stanford Heuristic Programming Project,* pp. 507-530, 1984.

[85] Y. Jiang, C.-S. Perng and T. Li, "Cloud Analytics for Capacity Planning and Instant VM Provisioning," *IEEE Transactions on Network and Service Management,* vol. 10, no. 3, pp. 312-325, 2013.

[86] M. Abbasipour, M. Sackmann, F. Khendek and M. Toeroe, "Ontology-based User Requirements Decomposition for Component Selection for Highly Available Systems," in *IEEE Information Reuse and Integration (IRI)*, San Francisco, 2014.

[87] M. Abbasipour, M. Sackmann, F. Khendek and M. Toeroe, "A Model-Based Approach for User Requirements Decomposition and Component Selection," *Formalisms for Reuse and Systems Integration,* pp. 173-202, 2015.

[88] M. Abbasipour, F. Khendek and M. Toeroe, "A Model-Based Framework for SLA Management and Dynamic Reconfiguration," in *International SDL Forum*, 2015.

[89] M. Abbasipour, F. Khendek and M. Toeroe, "Trigger Correlation for Dynamic System Reconfiguration," in *The 33rd ACM/SIGAPP Symposium On Applied Computing (SAC 2018)*, Pau, France, 2018.

[90] M. Abbasipour, F. Khendek and M. Toeroe, "A Model-based Approach for Design Time Elasticity Rules Generation," in *23rd International Conference on Engineering of Complex Computer Systems (ICECCS 2018)*, (Submitted), 2018.

[91] M. Abbasipour, F. Khendek and M. Toeroe, "A Model-Driven Framework for SLA Compliance Management and Dynamic System Reconfiguration," *Journal of Networks and Computer Applications (JNCA),* p. (Submitted), 2018 .

[92] M. Abbasipour, M. Sackmann, F. Khendek and M. Toeroe, "Ontology-Based User Requirement Decomposition for Component Selection for Service Provision". US Patent 9164734, 20 October 2015.

Appendix

Algorithm 1: Trigger Correlation

Input: TriggerSet, Configuration

Output: RelationGraphSet

```
1:  //creating relationgraphs
2:  RelationSet := Configuration.relations
3:  RelationGraphSet.relations := {}
4:  RelationGraphSet.triggers := TriggerSet
5:  For each relation in RelationSet
6:      If (TriggerSet.entities.includesAll(relation.entities)) then
7:          RelationGraphSet.relations := RelationGraphSet.relations ∪ { TransformRela-
            tion(relation, relation.entities.at(1).trigger, relation.entities.at(2).trigger)}
8:      End if
9:  End for
10: Return  RelationGraphSet
```

Algorithm 2: Selecting Applicable Elasticity Rules

Input: TriggerSet, ElasticityRuleSet

Output: ApplicableElasticityRuleSet

```
1:  // Selecting applicable elasticity rules
2:  ApplicableElasticityRuleSet:={}
3:  For each trigger in TriggerSet
4:      For each elasticityRule in ElasticityRuleSet
5:          If (elasticityRule.entityType == trigger.entity.entityType) then
6:              If (elasticityRule.scalingRule==trigger.scalingType) then
7:                  ApplicableElasticityRuleSet := ApplicableElasticityRuleSet ∪ {(trigger,
                    elasticityRule)}
8:              End if
9:          End if
10:     End For
11: End For
12: Return  ApplicableElasticityRuleSet
```

Algorithm 3: Selecting Optimal Action

Input: ElasticityRule, Measurement, Threshold, configuration

Output: OptimalAction

```
 1: //Selecting Optimal action based on the current situation
 2: ActionSet := ElasticityRule.actions          /// Set of all actions contained in the elasticity
    rule
 3: optimalAction := $action_0$
 4: optimalCost := $action_0$.midCost
 5: elasticityRuleFeasibility := false
 6: For each action in ActionSet
 7:    If (evaluate(action.condition,configuration,threshold,Measurement)) then
 8:          actionFeasibility := true
 9:          For each prerequisite in action.prerequisites
10:             If (not evaluate(prerequisite,configuration,threshold,Measurement)) then
11:                actionFeasibility := false
12:                Break
13:             End if
14:          End for
15:          If (not elasticityRuleFeasibility and not actionFeasibility) then
16:             If (optimalCost > action.midCost) then
17:                optimalAction := action
18:                optimalCost := action.midCost
19:             End if
20:          If (elasticityRuleFeasibility and actionFeasibility) then
21:             If (optimalCost > action.cost) then
22:                optimalAction := action
23:                optimalCost := action.cost
24:             End if
25:          Else if (not elasticityRuleFeasibility and actionFeasibility) then
26:             elasticityRuleFeasibility := true
27:             optimalAction := action
28:             optimalCost := action.cost
29:          End if
30:    End if
31: End for
32: Return  optimalAction
```

Algorithm 4: Applying Action Correlation Meta-Rules

Input: relationGraph, ApplicableElasticityRuleSet, configuration, Measurement, Threshold, configuration

Output: TriggerSet

```
33: //Sorting the triggers of a relation graph based on the relations between them
34: TriggerSet := RelationGraph.triggers
35: RelationSet := RelationGraph.relations
36: For each relation in RelationSet
37:   If (relation.isTypeOf() == Adjacency ) then
38:       adjcent1:= relation.vertices().at(1)
39:       adjacent2:= relation.vertices.at(2)
40:       If (adjacent1.indexAtTriggerSet () < adjacent2.indexAtTriggerSet() and adja-
          cent1.scalingType == increase and adjacent2.scalingType == decrease) then
41:           TriggerSet.swap (adjacent1,adjacent2)
42:       End if
43:   End if
44:   If (relation.isTypeOf() == Dependency) then
45:       dependentOptAction := optimalAction(dependent.applicableEr,configuration,
          Measurement,Threshold)
46:       sponsorOptAction := optimalAction(sponsor.applicableEr, configuration, Meas-
          urement, Threshold)
47:       If (dependentOptAction == scale-out) then
48:           If (sponsor.indexAtTriggerSet () < dependent.indexAtTriggerSet()) then
49:             TriggerSet.swap (sponsor,dependent)
50:           End if
51:       Else if (dependentOptAction == scale-up and sponsorOptAction == scale-up) then
52:               RelationSubset:= findOtherDependency (sponsor, RelationSet))
53:               If (findOtherScaleUpDependents (RelationSubset).notEmpty()) then
54:                   DependentSet := RelationSubset.dependents ∪ {dependent}
55:                   migrationSet := findMigrationSet (DependentSet)
56:                   scaleUpSet := DependentSet – migrationSet
57:                   TriggerSet.partialSort (MigrationSet, DependentSet)
58:                   RelationSet := RelationSet – RelationSubset
59:               Else if (findOtherScaleUpDependents (RelationSubset).isEmpty()) then
60:                   If (dependent.indexAtTriggerSet () < sponsor.indexAtTriggerSet())
                      then
61:                       TriggerSet.swap (dependent, sponsor)
62:               End if
63:       Else
64:           If (dependent.indexAtTriggerSet () < sponsor.indexAtTriggerSet()) then
65:               TriggerSet.swap (dependent, sponsor)
66:       End if
67:   End if
68: End for
69: Return  TriggerSet
```