Model-Driven Engineering for Analysing, Modelling and Comparing Cloud Computing Service Level Agreements

Fatima A A A Alkandari

Doctor of Philosophy

University of York

Computer Science

September 2014

Abstract

In cloud computing, service level agreements (SLAs) are critical, and underpin a pay-per-consumption business model. Different cloud providers offer different services (of different qualities) on demand, and their pre-defined SLAs are used both to advertise services and to define contracts with consumers. However, different providers express their SLAs using their own vocabularies, typically defined in terms of their own technologies and unique selling points. This can make it difficult for consumers to compare cloud SLAs systematically and precisely. We propose a modelling framework that provides mechanisms that can be used systematically and semi-automatically to *model* and *compare* cloud SLAs and consumer requirements. Using MDE principles and tools, we propose a metamodel for cloud provider SLAs and cloud consumer requirements, and thereafter demonstrate how to use model comparison technology for automating different matching processes, thus helping consumers to choose between different providers. We also demonstrate how the matching process can be interactively configured to take into account consumer preferences, via weighting models. The resulting framework can thus be used to better automate high-level consumer interactions with disparate cloud computing technology platforms.

Contents

List of Figures

List of Listings

List of Tables

Acknowledgements

I would like to thank my supervisor, Prof. Richard Paige, for his invaluable guidance, support and encouragement throughout my research.

I would like to thank my internal examiner, Dr. Dimitris Kolovos, for his comments that improved my thesis. I also thank my friends for their encouragement.

I would like to express my warm gratitude to my parents, my husband, Salah, and my sons, Mohammad, Abdullah and Ahmad, for their continuous support, encouragement and patience.

Author Declaration

This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References. Except where stated, all the work contained in this thesis represents the original contribution of the author. Parts of the work described in this thesis have been previously published as a workshop paper [37].

Chapter 1

Introduction

1.1 Introduction

The research in this thesis lies at the intersection between cloud computing and Model Driven Engineering (MDE). In particular, it aims to investigate how MDE can assist with particular domain challenges of cloud computing, and how users of cloud computing Service Level Agreements (SLAs) can benefit from the automation support that arises from the use of MDE tools, particularly for model management. Before defining the problem and research motivation, brief introductions to SLAs, cloud computing and MDE are provided in the following sections.

1.1.1 Service Level Agreements

An SLA is a contract between different parties: these are usually a service provider, a service consumer and perhaps a third party [124, 189] (e.g. consumers or providers might delegate the monitoring and reporting of violations to a third party [150]). An SLA can be used to identify the services to be provided to consumers, as well as the qualities associated with those services: for example, the availability or response time for service delivery. SLAs are fundamental to a variety of socio-technical or software systems, as they provide a basic means of negotiation and arbitrage amongst the parties that agree to be constrained by them.

For instance, suppose that a service consumer requires a *network service*. The service provider of a network service provides the details of the network service in the

SLA. The SLA contains a description and definition of the service. It might contain the expected *network service* performance, maintenance, monitoring and reporting functions. The service consumer and service provider can then negotiate on the SLA terms and agree to be bound by the SLA. This means that if the service provider fails to meet the agreement terms, penalties may apply.

An SLA defines measurable QoS parameters (e.g. availability and response time) and SLOs [181]. QoS parameters might be defined in an SLA with a threshold value (e.g. availability >99.9%); these combined form an SLO. An SLO is a term agreed between the service provider and the service consumer and is used as a means of defining and measuring the *quality* threshold values of a provisioned service. An SLO is an expression over a set of the defined QoS parameters [181]. The QoS parameter threshold values are monitored by at least one of the SLA parties to ensure that the SLOs are not violated. In case of violation, penalties may apply; to be certain that all parties agree to these said penalties, they may also be defined in the SLA (Chapter 2).

An SLA is important for service providers, so that they can try to avoid violations and penalties, understand the consumer's requirements, and know what could be done to satisfy the consumer [198]. From the consumers' perspective, an SLA is a way to define their requirements (including the required QoS) to meet their business goals.

In distributed systems research, SLAs are often treated as specifications that can be manipulated by computers [66, 70, 138] e.g. Web services or service-oriented architectures. As a result, substantial research has been carried out that explores the specification, automation and management of SLAs, including determining whether or not there is an agreement on the terms specified in the SLA (this is discussed further in Chapter 2).

1.1.2 Cloud Computing

Cloud computing is a distributed computing model that is gaining increased attention. It is based on the idea of providing a pool of virtualized computing resources (such as applications, servers and storage) that are accessed remotely on a pay-per-use basis [41, 82, 186]. These computing resources can be accessed remotely at any time and delivered on-demand to consumers. Resources can be dynamically and rapidly scaled up by acquiring resources on-demand and releasing them when not needed [133]. Many leading companies, such as Amazon, Google and Microsoft, have adopted this computing paradigm, and market different cloud offerings that provide different services.

Cloud computing differs from other distributed system models in terms of its support for virtualization (which is “a method, process or system for providing services to multiple, independent logical entities that are abstractions of physical resources, such as storage, networking and computer cycles” [86]), rapid elasticity (cloud computing capabilities are provisioned and released dynamically), and the models used for charging consumers for services (e.g. pay per consumption or a subscription basis).

The relationship between cloud providers and cloud consumers is normally governed by SLAs (as with other distributed computing models). The dynamic and complex nature of cloud computing arguably warrants complex SLA specification and management [76, 150]. One of the challenges in cloud computing is defining the SLA specifications “in such a way that has an appropriate level of granularity, namely the trade-offs between expressiveness and complicatedness, so that they can cover most of the consumer expectations and is relatively simple to be weighted, verified, evaluated, and enforced by the resource allocation mechanism on the cloud” [71]. Moreover, managing the cloud computing SLA requires an appropriate means to be consistent with the dynamic nature, autonomous and elasticity of cloud computing (i.e. self-service, rapid and dynamic resource allocation and release). The automation of the process of creating, selecting, negotiating managing and monitoring an SLA is addressed to tackle this challenge [26, 150, 197]. Furthermore, current cloud computing providers provide pre-defined cloud SLAs, which define the resource availability, while other QoS parameters, such as performance and elasticity, are not considered.

1.1.3 Model Driven Engineering (MDE)

MDE is a principled approach to system engineering that utilizes models as first-class artefacts of the development process [49, 50]. MDE is based on the construction and automated manipulation of precisely defined models of concepts from a problem domain; these models are manipulated (e.g. using automated tools) to achieve business and engineering goals, for example, the generation of code. Models in MDE are defined formally in terms of their conformance to *metamodels*. Metamodels are used to define the abstract syntax of modelling languages [161]. Operations can then be defined using metamodels to automatically manipulate models, e.g. for model transformation or code generation. The aim in using models is to raise the level of abstraction and increase the automation of the software development process.

1.2 Comparison and Selection of Cloud Computing Services

The vast number of cloud services from different providers makes it difficult for cloud consumers to decide which services from which provider to select and use. Several cloud computing studies addressed the problem of cloud services comparison and selection [83, 90, 119, 128, 199]. Four main concerns have been identified in comparing and selecting cloud services (see Section 2.4). First, the basis of comparison i.e. which parameters of the cloud services should be used as a basis for comparison? Second, the matching approaches i.e. how to match the consumer requirements with the provider offers? Third, how to select the cloud service or offer from among different alternatives. Finally, how can the consumers' requirements and offers be captured?

Different studies proposed different parameters as a basis for comparison. Studies such as [96, 119, 183] evaluate the performance of the cloud services offered by different cloud providers. This study provides performance comparison of the different providers. For example, a study in [119] compares public cloud providers (Amazon EC2, Windows Azure, Google AppEngine and RackSpace) and proposes a tool to compare the performance and cost of cloud providers.

However, QoS parameters rather than performance may be considered to be important by cloud consumers. These parameters may include static attributes like security and localisation [38, 83], or dynamic attributes that, for example, provide the status of the cloud service [36]. All such attributes may feed in to the cloud selection process.

Different approaches have been proposed to compare and select cloud providers based on these different parameters. Studies such as [90, 128] were based on the feedback from users to rank cloud services or providers.

For example, [128] suggested a platform to help cloud consumers in cloud providers selection. This selection is based on the evaluations submitted to the platform by other users. This study proposed a database model that defines the maturity model. This maturity model enables users to judge the quality of the service. Based on the information stored in the underlying database (i.e. other users' judgements), a user can select a provider based on the evaluation of the quality of service of different providers. This evaluation of each service is based on different criteria such as support, scalability, security, auditability, compliance, Data Centres, Interfaces, Certificates and SLA. The maturity of each service is calculated on the basis of the weighted arithmetic average of those criteria. [155] proposed to match SLAs because they define quality parameters to express consumer requirements and provider offers.

[155, 199] provide approaches to matching consumer requirements and provider offers. Functionality matching is introduced to return the service that is relevant to the consumer requirements [199], while SLA mapping is introduced in [155] as mapping between SLA elements based on their semantics and syntax. Current cloud providers provide different terminologies to define their quality parameters, which may have the same meaning. In the comparison, this difference is to be considered. [155] introduced a case-based reasoning solution to enhance the matching of different terminology used in the SLA. However, it selects the provider based on the total number of matched elements, which does not consider the consumer preferences regarding the quality parameters.

To compare and select a cloud service or offer from different alternatives, different criteria may be considered. The authors in [182] suggested that selecting cloud providers is an MCDM problem and proposed a multi criteria cloud service selection. Furthermore, the authors in another publication [183] compared the effect of different MCDM methods on cloud providers selection. Other publications, such as [18, 83, 85], discussed MCDM in cloud service selection.

For example, [83] proposed a framework to measure and rank cloud services. They provide the means for cloud consumers to compare their requirements with different cloud providers' offers. They suggested a quality model for IaaS providers, which included factors such as service response time, sustainability, interoperability, availability, reliability and elasticity. The paper included an evaluation the service provider offers using the MCDM approach.

In this thesis, we explore approaches to comparing cloud SLAs to support consumer decision making, pertaining to offers that *match* requirements; we explore the use of MDE to help to automate the comparison and selection processes.

1.3 Problem Definition

Cloud computing SLAs are arguably specified in ways that are ambiguous, imprecise and inconsistent; there is, as yet, no standard specification for cloud SLAs. Different cloud providers have different pre-defined SLAs; these SLAs are usually defined in terms of the technology they offer. As such, each provider describes their SLA in a platform-specific way, using the vocabulary and terminology that best fits their needs (including, for example, their advertising needs, i.e., for emphasising the cloud provider's unique selling points). This, in turn, makes it difficult for cloud consumers to rigorously *compare* SLAs, and also makes it challenging for consumers to specify and negotiate their own, bespoke SLAs that more accurately meet their own needs. Ultimately, cloud SLAs are important for cloud consumers in helping them to choose the most suitable provider that can help them to achieve their business goals.

The problem that we address in the thesis is: given the wide range of ambiguous, imprecise and not-yet-standardised specifications of cloud SLA offerings, how can we help cloud consumers and cloud providers to compare different SLAs and thereafter choose between them?

1.4 Motivation

Different studies have compared, and thereafter supported the selection of cloud services based on different parameters. An SLA has often been used in past work to define the quality parameters that can be used as a basis for comparison. As such, having a machine-processable SLA will support both cloud consumers and providers in expressing requirements and offers, and comparing them automatically.

The motivation of this work is to help cloud consumers precisely, efficiently and automatically compare different cloud computing SLAs. The increasing number of cloud providers with different offered services makes it challenging for consumers to match their demand.

There is a problem with this; a significant difficulty is that there is as of yet no standard vocabulary, metamodel or ontology for cloud SLAs. How does one compare two concepts that are defined in terms of different properties, terms or rules? Let us consider a small example. Different public cloud providers used different vocabulary to present *availability*, e.g. “Monthly Uptime” is used in the Amazon EC2 SLA [4], “Monthly Availability” is used in RackSpace [21], “ Monthly Connectivity Uptime Service Level” is used in Azure [32] and “Server Uptime” is used in GoGrid [16]. This different terminology makes it difficult for consumers to match their demands (e.g. availability) against these offers. Consumers have to compare this different vocabularies and their definitions to find matches between the cloud offers and their demands. Public cloud SLAs, such as the AWS EC2 SLA, provide only “Monthly Uptime” as an QoS parameter in the SLA; other quality parameters, such as response time, elasticity, interoperability, etc., are not included, although consumers may be interested in them [38, 197].

We aim to provide mechanisms to help consumers to decompose and formalise - using MDE tools and techniques - the comparison process.

1.5 Research Hypothesis

The research presented in this thesis investigates the following hypothesis:

Can MDE principles and tools support the precise modelling of cloud computing SLAs in such a way that cloud stakeholders can define their offers and demands? In addition, can the MDE principles and tools enable a semi-automated comparison process for cloud computing SLAs, in order to help cloud stakeholders to make better decisions about the appropriateness of the offerings of different cloud providers?

The main characteristics of the above statement are as follows:

1. Modelling cloud SLAs: By using MDE principles, artefacts can be produced for modelling a cloud SLA in a systematic, standardized and reusable way.

2. Comparison process: By using MDE principles and models of cloud SLAs, model comparison can be used to compare cloud SLAs semi-automatically, systematically and in a reusable way.

3. Comparing cloud providers: using the results from 1-2 above, the cloud consumer will be able to identify possible semi-automated matches between their demands and the cloud providers’ offers.

4. Supporting decisions: By using the results from 1-3 above, the cloud consumer will be supported in reasoning about the results of a comparison in order to help select a cloud provider.

5. Semi-automation: The proposed approach will provide the semi-automation of the comparison process, whereby some steps of the comparison process might need human intervention, e.g. to choose between the different possible comparison approaches.

1.6 Research Objectives

Motivated by the problem's definition and hypothesis, this thesis focuses on the following objectives.

1. To provide mechanisms for modelling cloud SLAs, based on MDE principles, techniques and tools.

2. To identify and describe precisely different scenarios for comparing cloud SLAs.

3. To provide mechanisms for automatically or semi-automatically comparing cloud SLAs, based on MDE principles, techniques and tools.

4. To propose mechanisms for presenting the results of comparing cloud SLAs in machine processable forms.

5. To evaluate the above mechanisms via examples inspired by real SLAs from current cloud providers.

1.7 Research Methodology

The aim of this section is to describe the methodology which was used to evaluate the hypothesis of this research. This exploratory method "focuses on determining what concepts to measure and how to measure them best" [153], which means providing more details where little information exists. It is used as pilot for other detailed studies

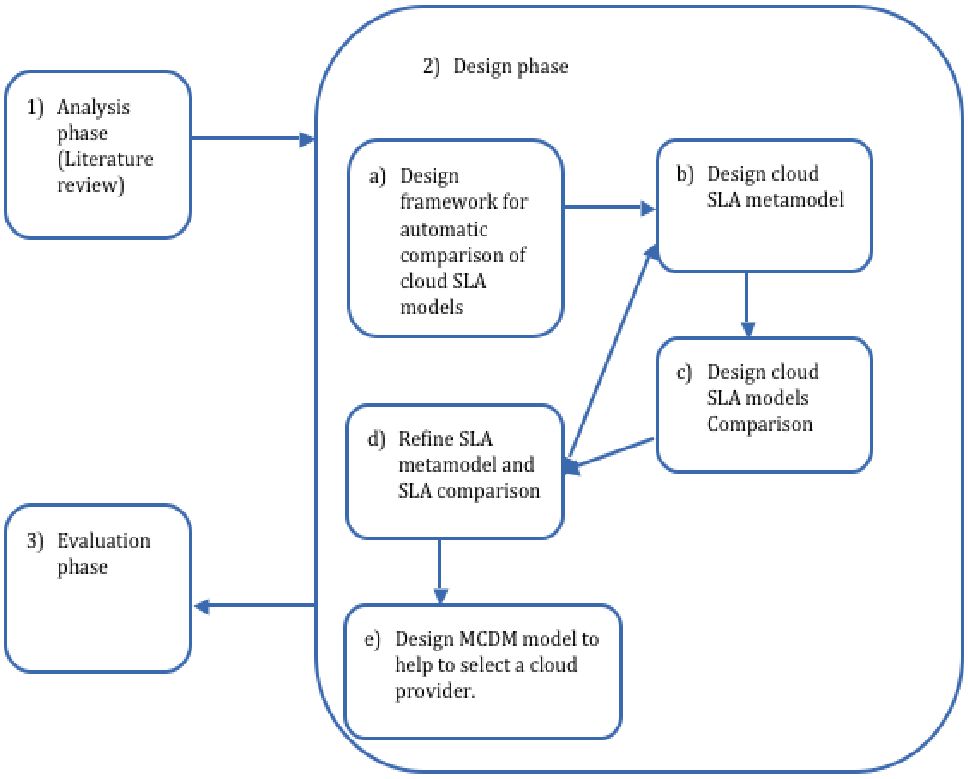

1) Analysis phase (Literature review)
2) Design phase
a) Design framework for automatic comparison of cloud SLA models
b) Design cloud SLA metamodel
c) Design cloud SLA models Comparison
d) Refine SLA metamodel and SLA comparison
e) Design MCDM model to help to select a cloud provider.
3) Evaluation phase

Figure 1.1: Research Methodology.

[153]. This methodology consists of the following phases: the *analysis*, *design and implementation* and *evaluation* phases [115]. Figure 1.1 shows the flow between phases of the research methodology. A detailed description of the phases is as follows:

1. The analysis phase is the literature review. In this phase, we analyse cloud computing SLAs, via the exploratory method [153]. We use this method because current cloud SLAs are ambiguous, imprecise and not yet specified in a standard format. Therefore, in this phase, studies of SLA are reviewed in terms of their technical aspects, specifications and management mechanisms (Section 2.1). We also review cloud computing, its common services and classifications (Section 2.2). We also discuss cloud computing SLAs in Section 2.2.4. Through this analysis, we identified challenges that further motivate the hypothesis and objectives of this research.

2. In the *design and implementation phase*, based on the findings of the *analysis*

phase, we propose, design and implement an approach in order to address the hypothesis. This phase includes:

(a) An approach to support automatic comparison of cloud SLA models. This including use cases and scenarios of cloud SLAs comparison to analyse and define the modelling assets that are involved in the comparison approach (Chapter 3).

(b) A cloud SLA metamodel was designed based on the findings of the cloud SLA comparison approach and the analysis phase. This metamodel was built incrementally and iteratively together with a comparison algorithm, which is explained in the following step (Chapter 4).

(c) A comparison algorithm for cloud SLAs was developed iteratively with the previous step. Each time the cloud SLA metamodel was changed, the comparison algorithm was refined (Chapter 5).

(d) Refining the cloud SLA metamodel and the comparison algorithm. These two processes underwent multiple iterations to refine the design that guided the implementation of the proposed approach.

(e) A process of analysing the outcomes of the comparison algorithm and forming as an MCDM problem was conducted. A detailed framework for cloud SLA models comparison has been designed (Chapter 6).

3. *The evaluation* phase was carried out by identifying case studies in order to assess the quality of our proposed approach (Chapter 7). This process was carried out iteratively for every major refinement of the implementation.

1.8 Thesis Structure

Chapter 2 provides an overview of SLAs by describing their main components and discusses their specifications. It also discusses cloud computing, its features and classifications. It provides a general overview of the MDE by describing its main concepts and principles and presents a review of the research that on the intersection between MDE and cloud computing.

Chapter 3 analyses and explores the problem and identifies the steps of the research plan to answer and address the hypothesis and objectives of this thesis. These are mentioned earlier in Sections 1.5 and 1.6. This Chapter starts with a motivating example and then defines the scenarios and the use cases. It explains the different comparison alternatives, which are: name-based, optimal and approximate, then presents the steps of the research plan and links them with the objectives of this chapter. The last section explains the general approach of cloud SLA comparison.

Chapter 4 illustrates the cloud SLA metamodelling approach. More precisely, section 4.2 describes different versions of the constructed metamodel. Then, section 4.3 describes the cloud SLA metamodel. This metamodel focuses on different concepts in the cloud computing SLAs, such as the QoS parameters.

Chapter 5 describes a general approach for matching two SLA models. It also introduces different comparison logics: optimal, approximate and name-based. The matching logics are split into several tasks. Different implementation details are discussed.

Chapter 6 illustrates our approach for comparing cloud SLAs to support cloud consumer decisions. It explains how the preferences of the consumer can be expressed as a weight model. This section provides a description of a cost model that can be used as the criterion for the decision-making. It also explains how the results of comparing the cloud SLAs with weight and cost models can support consumers to select a cloud SLA.

Chapter 7 describes the evaluation of the contribution in this thesis against the proposition of this thesis. It provides a case study to evaluate the work of this thesis. It also describes the limitation of this work.

Chapter 8 summarizes the findings and contributions of this thesis. It also discusses areas for future work.

Chapter 2

Literature Review

This chapter reviews topics that are considered in this research and have influenced the work; it also serves to allow the reader navigate the topics related to this work. This chapter provides a comprehensive overview of SLA, cloud computing and MDE, which has been used to carry out the research in this thesis and to identify the research gaps in this area. This chapter is divided into three parts. Section 2.1 presents an introduction to SLA. Then in Section 2.2.3, we present an overview of cloud computing in section 2.2, including its features and classifications and cloud SLAs. One of the objectives of this study is to provide mechanisms for modelling cloud SLAs, based on MDE principles, as discussed in section 1.6. Hence, section 2.3 explains MDE and its concepts related to our objectives. We present related work that discusses cloud computing service selection in Section 2.4. Then, a summary of the issues related to cloud computing selection modelling is discussed.

2.1 Service Level Agreement

Our proposed study focuses on modelling and comparing cloud computing SLAs. We briefly describe SLAs and their key topics. Before we start discussing SLAs, we explain what a service is in the context of distributed computing. The IT Infrastructure Library (ITIL) [17] defines a service as:

> *"a means of delivering value to customers by facilitating outcomes customers want to achieve, but without the ownership of specific costs and*

risks".

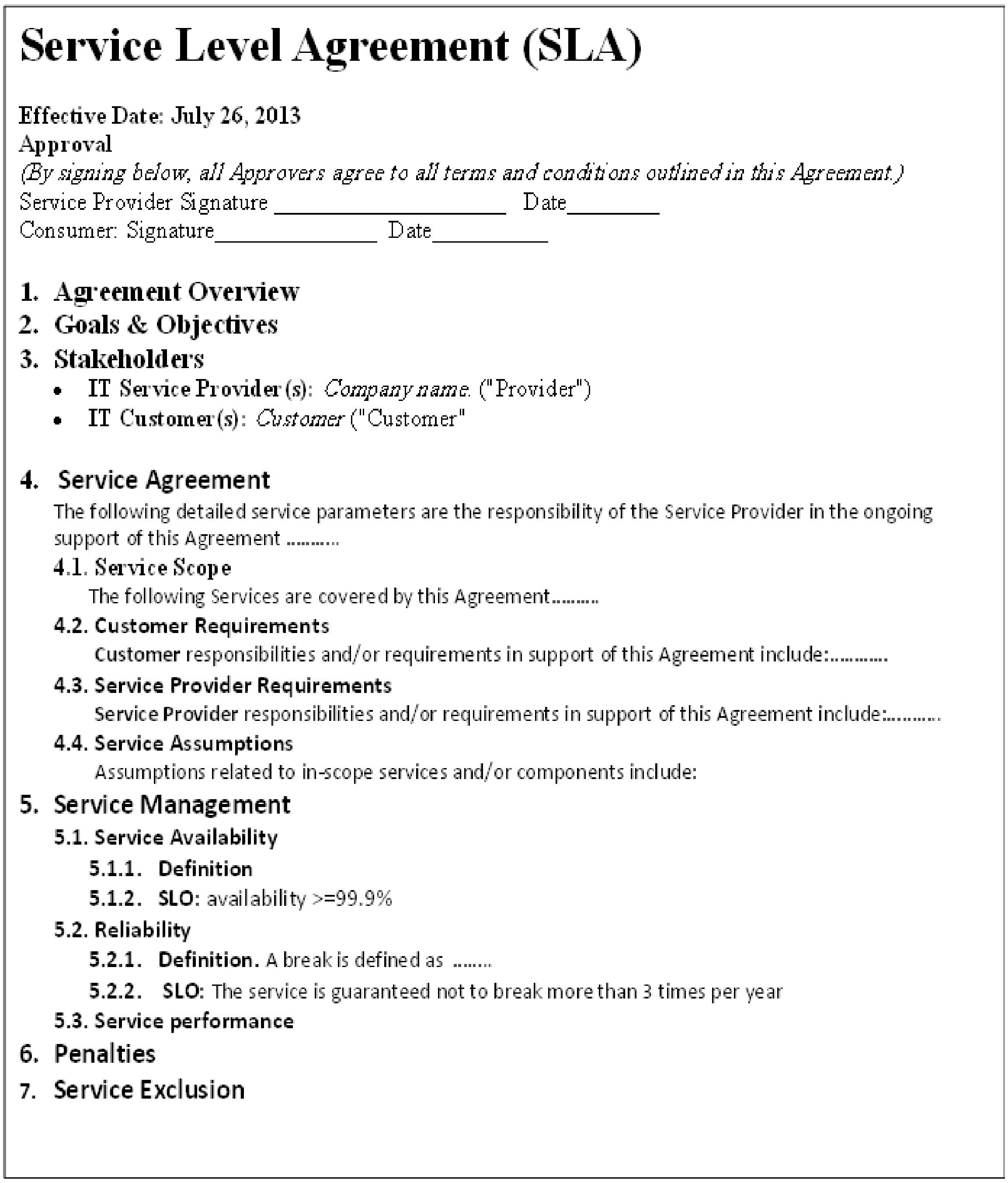
An SLA can be a part of a legal contract agreement [124] that is established between two or more parties. This agreement defines the framework of the relationship and responsibilities of the different parties. Service providers and service consumers are the main parties to the contract. [189] defines an SLA as:

> *"An explicit statement of expectations and obligations that exist in a business relationship between two organizations: the service provider and customer".*

An SLA can be used to remove ambiguity between parties by defining the service consumers' requirements and specifying the service qualities and responsibilities of each party. The intention behind specifying and using an SLA is that, by providing precise definitions and conditions, the parties can reduce the areas of conflict. As a result, an SLA is used most frequently to identify service consumer's requirements and service provider's capabilities.

There are several possible SLA structures, Figure 2.1 shows an example of such a structure. The main components, as in the example, are:

- *Duration of the SLA*: The agreement specifies the start and end dates regarding which the agreement conditions can be applied.
- *Parties*: This refers to the involved parties who take part in the SLA, these are usually a service provider, a service consumer and perhaps a third party.
- *List of services*: Each SLA: defines the list of services that are covered by the agreement. This is illustrated in Figure 2.1 as *service scope*.
- *Service definition*: This part includes a detailed description of the service and its features and functions.
- *Service Exclusions*: This describes the uncovered conditions and services in the SLA.
- *SLO*: This part defines a threshold level of service that a service provider promises to provide to the service consumer. For example, a service provider might promise to provide a service with 99.9% availability of the service time.



Service Level Agreement (SLA)

Effective Date: July 26, 2013
Approval
(By signing below, all Approvers agree to all terms and conditions outlined in this Agreement.)
Service Provider Signature ____________________ Date________
Consumer: Signature______________ Date__________

1. **Agreement Overview**
2. **Goals & Objectives**
3. **Stakeholders**
   - **IT Service Provider(s):** *Company name.* ("Provider")
   - **IT Customer(s):** *Customer* ("Customer"

4. **Service Agreement**
   The following detailed service parameters are the responsibility of the Service Provider in the ongoing support of this Agreement ....
   4.1. Service Scope
   The following Services are covered by this Agreement.........
   4.2. Customer Requirements
   Customer responsibilities and/or requirements in support of this Agreement include:...........
   4.3. Service Provider Requirements
   Service Provider responsibilities and/or requirements in support of this Agreement include:..........
   4.4. Service Assumptions
   Assumptions related to in-scope services and/or components include:
5. **Service Management**
   5.1. Service Availability
   5.1.1. Definition
   5.1.2. SLO: availability >=99.9%
   5.2. Reliability
   5.2.1. Definition. A break is defined as ....
   5.2.2. SLO: The service is guaranteed not to break more than 3 times per year
   5.3. Service performance
6. **Penalties**
7. **Service Exclusion**

Figure 2.1: An SLA example adapted from [28].

- *Penalty: This section of the agreement defines actions that might be taken when a service provider fails to meet the promised SLO (violation). Examples of actions that can be taken are: terminating the SLA, or an increasing/decreasing in the agreed payment.*
- *Agreement effective date*: specifies the duration of the service.

An SLA is an engineered artefact: it is developed through many different phases, from first creation through to termination. The engineering process for managing all of these phases is called SLA management. SLA management, which is defined as:

> *"the process of negotiation, SLA articulation, checks and balances, and reviews between the supplier and consumer regarding the services and service levels that support consumers' BP"* [118].

SLA management consists of several steps: [44, 45, 126]:

1. *SLA creation*: SLA management starts with this phase. In this phase service providers develop an SLA depending on their capabilities and provide *SLA Offers*. In the service consumers context, they define their requirements and refine then into *SLA requirements* [44, 68, 126].
2. *SLA discovery and selection*: Several studies have address SLA discovery and selection [160, 197] as one phase of the SLA life cycle. This entails discovering the offered services from different service providers' services to select services that satisfy the functional and non-functional requirements of the service consumers. The selection step may also include a decision-making process [83].
3. *SLA negotiation*: Negotiation starts between different SLA stakeholders to agree on and sign the terms of the agreement [160].
4. *SLA Monitoring*: This phase starts when the service is provisioned and activated for the service consumer. The service consumer starts monitoring and validating the SLA and detects any violations that may occur [197].
5. *SLA termination*: This occurs when a consumer or service provider decides to terminate when, for example, a violation occurs, or the service's SLA expires [197].

It is important for a service provider to avoid violating SLOs because, if this happens, penalties (possibly financial in nature) may apply. Thus, monitoring SLAs and violations is essential in order for a service provider to take appropriate action. Automation and autonomy for regarding SLA monitoring is introduced to provide continuous monitoring, evaluation and reporting and to reduce the need for a human intervention [77]. Therefore, approaches for electronic SLA specification and SLA management have been proposed, such as the Web Service Level Agreement (WSLA) [102], WS-Agreement [40] and SLA@SOI, which are discussed later (see Section 2.4.5). The objectives of these electronic SLAs, in addition to providing autonomous monitoring and reporting of the agreed SLAs terms, are to provide autonomous negotiation, identification and selection of services/service providers [104].

One of the main components used in SLA automation is the definition of QoS parameters and metrics, in order to monitor threshold values and ensure that these are not exceeded. Thus, QoS iwill form the topic of the next sub section.

2.1.1 Quality of Service (QoS)

In this work, we focus on the mechanisms used to model SLAs. We identify the key components of the SLA and how these can be used later in comparing and matching different SLAs. QoS parameters are an important part of SLAs and their comparisons; they are typically used in SLA guarantee conditions. The section provides a brief description of QoS.

SLO usually defines thresholds over the QoS parameters, e.g. *response time <= 5 seconds*. QoS is often considered as a part of the non-functional requirements [154, 167]. More precisely, QoS:

> *"often refers to a set of quality requirements on the collective behavior of one or more objects"* [59].

QoS parameters are used in SLAs to represent the level of services guaranteed by the service provider, e.g. *to guarantee that the service will be available 99.9% of the time*. QoS and its parameters have been undertaken in some SLA specification languages, e.g. WSLA [102] and WS-Agreement [40]. QoS has also been used as a basis for web service discovery and selection, as in [68, 154, 194]. Because of the

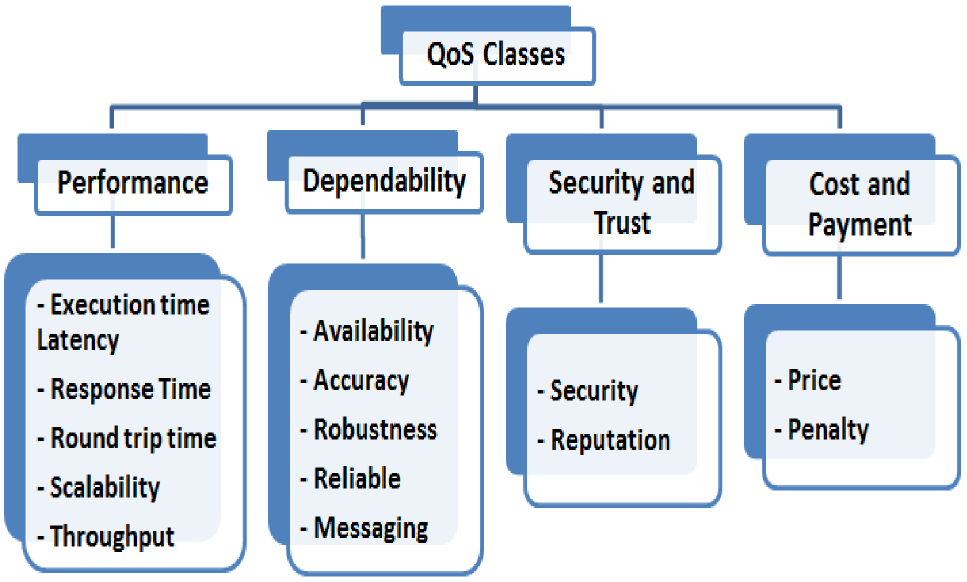


Figure 2.2: QoS Model [135].

objectives of this thesis, and because QoS parameters are defined in SLAs, we briefly discuss the studies of QoS that are relevant to our work. We describe research on QoS parameter specifications (Section 2.1.1.2), common QoS parameters that are defined in the SLA (Section 2.1.1.1) and how QoS parameters are used as a basis for service selection, in Section 2.1.2.

2.1.1.1 Common QoS and QoS classification

Terms such as *availability*, *reliability*, *accessibility*, *performance* and *cost* are examples of QoS parameters. QoS parameters can be categorized differently [83, 135, 154]. For example, the authors of [135] defined a QoS model fora web service SLA that categorized QoS parameters into four categories: *Performance*, *Dependability*, *Security and Trust* and *Cost and Payment*, as illustrated in Figure 2.2. QoS can be categorized from the perspective of the service party [46]. This is illustrated in Figure 2.3, which categorizes the QoS parameters of web services into three categories: *Developer Qualities*, *Provider Qualities* and *Consumer Qualities*. [83] defined the QoS parameters required by customers in order to select a cloud service, as: accountability, agility, assurance of service, cost, performance, security and privacy, and usability. QoS parameters may be defined in terms of their sub-parameters: e.g., availability is defined

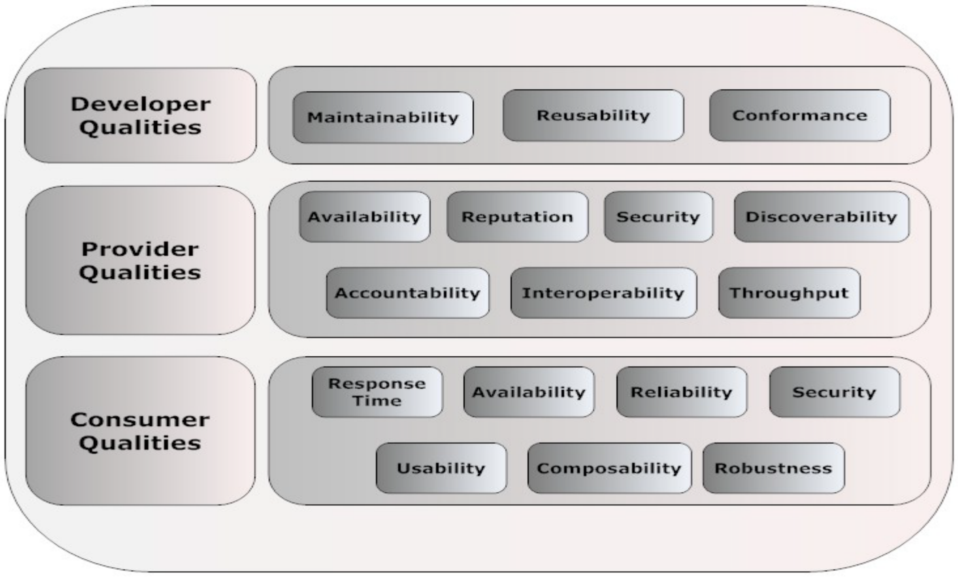


Figure 2.3: QoS Model [135].

in terms of uptime and downtime (Equation 2.1).

With differences in service functionality and service provider technologies, different QoS parameters may be defined in an SLA. Furthermore, each QoS parameter definition varies in the literature. For example, web service availability in [46] is defined in terms of *uptime* and *downtime*, as an Equation 2.1, while a different study [78] defined (power-station) availability in terms of Mean Time TO Repair (MTTR) and Mean Time Between Failure (MTBF), as illustrated in equation 2.2.

$$Availability = Uptime/(Uptime + Downtime) \tag{2.1}$$

$$Availability = MTBF/(MTBF + MTTR) \tag{2.2}$$

Different QoS specifications and ontologies have been proposed to define the QoS service offers and requirements. QoS ontologies (Section 2.1.1.2) and specifications were used as a basis for service selection and discovery (Section 2.1.2).

2.1.1.2 QoS Ontology

This section presents the notion of a QoS ontology; ontologies have been widely used in the automation of matching processes [131, 172, 194, 207]: e.g., to select web services. The concept of QoS ontology has been proposed as a way of facilitating the intercommunication between a service consumer and a service provider [75]; in particular, it has been used as a way to specify the QoS parameters. It is also used to facilitate machine reasoning [75].

A *QoS ontology* example is illustrated in Figure 2.4. This example defines QoS as a subclass of non-functional requirements and is defined by a *MetricName*, *Value*, *valueType*, etc. A later study suggested a set of QoS attributes to be considered when developing *QoS ontology*, which are: a *value*, a *value type* i.e. numeric or literal, *metrics* which describe how the parameters are calculated, and a *function*, which defines how metrics are related [74]. QoS ontologies focus on the structure of the QoS parameters and how they are related, which helps when reasoning about QoS information [176]. SLAs make use of QoS parameters and other components, such as the parties' responsibilities, SLOs and penalties, to assist with negotiating and monitoring services. QoS ontology provides a descriptive model of QoS (i.e. semantic representation) for machine understanding. In this study, we focus on the attributes of the QoS: *Value*, *valueType*, *unit*, *hasName* and *tendency*, in the cloud SLA comparison.

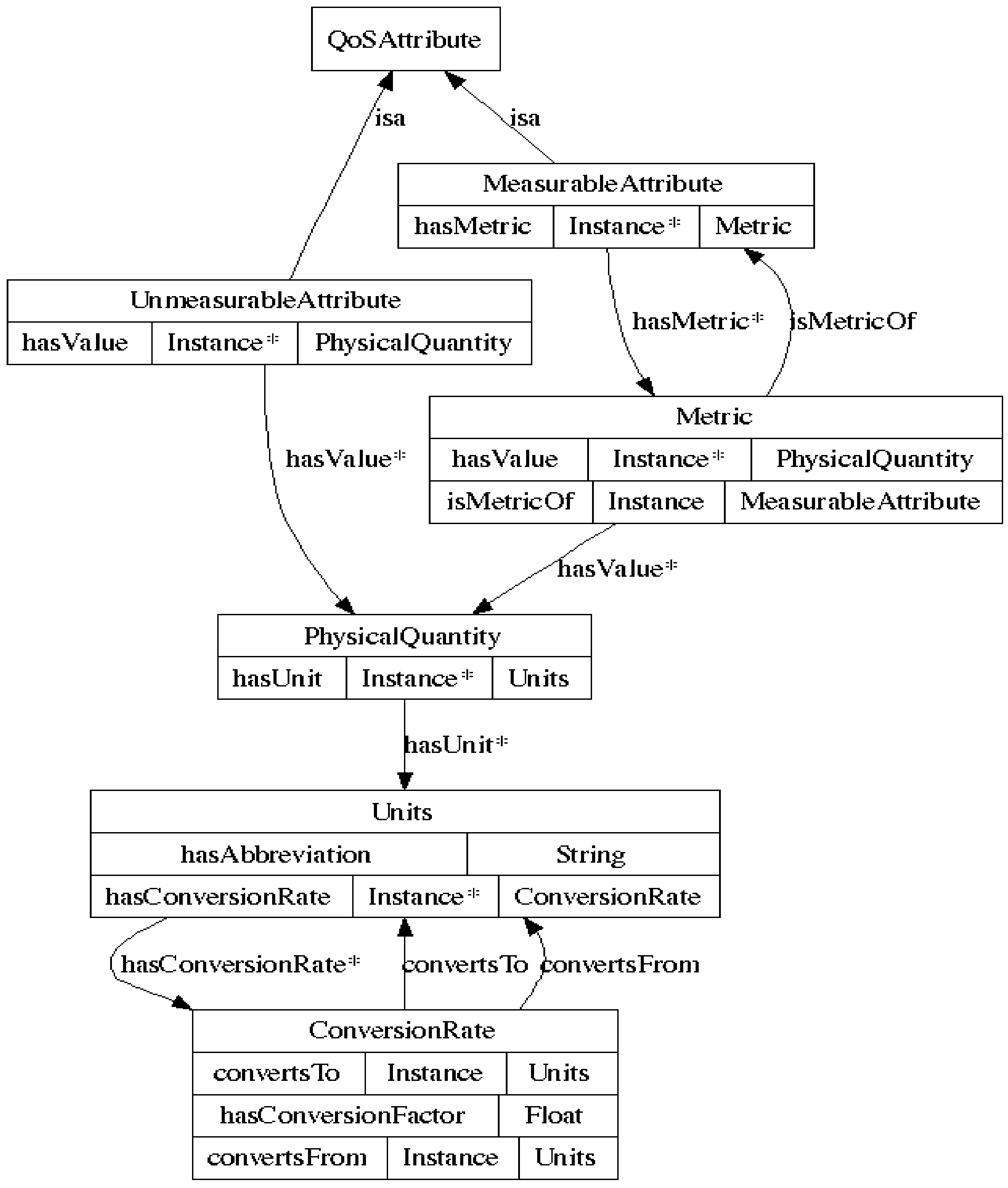

QoSAttribute
isa
isa
MeasurableAttribute
hasMetric | Instance* | Metric
UnmeasurableAttribute
hasValue | Instance* | PhysicalQuantity
hasMetric*
isMetricOf
Metric
hasValue | Instance* | PhysicalQuantity
isMetricOf | Instance | MeasurableAttribute
hasValue*
hasValue*
PhysicalQuantity
hasUnit | Instance* | Units
hasUnit*
Units
hasAbbreviation | String
hasConversionRate | Instance* | ConversionRate
hasConversionRate*
convertsTo
convertsFrom
ConversionRate
convertsTo | Instance | Units
hasConversionFactor | Float
convertsFrom | Instance | Units

Figure 2.4: QoS Ontology [75].

2.1.2 QoS-based web service selection literature

QoS parameters have been used as a basis for selecting web services. To illustrate the approaches to web service selection based on QoS, assume that a consumer requests a web service with functionality "X" and that two providers provide web services "X". The consumer wishes to be able to distinguish between the two services. Therefore, the consumer compares, e.g. the response time of each service, and chooses the provider service with the shorter response time. Several studies have proposed web service selection based on QoS. The general steps of such approaches [121, 131, 162, 194] are:

1. A consumer defines functional and non-functional requirements. Then non-functional requirements include the QoS parameters.
2. Providers define their offers, i.e. services and QoS parameters.
3. A QoS model defines the QoS concepts and values.
4. A process matches the consumer requirements and provider offers and then selects a provider.

For example, [121] proposed an extensible QoS model that includes generic and domain specific QoS parameters and a QoS registry. The QoS registry allows the service providers to register their services and QoS parameters. The QoS information in the registry is provided by the providers, and then computed by the consumers, based on monitoring service execution or user feedback. The QoS registry computes the values of QoS for each service provider. It also ranks the web service providers based on the QoS values, by defining a MCDM matrix. This matrix derives when to select, e.g. a service between multiple alternatives with multiple criteria [193, 195]. Another study [162] provided QoS-based web service selection by first verifying the syntax and semantics of the service and also by its QoS. It then measures the QoS values to compare them with the one advertised.

The above studies are proposed for web services rather than cloud computing, which is our current focus. The nature of cloud computing (which is explained in Section 2.2), where more related QoS properties and its model are considered (see Section 2.4.3). Another difference between the above studies and the current one is the fact that the comparison is based on evaluating or analysing the QoS parameters while the comparison in this study is based on the QoS parameters provided by the providers and consumers as part of the SLA contract. These studies are similar to our work in that they employ QoS as a basis for selecting between different alternatives and defining the MCDM matrix (see Section 2.4.1.1 and Chapter 6); however, in this thesis, one of our objectives is to specify a cloud SLA using MDE to allow providers and consumers to define their offers and requirements (see Section 1.6).

Several studies, for example [194], have proposed a QoS-based selection by specifying a set of QoS ontologies and vocabularies. They used Web Services Modelling Ontology (WSMO) to define the service and QoS parameters. The QoS parameter is defined by *MetricName*, *ValueType*, *Value*, *MeasurementUnits*, *ValueDefinition*, *Dynamic/Static*, *isOptional*, *hasTendency*, *isGroup*, and *hasWeight*. The last four attributes were defined for the purpose of selection. During the selection process, these attributes are checked and matched semantically. A selection algorithm based on these QoS parameters forms a MCDM matrix for selecting a service provider. The common QoS metrics ontology in many studies is [75, 148, 175, 194]: name, unit, value, type and tendency.

A dynamic service selection via an agent, coupled with QoS ontology, has been proposed by [131]. Consumers and providers, by using a QoS ontology, express their preferences and policies. This study distinguished three types of QoS: upper, middle and lower. The upper QoS defines generic concepts of QoS such as, *QMeasurement*, while the middle QoS defines a set of QoS parameters, such as availability and performance. It also defines sub-classes for these parameters, e.g. MTTR and uptime are sub-classes of the availability parameter. The agent then finds a match from the providers' services and ranks the service based on the consumer requirements. Similarly, we specify set of QoS parameters such as availability and performance, and then one or more attributes within each QoS parameter (see Chapter 4).

As discussed in Section 1.6, one of our objectives is to provide a mechanism for modelling cloud SLAs. Therefore, we discuss in the next section cloud computing technology and cloud SLAs.

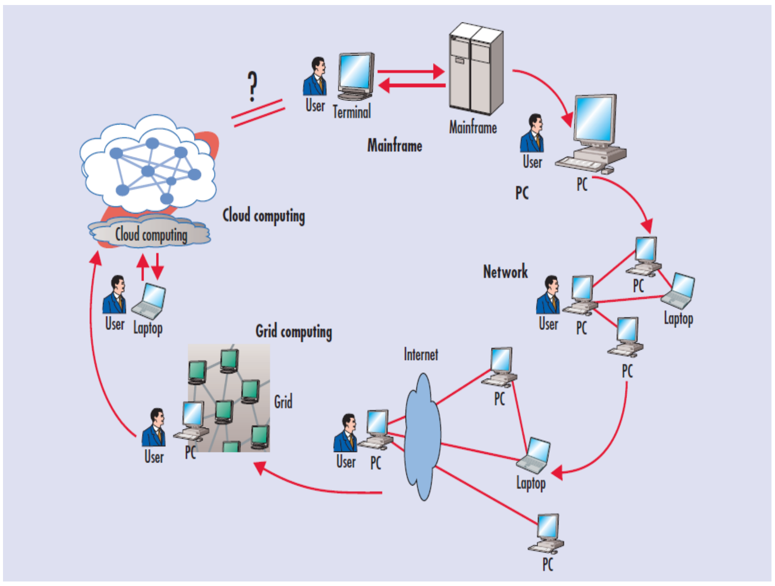


Figure 2.5: Computing Paradigm Shift [191].

2.2 Cloud Computing

In this section, we provide an overview of cloud computing as a basis for developing models of cloud SLAs (as discussed later in Chapter 4). We first provide an overview of the general concepts and characteristics of cloud computing, then outline different deployment mechanisms and service categorization of cloud computing. Following this, we discuss the cloud computing service terminologies. Finally, we discuss commercial cloud computing SLAs and their terminologies.

2.2.1 Introduction

Many definitions have been proposed for cloud computing, each of which, defines the phenomenon from a distinct point of view. The National Institute of Standards and Technology (NIST) defines cloud computing as:

> “a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interac-

> tion. This cloud model is composed of five essential characteristics, three service models, and four deployment models" [133].

Other studies define cloud computing as a parallel and distributed computing system [55], or as providing scalable and QoS guaranteed network-enabled services. [192] and [186] defined a cloud as a pool of virtual, dynamically scaled resources which are exploited by a per-use model. A well-known example of cloud computing is Amazon Web Services (AWS).

Cloud computing is a topic that is attracting significant attention in both industry and academia. It has been described, in [191], as a computing paradigm shift. The authors argue that this paradigm consists of 6 phases, as shown in figure 2.5. In the first phase, users used a dummy terminal with monitors and keyboards, connected to mainframes. In the second phase, users used stand-alone PCs. In the third phase, users share resources connected by a network. In the fourth phase, users are connected to a global network (*internet*), which is formed of local networks connected to each other, using PCs to access applications and resources. In the fifth phase, users using PCs can connect to a computational grid. Grid computing involves combining distributed resources from different organizations to reach a common goal [53]. The sixth phase is cloud computing, where scalable software and hardware resources are provided as a service and can be accessed remotely.

Cloud computing shares aspects with, and is built on, existing technologies. For example, AWS provide cloud services using pre-defined Application Programming Interfaces (APIs) described as *Web services* that are implemented across HTTP. Web services are used to build web applications and can be described using WSDL, which is based on Extensible Markup Language (XML). WSDL describes a collection of operations that web services expose.

Cloud computing is often defined in relation to *Grid Computing* [82, 187]. *Grid Computing* is distributed computing that coordinate resources using standard, open, general-purpose protocols and interfaces for a common goal [82, 186]. Cloud computing and grid computing share several features in common, such as employing distributed resources and supporting the aggregation of heterogeneous hardware and software resources.

However, cloud and grid computing differ in other aspects, such as resource sharing. Grid computing aims to enable fair resource sharing across organizations and coordinates resources whereas, in cloud computing, cloud providers provide resources to cloud consumers as a dedicated service/resource. They differ in other aspects, such as their business model; in cloud computing, the payment model is based on consumption, which is not usually the case in grid computing. Cloud computing is thus a form of *Utility Computing*, which refers to providing computing resources like other utilities, such as electricity. Cloud consumers are expected to pay bills on a consumption basis.

Clouds rely on *Virtualization* technology. [156] describes virtualization as an approach to abstracting the complexity of the physical resources and providing virtual resources to cloud consumers. There are different levels of virtualization, such as server, storage and network virtualization. In server virtualization, the physical server hosts several virtual servers. These virtual servers may run different operating systems and share the same physical resources in the host server. Hypervisor software, such as Xen, KVM and VMware, are examples used to create a virtual machine in the cloud. Cloud computing, by using virtualization, can dynamically add, resize or remove virtual resources (e.g virtual servers) on demand.

Cloud computing has its own features that make it distinct from other technologies. Some of these features will be discussed in Section 2.2.2.

In the context of this work, we need to define the terms: *cloud provider* and *cloud consumer*, which will be used frequently throughout this thesis. A *cloud consumer* (cloud service consumer) is a user or organization that uses or consumes one or more cloud computing services. *Cloud consumers* can be end users, enterprises, developers and cloud resellers. A *cloud provider* (cloud service provider) is an organization that provides cloud services to consumers.

2.2.2 Cloud computing features

In this section, we list the features of clouds that are found in the previous studies about cloud computing. These features are general and independent of the different classes or categories of cloud, which are introduced later in Section 2.2.3.

- *Autonomous*: Cloud computing is typically described as self-service, where consumers can request, manage, configure, release and access their resources, without the need to interact directly with the cloud providers. These activities can usually be performed via automated systems.

- *Pool of Resources*: Cloud computing provides a pool of computing resources as a service to consumers. These resources are generally computing power, memory, storage, networking, operating systems, platforms and applications [82, 158, 186].

- *On-demand service provision*: Resources in the cloud are provisioned and offered on-demand. Consumers can release resources when they are not required. For example, a consumer could request a virtual machine in Amazon EC2 to use for as long as needed; when finished, they can release this computing power [41, 55, 186]. This characteristic can potentially help to reduce the operating expenses, as resources are only paid for as long as they are used.

- *Ubiquitous network access and Location independent*: Resources can be accessed remotely from anywhere and at any time. Cloud computing resources are provided and are available through the network. They can be accessed by using thin and thick clients (e.g. PCs, laptops and mobile phones) [133].

- *Rapid Elasticity*: Because of their self-service and on-demand characteristics, a consumer can rapidly scale-in (decrease resources) or scale-out (increase resources). The provisioning of services, in many cases, can be done automatically. For example, cloud providers such as Amazon EC2 [3] claim that consumers can add a number of virtual servers within a very short time; e.g., minutes rather than hours or days.

- *Metering/Measured services and SLA*: A cloud system provides a metering capability for distinct resources, and charges consumers based on their usage of those resources. Measuring resource usage is a key element needed to charge the consumer accurately. This is important for both billing and capacity planning. Therefore, cloud providers such as Amazon provide tools that automatically and continuously monitor and manage the usage of resources.

  For example, AWS provides a cloudWatch [2] service and resource monitoring, such as CPU utilization within a time period (say 5 minutes) [2]. By monitoring these metrics, a consumer may decide to take some action. Assume that a

CPU utilization is desired to be between 20-90%; when it exceeds the upper limit, a consumer may wish to run another Virtual Machine (VM), and when it falls below the lower limit, this VM, which is considered to be an underutilized resource, can be shut down to reduce costs. Monitoring metrics (e.g. *uptime* and *downtime*) allows consumers to ensure that the guarantees applied by cloud providers in an SLA (e.g., *availability*) are fulfilled.

- *Payment Model*: As discussed earlier, cloud computing uses a pay per consumption model. Consumers pay only for the actual usage of resources. Cloud providers charge on a pay as-you-go or or subscription basis. With the former, consumers pay for a unit of service (e.g., GB, CPU), usually for a certain period of time (e.g. an hour), while, with the latter, cloud consumers sign a contract and pay *"for using a pre-selected combination of service units for a fixed price and longer time frame, usually monthly or yearly"* [196].

The features of cloud computing have been claimed to encourage enterprises to investigate how they can benefit from their applications by migrating to the cloud. This, however, raises several issues concerning aspects such as *reliability and availability*, *interoperability* and *security and privacy* [41, 71, 208]. [89, 149] suggest that the migration process (i.e., application or data) to the cloud does not have to be "all or nothing", as some services can be migrated while others are not. Thus, before moving to he cloud, these challenges should be planned and studied [64, 89].

Cloud computing provides various services. Clouds have been classified based on the services they provide. It is essential to understand the difference between these services and classifications in order to, better plan whether moving to the cloud and achieving the enterprise's goals. The next section discusses the different classifications of cloud computing.

2.2.3 Cloud classification

In this section, we discuss cloud computing classifications to give the reader a better overview of the cloud domain. This will help to explore the domain of cloud computing and understand the cloud terminologies. [38] stated that different categorizations of cloud computing have different QoS parameters to be defined in an SLA. This section presents two classifications of cloud computing.

Cloud computing can be classified depending on the service model and deployment model. Deployment models differ in terms of who the consumers of the cloud services are and who owns the cloud. Existing deployment models include: *public* clouds, *private* clouds, *hybrid* clouds and *community* clouds. Service models differ in terms of the types of services that are provided by the cloud. The most prominent service models are: Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) [151]. We briefly explain both the deployment models and service models, which are summarized in Tables 2.1 and 2.4.

2.2.3.1 Cloud computing deployment model

- *Private cloud*: The services of private clouds are provisioned for exclusive use by an organization. The infrastructure of the cloud is hosted and managed within the organization or can be run and managed by a third party, on behalf of that organization alone [133]. Reduced capital cost is not generally a key goal for such clouds [127, 163]. However, privacy concerns are reduced when compared with public cloud computing. In private clouds, the enterprise owns the data, and they are not shared with others, such as in public clouds [99].

- *Public cloud*: Public clouds provide cloud services to public consumers, who may belong to different organizations. Each cloud provider applies its own policy and pricing models. Examples of such clouds are; AWS, RackSpace, Google, Windows Azure and GoGrid. Consumers of such types of clouds buy services and pay on a per-consumption basis.

- *Community cloud*: In community clouds, cloud services are offered to consumers of the same community with common policies, values and concerns (e.g. security requirements), where they share the same infrastructure [133]. It is similar to a private cloud but it is provided to a set of organizations rather than a single one [97].

- *Hybrid cloud*: Hybrid clouds involve combining of more than one cloud (private and public) to provide a cloud environment, i.e., a cloud infrastructure of hybrid clouds consists of both public and private clouds. Each cloud is independent but they are bound together by standardized or proprietary technology which enables data and application portability [133].

Table 2.1: Deployment models of cloud computing

Deployment Model	**Characteristics**	**Advantages**	**Disadvantages**
Private Cloud	Services exclusively provided for a single enterprise	Critical and core applications can be applied, where the privacy is not the main concern [99]	There can be a significant upfront cost [163]
Public Cloud	Services are provided for public use.	Reduced upfront cost [127], Pay per consumption basis	Security, privacy and jurisdiction [92]
Hybrid Cloud	Combination of private and public cloud	More secure than public ones, where critical data can be stored in a private cloud and is more cost-efficient than private and public clouds [132]	Data portability, planning of load distribution across private and public resources adds to complexity [132, 200]

2.2.3.2 Cloud computing service model

- *SaaS*

  In this service model, cloud providers host the applications. The consumers access applications through networks via thin or thick client interfaces. Consumers do not have to concern themselves with managing the underlying cloud infrastructure, maintenance, installation and updating the application[114, 116] “with the possible exception of limited user-specific application configuration” [133]. In this cloud, instead of buying software licenses and installing applications on in-house servers, the applications are hosted by a third party, which can conceivably reduce the capital costs [203]. Examples of this kind of service are Salesforce.com (Customer Relationship Management (CRM)) and Google Apps (such as Google Docs). SaaS is a multi-tenant platform [156], which supports many consumers using a common infrastructure [173].

- *PaaS*

  PaaS clouds provide an environment in which to build, deploy, test and host applications, which tend to operate online [116, 156]. The PaaS cloud providers provide the hardware and software to build an application, which frees the developers from concerns about the underlying system. Lawton [116] states that PaaS provides the developer with an OS, API programming languages and management capabilities.

  For software developers, the complexity of the underlying physical system is hidden and they only need to focus on the application development process. The developer does not need to worry about how to configure the servers for building applications, and thus has the potential to increase programming productivity [91].

  Cloud providers, such as the Google App Engine [13], Force.com [12] and Heroku [34], are examples of PaaS. Some PaaS clouds allow developers to work offline e.g. the Google App Engine, whereas others, such as Bungee Connect must to be connected to the internet to build their applications [190].

  For example, Google App Engine enables developers to build web applications using standard Java and Python technologies. The application is run under the Google infrastructure. Google provides distributed data storage services as well as, API for authenticating users and sending email using Google accounts [31]. The Google App Engine provides a SDK, to allow developers to work offline and then upload their application to the Google App Engine. They can also upload a new release of their application as a new version.

- *IaaS*

  IaaS is a service model that offers users computing power, storage and network topologies as a service [133]. Cloud providers own the physical resources of

Table 2.2: IaaS, PaaS and SaaS service models

Service Model	Service Contents	Characteristics
SaaS	*Applications*, e.g. Social networks, Office suites, CRM, Human Resource Management (HRM)	– Consumers are provided with an application, which is accessible remotely at any time, and do not have to worry about managing or controlling the underlying infrastructure and application platform [114, 116, 133] – Consumers can configure their applications [133]
PaaS	*Programming languages and environments*, e.g. Development tools, Frameworks, Deployment tools, Databases, Data storage, Message queue	– Consumers are provided with a platform to develop their application and do not have to worry about managing or controlling the underlying infrastructure, servers or operating system [91] – Consumers can develop and deploy applications using the platform and tools provided by PaaS cloud providers [91]
IaaS	*Servers, Storage, Network* (e.g., firewalls, load balancing)	Consumers are provided with virtual hardware and do not have to worry about managing or controlling the underlying infrastructure [132] Consumers can deploy and manage their own software OS on virtual hardware and manage it [91]

IaaS, and supply a pool of resources. The service contents are *computing resources* (servers), *storage devices* and *networking*. Consumers are free from the concerns related to managing the underlying infrastructure, so they can focus on managing the virtual hardware [91, 132]. They can acquire and access these resources (servers, storage and network) as a service remotely. Cloud consumers do not control the underlying cloud infrastructure, but do control the provisioned service, as well as. the operating system, storage and certain network functions such as a firewalls [43].

Amazon EC2 and S3 [1, 5], GoGrid [15], and Rackspace [21] are examples of commercial IaaS clouds, while Eucalyptus [33] and OpenNebula [35] are examples of open source IaaS clouds. In public clouds, providers compete on the basis of the performance of their resources and also pricing. IaaS clouds offer APIs to start, stop, scale and manage these services. In this thesis, one of our objectives is to model SLAs for clouds (Section 1.6), specifically IaaS clouds. Therefore, we discuss the IaaS attributes in more detail. The key IaaS attributes are as follows:

– Computing Power: IaaS cloud provides *servers* as a service; they are accessed remotely ,as in Amazon EC2 [3]. It allows consumers to increase their computing capacity rapidly and without buying and investing in new hardware. The IaaS cloud is based on virtualization technology (see Section 2.2.1), which allows the creation of multiple servers with various operating systems on one physical server; the consumer then acquires the server. Consumers can control the server and its operating system and deploy applications on it, then release or stop the server when it is not needed, to reduce costs. Providers offer pay as-you-go models for payments. In the *subscription* model, consumers pay in advance for certain servers for a certain period of time, whereas, in the *on-demand* payment model, consumers pay for resource usage only for a short time unit (an hour). The price of a server depends on the server performance (RAM, Compute Unit), operating system and payment model.

– Storage: Another facility offered by a typical IaaS provider is on-demand storage. Consumers can store and retrieve their data with the cloud, typically using web access, such as AWS S3 [5] and RackSpace cloud files [21]. Most providers offer storage and charge consumers based on the space they use on a monthly basis. The price depends also on the storage type, which may include: *Object storage* and *block storage*. *Object storage* (such as RackSpaces cloud files or AWS S3) provides access to an object through an API interface, and, contains a collection of objects and each object contains object data and object meta-data [79]. *Block storage* is file level access, which is a "logical array of unrelated blocks, addressed by their index in the array (i.e., their Logical Block Address (LBA))" [79] and can be used when consumers need to increase their hard drive utilisation.

– Networking: Cloud IaaS provides networking and communication services such as load balancing, firewalls, and public IPs. Some providers allow users to create a *Virtual Private Cloud* (VPC). Cloud providers charge consumers for the transferred data or bandwidth. They generally charge based on the amount of data transferred within a month. Many IaaS clouds transfer *data-in* to the service of the service provider, which is a free service or costs less than transferring *data-out* of the service [3, 21].

To provide a better explanation of the IaaS model, let us consider Amazon EC2 as an example [1]. Consumers request computing power from EC2 in the form of VMs; these VMs are launched within a short time. The VMs are created from a pre-configured Amazon Machine Image (AMI), or by creating an AMI that contains the consumer's application, data and configuration. Consumers can store these AMI with data and configuration settings using Amazon S3 as repository, then create VMs rapidly based on these AMIs. They can choose the size of computing power offered by Amazon, such as the operating system, small instances, large instances, extra-large instances, etc., then configure the security group and network access to the EC2 instances. Depending on a set of conditions, they can auto-scale. The pre-defined conditions with the continuous monitoring of VM allow rapid scaling when these pre-conditions are met.

In this section, we discussed cloud computing and the different kinds of provided services, mainly IaaS. We focused on the main concepts of the IaaS service. The main *functional* properties of the IaaS cloud are: computing power, storage,and networking.

We also considered how each service can be charged. Understanding cloud services helps a cloud consumer to decide what is to be outsourced in the cloud. This includes details of how the service is monitored and managed. Given sufficient details, cloud consumers and cloud providers can negotiate over the offered services. Cloud consumers can thereafter compare different offers from various cloud providers based on QoS and cost [64]. Concepts such as computing power, storage and networking can be used as a basis for cloud computing comparison and matching cloud computing services[141, 206].

For cloud consumers, the challenge is to define their demands for their business goals. clearly They have to provide a clear definition of the required service(s) and QoS, while also minimizing the cost. Those requirements are used as a basis for comparing different offers. Like web services and other distributed systems, QoS is used as a basis for comparing the services of different service providers. Therefore, the next section discusses cloud SLAs.

2.2.4 Cloud SLAs

After presenting an overview of SLAs in section 2.1 and cloud computing concepts in section 2.2, we now discuss commercial cloud SLAs. One the objectives of our thesis (Section 1.6) is to provide mechanisms for modelling cloud SLAs. This section investigates the cloud SLA domain, however, an overview of cloud computing researches that discuss the QoS of cloud computing and the comparison and selection of cloud services is presented later in section 2.4.

2.2.4.1 Commercial Cloud SLAs

Our objective of this thesis is to model and compare cloud provider's SLAs and cloud consumer's requirements, and thereafter demonstrate how to use model comparison technology to automate various matching processes. We first analyse and compare the SLAs of commercial cloud providers.

Commercial cloud providers, such as Amazon [4, 6], GoGrid [16] and RackSpace [23], provide a static or *pre-defined* (non negotiable) SLA. [92] argues that there is a lack of well defined SLAs from the commercial cloud providers.

Table 2.3: Common components in SLAs of commercial cloud computing

	EC2	**Azure**	**RackSpace**	**GoGrid**
Commitments	EC2 Availability 99.95%	Monthly Connectivity Uptime Service Level 99.95%	Network 100% for cloud server failure restoration or repair will be complete within an hour of problem identification	100% server uptime 100% uptime of the internal network
Credit	<99.95% monthly uptime: 10% credit, <99% monthly uptime: 30% credit	<99.95% monthly uptime: 10% credit, <99% monthly uptime: 25% credit	Network: 5% of the fees for each 30 minutes of network downtime, up to 100% of the fees Cloud Server Hosts: 5% of the fees for each additional hour of downtime, up to 100% of the fees	Credit equivalent to 100 times the Customer's fees for the impacted Service feature for the duration of the Failure, up to 100%
Credit Request	Email	Customer contact Support	Customer contact Cloud team	Customer completes the automated SLA Credit Request process online
Interval time to request credit	by the end of the second billing cycle after violation	5 business Day	30 days	48 hours
Service Time	Monthly	Monthly	Monthly period	Current period

Commercial cloud providers define different SLA structures and terminologies. For example, Amazon EC2 defines *Monthly uptime*, while RackSpace define *Monthly Availability* in their SLAs. Gogrid, as another example, defines different cloud services, their QoS parameters and SLO for each service in one cloud SLA, while Amazon EC2 defines two SLAs; one for the EC2 service and the other for the S3 service.

In this section, we compare some of the *pre-defined* and non-machine readable SLAs of IaaS clouds, which are: Amazon [4], GoGrid [16] and RackSpace [23], to identify the similarities and differences between the SLAs. In the comparison, we sought any similarities and differences in the structure of the SLA components, such as services (e.g. server, storage, network) covered by the SLA, SLO and QoS.

The general structure of pre-defined SLAs is quite similar, as observed in Amazon EC2 and S3 [4, 6], GoGrid [16], Windows Azure (Cloud Services SLA) [32] and RackSpace [23]. The general common structure of pre-defined cloud SLAs consists of:

- *Service*: each cloud provider defines the services covered by the SLA. For example, Amazon ECs SLA covers the services *Amazon Elastic Compute Cloud and Amazon Elastic Block Store*, while there is another SLA for *Amazon Simple Storage Service* which is called Amazon S3. The Gogrid SLA covers *computing service, cloud storage and networking* in one SLA.

- *Definition section*: defines and describes terms used in the SLA. For example, Amazon EC2 defines *monthly uptime* and *service credit* and this is applied to the other clouds SLAs.

- *Obligation section*: contains the SLOs of the service. These define precisely the thresholds of the quality values and describe the action to be taken when a violation occurs. In some pre-defined SLAs, this section is called commitment. The most common QoS indicator introduced in these SLAs is *availability*, particularly its attribute *uptime*.

- *Credit*: This section describes how the credit is calculated. In case of violation or failure of an SLA, certain remedies may apply. In the case of cloud computing (Amazon, GoGrid, RackSpacae and Windows), these remedies are credits, which are calculated and defined and given by the cloud provider to the cloud consumer.

- *Credit Request*: This describes the requirements for requesting credits e.g. requests within 5 working days of when the violation occurred and providing evidence of the violation (service logs).

Table 2.4: Differences in terms of QoS parameters referred to in commercial cloud SLAs

	EC2	Azure	RackSpace	GoGrid
Availability	99.95% Monthly Uptime Percentage	99.95% Monthly Connectivity Uptime	99.9% Monthly Availability	100% Individual servers uptime
Time to resolve	-	-	1 hour	-
Support response time	-	-	-	30 m (Emergency), 120 m(Others)
(Internal) Network performance	-	-	-	Packet loss <0.1, Latency <5ms
Outage	Running instances have no external connectivity, Elastic Block Store (EBS) perform zero read write IO	Connectivity Downtime	API error	Failures in the hardware and hypervisor

However, these pre-defined SLAs differ in terms of the details of those sections, depending on the technology offered by the provider, as illustrated in Table 2.4. We observed that availability is a common QoS indicator in cloud pre-defined SLAs. The terms *availability* or *uptime* always appear in publicly available cloud provider SLAs, but availability properties, such as uptime and downtime, are defined and calculated differently. These attributes, together with their threshold (which are used to evaluate availability), are defined in the obligation section of each SLA. This difference in terminology (and the semantics of the widely used terms) makes it difficult for consumers to make meaningful comparisons of cloud SLAs. The challenge is to enable the automated comparison of similar QoS parameters, while keeping the SLA flexible to allow cloud providers and cloud consumers to define their own terminologies.

We aim to provide support for comparing different cloud SLAs, using different scenarios then use the results of these comparisons for further processing; e.g., different visualizations. The objective of this thesis (see Section 1.6) is to model cloud SLAs and automate the comparison process using MDE. MDE supports automated processing and standard representations and is used successfully in software development processes. Therefore, the topic of the next section (see Section 2.3) is MDE.

2.3 Model Driven Engineering (MDE)

MDE uses models as first-class artefacts of the development process. It is argued that MDE can be used to manage the complexity of software development, by using standardized languages and the automated management of engineering artefacts (models). In MDE, models are expressed at different levels of abstraction to capture various concepts, knowledge and also the requirements of the problem domain. MDE "allows the exploitation of models to simulate, estimate, understand, communicate and produce code" [84]. MDE aims to improve the productivity and quality of software development, to improve communication and information sharing between stakeholders and also automation [136]. This approach is relevant to abstracting the complexity of the software development process.

In this thesis, we present cloud SLAs as standard artefacts to automate the process of cloud SLA comparison and selection. Thus, to achieve these goals and as explained in the introduction (see Section 1.6), we employ MDE; hence, this section gives an overview of MDE concepts. MDE involves the key concepts of: models, metamodels, model transformations and model management. This section describes these MDE artefacts and relates them to the previous discussion on cloud computing and SLAs.

2.3.1 Models

Models are the core artefacts in MDE. As in object-oriented technology, where *"Everything is an Object"*, MDE is based on the principle that *"Everything is a model"* [49]. There have been many definitions of the term *model*: *" A model is a representation of a concept. The representation is purposeful: the model purpose is used to abstract from the reality the irrelevant details"* [168]; *"A model is a purposely abstracted, clear, precise and unambiguous conception"* [80]; *"A model is a purposeful abstraction that allows one to reduce complexity by focusing on certain aspects"* [184]. Thus, a model is an *abstraction* of reality, which can be a system, phenomenon, object system or a system under study (SUS) [113]. The model represents the main properties of the system under study. By focusing on the main properties and discarding ones, models simplify and reduce the complexity of the system under study.

A map is an example of a model; it represents a certain geographical area. For same area, there can be different maps, each of which is represented in a way that supports a certain goal [49], e.g., to describe political boundaries, the geological structure, etc.

Models are used widely in the field of science. Some examples of model usage in science are mathematical models, statistical models and biological models. In software development, models are used generally; for example, to describe software architecture or the behaviour of a system. In the context of software development, one of the aims in using MDE is to increase the automation of the software development process [88, 105]. One of the definitions of a model, in the context of MDE, is

> "a description of (part of) a system written in a well-defined language. A well-defined language is a language with well-defined form (syntax), and meaning (semantics), which is suitable for automated interpretation by a computer" [103].

A model is expressed in a modelling language. A modelling language is defined in terms of metamodels. A metamodel describes the abstract syntax of a language.

2.3.2 Metamodel

The relationship between a model and a metamodel is called instantiation. A model *conforms to* a metamodel [49]. A metamodel defines the modelling language in a high level of abstraction (e.g. above the code level). A model conforms to a metamodel, which specifies its elements based on the concepts that are defined in the metamodel, and uses these elements according to the rules or constraints of the metamodel. A valid model is one that conforms to a metamodel and satisfies the constraints captured by its metamodel [146].

For example, classes, data types and packages could represent the key construct (abstract syntax) of the modelling language. Figure 2.6 provides an example of the metamodel abstract syntax, which defines a model. This model consists of a collection of types, each of which has a name and attributes which also have a type. Concrete syntax is the notation that is used to enable the description and presentation of the model [61]. Concrete syntax can be textual or graphical [61]. A common example of a general purpose modelling language is the Unified Modelling Language (UML) [29].

Returning to the abstract syntax shown in Figure 2.6, a possible concrete syntax for this abstract syntax is listed in Listing 2.1. Semantics are used to describe the meaning of the concepts in the language.

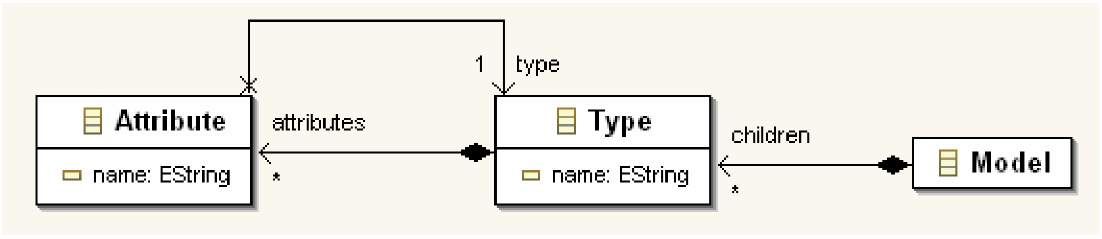


Figure 2.6: Metamodel of abstract syntax for a simple language [137].

service requester SLA model with provider SLA model]

Listing 2.1: Concrete Syntax example [137]

```
Type Mail {
 From : User
 To : User
}
Type User {
 Name : String
}
Type String;
```

Several metamodelling frameworks have been proposed, which can be used to define metamodels and models that conform to their metamodels, such as the OMG Meta Object Facility (MOF) [19] and Eclipse Modelling Framework (EMF) [9].

2.3.3 Model management

The aim of MDE is to raise the level of abstraction and increase the automation in the software development process. A typical MDE process involves various inter-related models that capture different level of abstractions and may use different modelling languages. Model management is the discipline of manipulating models (e.g., via transformation, comparison, merging) using automated tools. Model management tasks, such as model transformation, validation and comparison, are discussed in this section.

2.3.3.1 Model Transformation

Transformation is a fundamental operation in MDE. A transformation transforms a *source* model into one or more *target* models, based on the transformation rules. The rules describe how concepts in the source model relate to those in the target models; these relations are expressed in terms of the source and target metamodels [103, 166]. Transformations can be used to transform models from higher levels of abstraction to lower ones. They can also be used to transform models at the same level of abstraction. Another transformation can be performed from a low level to a higher level (e.g. in reverse engineering). Transformations can also generate code.

There are different approaches to transformations such as Model-to-Model (M2M), Model-to-Text (M2T) and Text-to-Model (T2M). In M2M, the target model is derived from the source model. The transformation in M2M is usually defined by a set of rules, each of which specifies how the source element(s) is transformed into the equivalent target element(s) [109]. Many M2M transformation languages have been proposed, such as the Epsilon Transformation Language (ETL) [109], Query/View/Transformation (QVT family) [20] and ATL (ATLAS transformation language) [98]. M2M languages can be classified as imperative, declarative or hybrid [109]. In the imperative transformation, the scheduling of the rule execution is defined by users explicitly, whereas in declarative languages, an execution engine decides which rule is to be executed. Hybrid languages are a mix of implicit and explicit rule scheduling. An example of a M2M transformation written in ETL (a hybrid language) is shown in listing 2.2. ETL syntax is explained later in section 2.3.4.7.

In an M2T transformation, the *target* is text, e.g. code or non code artefacts such as documentation. Common approaches to M2T transformation are template-based approaches [65]. The Epsilon Generation Language (EGL) is an example of M2T transformation language, as explained later in section 2.3.4.6.

The third type, T2M transformation, transforms text into generate structured artefacts that are used to form models. The different transformation approaches are presented in [65].

Listing 2.2: Example of ETL rule from [11]

```
rule Tree2Node
transform t : Tree!Tree
to n : Graph!Node {
 n.label = t.label;
 if (t.parent.isDefined()) {
  var edge = new Graph!Edge;
  edge.source = n;
  edge.target = t.parent.equivalent();
 }
}
```

2.3.3.2 Model Comparison

Model comparison is an operation executed on two models to identify any matching elements between them [54, 106]. Model comparison is an important task in MDE; it supports model development processes (e.g. for model versioning) [60], model composition and model transformation testing [107]. For two model elements, we need a *calculation* or *algorithm* in order to be able to compare or match two models [54]. The outcome of this calculation may identify any differences; e.g., elements for which matching elements do not exist in the opposite [107] model.

Several model comparisons apply different calculations and methods to match two models. [110] categorized several approaches to model matching. In *Identity-based matching*, elements are matched based on their identities, assuming that each element has a unique, static and persistent identity. This approach finds any differences between two versions of the same model, and so does not work for independent models. The second approach is *signature-based matching*. This is similar to identity-based matching, in that it matches the identities of elements. However, the identity in this matching is not static but calculated dynamically. This approach applies to comparing independent models. A third approach is *similarity-based matching*. This approach seeks to calculate the aggregate similarity of the features of elements. It also specifies a weight for each feature. Finally, in *custom language-specific matching*, users can specify a comparison algorithm using a specific language for comparison. While this approach can provide more accurate matching results, it requires significant effort to develop the comparison algorithm. ECL is an example of this approach. This thesis' objective, as described in Section 1.6, is to compare cloud SLAs using MDE principles; thus we use one of the model comparison approaches, i.e., ECL, which is explained in section 2.3.4.4.

2.3.3.3 Model Constraints

In a large software development process, where different but related models are created for different perspectives [51], it is essential to detect and report any inconsistencies between the models [51]. Models with missing information (*incompleteness*) and incompatible information (*contradiction*) are forms of inconsistency [105].

The Object Constraint Language (OCL) is a prominent language used to specify the consistency constraints in UML. Epsilon Validation Language (EVL) is another example of a model validation language, as discussed in section 2.3.4.7. The EVL supports the repairing of inconsistencies as well as inter-model constraints, which is not the case with OCL [105].

2.3.4 MDE Tools

A number of model management frameworks exist, including AMMA [48], EMF [169], and Epsilon[10]. In the following sections (2.3.4.1 and 2.3.4.2), we discuss

EMF and Epsilon, as these technologies are employed in this thesis.

2.3.4.1 EMF

The Eclipse Modelling Framework (EMF) is built on the Eclipse platform, which is in itself a tool-integration platform for software development. It provides tools such as a graphical editor to define the metamodels and tools to generate model editors form the metamodels. The EMF framework includes three main components: Ecore, which is used to specify metamodels, and *EMF.Edit*, which is used for building editors for new modelling languages and *EMF.Codegen*, which is a code generation for facilitating the building of an editor for the EMF model. The EMF model editor consists of; a navigation view to specify the models' elements and property, and, is used in the model examples in Chapter 4. Several tools have been built for use with the EMF, such as the Graphical Modelling Framework (GMF) [14] and Epsilon [10], as described in section 2.3.4.2.

2.3.4.2 Epsilon

We use Epsilon in this thesis to implement model comparison, transformation, management and and creation. Epsilon [108] stands for *Extensible Platform of Internet Language for mOdel maNagement*. It is a family of task-specific languages which can be used to manage EMF models. Epsilon can be used to perform different MDE tasks, such as model-to-model transformation, code generation, model merging, comparison and validation. For each of these MDE tasks, Epsilon provides a task-specific language. *EOL* is the core language and other model management languages are built atop it. In the following sections, we explain each of the Epsilon task-specific languages that are used later in this thesis.

2.3.4.3 Epsilon Object Language (EOL)

EOL [11, 108] is a model management language used to manage models from various technologies such as EMF and XML. It allows the creation, modification and querying of models. EOL contains different features. It supports model querying operations such as select(), collect(), etc. It has the capability of model modification, such as modifying element's properties, *adding*, *removing* and *deleting* model elements.

Furthermore, EOL supports access to multiple models that are possibly conforming to different metamodels. EOL users can reuse their defined operations, not only in other EOL programs, but also in other Epsilon languages. Finally, it supports common programming constructs such as *while and for loops* and *if..else statements*, and also provides support for user interaction. EOL has been used in the cloud SLA comparison and selection process in Chapters 5 and 6.

2.3.4.4 Epsilon Comparison Language (ECL)

ECL is a rule-based, metamodel-independent language for comparing the arbitrary models of arbitrary metamodels [11]. As with other Epsilon languages, ECL is built on EOL language. With ECL, users can define comparison rules for the elements in two models. The result of the ECL comparison is a trace that consists of a number of matches. Each match has a reference to the compared elements (left and right) and a Boolean to indicate the matching result. It also has a reference to the match rule that led to the decision.

As ECL is used in this thesis for comparing and matching elements of cloud SLA models, we provide further details of it. The concrete syntax of an ECL rule is displayed in Listing 2.3. ECL rules are specified in modules *ECLModule*. In this module, a user can import other epsilon modules, a set of match *rules* and *pre* and *post* blocks [106]. Statements defined in the pre part are executed before executing the match rules, while statements in the post part are executed after executing the match rules that are defined in the module. Both the pre and post parts are optional. The name of the rule follows the keyword *rule*. This rule declares two parameters *leftParameter* and *rightParameter*, e.g. two elements from two models. The left parameter is defined after the *match* keyword and the right parameter is defined after the *with* keyword. The rule can optionally extend other rules. These are separated by a comma and defined after the *extends* keyword. Rules that are specified in the extends part are executed before the comparison statement and, if the rules in this part are not satisfied, the compare part is not executed. This extends is an optional part. The rule can optionally specify a *guard* expression or block of statements. The guard statement narrows the set of matched elements; if the guard fails for specific parameters, the compare part is not executed.

After the *compare* keyword, users specify comparison expressions or block of statements. Expressions after *guard* and *compare* follow a column. The block of statements is defined between curly brackets ({}) after the *guard* and *compare* keyword. In this comparison part, a user can specify their matching statements to indicate whether the left and right parameters are matched or not. This compare part returns a Boolean value to indicate whether the two parameters (left and right) are matched or not. If this Boolean value indicates that a match has been found (true value), an optional *do* is executed and if the comparison result is true, a user can specify any additional actions. If the left and right parameters are matched, the do part allows a user to define any other actions. An ECL rule is executed automatically for the non-lazy and non-abstract rules. The operation *match()* invokes the lazy rule.

Listing 2.3: Concrete Syntax of a MatchRule from [11]

```
(pre <name> {
 statements+
})?
(post <name> {
 statements+
})?

(@lazy)?
(@greedy)?
(@abstract)?
rule <name>
match <leftParameterName>:<leftParameterType>
with <rightParameterName>:<rightParameterType>
(extends (<ruleName>,)*<ruleName>)? {
 (guard (:expression)|({statementBlock}))?
 compare (:expression)|({statementBlock})
 (do {statementBlock})?
}
```

2.3.4.5 Epsilon Transformation Language (ETL)

ETL is a hybrid model-to-model transformation language, built on EOL; this allows the transformation language to integrate with other Epsilon languages and share and reuse operations [109]. ETL transforms input models into arbitrary output models. As in ECL, ETL rules are organized in modules *ETL*. ETL modules include also *pre* and *post* blocks, as in the ECL module. Listing 2.4 displays the concrete syntax of ETL.

Each rule defines a name, a *sourceParameter* and one or more *targetParameter*. The sourceParameter is used to specify an element of the source model that is involved in the transformation, while the targetParameter specifies the target element(s) of the target model. Similar to ECL, an optional *extends* specifies the rules that it extends, separated by a comma. After the optional *guard* keyword, a user can define simple EOL expressions following column (:) or as a block of statements between curly brackets ({}). The ETL statements are executed if the optional parts extend and guard rules and expressions are satisfied. Finally, a sequence of EOL statements form the body of the transformation rule that is executed, which means a target element(s) is/are created. These statements populate the contents of the created element(s). The ETL is used in this thesis to transform the outcome of the matching process to a decision matrix (Chapter 6).

Listing 2.4: Concrete Syntax of a TransformationRule from [109]

```
(@abstract)?
(@lazy)?
(@primary)?
rule <name>
transform <sourceParameterName>:<sourceParameterType>
to (<targetParameterName>:<targetParameterType> (, <
    targetParameterName>:<targetParameterType>)*
(extends (<ruleName>,)*<ruleName>)? {
 (guard (:expression)|({statement+}))?
 statement+
}
```

2.3.4.6 Epsilon Generation Language (EGL)

EGL is a model-to-text transformation language. It is a template-based language for code generation and documentation [159]. It includes standard features, such as decoupling content from the destination, mixing generated and hand-written code, and coordinating templates. EGL defines sections which are: *static* sections, *dynamic* sections and *dynamic output* sections. The contents of a *static* section appear verbatim in the generated text. The executable code (expressed in EOL) is defined in the *dynamic* section and is enclosed between [% %]. The results of evaluating the expressions in dynamic output, which has the syntax [%=expr%], are included in the generated text. EGL is used in this thesis to generate HTML documents for the outcome models of the matching process (as described in Chapter 7).

2.3.4.7 Epsilon Validation Language (EVL)

EVL is a validation language built on EOL, which is used to evaluate constraints on models [11]. This task-specific language allows users to distinguish between errors (*constraints*) and warnings (*critiques*), and to specify *fix*es for failed constraints. Figure 2.5 describes the concrete syntax of the EVL context. A context part specifies an element type that is involved in the evaluation in the *invariant* part and is a *guard*. This guard is used to limit the instances involved in this context. The *invariant* can be (*constraints*) or (*critiques*). Figure 2.6 describes the concrete syntax of the invariant. Both constraints and critiques define an optional *guard* part, which is defined to further limit instances that are involved in this operation. This guard may have an expression or a block of statements. If the expression (or block of statements) is satisfied, then a *check* part is executed. This part specifies an expression or block of statements. Similar to the guard part, if the expression (or block of statements) is satisfied, it executes the *message* part. The message part, which consists of expressions or blocks of statements, returns a String. This String describes why a constraint has failed for an element. The optional *fix* part consists of a block of statements that can be used by users to define fixing actions. EVL is used in this thesis to define the constraints on cloud SLA models, as described in Chapter 4.

Listing 2.5: Concrete Syntax of an EVL context[11]

```
context <name> {
 (guard (:expression)|({statementBlock}))?
 (invariant)*
}
```

Listing 2.6: Concrete Syntax of an EVL invariant from [11]

```
(@lazy)?
```

```
(constraint|critique) name {
 (guard (:expression)|({statementBlock}))?
 (check (:expression)|({statementBlock}))?
 (message (:expression)|({statementBlock}))?
 (fix)*
}
```

We reviewed the MDE and the tools that will be used in this thesis. Cloud computing SLAs modelling and the automated comparison of cloud SLAs will be been discussed with regard to thesis objectives (see Section 1.6) in Section 2.4. In Section 2.4 we will discuss the rautomated comparison of cloud SLAs will be been discussed with regard to thesis' objectives (see Section 1.6), including SLA specifications.

2.4 Comparison and Selection of Cloud Computing

As described in Section 1.5, the thesis' hypothesis investigates how cloud SLAs can help consumers to select between different offers.

Web service selection has been discussed in the literature (such as [94, 121, 131] and [162]) using different approaches. [204] provides a classification of different the approaches to web service selection, which used consumer preferences as one of the criteria that used for classifying the various selection approaches. [94] suggested that service comparison and selection, that help consumers to choose an appropriate web service, consists of 3 phases: functional matching, text-based QoS matching and finally QoS numeric-based QoS matching.

When considering how to automate cloud selection, we can consider the questions raised in Section 1.2. The first question asks about the basis for the comparison - what parameters are involved in the selection/matching process, and how are they named and modelled (see Sections 2.4.3 and 2.4.4). The second question relates to how to select an offer amongst different offerings (see Section 2.4.1). The third question is how to match a requirements and offers to filter the similarities to be involved in the selection process (see Section 2.4.2). The final question is how these requirements and offers are captured to automate the comparison and selection process (see Section 2.4.5).

The next section discusses the literature on service selection (i.e. web services and cloud services) which are based on numeric-based QoS matching. Then, we discuss the studies that have addressed text-based QoS (including functionality matching). After that, this section discusses the QoS parameters that are of interest in the domain of cloud computing and to be included in the matching process. Usually, QoS is parts of SLAs. We discuss some of the electronic SLAs that are used to define web services or cloud computing SLAs. These electronic SLAs are used to automate the comparison process regarding cloud offers. After that, we conclude by identifying the research gap and the contribution of this work, before presenting a summary of this chapter.

2.4.1 Cloud computing QoS-based selection

Like web service selection, in the domain of cloud computing, several studies have discussed cloud services selections [83, 90, 119, 128, 199]. Various studies have discussed QoS matching (i.e. numeric-based) in the domain of cloud computing. This matching is used to distinguish the appropriate cloud service offer from other cloud services offers that have the same functionality and are based on the same QoS criteria (i.e. the results of functional and text-based matching). This numeric-based matching is discussed in [83, 85, 183] as an Multi-Criteria Decision Making (MCDM). The MCDM approach is widely used in to solve selection problems [93, 204] (e.g. web service selection). In this approach, calculation processes are performed using different criteria selected from among different alternatives (e.g. cloud service offers from different cloud providers) to enable consumers to make decisions to select appropriate alternatives (see Section 2.4.1.1). In MCDM, decision makers (e.g. consumers) can provide their preferences based on the criteria, which is (i.e. consumer preferences) an essential factor in the selection approaches [204].

For example, [18] used MCDM to compare between different cloud providers. In this study, the criteria are determined based on the literature and expert interviews. [18] defines 3 main criteria (i.e. Provider perspective, Service perspective, Support perspective) and 8 sub-criteria. The importance of these criteria is based on the experts' opinions.

However, this thesis is interested in the consumer requirements as a basis for cloud provider selection (see Section 1.5). [83] provides a mathematical framework as a means to for selecting a cloud offer based on consumer requirements. The authors suggested a QoS model for IaaS providers and then, by providing consumer requirements and collecting offers from different providers and based on the QoS models as a criterion, a calculation is performed to select the provider offer. The authors did not discuss how these criteria (i.e. consumer requirements) were captured for matching similar criteria from the providers' offers. Furthermore, it is not discussed in this study, i.e. [83], how the cloud providers' offers for the same requested criteria were filtered to form MCDM (i.e. text-based QoS matching).

[183] compared 13 cloud services based on performance attributes and applied MCDM. They applied different MCDM techniques, such as Min-Max, and assumed that all of the compared criteria have the same consumer preferences (i.e. the same weights). In MCDM different techniques cause significant differences in the results. Consumers (i.e. decision makers) may select and apply MCDM techniques depending on different factors [179]. The following Section (2.4.1.1) describes the MCDM approach.

2.4.1.1 MCDM

This section presents a brief description of MCDM (or Multi Criteria Decision Analysis MCDA [177]), which "is concerned with designing mathematical and computational tools to support the subjective evaluation of a finite number of decision alternatives under a finite number of performance criteria by a single decision maker or by a group" [122].

As explained in [183], the MCDM is illustrated as an evaluation matrix. This evaluation matrix (as shown in Figure 2.7) has different m alternatives ($A_1, ..A_m$) and n criteria ($c_1, ..c_n$). In this figure, each row has a number of values, a_{ij} represents the value of the alternative A_i against the c_j criteria. A weight value can be assigned to each criterion to add importance to the criteria according to the decision maker. This is represented in the figure as w_j where each criterion is assigned a weight. The values (a_{ij}) in the matrix may have different measurements. Therefore, the first operation performed in the matrix is Normalization. Then calculations are performed to find the best alternative.

$$\begin{array}{c|cccc}
 & \multicolumn{4}{c}{\textbf{Criteria}} \\
 & C_1 & C_2 & \dots & C_n \\
\textbf{Alternatives} & (w_1 & w_2 & \dots & w_n) \\
\hline
A_1 & a_{11} & a_{12} & \dots & a_{1n} \\
A_2 & a_{21} & a_{22} & \dots & a_{2n} \\
 & . & . & . & . \\
 & . & . & . & . \\
A_m & a_{m1} & a_{m2} & \dots & a_{mn}
\end{array}$$

Figure 2.7: MCDM matrix [178]

Different MCDM methods for calculating the best alternative have been proposed in the literature, each of which has its respective advantages and disadvantages [58]. Several studies compare these methods; e.g., the Weighted Sum Method (WSM), Weighted Product Method (WPM), Analytical Hierarchical Process (AHP), Technique for order preference by similarity to ideal solution (TOPSIS), VlseKriterijumska Optimizacija I Kompromisno Resenje (VIKOR), etc., which may return different results and how to select one of them [58, 152, 178].

The next section discusses QoS matching in the cloud computing domain to filter the alternatives that define the same service functionality and QoS properties as required by the consumer.

2.4.2 Cloud computing matching

In this section, we present the studies that match the services that define both similar functionality and QoS terminology as required by consumers. These studies are based on ontology matching, which is "the problem of finding the semantic mappings between two given ontologies" [73]. Semantic matching "is an approach where semantic correspondences are discovered by computing, and returning as a result, the semantic information implicitly or explicitly codified in the labels of nodes and arcs" [147]. The semantic matching approach, as proposed by [147], identifies the degree of similarity between the offered and required services.

For example, [47, 120] discussed matchmaking approaches for functional and non-functional (i.e. QoS parameters) requirements. The matchmaking algorithm calculates the matching degree, which is identified as: *exact*, *plugin* and *subsume* and *fail* [47]. The matching degree is based on a ranking that depends on the number of matching elements found in an offer, which ignores the values of the parameters and the consumer preferences.

In the domain of cloud computing, [100, 142, 155] employ matching cloud services approaches that are based on ontology-based semantic matching. For example, [142] used OWL-S. OWL-S, which is a language that specifies the service profile as well as the service-process model but not the QoS ontology. [194] proposed an ontology for the QoS model based on an existing ontology model. Then, based on this ontology language, the services are filtered by matching the functional properties, then matching the QoS properties. [74] presents a unified QoS/SLA ontology for QoS-based web service selections.

However [129, 155, 157, 202] introduced SLA mappings, which are argued to increase the market liquidity, where the SLA parties negotiate the already understood and mapped SLA parameters. Unlike ontology, SLAs can be used by consumers and providers in other management processes, e.g. SLA negotiation and monitoring. SLAs in [155, 157] are expressed using WS-Agreement or WSLA.

For example, [155] proposed to match SLAs because they define quality parameters to express consumer requirements and provider offers. [155] used a machine learning algorithm to automate the management of market knowledge; this approach is based on building market knowledge based on user service requirements. This market knowledge is used in automatic SLA matching and automatic SLA selection. The automatic matching is based on matching two SLA elements semantically, followed by automatic mapping to show the differences between their syntax specifications. The automatic matching of SLAs is performed first by calculating the similarities between the element definitions using a character-based similarity metric, then checking the similarity between the element metrics to see if they are semantically similar, and if their functions are logically similar. Then the similarities between all of the properties are combined to calculate the probability of element equality. When two SLA elements of two different SLAs are matched, an SLA mapping is created and recommended to users. A provider is automatically selected of by iterating a set of public SLAs and checking the similarity between all of the elements that are required by users. The SLA that has the highest probability and exceeds a predefined threshold is chosen as the optimal offer.

However, [155] suggested selecting a provider based on the total number of matched elements, which fails to consider the consumer preferences or priorities regarding the quality parameters. Another issue is that the correctness of the SLA mapping depends on user feedback, who may submit contradictory feedback.

Another matching approach is [90], who proposed a recommender system (RS) to help consumers to select the best services from different cloud providers which match their requirements. This is based on user feedback, while [62] showed that users are likely to ignore explicit ratings. [170] is based on the Cloud Service Provider Index (CSP-Index), which captures the similarities between various properties of the cloud providers. The cloud broker collects service properties from the cloud providers, such as service type, unit cost, and available resources. Each service is defined by an *indexing key*. However this approach does not consider the variation in QoS values between provider offers.

As discussed in Section 1.6, this study focuses on modelling cloud SLAs, which is used by cloud parties, to provide their offers and requirements and can be used for service comparison and selection, as introduced in [155]. Therefore, the following section discusses the QoS parameters' terminology and model, which can be used as a basis for cloud SLAs selection (see Section 2.4.1).

2.4.3 Cloud computing QoS parameters and modelling

1. *Which QoS parameters should be considered in cloud SLAs?*

   QoS parameters were used as a basis for comparing and then selecting the offers that fits the consumer requirements [83, 183, 194]. Therefore, this thesis uses cloud QoS parameters, which are a part of cloud SLAs, as a basis for comparing cloud computing offers.

   To compare these QoS parameters, we first explore what these are. Are the QoS parameters of cloud computing SLAs similar to other computing technology, such as web services or grids? Some QoS parameters are specific to cloud computing, such as auto-scaling [38] or elasticity [83]. Are the QoS parameters of cloud computing SLAs similar to other computing technology, such as web services or grids? [197] argued that cloud SLAs are different to traditional web services, as more QoS parameters related to energy, security, privacy and trust should be considered. [38] distinguishes the main SLA metrics according to the cloud service models i.e., SaaS, PaaS, IaaS, and general terms of SLA. For example, in IaaS clouds, the important SLA metrics are: CPU capacity, memory size, boot time, storage, scale up, scale down, scale up time, scale down time, auto-scaling, availability, response time and the maximum number of VMs that can be configured on a physical server. Another set of QoS parameters are: availability, accessibility, arrival-rate, non-repudiation, isolation, regularity and completion-time; these are defined in the SLA@SOI project, as discussed earlier in section 2.4.5. Cost is one of the factors that affect cloud consumers' preferences for one cloud provider over another. These QoS parameters should be considered when designing our cloud SLA model.

As we can see, different studies suggested different cloud QoS parameters; some of these parameters are specified depending on the services provided by the cloud.

The comparison approaches [121, 162, 194] are based on the terminology of the QoS parameters. These studies specify the QoS terms in order to select a web service based on its QoS. As we discussed in Section 2.1. SLAs include the QoS parameters. We want the SLA specification, which includes the QoS parameters, to facilitate the comparison and selection approach. Therefore, like [121, 162, 194], to compare and select an appropriate cloud provider offer based on the consumer requirements, this thesis proposes to specify the QoS terms in the cloud SLAs.

However, as discussed in Section 2.2.4.1, the pre-defined cloud providers may use different QoS terminology in their SLAs. Therefore, designing cloud SLAs that specify the QoS terminology while allowing a degree of flexibility for the cloud parties (e.g. cloud providers) to define their own QoS terms is proposed in this thesis (see Section 4.2.1)

2. *What is the QoS parameter model?*

We include this question because one of our goals in this thesis is to model cloud SLAs, especially QoS parameter definitions. [26] defined QoS parameters and how they are measured, which is different to the former study. For example, reliability is defined in [83], as illustrated in Figure 2.8, whereas 2.9 Figure illustrates the reliability in [27]. In this example, reliability is defined differently in these two studies, in [83], reliability is defined in terms of the number of failures and the promised mean time of failure, whereas, in [27] is defined in terms of the total number of invocations of the number of failures. [56] defined cloud reliability as illustrated in equation 2.3. From the above examples, we can detect the existence of differences when defining QoS parameters, i.e., reliability in this example. The QoS parameters were defined using different terms and metrics.

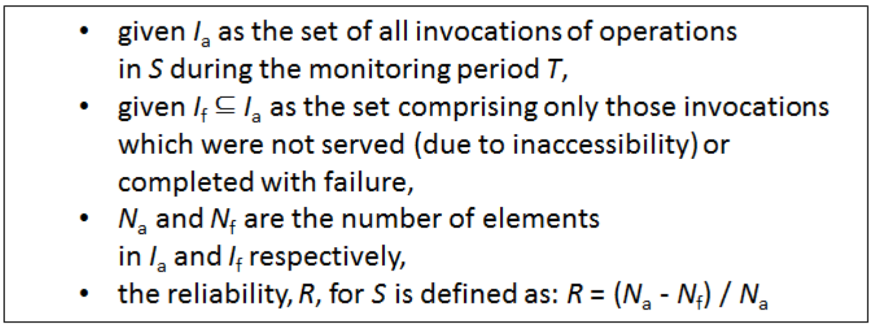
Therefore, we design SLA specifications that allow of these differences in the sub-parameters, by specifying the QoS concept, while allowing the SLA parties to define their sub-parameters within the QoS concepts (see Chapter 4).

- *Numfailure:* is the number of users who experienced failure in the amount of time less than promised by the Cloud provider
- *n:* is number of users.
- *p_mttf :* is the promised mean time to failure

- *Reliability = probability of violation × p_mttf*

$$=\left(\frac{1 - numfailure}{n}\right) * p\_mttf$$

Figure 2.8: Reliability definition [83].



- given I_a as the set of all invocations of operations in S during the monitoring period T,
- given $I_f \subseteq I_a$ as the set comprising only those invocations which were not served (due to inaccessibility) or completed with failure,
- N_a and N_f are the number of elements in I_a and I_f respectively,
- the reliability, R, for S is defined as: $R = (N_a - N_f) / N_a$

Figure 2.9: Reliability definition SLA@SOI [27].

$$CloudServiceQOS_{Reliability}(S) = R/M \tag{2.3}$$

where:

S = service

R = the times of called and successful implements of S

M = total called times of S

3. What is the model of the QoS metric?

   The metrics describe how the parameters are assigned value. As described in Section 2.1.2, the parameter has: a name, value, type and unit. Other properties an be defined based on the usage purpose.

As discussed in this section and Section 2.2.4, the terminology and model of QoS parameters in cloud computing may differ. In this work, we model an SLA for a specific domain of cloud computing, which is IaaS, where cloud providers provide different resource units. These resources, which are based on the consumer requirements affect the cost of the provided cloud service and consumer choices. Therefore, cloud resource units are discussed in the following section.

2.4.4 Cloud computing resources modelling

This section summarizes the IaaS cloud resource modelling discussed in Section (2.2.3.2). These resources, such as computing resources, are provisioned according to different specifications (e.g., CPU) and prices. Cost is an essential property that affects the choices of the decision makers. Therefore, cost is considered in the SLA agreements. Cloud providers offer the resources and their prices, which depend on various properties, such as: the payment method (see Section 2.2.2), resource location and duration of the resource usage. Therefore, we discuss cloud resource modelling as it might be considered in the cloud SLAs. As discussed in Section 2.2.3.2, the common IaaS resources are: Computing, Storage and Networking. In the following, we summarize the main resource properties that form a cloud resource unit and may affect the resource prices.

Computing resources consist mainly of: CPU speed, the number of cores, RAM, OS and Storage Capacity. Usually, the computing resource is a virtual machine, although cloud providers, such as GoGrid, provide a physical computing resource. The storage resource is defined by its size, which tends to have a minimum and a maximum size. The different types of storage are discussed in Section 2.2.3.2, which are block storage and object storage.

The Networking resource is defined by he size of its bandwidth usage and the direction of the data, which can be *into* the cloud, *out from* the cloud, or between different resources within the cloud.

Other properties affect the cloud resource, which are common to different kinds of cloud resources. The location of the resource, payment model and payment period affect the resource price, e.g. [3].

The QoS parameters, including the cost, are considered as a component of SLAs; therefore, in the context of specifying QoS and cloud unit resources terminology, we discuss the specifications of SLAs in the next section.

2.4.5 Digital SLAs

In this thesis, one of our objectives is to provide mechanisms for modelling cloud computing SLAs (see Section 1.6). Digital SLAs are used to automate the negotiation and monitoring of SLAs. In section 2.4.1 we discussed SLA selection and comparison based on QoS. The QoS parameter terminology plays an essential role in this selection and matching process. Therefore, in this section, we review the existing SLA specifications based on whether they specify the QoS parameter terminology. In addition to

QoS terminology, we investigate the IaaS cloud resource unit specification and terminology in order to calculate the total cost based on the requirements. In this section, we outline some of the existing SLA specifications.

WSLA is a framework for describing SLAs for web services created by IBM. It specifies the service measurements of the agreed services between the SLA parties, and is used for monitoring the agreement. WSLA provides a language for specifying and monitor an SLA, and in particular allows the SLA parties to specify measurement directives, which are special functions for composing aggregate metrics and predicates to evaluate specific metrics [130].

The specification of WSLA contains three main sections, as described in [30, 102] and illustrated in Figure 2.10:

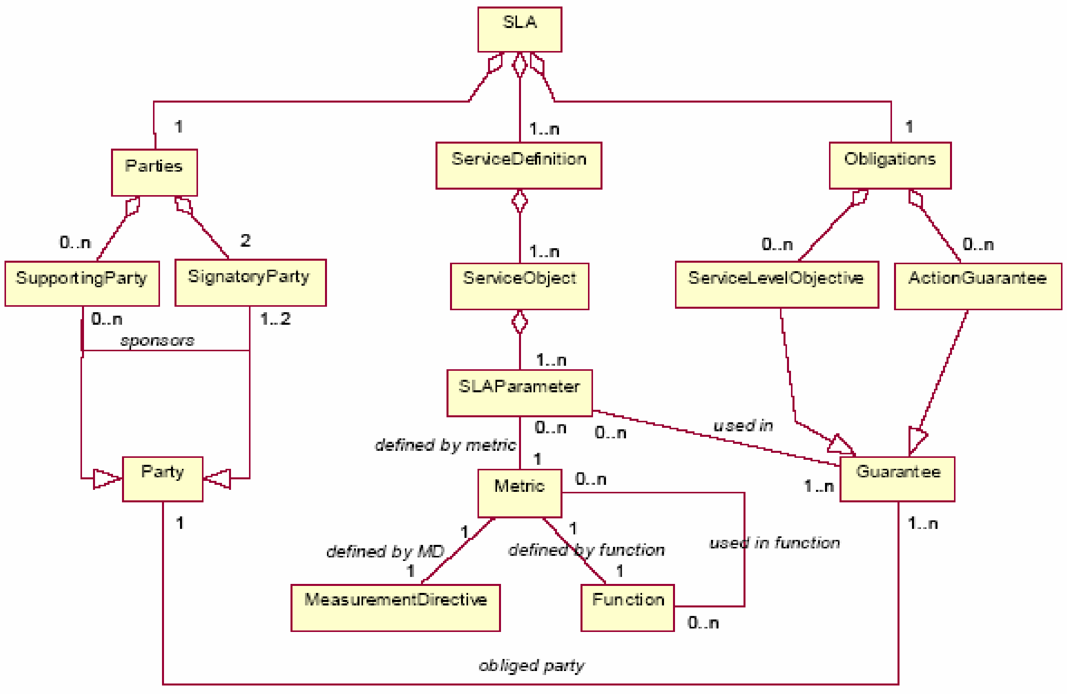


Figure 2.10: WSLA Structure [150].

1. Parties section: defines the *Signatory Party*, i.e. service provider, service consumer and the *Supporting Party*. The *party* captures the technical properties of the parties such as the contact information i.e. address.

2. Service definitions: describes a service and SLA parameters that are related to a service. This definition contains one or more *service objects*. Each *service object* defines the associates with one or more *SLAParameter*. Each *SLAParameter* has a *name*, *type* and *unit*, and is defined by a *metric*. The *SLAParameter* refers to one *Metric* that has a *MeasurementDirective* type (that defines how the metric value is measured by the provider), or special *Function* to compose aggregate metrics and predicates to evaluate specific metrics.

3. Obligations: specifies, by using logical conditions, the acceptable threshold of metrics, and describes the actions to be taken when a violation occurs. It describes the guarantees and constraints on SLA parameters, by using logical conditions. As shown in Figure 2.10, the *Obligation* defines the *ServiceLevelObjective* and *ActionGuarantee*. Both *ServiceLevelObjective* and *ActionGuarantee* refer to the *SLAParameter* defined in the service definitions.

The specifications of WSLA allow the parties to monitor the QoS parameters, which are separated from the contractual terms, to detect any violations. However, WSLA does not specify the terminology for the QoS parameters, as this terminology can be used, for example, in negotiations, where different parties may define different QoS and use different terminology for the same QoS which make it ambiguous to the SLA parties.

Another SLA specification isthe WS-Agreement (Web Service Agreement) which was developed by the Grid Resource Allocation and Agreement Protocol working group. WS-Agreement is an XML-based language that is used for the specification and negotiation of SLAs between parties [40]. The QoS parameters and their metrics are considered as domain-specific terms; thus it is not specified and makes it more flexible than WSLA. Therefore, to automate the monitoring of SLA agreements, the structures of QoS metrics, such as predicates, units and parameters, and QoS ontology concepts, were proposed in [144]. The WS-Agreement is composed of the main parts *name* (optional), *Context* and *Terms*, as illustrated in Figure 2.11.

The agreement *Context* contains information about the agreement parties and the duration of the agreement. The attributes of the *Context* are: *AgreementInitiator* (optional), *AgreementResponder* (optional), *ServiceProvider*, *ExpirationTime*, *TemplateID* (optional), and *TemplateName* optional. The *Terms* section is the core part

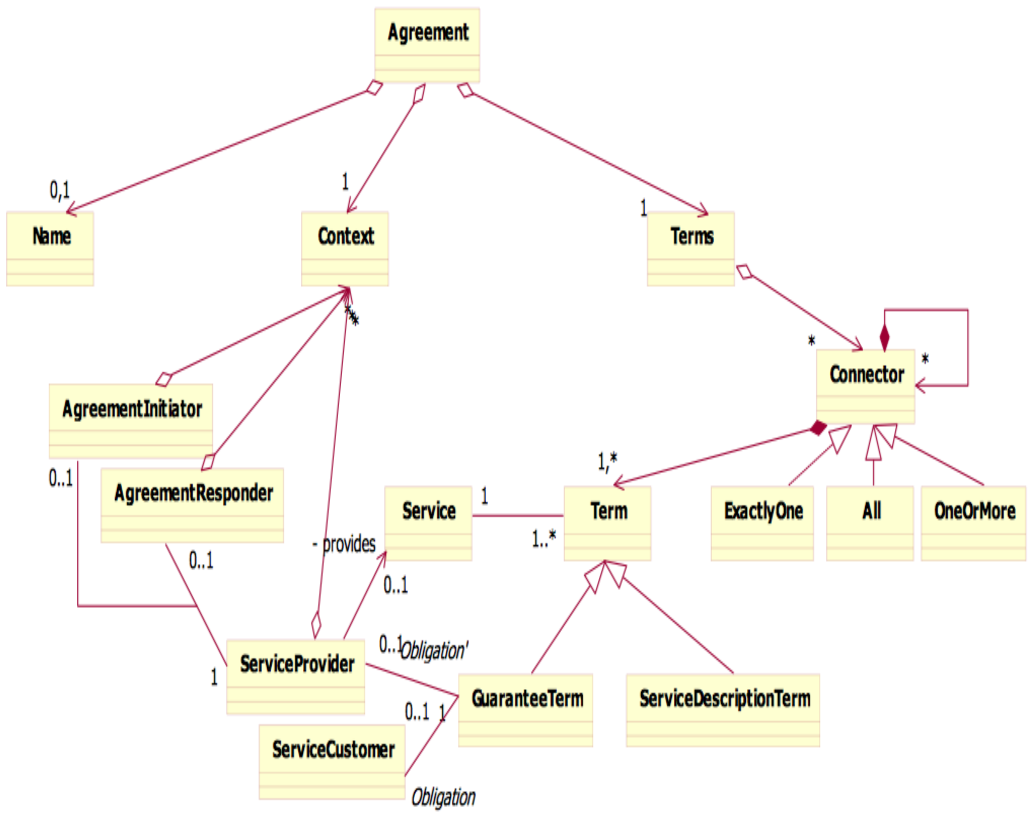


Figure 2.11: WS-Agreement Structure [123].

of the agreement offer. It defines one or more *Term*, where the terms are related using the term compositor: *All*, *OneOrMore* and *ExactlyOne*. Each *term* refers to a service. There are two types of terms, which specify the functional and non-functional properties:

- Service Description Terms: describe the functionality of the service. It consists of: a name of Service Description Terms, a name of the required or offered service and its functionality. It is a domain dependent on the description of the service functionality.
- Guarantee Terms: define the SLO and qualifying conditions for a service (i.e. described in the *Service Description Terms*) that can be fulfilled or violated. It

defines three states: *fulfilled*, *monitored* and *not determined*.

The assurance of the service quality (or availability) is associated with the service described in the *Service Description Terms*. Each agreement contains zero or more *Guarantee Terms*. *Guarantee Terms*, which consist of:

- *Obligated*: specifies the obliged party.
- *ServiceScope*: specifies the list of services to which the guarantee term applies.
- *QualifyingConditions*: specifies an optional precondition to be met to enforce a guarantee.
- *ServiceLevelObjective*: specifics the condition that must be met under which a guarantee holds.
- *BusinessValueList*: one or more business values, i.e. *Penalty* and *Reward* are associated with the *ServiceLevelObjective*.

Both WSLA and the WS-Agreement define the SLA using XML, while SLAng uses the Unified Modelling Language (UML) to model and define the language [164]. The SLA itself can be expressed in XML or Human-Usable Textual notations (HUTN). SLAng defines a vocabulary for web services. It defines the terms of the behaviour of the services and the clients who are involved in service usage. Corresponding to the service usage, SLAng defines six types of SLA (i.e. levels) that are classified into two classes: Vertical and Horizontal SLAs. The Vertical SLAs, which manages the interaction between two subordinate peers, are: Hosting, Persistence and Communication, while the Horizontal SLA, which manages the interaction between two coordinate peers, are: ASP (Application Service Provider), Container and Networking.

SLAng defines, for each type of SLA (i.e. Hosting, Persistence, etc.), a general structure including *Responsibility* for the *Client* of the service, the *Server* provider of the service, and *Mutual*, which is the responsibility of both the service client and service provider (see Figure 2.12). Each responsibility (i.e. client, server and mutual) defines a set of SLA parameters (such as availability, performance, maintenance, monitoring, security, backup and throughput) specific to the type of SLA.

Like the WS-Agreement specifications, the model is composed of:

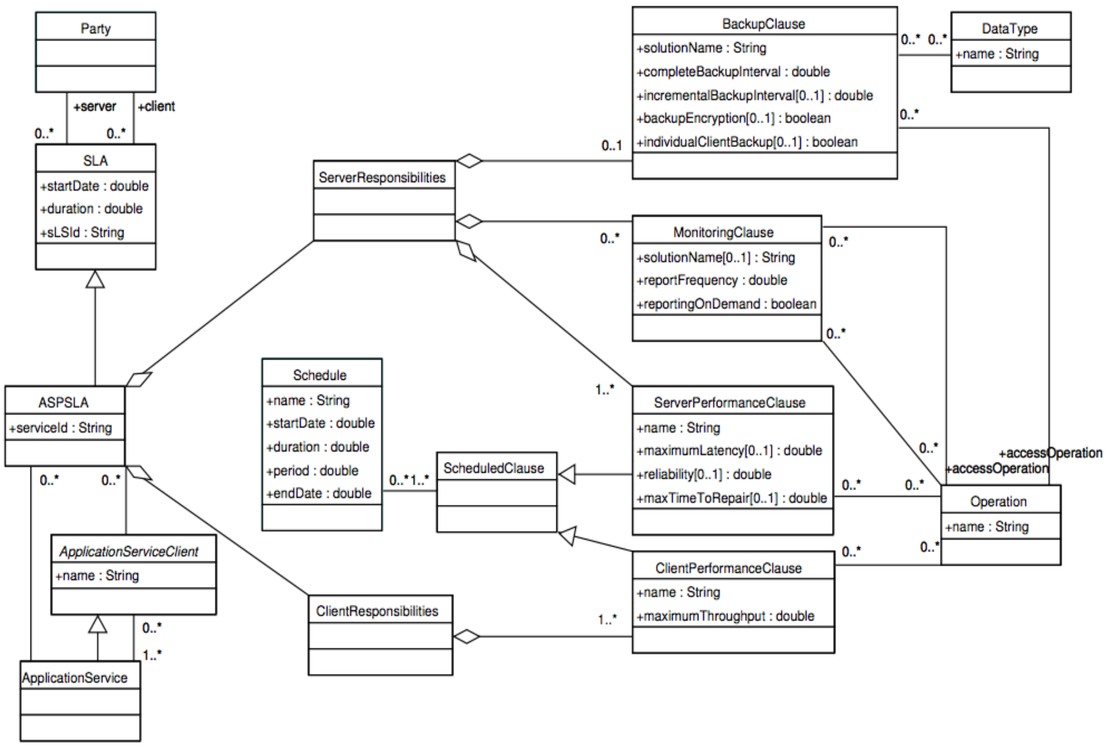


Figure 2.12: SLAng Structure [165].

- agreement parties
- offered services
- agreement terms which specify the QoS guarantees and the parties' obligations.

However, SLAng does not define the financial terms, such as costs, prices of the service and penalties in case of violation. Furthermore, the specification of this SLA is defined for electronic services with defining a limited set of QoS parameters. [38, 83] argued that more QoS parameters should be considered in the cloud SLAs (see Section 2.4.3) than are provided in this specification.

SLA* is an abstract syntax for defining digital SLA agreements [63, 101]. It was developed as a part of SLA@SOI, which is a project that provides solutions for the automation of SLA negotiation and management, which is initiated for the Service Oriented Infrastructure. It proposes a SLA(T), i.e. SLA Template, that defines the structure of the SLA. It specifics built-in QoS terms. The SLA* is inspired by the structure of the WS-Agreement, but differs from the WS-Agreement in the sense that it includes domain vocabulary and constraint language. The SLA is composed of the following main parts: *Domain Vocabularies*, *Party*, *Interface Declarations*, *Variable Declarations* and *Agreement Terms*. Many built-in QoS parameters' terminologies are defined in SLA* [63, 101], which are: throughput, MTTR, MTBF, availability, accessibility, arrival-rate, non-repudiation, isolation, regularity and completion-time.

SLA* defines the terminology for SLA (i.e. SLA vocabulary) which describes, for example, the SLA states (i.e. *agreed*, *terminated*, *violated*, etc.), the SLA parties and their role (i.e. *Party* and the role can be *provider* or *customer*), agreement terms, action and others. The goal of using these terms is to manage the SLA life cycle from initialization to termination.

All of the discussed SLA specifications define *party* section. The service description or definition and SLO, are presented in WSLA and the WS-Agreement. The metrics are defined in WSLA and SLA*, while QoS terminologies are specified in SLAng and SLA*.

Table 2.5 summarizes the differences between SLAs specifications.

Table 2.5: Comparison of SLAs specifications

SLA Specification	QoS terminology	Cloud resource units terminology	Metrics	Business value (e.g. cost and penalties)
WSLA	NA	NA	specifies measurement directives to compose metrics	specifies action in case of violation
WS-Agreement	NA	NA	does not specify metrics and is considered as domain-specific terms	specifies the terms for penalties and rewards
SLAng	specify limited set of QoS terminology	NA	does not provides metrics but is based on the behaviour of the parties	NA
SLA*	Specifies a built-in QoS terminology	NA	yes	specifies action terms in case of violation and payment terms

Unlike the research presented in this thesis, the above studies are not defined specifically for cloud computing, so we may need to consider different QoS properties or cloud resources (see Sections 2.2.3.2 and 2.4.3).

2.5 Summary of Cloud Computing SLA Modelling and Comparison issues

In section 2.4, we identified the following categories of related work: (1) Digital SLAs, (2) Selecting a cloud offer from different cloud provider offers, (3) Matching cloud computing requirements and offers, (4) The QoS properties and models to be considered in the cloud SLA, and (5) The cloud resources and their models that can be considered in the cloud. In the this section, we summarise some of the issues related to the modelling and comparison of cloud SLAs.

- Cloud computing providers define their SLAs, using their own terminology that is dependent on cloud technology (see Section 2.2.4). These SLAs are defined largely in natural language. Furthermore cloud providers can update these SLAs and the QoS, e.g., the last version of EC2 SLA was updated in 2013 with the monthly uptime percentage as a commitment, while the older version was from 2008, with the annual uptime percentage [4]. The comparison of two SLAs, defined in natural language, which may be structured differently and use different terms, is difficult to automate. Therefore, structured SLAs were proposed to facilitate the automation of SLA management and comparison.
- As described in Section 2.4.5, the widely mentioned WSLA and WS-Agreement do not specify QoS terminology, which are used for comparing cloud SLAs. The SLAng language fails to specify financial aspects, such as costs and penalties, which is an important property in the consumers' decision-making process. The SLA* is a service-oriented SLA, which does not specify the service and resource units terms, which is needed to compare SLAs for the same service.
- As noted in Section 2.4.3, there is a lack of agreement about the QoS properties and terminologies to be specified for the IaaS cloud.
- In selecting cloud computing approaches (see Section 2.4.1), where MCDM were proposed, some of the studies, which are reviewed in Section (2.4.1), failed to define how the requirements and offers were captured, while others failed to define how these data were filtered to form the MCDM problem.

- In matching cloud computing, the studies based on the SLA mapping approaches failed to consider supporting the consumer decision-making by providing MCDM, which allows consumers to add their preferences regarding the various QoS parameters and constraints on the QoS values. MCDM considers the difference between the offered QoS values and the required QoS values, while matching approaches in Section 2.4.2 do not consider the difference between values of QoS properties.

- The approaches discussed in Section 2.4.1 match the QoS properties. Cost is an essential property in the decision-making process [83, 183]. The cloud providers offer their resource units and prices separately from the SLA. To include the cost and as part of an automated comparison, the specification of the resource unit is required to match it with the offered cloud resource unit and then calculate the cost. None of the digital SLAs (which are presented in Section 2.4.5) specify the terminology for IaaS clouds' resource units. Furthermore, based on the matching terminology, there will be more than one match found when the terminology of the required resource unit matches that of the offered resource units. For example, the required resource is defined as follows: VM with (CPU core = 1, RAM = 4.75 GB and storage size = 32 GB). The offered resource units in a cloud provider are defined as follows: VM1 with (CPU core = 1, RAM = 3.75 GB and storage size = 20 GB) and VM2 with (CPU core = 2 , RAM = 7 GB and storage size = 40 GB). The matching that is based on terminology matching and the structure of the VM resource returns two matches. In this case, a further step might be required in order to select one of the offered VMs. In general, for the cloud provider and consumer SLAs, there is no single SLA specification that specifies the QoS terminology and cloud resource units that can be used to select an offer from a range of different offers in an automated fashion.

- In general, there is no single definition suitable for cloud SLA comparison and selection. Therefore, we need a *framework*, including a customisable and user-configurable technique that can be used to define a project or domain-specific comparison of SLAs.

This thesis focuses on the comparison of cloud SLAs. The hypothesis and objectives are presented in Chapter 1. This approach proposes metamodels to allow cloud

SLAs to be precisely captured using models, and to automate the comparison process by two phases. The first phase involves matching the required SLA with the offered SLA. The second phase entails supporting the consumers' decision-making by the selection one offer form a range of different offers using MCDM.

2.6 Chapter Summary

In this chapter, we described the SLA concepts and specifications. An SLA forms part of the legal contract between the service consumer and service provider. SLA specifications and electronic SLAs have been introduced. The main components of SLAs are the QoS parameters and metrics. Different services, service providers and consumer requirements require different QoS parameters. Therefore, several studies focused on service/provider selection which are based on SLA and QoS. Various researches have also proposed QoS ontologies to create precise definitions and facilitate auto-selection of services.

This chapter also described how cloud computing technology provides different types of services and how the dynamic nature of clouds requires continuous monitoring to ensure the required/agreed service level. Different studies defining different QoS parameters and models have been described. A comparison of public clouds that provide various SLAs with different terminologies, has been discussed. There is a lack of standard specifications for cloud SLAs and the QoS terminologies.

We described the MDE principles and tools and the intersection between cloud computing and MDE. We then described the related works on the selection of a cloud offer from a a range of different offers. This selection is a post process for matching consumer requirements with a provider's offer. We discussed various matching approaches, many of which are based on terminology matching. Therefore, we presented the QoS parameters and digital SLAs, as they are used as input for the matching process. After that, we discussed the issues associated with the related work, which are summarized in Section 2.5.

The next chapter (Chapter 3) explores, analyses and discusses the ways to approach thesis' hypothesis and objectives, as proposed in Chapter 1.

Chapter 3

Problem Analysis

3.1 Introduction

Chapter 2 introduced the concept of an SLA, which is part of a contract between different parties, usually a service provider and a serviceconsumer. An SLA consists of the definition of services and QoS parameters associated with those services (Sections 2.1.1 and 2.4.3). SLAs define service level objectives which associate qualities with threshold values in cases where violation penalties may apply (Sections 2.1,2.1.1). An SLA plays a fundamental role in web services and SOA as it is used as a basic means for negotiation as well as for monitoring the quality of the service. SLAs are often treated as specifications to be manipulated by computers. As a result, many studies have been carried out that discuss the specifications, automated processing and management of an SLA (Chapter 2).

Cloud computing was introduced in Chapter 2. We also discussed cloud SLAs and their impreciseness and differences in terminology (Section 2.2.4). A challenge with this is that there is, as yet, no standard vocabulary, metamodel or ontology for cloud SLAs.

A number of SLA languages have been proposed (see Section 2.4.5). These SLA languages are not specific to the cloud computing domain. They were designed to: help to monitor QoS parameters by providing measurement directives and special functions to compose aggregate metrics [102], specify the negotiation of SLA between different parties, and automate the SLA negotiation and management process [26, 40].

Cloud selection and comparison processes have been reviewed in Section 2.4. Matching process is performed based on finding matches between two SLAs (see Section 2.4.1), while the selection process is based on selecting an offer between various offers that fits the consumer requirements (see Section 2.4.2). The matching process is founded on matching QoS terms and structures, whereas the selection process is founded on calculations that involve values of QoS terms and parameters.

Our approach in this work is to utilise MDE principles and technologies to achieve the objectives mentioned in Section 1.6. Our goal is to provide mechanisms for modelling cloud SLAs, based on MDE principles, techniques and tools and then automatically or semi-automatically to compare and select cloud SLAs.

This chapter is concerned with eliciting and specifying the detailed plans and requirements of a solution to the problem identified in Chapter 1. We use a standard Requirement Engineering (RE) process to develop a general understanding of the problem domain. An RE process is:

> “ a coordinated set of activities for exploring, evaluating, documenting, consolidating and changing the objectives, functionalities, assumptions, qualities and constraints that the system-to-be should meet based on the opportunities and capabilities provided by new technologies” [185].

An RE process includes developing a sufficient understanding of the problem and its domain. It also includes defining the objectives to be satisfied, the functional services, constraints and assumptions, and assigning responsibilities among different components. There are multiple artefacts and techniques in RE, such as scenarios [185] and use cases [112], which are used in this chapter. Use cases are used to identify the requirements, and the behaviour of the use case can be described by *scenarios* [171]. To explore the steps of our plan that will shape the solution to the problem explained in Chapter 1, we provide scenarios and use cases in this chapter. Before that, we start by introducing a motivating example to explain the problem and the goals to be achieved in order to solve it. By defining the use case and scenarios, we can use them to capture functional and non-functional requirements for semi-automating cloud SLA comparison (Section 3.3). Then, a list of requirements for satisfying the objectives is described in Section 3.4.

3.2 Motivating Example

Assume that an organisation wants to use a public cloud provider and is looking for VM and storage services. This organisation has business goals that it anticipates to achieve by migrating to the cloud; hence, it prepares and defines its required services and qualities, e.g. the required VM(s) and their performance, the required storage, the availability threshold for the VM and availability for the storage service, and their total budget. The organisation then looks at offers from the available service providers.

Suppose that an organisation wants to a find public cloud provider that matches their needs and starts to compare different cloud offers. This comparison can be based on different criteria, e.g. service functionality, price and QoS. Usually, the parameters of the QoS are defined in an SLA (Section 2.1.1). However, there are many public clouds that provide VM and storage as services. The increasing number of cloud providers, the variation in their service details (e.g. one cloud provider may provide VM and dedicated servers while another only VM and Messaging Services), different qualities of service and the varying pricing models makes it difficult for the organisation to compare different cloud services or providers in terms of the organisation's needs. The comparison process starts with the organisation trying to find cloud providers who offer the required service(s) that meet the organisation's needs, which narrows the search space. Then the organisation selects the cloud that *best* matches the organisation's needs. The wealth of cloud services with different QoS parameters makes the process of selection a difficult task; [83, 183] argue that the problem of selecting cloud computing is an MCDM problem. In such a problem, a decision-maker has multiple criteria with multiple alternatives and different preferences. The decision-maker has to identify and evaluate the criteria based on these preferences [201].

The problem is: how do we systematically compare cloud SLAs, thus helping the organisation move into cloud computing? How can we systematically narrow the search space? Can the organisation automatically or semi-automatically select the service provider that best matches its business needs? To determine how these questions can be answered, we first precisely identify what the goals are and what the solution to the problem should achieve. This is discussed in the next section.

Before defining the goals for this motivating example, in the next section, we continue with the example to identify the parties involved. In RE, it is essential to understand who is involved to achieve the objectives of the problem and what are their responsibilities are [185]. The main party in this example is the *service requester*. The *service requester* is a party who wants to match their needs against SLA offers. The *service requester* can be a cloud consumer or cloud provider. In our example, the cloud consumer (i.e the organisation) is the *service requester*. The other party is the cloud provider who provides the pre-defined SLAs that will be compared against the needs of the service requester.

3.2.1 Goals to be Satisfied to Solve the Problem

This section describes the general goals that are to be achieved to address the problem presented in Chapter 1: can we systematically help an organisation (cloud consumer) to make decisions about different cloud computing offers by comparing cloud SLAs? In web service studies, automated approaches have been proposed to address such a problem [42, 125]. In some studies [94, 125] the automation of web service selection is based on QoS. Current cloud providers provide their pre-defined SLA with some of the QoS parameters, such as availability and uptime, e.g. Amazon EC2 [4] (Section 2.2.4.1). Thus, as in web service studies, we identify the following goals:

- *Language for SLA*: to facilitate the communication between the service requester and the service providers, where an organisation uses this language to specify their needs.

- *Automation of Comparison*: to narrow the search space for the comparison process among a wide range of providers, taking into account the different criteria for matching.

- *Automation of Selection*: to help the organisation to select the best provider offers that match their needs.

3.3 Scenario and Use Cases

In this section, we describe the comparison scenarios and the actors involved. We assume that there is a repository of SLAs from different cloud providers. We assume that this repository can be implemented as models conforming to a common SLA metamodel; a metamodel is presented later in Chapter 4. By having a standard language (metamodel) and editing tools to create models, a repository of cloud SLA models can be implemented. A service requester can be any consumer that wants to compare and find cloud services that match their needs. The general scenario is illustrated in Figure 3.1 and use case 3.1. The template for expressing the use cases in this chapter is adopted from [112].

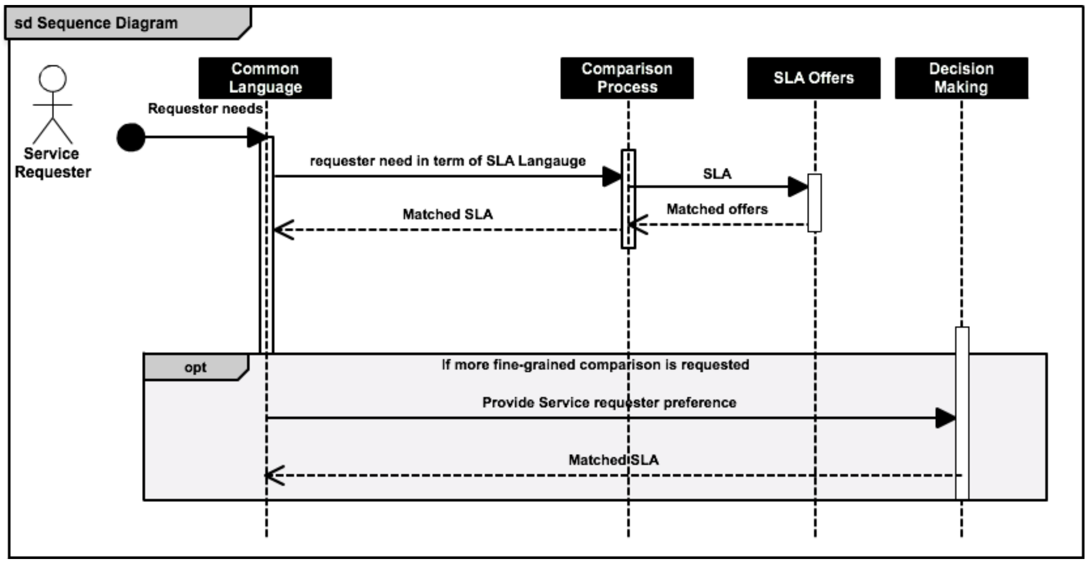


Figure 3.1: Comparison Scenario.

Figure 3.1 shows that the service requesters need to provide their demands. The requester provides their needs using common terminologies (in an SLA language, Section 2.4.5). A set of providers SLA offers are required (e.g. stored in a repository) in order to compare them with the service requester's demands. A comparison process is required to compare the service requester's demands and providers' SLA offers (e.g., in the repository), which generates any match between the requester needs and the offers. To this end, a smaller set of provider offers that match the requester's needs can then be post-processed. This post-processing is used to achieve more fine-grained match requirements, which can help the service requester to select a better offer.

Use case 3.1 illustrates more details of the comparison process. For SLA comparison, the service requester (usually the cloud consumer) provides their needs (SLA). Then a comparison process is invoked to calculate the matches between the cloud

Use Case 3.1: SLA Comparison, Use Case template is adopted from [112]

Use Case Name:	SLA Comparison
Summary:	System Context use case. A cloud consumer provides their requirements. The system then finds the matching offers of the cloud providers.
Basic Course of Events:	1. A cloud consumer provides their needs with regard to cloud services and SLO/QoS in terms of an SLA. 2. The system finds the matching elements of the cloud consumers' needs against different cloud providers' offers stored in the repository. 3. The system responds by providing a set of offers in the repository that matches the cloud consumer's needs.
Assumptions:	Cloud provider SLA offers stored in a repository
Post Condition:	1. Cloud consumer provides preferences on their needs. 2. The system responds by providing a further processing based on the consumers preferences to select a better offer.

consumer's needs]and the cloud provider's offers. This comparison process provides matching offers to cloud consumers. A cloud consumer may request further processing, e.g. to find more fine-grained matches, which is described in this use case as a post condition.

In the next sections (3.3.1 and 3.3.2), we provide more detailed use cases covering alternative cases. The purpose of these use cases is to show the interactions with different actors and more details of the comparison alternative scenarios. The alternative scenarios provide different options for the actors.

3.3.1 Consumer-Providers/Provider-Providers

This section describes use cases to show the interactions between the service requester and the comparison process. For example, can a comparison process compare an SLA offer of a provider against other SLA offers stored in the repository, or only compare the consumer needs (SLA) with the SLAs in the repository? The goal of this case is to help providers to understand how they differ from the competition. Some cloud providers, indeed, provide a comparison of their service offers with other providers' offers (i.e. Amazon AWS) [24, 25]. To this end, we define two use cases to show the interactions: first, the interaction with the service requester as a cloud consumer provides their needs in term of an SLA. The second use case interaction is when a service requester compares one of the SLA offers against another SLA in the repository.

Use Case 3.2: Consumer-Provider Comparison

Use Case Name:	Consumer Provider Comparison
Summary:	System Use Case Context. This use case explains that cloud consumer needs are used in the comparison process.
Basic Course of Events:	1. Service consumer provides their needs in terms of SLA. 2. The matching process is invoked. 3. The matching process returns providers' SLA offers that match consumer needs.
Assumptions:	Cloud provider SLAs are stored in a repository

The aim of the *consumer-provider* comparison (Use Case 3.2) is to help the cloud consumer (e.g organisation) to select the cloud SLA offers that match the consumers' SLA. This may also help the cloud provider to compare their offers with other competitors in order to improve their services. Another possibility is that the service requester wants to compare one of the SLAs in the repository against other SLAs in the repository; for example, a service provider may want to compare their offers against other provider offers. The two use cases are similar; the differences are the provided SLA, whether it is a consumer's SLA or one of the provider offers stored in the repository.

Use Case 3.3: Provider Comparison

Use Case Name:	Providers Comparison
Summary:	System Use Case Context. This use case specifies that the cloud provider offer is compared with other SLA offers that are stored in the repository.
Basic Course of Events:	1. A service requester provides an SLA offer of a cloud provider using the provider to be compared with other SLA offers. 2. The matching process is invoked. 3. The matching process returns providers SLA offers that matches the service requester's input.
Assumptions:	Cloud provider SLAs are stored in a repository

The reason for having two use cases is as in the case of Consumer-Provider Comparison use case (Use Case 3.2), we have to take into account the variations between providers' SLAs and imprecise and incomplete consumer's requirements [68].

3.3.2 Comparison of Alternatives Scenarios

This section illustrates the comparison of alternative scenarios that were introduced in Use Case 3.1 (Section 3.3). These scenarios capture additional details about the alternative requirements for comparison: for example, matches could be in terms of different properties or a different match degree. A number of options exist:

- compare the SLAs in terms of an *optimal match* of their QoS values, e.g., their service uptime, their service performance [134, 176].
- compare the SLAs in terms of an *approximate match* of their QoS values; for example, two SLAs may be considered approximate if their service uptimes are within a certain delta of each other [69, 140].
- compare the SLAs in terms of a domain-specific similarity measure based on their service-level objectives; for example, two SLAs may be *name-based* simi-

Use Case 3.4: Comparison Alternative Scenario

Use Case Name:	Matching Scenarios
Summary:	This use case explains that the service requester has different options for matching scenarios.
Pre–condition:	An SLA to be matched against other SLAs in the repository (input) is provided.
Basic Course of Events:	1. Service requester provides the required matching scenario. 2. The matching scenario is invoked. 3. The matching scenario responds by returning the SLA offers that match the service requester input.
Alternative Paths:	In step 1, the service requesters have several options of the match scenario. They can choose from: 1. *Name-based* match scenario (Section 3.3.2.1). 2. *Optimal* match scenario (Section 3.3.2.2). 3. *Approximate* match scenario (Section 3.3.2.3).
Assumptions:	Cloud provider SLA offers stored in a repository.

lar if they are structurally similar (i.e., defined using similar language) and, based on feedback from a domain expert, their service-level objectives are similar.

As discussed in Section 2.4.4, in order to include the cost of the cloud offer to be used in the decision-making problem (i.e. selection process), a matching of the required cloud resource units and offered cloud resource units should be performed first. By doing matching terminologies, the matching process returns the resource unit that define the same resources, e.g., a required VM can be matched with any offered VM, without taking into account the different capacity between them. However, to include the values, e.g. VM capacity, may require adding some constraints to the matching process. For example, consumer requires a VM with price that does not exceed $X/month$.

Therefore, sections 1.2 and 2.4 discussed different comparison approaches in cloud computing. In general, there is no single definition suitable for SLA comparison, and that, more concretely, we need a *framework*, including a customisable and user-configurable technique that can be used to define a project or domain-specific comparison for SLAs. We also argue that such a comparison needs to take into account consumer preferences, which allows the consumer to decide which characteristics of an SLA are to be emphasised in the comparison process.

3.3.2.1 Name-based Matching

This scenario describes the first alternative in Use Case 3.4. Cloud SLA In this comparison scenario, we find the matched elements, regardless of their values. For example, if a consumer defines a VM, then this scenario will return all VMs that are offered by the provider regardless of, e.g., number of cores and RAM size. This comparison scenario is provided if the service requester provides abstract definitions of QoS (i.e. they are incomplete and imprecise) [193]. To explain this, assume a service requester is able to define preferences for a certain QoS, e.g. higher availability (without defining the value) is more important than lower price. The service requester does not define any desired values of the required availability. In this case the comparison scenario matches all offers that define availability regardless of their values.

This matching logic is used in the later scenarios Sections (3.3.2.2 and 3.3.2.3) implicitly. However, we provide it as a stand-alone use case that can be used when the *values* of, e.g., the QoS parameters are not the main concern of the service requester.

3.3.2.2 Optimal Matching

In the previous comparison scenario (Section 3.3.2.1), the main focus was on matching elements, regardless of their values. In this scenario, which is the second alternative in Use Case 3.4, the focus is on the required and provided *value(s)* of the QoS parameters. Assume a service requester is trying to find a specific cloud service with, e.g., specific performance values or prices. This comparison scenario finds the exact or better *values* of two elements. This is the case when a service requester wants to find at least the required values, e.g., an *availability* threshold value. The *values* in the QoS parameters could be a Boolean, a single numeric, a range numeric or a string [83].

3.3.2.3 Approximate Value Matching

In the *optimal match* scenario in Section 3.3.2.2, consumers may be too strict in matching the QoS parameters, where there is a risk of failing to find services that match their requirements when using certain values [69]. Therefore another possible scenario arises, whereby consumers relax their constraint on the matching, and thus try to match approximate values for the QoS parameters; this is the third alternative in the Use Case 3.4. For example, two SLAs elements may be considered approximate if their service uptimes are within a certain delta of each other. Another example, is when two cloud units are matched while their prices do not match (e.g. 0.9$, 0.8$). An approximate match defined with delta=0.1, matches their prices.

3.4 Steps of the Research Plan

In the previous sections, we described the use cases and scenarios. We used use cases to analyse the problem at hand, and to establish a more detailed research plan, which addresses the thesis objectives. We describe this research plan further in this section.

Once again, the objective of this research is to model cloud SLAs and semi-automatically compare cloud SLAs based on MDE principles, techniques and tools (Section 1.6). Therefore, we link these steps to the objectives of this thesis. To do so, we first discuss the steps that satisfy the goals stated in Section 3.2.1, and link them with the use cases provided in this chapter. Then, we identify the MDE-based steps.

From previous scenarios and use cases (Section 3.3), we can identify a set of steps in order to satisfy the goals discussed in Section 3.2.1. Table 3.5 summarises the identified steps.

The first step is that the framework should provide means for the SLA specification which can be used by a service requester to describe their needs. Also, the SLA offerings of the cloud provider (which are stored in a repository) are to be defined using similar language. We assume that, in all use cases, the framework should have a repository to store the SLA offers. One of the main objectives is the comparison algorithm. Different comparison algorithms that support the uses cases in Section 3.3 should be supported. These are the general steps to (semi)-automate an SLA comparison process.

Table 3.5: Platform independent steps of a research plan

#	Step	Applicable use case
S1	The framework shall provide an SLA language for the service requester	• Use Case Consumer-Provider Comparison 3.2.
S2	The framework shall provide a language for cloud offers	• Use Case Provider Comparison 3.3.
S3	The framework shall have a repository to store cloud offers	• Use Case Consumer-Provider Comparison 3.2. • Use Case Provider Comparison 3.3. • Use Case SLA Comparison 3.1. • Use Case Comparison Alternative Scenario 3.4.
S4	The framework shall provide comparison algorithms for different comparison scenarios	• Use Case SLA Comparison 3.1. • Use Case Comparison Alternative Scenario 3.4.
S5	The framework shall allow the service requester to provide their preferences (e.g. weighting)	• Use Case SLA Comparison 3.1.
S6	The framework shall support more fine-grained match requirements	• Use Case SLA Comparison 3.1.
S7	The framework shall provide the results of the comparison scenarios	• Use Case SLA Comparison 3.1. • Use Case Comparison Alternative Scenario 3.4.

One of the objectives of the thesis is to provide mechanisms for automatically or semi-automatically comparing cloud SLAs, based on MDE principles, techniques and tools. The comparison process will compare the different cloud provider SLA offers against cloud consumer SLAs. This requires the SLA models for both cloud providers and cloud consumers to be defined; these are explained in the following required steps.

Sp1 The input to the framework shall be a cloud consumer SLA model. The SLA model is used by the comparison scenarios (Section 3.3).

Sp2 The framework shall have a repository of SLA models of cloud provider offers. This work assumes a repository containing a set of SLA models from different cloud providers. These models conform to a cloud provider SLA metamodel.

Sp3 The framework shall define a cloud SLA Metamodel(s) which can be used to produce SLA models of both cloud providers and cloud consumers. Using a common language (metamodel) facilitates the communication and matching the elements from both the required and offered SLAs.

A metamodel defines the abstract syntax of the SLA, e.g, it defines the main classes of the SLA, their attributes, constraints and rules to form consumer SLA models. Therefore, an SLA metamodel is a means for cloud consumers and cloud providers to define and specify their SLA.

Sp4 The framework shall provide different comparison algorithms. These algorithms are applied to the comparison alternatives in use case 3.4 (Section 3.3.2).

Sp5 The framework could provide a cloud consumer preference model. The model is used to provide a more fine-grained comparison (e.g weighting elements in the SLA model). Many QoS-based selection studies provide a weighting schema. Each element is given a weight by the consumer to show an importance or priority of this element among other elements for the consumer.

Sp6 The framework shall provide an output model of the comparison process. As discussed in Section 2.3.3.2, the result of the model comparison should be represented in some form. These results can be a subject of further processing.

Sp7 The framework could provide a method for visualizing the outcome of the comparison process. The outcome of the matching algorithm needs to be visualizable in human-readable forms, which allows the consumers and providers to realise and analyse the differences between the models.

Table 3.6 specifies the required steps that are MDE-dependent. The table links those steps with the objectives of this research (Section 1.6). The achievements of the steps (Table 3.6) will be assessed in terms of completeness after considering the case studies. A methodology for comparing cloud SLAs based on these steps is developed in the following section.

Table 3.6: MDE-based steps related to the thesis objectives (Chapter 1) and the platform-dependent steps (Table 3.5)

#	Step	Applies to thesis objective	Applies to platform-dependant Requirements
Sp1	Cloud Consumer SLA Model	Objective # 1 (Section 1.6) modelling cloud SLAs based on MDE principles	**S1**
Sp2	Cloud Provider SLA Model	Objective # 1 (Section 1.6) modelling cloud SLAs based on MDE principles	**S2**
Sp3	Cloud SLA Metamodel	Objective # 1 (Section 1.6) modelling cloud SLAs based on MDE principles	**S1, S2**
Sp4	Model Comparison and different matching algorithms	Objective # 2 (Section 1.6) automatically or semi-automatically comparing cloud SLAs, based on MDE principles	**S4**
Sp5	Cloud consumer Preference Model	Objective # 2 (Section 1.6) automatically or semi-automatically comparing cloud SLAs, based on MDE principles	Service requester preferences (e.g. Weighting scheme) (**S5**)
Sp6	Matching Results Models and Metamodel	Objective # 4 (Section 1.6) presenting the results of comparing cloud SLAs in machine processable forms	**S6**
Sp7	An MDE-based method for results visualisation		Provide results of the comparison algorithms (**S7**)

So as to be able to use cloud SLAs for both consumers and providers - as discussed in Section 2.4 - when constructing a cloud SLA we should capture the following information:

- The QoS terms that will be used in both matching and selection.
- The service types of the cloud SLA and cloud resource terminologies, structure and prices to be compared and matched.
- The weights that will be assigned by consumers to provide the preference model.
- A different model will be needed to further process the output results of the model comparison process (i.e. model for MCDM).

3.5 An approach for Cloud SLAs Comparison

This section describes our approach to comparing cloud SLAs to help consumers to select a cloud SLA offer using MDE principles. The purpose of this section is to provide an overview of our proposed approach before going into the details in the following chapters. As we described in Section 3.4, metamodels for cloud SLAs are one of the requirements for this approach. Models that conform to this metamodel can be compared. This is illustrated in Figure 3.2. Cloud SLA models of providers and consumers conform to our proposed metamodels, which is described in Chapter 4. The conformant SLA models are involved in a comparison process. This comparison process consists of a number of matching processes, whereby each process matches two models. This comparison is described in Chapter 5. The outcome of the matching consumer model with a provider model is a model. Having a number of providers to be matched with a consumer, the result will be a number of matched models. Different models' management tasks, such as model transformation, can be performed to analyse the results, which may need human interaction at certain points. The different outcome models can be transformed into a decision-making matrix. This is presented in Chapter 6.

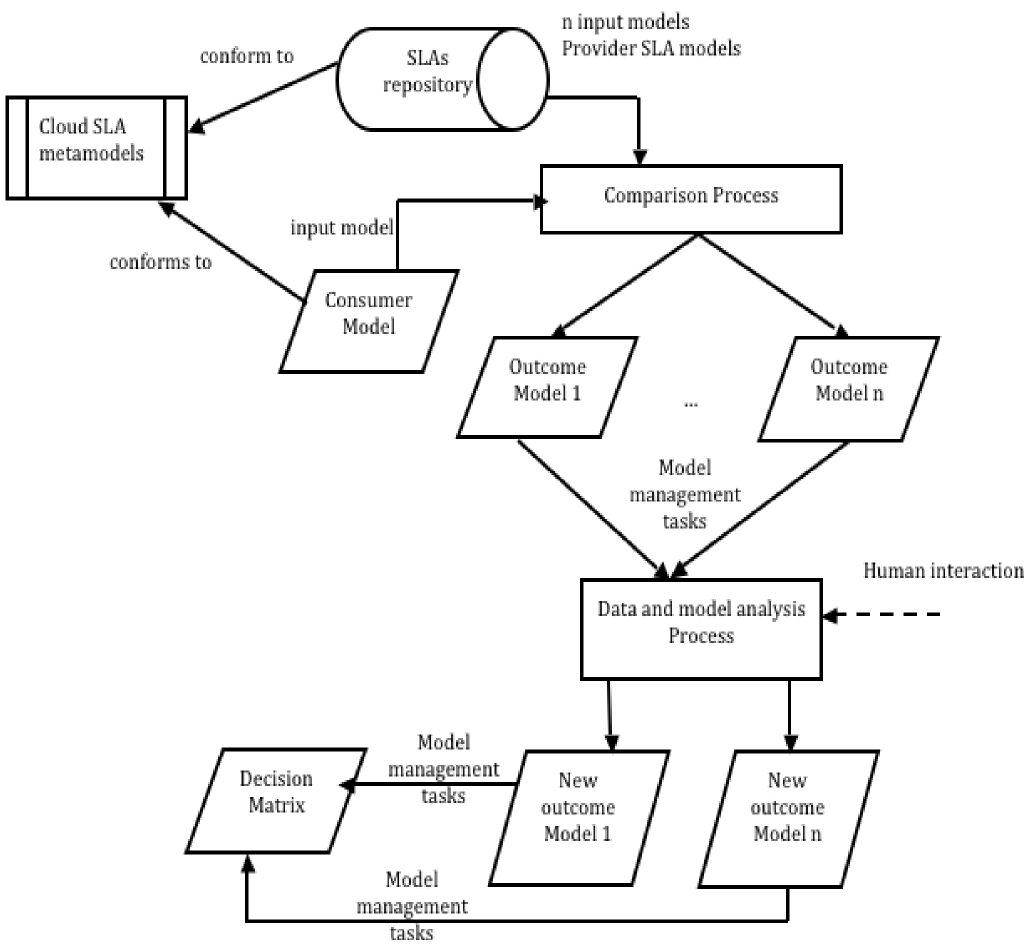


Figure 3.2: Our proposed approach for cloud SLA models comparison

Chapter 4

Domain Analysis and Metamodelling

Chapter 3 introduced use cases and scenarios for cloud SLA comparison, and specified the requirements for addressing the research objectives described earlier in Section 1.6. To compare cloud SLAs based on MDE tools and principles, consumer and provider models are required. As discussed earlier in Section 2.3, in MDE, a model conforms to a metamodel. Thus, a metamodel is specified as one of the goals in Table 3.6 in Section 3.4. As discussed in Section 3.4, we constructed two metamodels for cloud SLAs: one for cloud providers and the other for cloud consumers. However, we ended up with one cloud SLA metamodel for both cloud providers and consumers, which is discussed in Section 4.2.

This chapter presents not only the metamodel for describing cloud SLAs, but also the process by which it was developed, including the different versions of the metamodel that have been elaborated. This chapter starts with an introductory section. Then, Section 4.3 presents the construction process and describes the cloud SLA metamodel used in our proof-of-concept tool support. Then Section 4.2 presents the early versions of the cloud SLA metamodel.

4.1 Introduction

This study explores the research hypothesis (described in Section 1.5) on how MDE can help consumers in cloud computing to compare different cloud providers' SLAs. In Chapter 3 we specified the goals needed to address the thesis' objectives. A cloud

SLA metamodel, a cloud provider SLA offering model and a cloud consumer SLA requirements model are needed to address objectives 1-3 (Section 1.6). This chapter describes the cloud SLA metamodel and how it has been constructed.

To construct an SLA metamodel for cloud computing, we analysed the cloud computing domain (Sections 2.2.4, 2.2.3.2). As mentioned earlier (Section 2.2.3.2), the domain of clouds can be categorised into IaaS, PaaS and SaaS. Different kind of services provided by cloud computing may have different QoS, and thus different QoS parameters might be required to be specified for each service [38]. In this thesis, our focus will be on IaaS cloud computing as a proof of concept. Other cloud services, i.e. PaaS and SaaS, may potentially extend this metamodel by following the example process described in this chapter.

In constructing a cloud SLA metamodel, we studied the pre-defined public IaaS cloud SLAs. These pre-defined cloud SLAs structures, are in general, similar to other SLAs; they consist of: service definitions, service guarantee, credit in case of violation and credit request (Section 2.1). Examples of pre-defined cloud SLAs are: AWS EC2 [3] and S3 [5], RackSpace SLA [21] and GoGrid SLA [15]. A cloud consumer, as in web services [94, 121, 162, 176, 194], may use the QoS as a criterion for selecting better cloud services.

We identified the most common QoS parameters in cloud computing SLAs offerings. In public cloud SLA offers, availability is the most common QoS concept, while cloud computing studies, such as [38, 81, 83], present other QoS concepts that should be considered in the cloud computing SLA (Section 2.4.3). For example, Amazon EC2 and S3 SLAs define “Monthly Uptime Percentage”, RackSpace defines “Monthly Availability” and GoGrid defines “Server Uptime” (Section 2.2.4.1). We started constructing the cloud SLA metamodel with the common QoS *availability*. In constructing a cloud SLA metamodel, we provide *Availability* and *Maintainability* as QoS concepts as a proof of concept. Other QoS concepts were discussed in Section 2.4.3 by following the example of the QoS concepts described in this metamodel. The next section provides a detailed description of the development of the cloud SLA metamodel.

4.2 Constructing the cloud SLA metamodel

We construct a cloud SLA metamodel iteratively: we first analyse the domain of cloud computing and cloud SLAs; construct a metamodel; run small examples using a comparison process (testing); and extend and refine the metamodel. The cloud computing domain is a large domain with a number of concepts. Therefore, we constructed a metamodel incrementally, i.e. we started constructing a metamodel for one of the concepts. There is a possibility that a model that conforms to a metamodel may not correctly capture the concepts for a specific purpose. Thus, we started with a small concept and generating models. These models are evaluated (for comparison processes). If the evaluation fails, we take a step back to refine the metamodel; otherwise, we extend the metamodel to include other concepts.

We start the metamodel construction using two pre-defined SLAs, Amazon EC2 and S3, as examples. Amazon AWS is one of the leading providers of cloud computing and has been used as an example in many studies, such as [95, 111, 143]. Other cloud provider offers were considered, including RackSpace and GoGrid (both are IaaS public cloud. RackSpace is a common cloud that usually is compared with Amazon AWS [83, 119] and GoGrid has a different cloud SLA structure). The previously mentioned public clouds are IaaS clouds: they provide computing power, storage and networking. In the next section we describe the proposed metamodel. Section 4.2.1 provides a general description of the cloud SLA metamodel and an example model of a cloud SLA. The details of this metamodel are provided in Section 4.3. Then sections 4.4.1 and 4.4.2 provide a brief description of the early phases of constructing a cloud SLA metamodel.

4.2.1 Cloud computing SLA metamodel

We determined, after a number of experiments, that the two metamodels largely overlap (modulo naming conventions and a few additional constructs), and thus combined them. The new metamodel is different to the previously explained metamodels (Section 4.4.2). The cloud SLA metamodel illustrated in Figure 4.1 is thus composed of several metamodels. The figure shows the general composition while the details of each metamodel are explained later in Section 4.3. The cloud SLA metamodel is composed as follows:

- A Contract metamodel that includes different SLAs (Section 4.3.10).
- A cloud SLA metamodel (Section 4.3.9), which support specifying QoS parameters and SLO for service in an IaaS cloud.
- A service metamodel (Section 4.3.2), which specifies the services of IaaS cloud computing.
- A small metamodel for IaaS cloud service units was constructed, which allows service providers to specify service units separately from the SLA (which is the case of the public cloud provider). These can be reused independently of the main cloud SLA metamodel (Section 4.3.3.1).
- A price metamodel, because the public clouds provide different prices for the same type of unit. This depends on the payment method (on-demand or subscription), and can use different currencies depending on the location of the service (Section 4.3.8).
- An Obligation metamodel which specifies SLO and credits in case of violation of the SLA (Section 4.3.7).
- A generalTypes metamodel which specifies different concepts that are used by other metamodels (Section 4.3.1).
- A small metamodel for weighting, which allows cloud consumers to provide their preferences (Section 6.3).

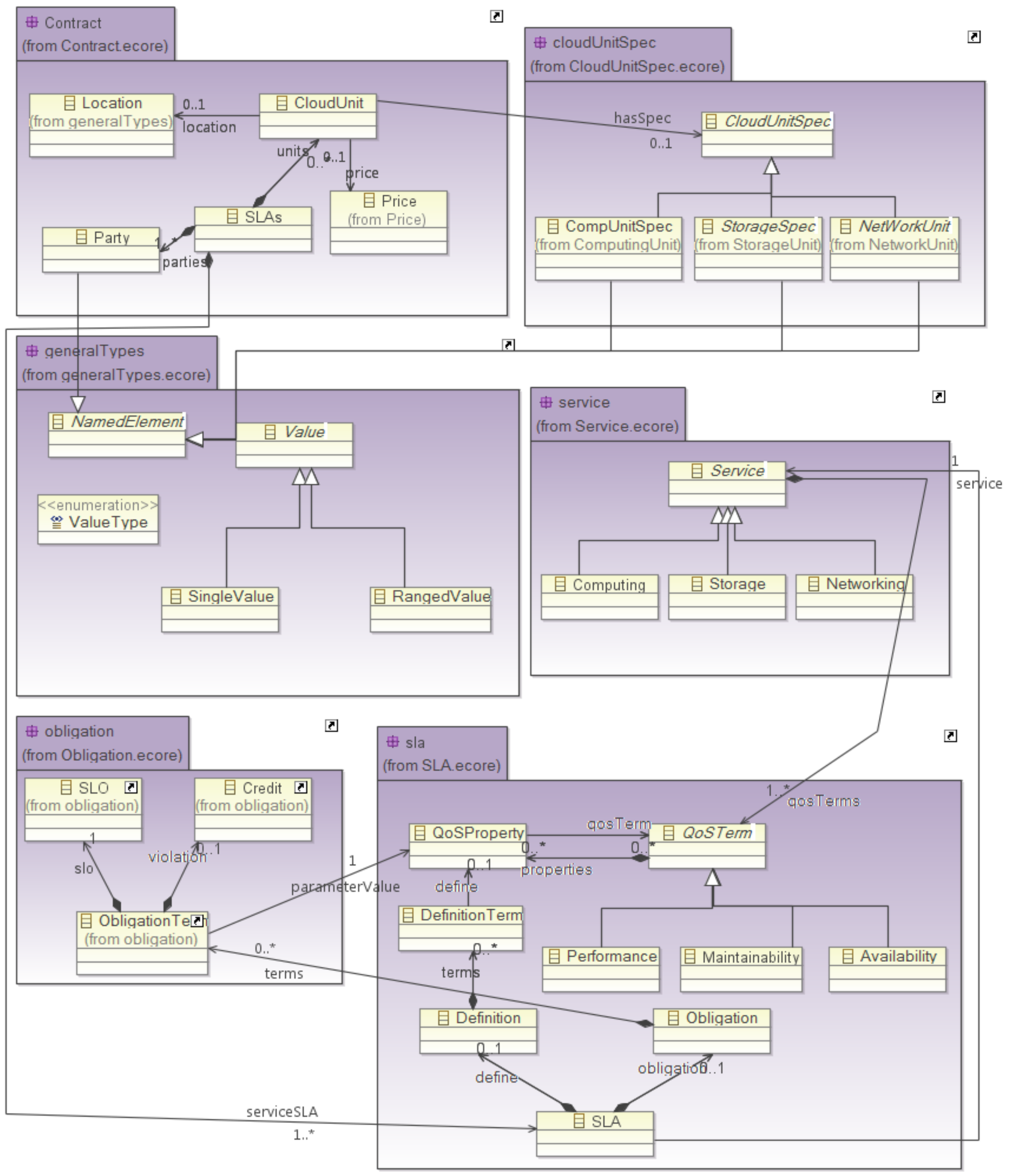


Figure 4.1: Main components of the cloud SLA metamodel

4.3 Cloud SLA Metamodelling

Section 4.2.1 provided an overview of the cloud SLA metamodel. This section discusses it in more detail. In each of the following sections we present a description of a (*Package*) of the main metamodel. Each section provides a general description of the package, abstract syntax, an example model created by the built-in EMF tree-based editor and constraints for certain models.

We provide an example model to explain the general cloud SLA metamodel. Assume a cloud consumer defines the following requirements for computing service that defines uptime = 99.9%, response time = 60 -160 GB and cloud unit (i.e. VM) with Linux operating system, number of cores 2, RAM = 10 GB and storage = 500 GB. Figure 4.2 shows the object diagram of these requirements as an SLA.

This figure shows an SLAs model that has a name, modelA, and defines a consumer as a party to this SLA. This model defines an SLA for a computing service. The SLA of the computing service defines two parameters: upTime and response time. The upTime parameter belongs to the QoS term Availability, while the response time parameter belongs to the Performance QoS term. Both the Availability and Performance terms are defined as QoS terms in the computing service. Uptime is defined as a single value = 99.9 and has a percentage as a unit, while the response time is defined as a ranged value type. The minimum value is 60 and the maximum value 120 of type time unit (i.e. second). The ModelA SLA defines a computing unit (i.e VM) with name VM1 and Linux OS. It has a cpu speed defined as the number of cores = 2, a RAM as a single value = 10 and the size unit type is GB. The price of VM1 is 0.06 USD (i.e. $) and the location of this computing unit is the US.

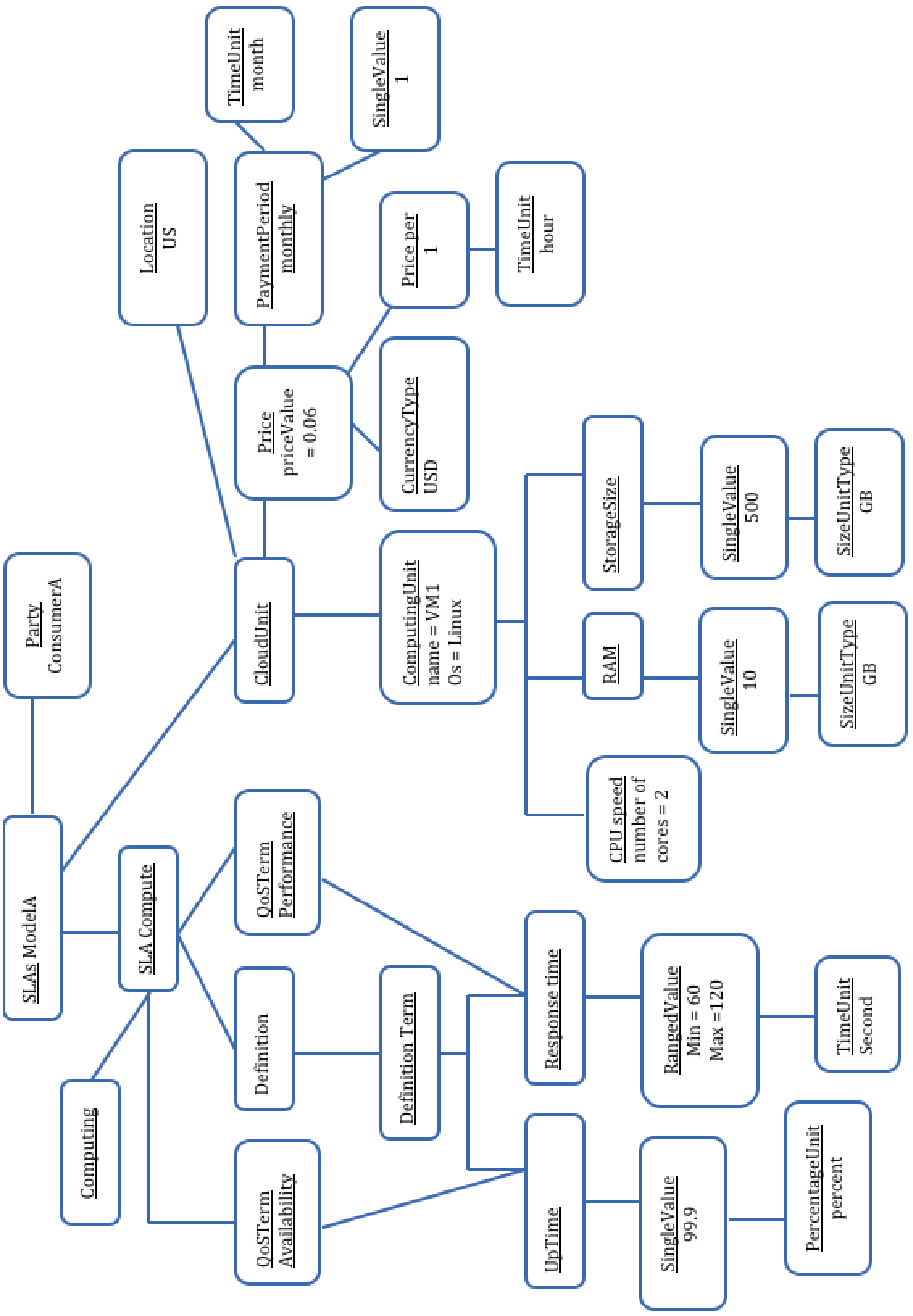


Figure 4.2: Object diagram of consumer requirements

4.3.1 General Types Package

The aim of the *GeneralTypes* package is to specify a set of common concepts that can be reused by other packages, such as value and units. QoS parameters and SLO thresholds have *values*, *units* and *types*, e.g. Latency <5 ms, where 5 is the *value*, *ms* is the *unit* and the *type* of this value is numeric. The units are used in defining cloud computing services, prices and SLAs and are as follows:

- Time unit specifies the time units e.g. *Monthly* payment period, *Hourly* price or *Monthly* uptime. The common time units identified from pre-defined cloud SLAs are hour, month and year. Time is defined as a value and a unit, e.g. 1 month (1 is the value and the month is the unit). As shown in Figure 4.2, the response time is defined per second, the price value is defined per hour and the payment period is defined per month.
- The capacity unit specifies the RAM and storage size, e.g. RAM = 10 GB or Storage = 500 GB. The common capacity units identified are: MB, GB and TB. Capacity is defined as a value and a unit, e.g. 10 GB where 10 is the value and GB is the unit.
- Percentage: specifies a percentage associated with QoS parameters. For example, in pre-defined cloud SLAs, the commonly used percentage is upTime percentage = 99.9%.
- Currency unit: specifies the currency that is used by the cloud party. For example, Amazon EC2 uses USD currency in their offers, while RackSpace provides the prices using the USD and GBP currencies.
- Location: specifies the location of the cloud services. For example, Amazon EC2 provides VM instances in different regions, e.g. US East, US West, EU etc. [8].
- Value type: specifies the value associated with *SLA* elements, e.g. QoS parameter, size value. A value type can be, e.g., a numeric, string or unordered set [83]. A numeric value can be a single value or a range value, e.g. size = 10 GB and response time = 60-120 second.

Thus a metamodel that includes the main units has been created. The following section presents the abstract syntax of the General Types Package.

4.3.1.1 Abstract Syntax

Figure 4.3 illustrates the GeneralTypes abstract syntax.

- Value: This is an abstract class and has two attributes which are:

  isPositive (Boolean attribute) and *valueType*. The *valueType* describes the type of a value e.g. String or Integer. The attribute *isPositive* is true when, the higher a value, the better its quality; and false when the lower a value, the better its quality (when the *valueType* is numeric).

- SingleValue: This class extends the abstract class *Value* and contains the *unit* of type *UnitType*. Moreover, it has one string attribute which is *value*. This is used to define numeric, string and Boolean single values (e.g. SingleValue(value = 99.9, type = float, isPositive = true, unit = percentage)).

- RangedValue: This class extends the abstract class *Value* and may specify a *maxUnit* and a *minUnit* of type *UnitType*. Furthermore, it specifies two attributes: *min* and *max*. As illustrated in Figure 4.2, the response time is defined as a ranged value with min = 60 and max =120.

- UnitType: It is an abstract class that is extended by other classes. This is used to define ranged numeric values, e.g RangedValue (min =99.9 max = 100, type = float, isPositive = true, minUnit = percentage, maxUnit=percentage).

- SizeType: This class inherits from the abstract *UnitType* class. It has an attribute *unit* of type *sizeUnitType*. This class is used to define the capacity size, e.g. storage size.

- Percentage: This class specifies the attribute *unit* which is of type *PercentageUnit* and extends the *UnitType*. This is used to define QoS parameters that have percentage values, e.g. availability 99.9%.

- TimeType: The *unit* attribute defined in this class is a *timeUnit* type. As *SizeType* class, it extends the *UnitType*. This is used to define the time periods. For example, in AWS, a price is defined per hour for cloud VMs, while it is defined per month for cloud storage.

- RequestType: The *unit* attribute has a *RequestUnit* type. This class also extends the *UnitType*. The cloud provider charges the cloud consumer for the bandwidth and requests. A request is a HTTP method such as GET or POST. This class is used to define the request type.

- CurrencyType: The *unit* attribute has a *Currency* type. This class also extends the *UnitType* class. This is used to define the different currencies.

- NamedElement: This abstract class specifies an attribute *name* which is a string type. This class is extended by other classes in other packages. This is used to define the names of different objects.

- Location: This class specifies three attributes *country*, *region* and *area*. This is used to define the regional location of the service. This is used to define the location of the services.

- TimePeriod: This class has an Integer attribute *interval* and a *TimeUnit* attribute *timeUnit*. The *TimePeriod* extends the *NamedElement* class. This is used to define the time units; for example, the time unit of the payment periods (monthly or yearly).

- PayementPeriod: This class extends the abstract class *NamedElement*, has an attribute *payementType* and specifies a *timePeriod*. This class is used to define the payment method for a service. As discussed, (see Section 2.2.2), the payment methods can be pay-as-you-go or on a subscription basis. Cloud providers provide for the same cloud unit, but with different prices based on the different payment methods. For instance, the price of the Amazon EC2 VM named “m3.medium” is $0.07 per hour on-demand, while the price of the reserved service (i.e. subscription) is $0.05 per hour.

- ValueType: It is an enumeration that has three values *Integer*, *String* and *Float*. This is used to define the type of the value, which is captured in the abstract class *Value*.

- TimeUnit: An enumeration that defines values: *second*, *minute*, *hour*, *day*, *month* and *year*. This is used by the *TimeType* class and defines the measurement unit for time.

- SizeUnitType: The value of this enumeration is: *Mbit*, *Gbit*, *MB*, *GB* and *TB*. This defines the attribute *unit* in the *SizeType* class. This enumeration is used to define a set of the measurement units for the capacity size.

- PaymentType: The value of the enumerations are: *subscribe*, *onDemand*, *spot*. This defines the type of the attribute *paymentType* of class *PaymentPeriod*. This enumeration specifies a set of payment types (Section 2.2.2).

- OS: This enumeration specifies the common operating systems used in cloud computing, e.g. *Windows*, *Linux, Unix* (Section 2.2.3.2). In cloud computing, the VM is defined in terms of a set of parameters, one of which is the OS. This OS is used in defining the VM in IaaS clouds.

- Region: This is used to define a set of regions to define the location of the service, these are: *NorthAmerica EU Asia SouthAmerica Africa Australia*.

- Country: This is used to define a set of countries that can be used later to define the location of the service. As a proof of concept, for simplicity in the comparison, we specify this enumeration to define a set of countries, to define the location of the cloud services.

- Currency: This enumeration is to define the possible currencies that a cloud SLA party can use to define their offers or needs. Similar to the *TimeUnit* enumeration, we specify a set of currencies to be used in the comparison process as a proof of concept.

- logicalOperator: This enumeration defines a set of logical operators *less lessEqual greater greaterEqual equal notEqual*. This is used to define the SLO logical condition, e.g. availability *lessEqual* 99.0%.

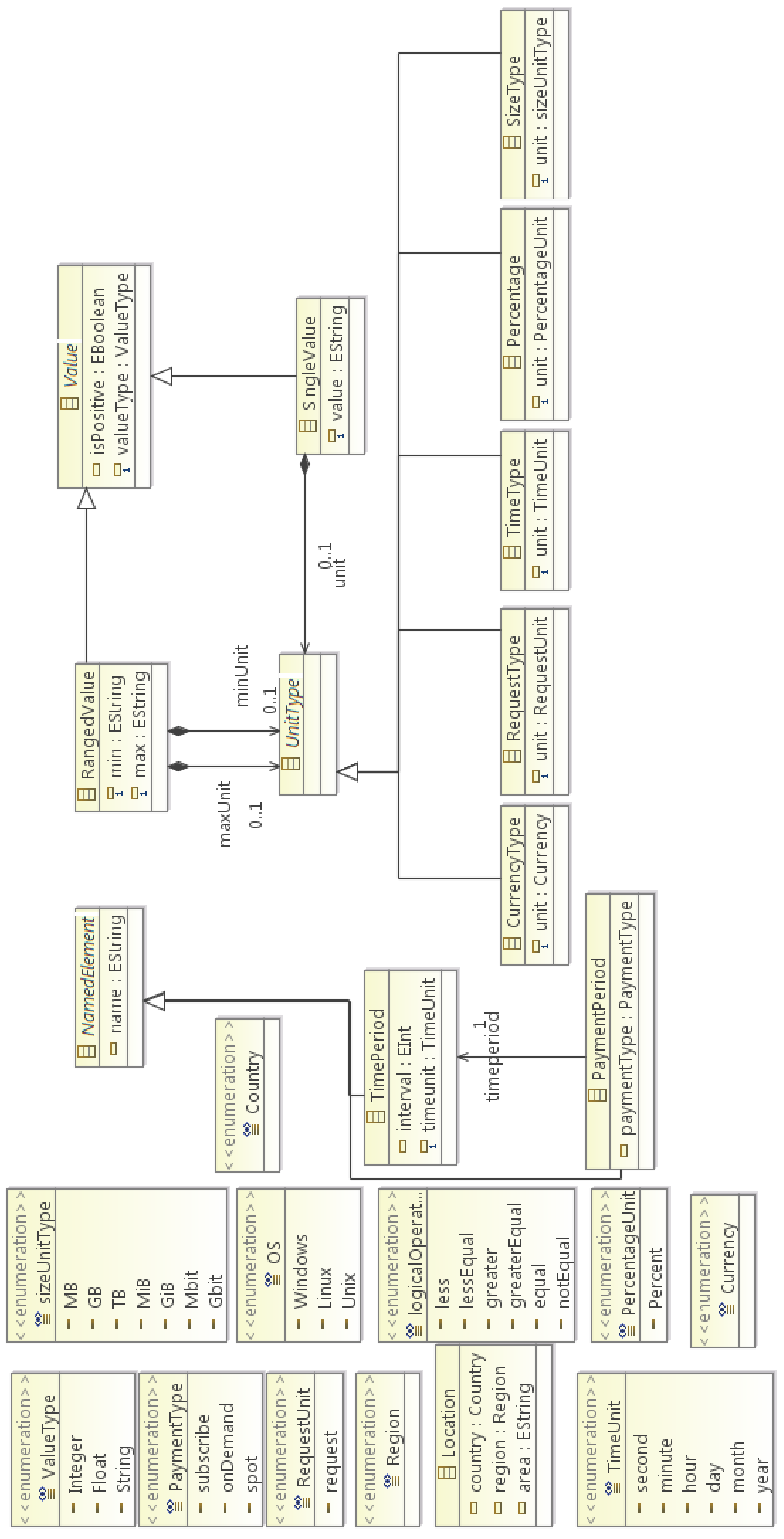
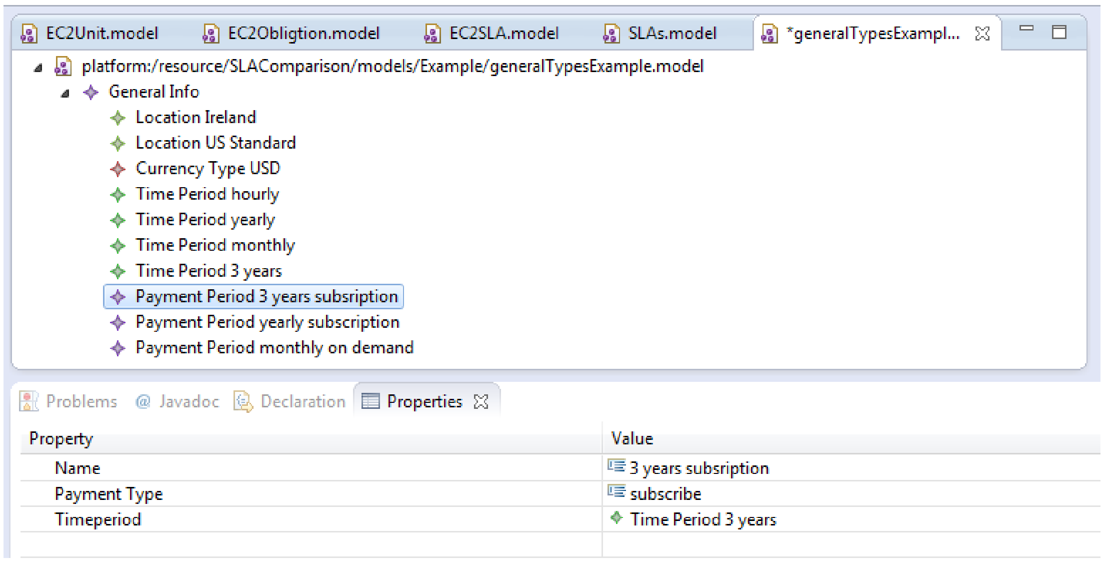


Figure 4.3: Cloud SLA general types Abstract Syntax

4.3.1.2 Example of GeneralTypes Model

Figure 4.4 illustrates an example use of the *GneralTypes* metamodel. In the example, two Locations are defined, Ireland and the US; thus, the cloud provider provides services in those two locations. The USD currency means that the cloud party defines these service units in this currency only. It is possible to define different currencies, thus allowing the units to have prices in different currencies. This is necessary, since some cloud providers provide services in different locations, and using a different currency in each one, e.g., RackSpace provides prices in USD in [21] and GBP in [22]. The diagram also illustrates that different time periods were defined which are: hourly, monthly, yearly and a 3 year period, which are used to define the payment periods. There are also three payment periods which are defined based on the time periods.



platform:/resource/SLAComparison/models/Example/generalTypesExample.model
- General Info
  - Location Ireland
  - Location US Standard
  - Currency Type USD
  - Time Period hourly
  - Time Period yearly
  - Time Period monthly
  - Time Period 3 years
  - Payment Period 3 years subsription
  - Payment Period yearly subscription
  - Payment Period monthly on demand

Property	Value
Name	3 years subsription
Payment Type	subscribe
Timeperiod	Time Period 3 years

Figure 4.4: Example of Cloud SLA general types model

We can also define constraints on the *SingleValue* and the *RangedValue* using EVL. *SingleValue* and *RangedValue* specify the attributes value, min and max which can have a string type. To ensure that these attributes can be defined in the model, we specify an EVL constraint shown in Listing 4.1.

The *context SLAs!SingleValue* defines two constraints on the *SingleValue*. The first constraint is *constraint AllDigits*, which check that the string *value* is real or an integer, when the *valueType* is defined as a *Float* or *Integer*. The other constraint, *Boolean-*

Value, checks that only *true* and *false* are defined as values for the attribute *value*, when the *valueType* is defined as *Boolean*; otherwise, a message is produced. In effect, this defines a domain specific type system for a particular SLA.

The constraint *AllDigitsRanged* is defined to check that the min and max string values defined in *RangedValue* have a numeric (Float or Integer) type only. The constraint *minLessThanMax* checks that the minimum value (min) is less than or equal to the maximum value (max). If not, then this can be repaired by swapping the min and max values.

Listing 4.1: The EVL Constraint that validate instances of SingleValue and RangedValue

```
context SLAs!SingleValue{
 constraint AllDigits {
  guard: self.valueType.name = "Integer" or self.valueType.name =
      "Float"
  check : (self.value.isReal() or self.value.isInteger())
  message : self.value + " is not a valid Numeric Value"
 }
 constraint BooleanValue{
  guard : self.valueType.name = "Boolean"
  check : (self.value = "false" or self.value ="true")
  message : self.value + " is not a valid Boolean Value"
 }
}
context SLAs!RangedValue{
 constraint Numeric{
  check : self.valueType.name = "Integer" or self.valueType.name
     = "Float"
  message : self.min + " to " + self.max  + " should have a
     Numeric type"
 }
 constraint AllDigitsRanged {
  check : (self.min.isReal() or self.min.isInteger()) and (self.
     max.isReal() or self.max.isInteger())
  message : self.min + " to " + self.max  + " not a valid Numeric
      Value"
 }
 constraint minLessThanMax {
  check : self.min.asReal() <= self.max.asReal()
  message : self.min + " should be less or equal to" + self.max
  fix {
   title : "Swap Min " + self.min + " with " + self.max
   do{
    var swap : String;
    swap = self.max;
    self.max = self.min;
    self.min = swap;
   }
  }
 }
}
```

4.3.2 Service Package

As described in Section 2.2.3.2, IaaS clouds provide three main services which are: *Computing Power*, *Storage* and *Networking* [206]. In typical pre-defined commercial cloud SLAs (such as Amazon and RackSpace), there is an SLA for each service. For example, Amazon AWS provides two SLAs: one for EC2 (Computing) and one for S3 (Storage); while RackSpace provides SLAs for cloud servers, cloud files (Storage), and the Load Balancer (Network). To define an SLA, the cloud party first defines the services to be included in the SLA, and then defines the QoS parameters related to each service. Back to Figure 4.2, the SLA model specified Computing as a service. The next section explains the abstract syntax of the Service Package.

4.3.2.1 Abstract Syntax

The abstract syntax is illustrated in Figure 4.5.

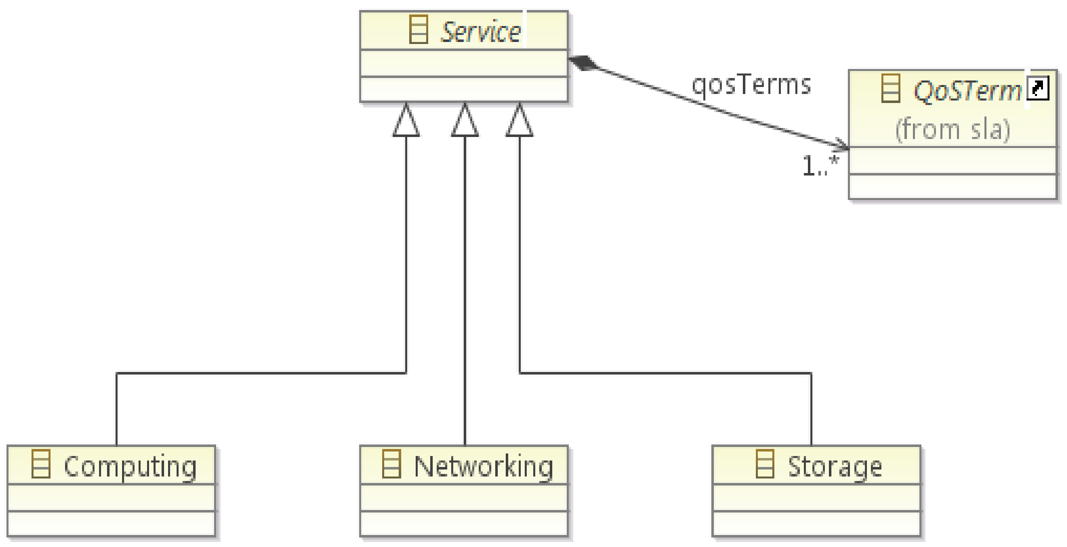
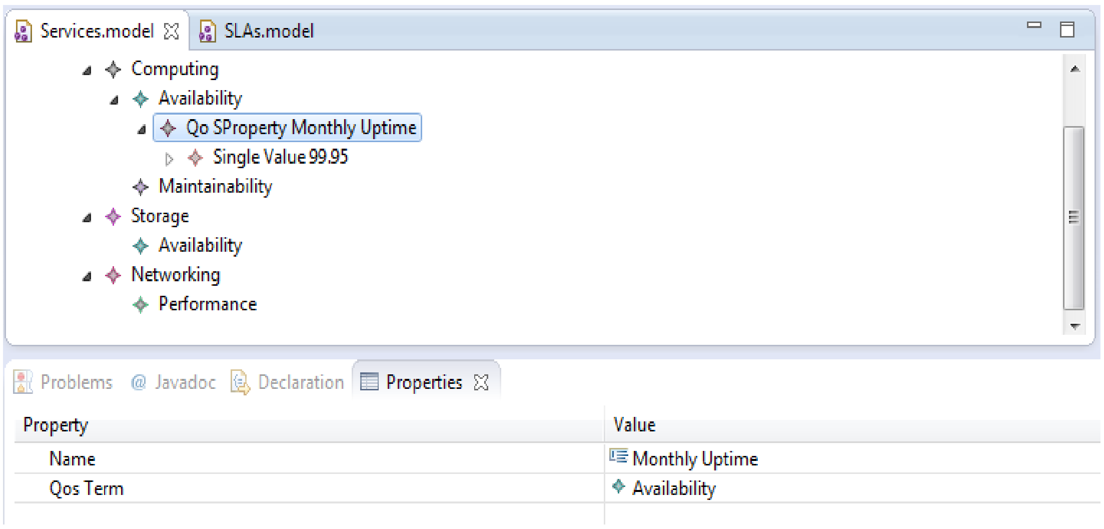


Figure 4.5: Service Abstract Syntax

- Service: an abstract class which contains at least one *qosTerms* of type *QoSTerm* (Section 4.3.9). This abstract *Service* class is used to specify the service type in an SLA model. The *Computing*, *Storage* and *Networking* services extend this class.
- Computing: extends the Service class, used to describe the service type of the cloud SLA (which in this case is computing).
- Storage: acts as a storage service of the SLA which extends the Service class.
- Networking: acts as a Networking service of the SLA which extends the Service class.

4.3.2.2 Example of Service Model

Figure 4.6 illustrates an example of a *Service* model. There are three services created: Computing, Storage and Network. The Computing service defines two *qosTerms*: availability and maintainability, while the Storage service specifies one *qosTerm*: Availability. In general, cloud parties create a service and its QoS parameters (*QoSTerm*). Then, when the cloud parties define their SLAs, they refer to the created service and



Services.model SLAs.model

- Computing
  - Availability
    - Qo SProperty Monthly Uptime
      - Single Value 99.95
  - Maintainability
- Storage
  - Availability
- Networking
  - Performance

Problems @ Javadoc Declaration Properties

Property	Value
Name	Monthly Uptime
Qos Term	Availability

Figure 4.6: Example of Service Model

its QoS parameter (*QoSTerm*). Thus, we created a separate Package for the *Service* types and the *QoSTerm*. Each *QoSTerm* (Section 4.3.9.1) is defined once in a service; hence, an EVL constraint needs to be specified. This is illustrated in Listing 4.2. This EVL constraint below is used to ensure that a model defines the QoS concept (e.g. Availability) no more than once within the same service.

Listing 4.2: The EVL Constraint generated for QoSTerm

```
context SLAs!Service{
 constraint OneQoSTermTypeForEachService {
  check {
   var notfound : Boolean;
   notfound = true;
   for (qos in self.qosTerms) {
    if (self.qosTerms.select(e | e.EClass.Name = qos.EClass.Name)
       .size > 1){
     notfound = false;
     break;
    }
   }
   return notfound;
  }
  message : self.qosTerms + " has duplicate QoSTerm"
 }
}
```

4.3.3 CloudUnitSpec Package

This package is constructed to specify IaaS cloud service units. The pre-defined public cloud SLAs do not provide specifications of the provided service, e.g. the CPU and RAM of a VM. However, these service units are specified to include the cost of the service in the SLA comparison. For a computing service, a unit is a ComputingUnit (called an instance in Amazon EC2, or a cloud server in RackSpace and GoGrid), while the storage unit can be cloud storage, block storage and backup storage. The network unit describes transferring types of data (i.e. in to the cloud service, out from the cloud service or between cloud services). Each service unit has a price and the cloud consumer is charged for consuming it. For example, an EC2 instance that has vCPU= 1, RAM = 3.75 GB, Storage = 1 x 4 SSD GB and Price = $0.070 per Hour is considered a service unit of *Computing Unit*. In the following sections (4.3.4- 4.3.6), a metamodel of the service units for *Computing Unit*, *Storage* and *Networking* is provided.

4.3.3.1 Abstract Syntax

Figure 4.7 illustrates the abstract syntax of *CloudUnitSpec*:

- CloudUnitSpec: This is an abstract class, which is extended by *ComputingUnit*, *StorageSpec* and *NetWorkUnit* classes. The *CloudUnit* class (explained in Section 4.3.10.1) is associated with the *CloudUnitSpec* class.

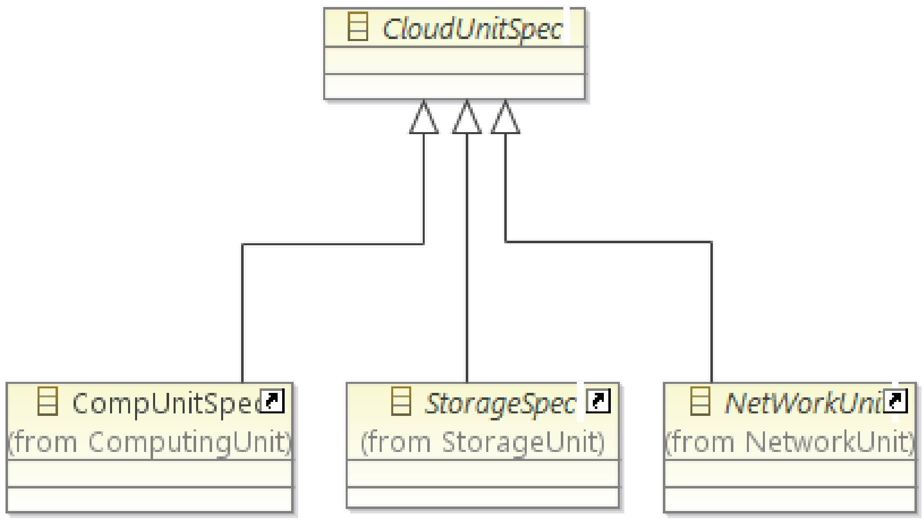


Figure 4.7: Cloud Unit Abstract Syntax

4.3.3.2 Example of CloudUnitSpec model

As illustrated in Section 4.3.3.1 the *CloudUnitSpec* class is an abstract class extended by other classes: *ComputingUnit*, *StorageSpec* and *NetWorkUnit*. These classes are used to define the different units of services of the IaaS cloud, e.g. (*CompUnitSpec*), *CloudStorage* and *BlockStorage*. These units are explained in the following sections (4.3.4- 4.3.6).

4.3.4 ComputingUnit Package

This package is provided to specify the Computing Power of an IaaS cloud (Section 2.2.3.2). Computing power can be expressed in terms of VMs (e.g. Amazon EC2, RackSpace and GoGrid) or dedicated servers (e.g. GoGrid and RackSpace). From the different cloud providers, several recurring characteristics have been identified in defining computing units. The common specifications are:

- Each server is assigned a name, e.g., "m3.medium',' the VM name in Amazon EC2 [7].

- Each server is assigned with a number of vCPU (cores), e.g. "m3.medium" has vCPU = 1 [7].

- Memory (RAM), e.g. “m3.medium” has memory = 3.75 GiB [7].
- Storage, e.g. “m3.medium” has storage = 1 x 4 GB[7].
- OS, e.g. Amazon EC2 provides support for different OSs, such as Linux and Windows [8].

4.3.4.1 Abstract Syntax

Figure 4.8 illustrates the abstract syntax of the computing power unit.

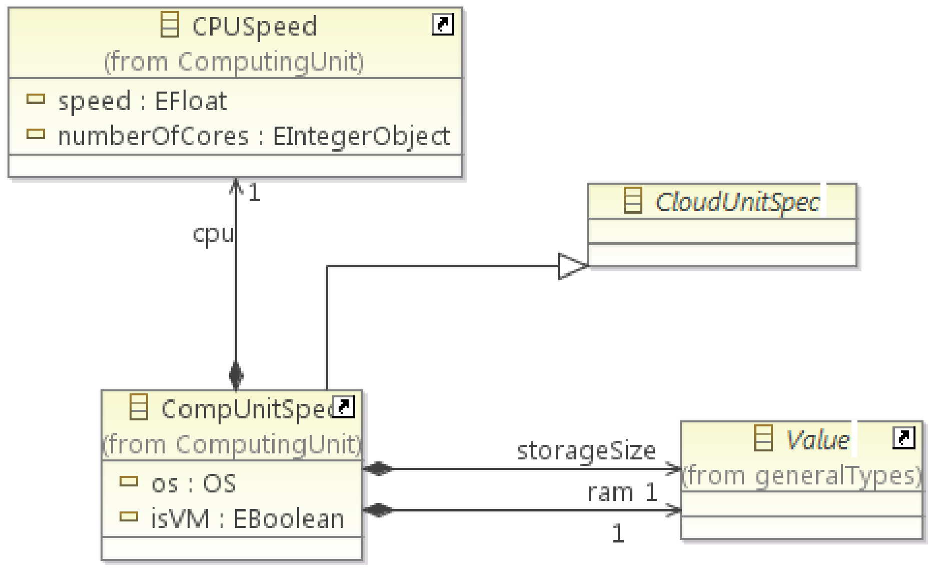


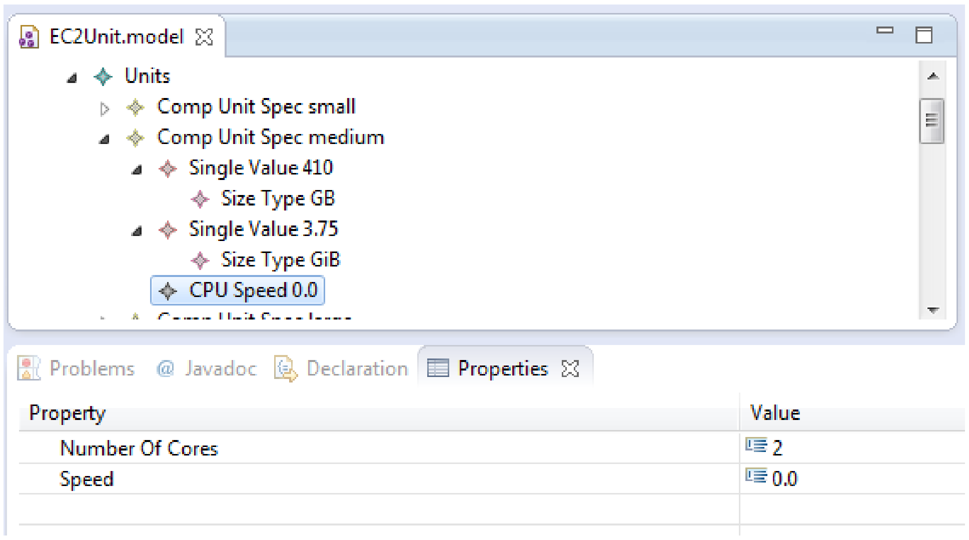
Figure 4.8: Computing Unit Abstract Syntax

- CompUnitSpec: This class has a name and extends the abstract class *CloudUnitSpec* (Section 4.3.3.2) and the *NamedElement* class (Section 4.3.1.1). It also defines a Boolean attribute *isVM*, which is true if the computing unit is a VM and false otherwise. Moreover, the *OS* attribute is defined to specify the operating system of the computing unit. This class associates with *CPUSpeed* class and *Value* class. The *Value* class, which is explained in Section 4.3.1.1 supports expressions of memory and *storageSize*.

- CPUSpeed: Defines the CPU specifications, which has the attributes; speed and *numberOfCores*.

4.3.4.2 Example of a ComputingUnit Model

In this section, we provide a small example of the VM specifications' model. As illustrated in Figure 4.9, a *CompUnitSpec* named medium, storageSize (*SingleValue*) = 40 GB, ram (*SingleValue*) = 3.75 GiB and number of cores = 2.



```
Units
    Comp Unit Spec small
    Comp Unit Spec medium
        Single Value 410
            Size Type GB
        Single Value 3.75
            Size Type GiB
        CPU Speed 0.0
```

Property	Value
Number Of Cores	2
Speed	0.0

Figure 4.9: Example of Computing Unit model

4.3.5 Storage Package

The storage package is constructed for the purpose of specifying cloud storage service units. Cloud providers provide storage in different types and at different prices, such as: cloud storage, block storage and backup storage. Amazon S3 and RackSpace are examples of cloud storage, while Amazon EBS, RackSpace Cloud Block Storage and GoGrid Block Storage are examples of block storage. Backup Storage is also provided by Amazon, RackSpace and GoGrid. Why do we include these storage types in the metamodel? Cloud storage is included in pre-defined cloud SLA offers (e.g. Amazon S3 SLA and RackSpace cloud files SLA). Block storage is also usually related to the

VM which affects its price, while the backup storage is used by Amazon, RackSpace and GoGrid in their offers.

This package allows the consumer to define their requirements which can be then compared with the cloud providers offers, compare the prices, and thereafter calculate the total cost of the cloud service.

4.3.5.1 Abstract Syntax

The abstract syntax of the storage services is as follows (Figure 4.10):

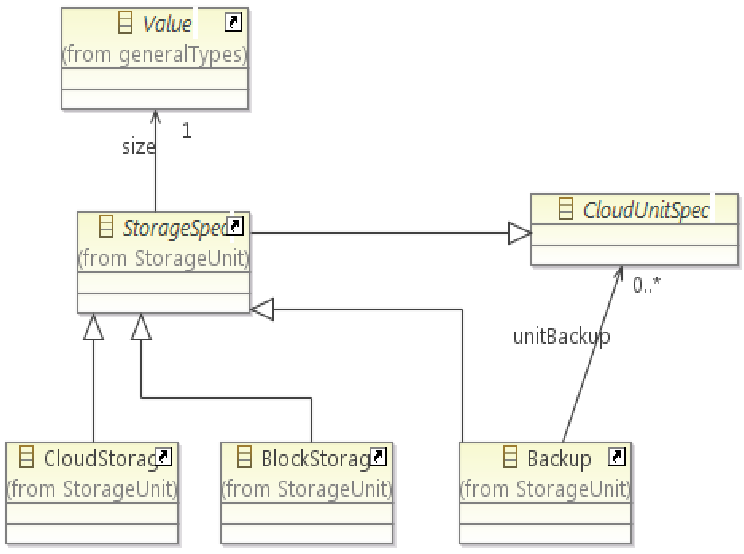
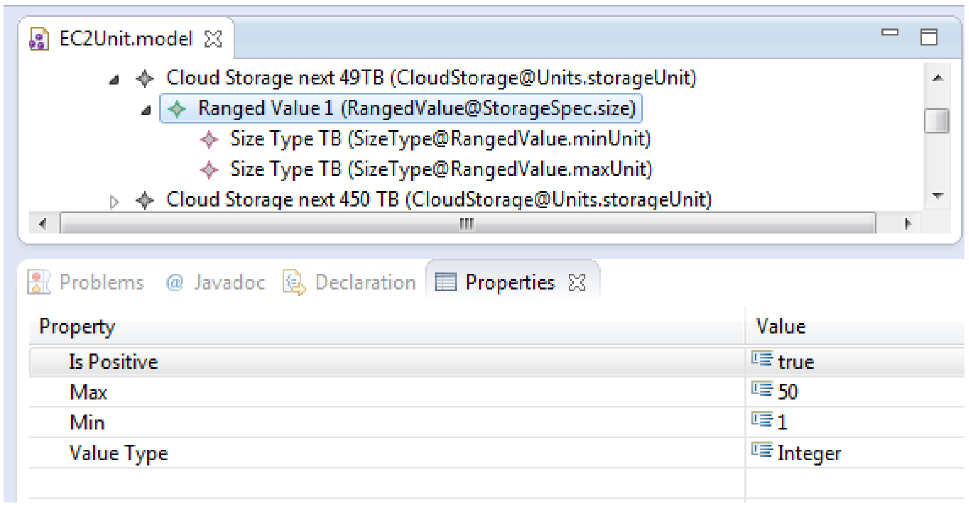


Figure 4.10: Storage Unit Abstract Syntax

- StorageSpec: This is an abstract class and specifies a *size* of type *Value* (Section 4.3.1.1). It inherits the *CloudUnitSpec* class (Section 4.3.3.1). This class is extended by three other classes which are explained as follows: These three classes are provided to distinguish the different storages that are provided by the cloud provider.
- CloudStorage: This class extends the *StorageSpec* class. This class is used to specify cloud storage such as Amazon S3 and RackSpace.
- BlockStorage: This class extends the *StorageSpec* class. This class specifies block storage (such as Amazon EBS).



EC2Unit.model

Cloud Storage next 49TB (CloudStorage@Units.storageUnit)
Ranged Value 1 (RangedValue@StorageSpec.size)
Size Type TB (SizeType@RangedValue.minUnit)
Size Type TB (SizeType@RangedValue.maxUnit)
Cloud Storage next 450 TB (CloudStorage@Units.storageUnit)

Problems | Javadoc | Declaration | Properties

Property	Value
Is Positive	true
Max	50
Min	1
Value Type	Integer

Figure 4.11: Example of Storage Unit model

- Backup: This class extends the *StorageSpec* class and specifies a *unitBackup*. This class specifies backup storage (available from all cloud providers).

4.3.5.2 Example of a Storage Model

As illustrated in Figure 4.11, a storage unit, of CloudStorage type is defined with a size of range (*RangedValue*) = 1-50, thus we assign min = 1 and max = 50; and the units of minUnit and maxUnit = TB. The attribute isPositive = true (in the properties window) which means that a larger value of size is better.

4.3.6 Network Package

This package is constructed to specify the network service units. IaaS public clouds provide networking services, such as transferring data into/out of cloud services and between cloud services. Other networking services may also be provided, such as load balancing and firewalls. These services are provided with respect to other services, i.e. they are related to Storage and Computing services. Moreover, cloud providers specify prices for these services. As a consequence, we include such network services in our metamodel. In addition, some cloud providers (e.g. GoGrid) include references to network performance in a cloud SLA. Some cloud providers also distinguish data transferring *in* to the cloud provider services, and *out* of the cloud provider services.

They distinguish services in different geographical locations. For instance, data transfer from Amazon EC2 to another Amazon EC2 in a different geographical locations costs $0.01 [8].

4.3.6.1 Abstract Syntax

The abstract syntax of the Network Package is illustrated in Figure 4.12.

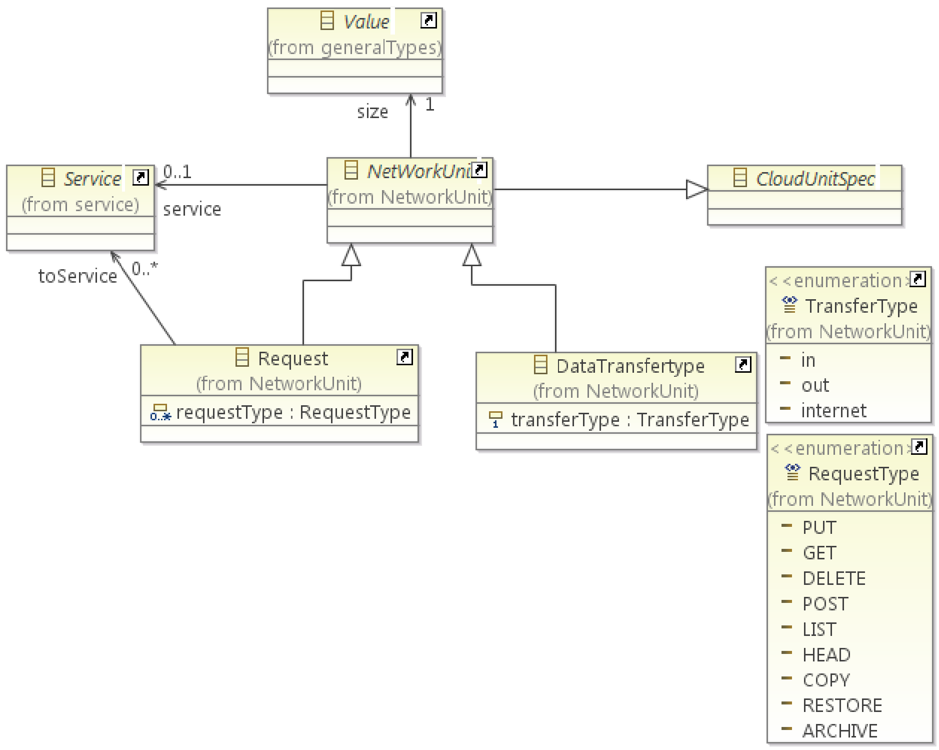


Figure 4.12: Networking Abstract Syntax

- NetworkUnit: This is an abstract class that extends the *CloudUnitSpec*. It specifies the *size* of type *Value* (which is explained in Section 4.3.1.1). This class may have a *service*.
- RequestType: This class extends the abstract class *NetworkUnit* and has an attribute *requestType*. Usually public cloud providers (e.g. Amazon and RackSpace) define a request and a price for this service depending on the number of requests.

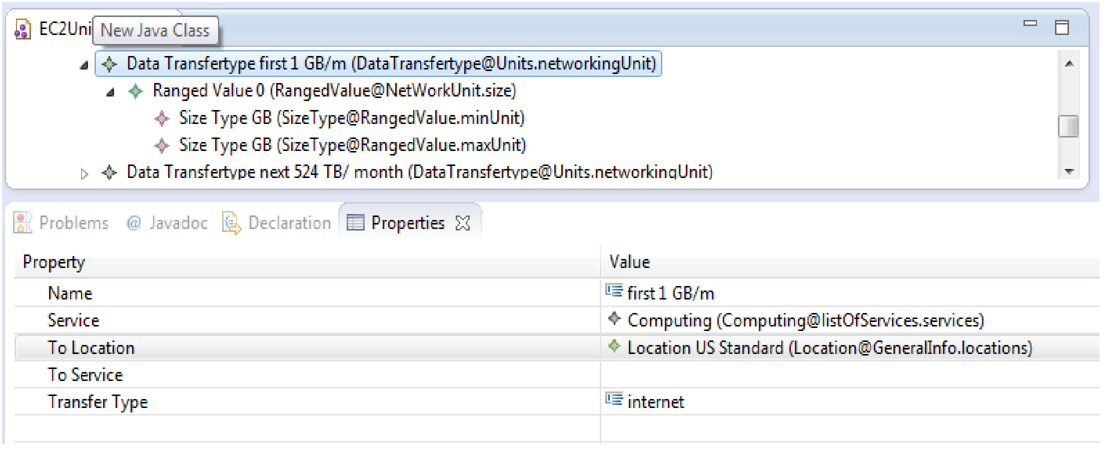
Typical request examples are: put, get or post. Thus this class is constructed to specify the price of different requests.

- TransferType: This class extends the abstract class *NetworkUnit* and has an attribute *transferType*. In public cloud providers, transferring data into the cloud has different prices compared to transferring them out of the cloud services. The price of this service depends on the size of the data and to where they are transferred.

- TransferType: This is an enumeration and has three values *in*, *out* and *internet*. The *in* is to specify data transferred into the cloud provider service. The *out* is to specify data transferred out of the service to another service but within the same provider. Finally, *internet* is used to specify that data are transferred out of the cloud provider services.

- RequestType: This is an enumeration and has values *PUT, GET, DELETE, POST, LIST, HEAD, COPY, RESTORE*, and *Archive*. These values are used to specify the request type.

4.3.6.2 Example of Network Model



```
◢ Data Transfertype first 1 GB/m (DataTransfertype@Units.networkingUnit)
    ◢ Ranged Value 0 (RangedValue@NetWorkUnit.size)
        Size Type GB (SizeType@RangedValue.minUnit)
        Size Type GB (SizeType@RangedValue.maxUnit)
▷ Data Transfertype next 524 TB/ month (DataTransfertype@Units.networkingUnit)
```

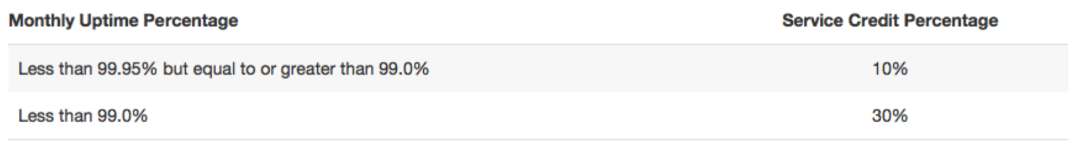
Property	Value
Name	first 1 GB/m
Service	Computing (Computing@listOfServices.services)
To Location	Location US Standard (Location@GeneralInfo.locations)
To Service	
Transfer Type	internet

Figure 4.13: Example of cloud Network Model

DataTransferType can be defined by a name and a size, as illustrated in Figure 4.13. We can also associate the *DataTransferType* with a service that is using this network service, e.g. transferring data from the cloud servers (*Computing*) to the internet (out side the cloud services), as in the figure.

4.3.7 Obligation Package

An obligation, in the context of a cloud SLA, is a commitment of the cloud provider in case of violations. Public cloud providers define a penalty in the form of credit as a commitment in case of SLA violation. Cloud providers usually provide different threshold values for a service (SLO Section 2.2.4.1) as a commitment, and different credit values for each threshold, as illustrated in Figure 4.14. Therefore, we define an obligation as a number of terms (*obligationTerm*), each of which consists of an SLO and credit. Overall, this package is constructed to specify the obligations which are composed of SLO and credit as explained in the following section.



Monthly Uptime Percentage	Service Credit Percentage
Less than 99.95% but equal to or greater than 99.0%	10%
Less than 99.0%	30%

Figure 4.14: A screen shot of Amazon EC2 Credits [4]

4.3.7.1 Abstract Syntax

The abstract syntax of the obligation package is illustrated in Figure 4.15.

- ObligationTerm: Each *Obligationterm* contains *slo* of type *SLO*, *parameter-Value* of type *QoSPreperty* (Section 4.3.9.1) and may specify *violation* of type *Credit*.
- SLO: This class specifies a *value* and an *operator*. This class is constructed to specify an SLO logical condition, e.g. uptime <99.95.
- Credit: This class defines a *creditValue* and may specify a *creditType*, *minValue* and *maxValue*. This class is used to specify the credit value in case the logical condition (SLO) is true.

- ServiceCreditType: This is an enumeration and has two values: *serviceTime* and *servicePrice*. This class is created to support the comparison process, to differentiate how credit is calculated. For example, in Amazon EC2 the credit is calculated based on the percentage of the total charges paid by a consumer for a monthly billing period, while in GoGrid, the credit is calculated as one hundred (100) times the Consumer's fees for the impacted Service feature for the duration of the Failure.

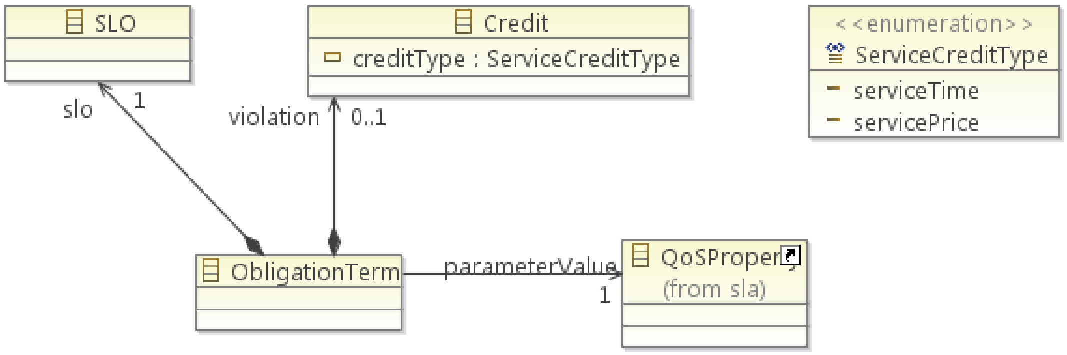
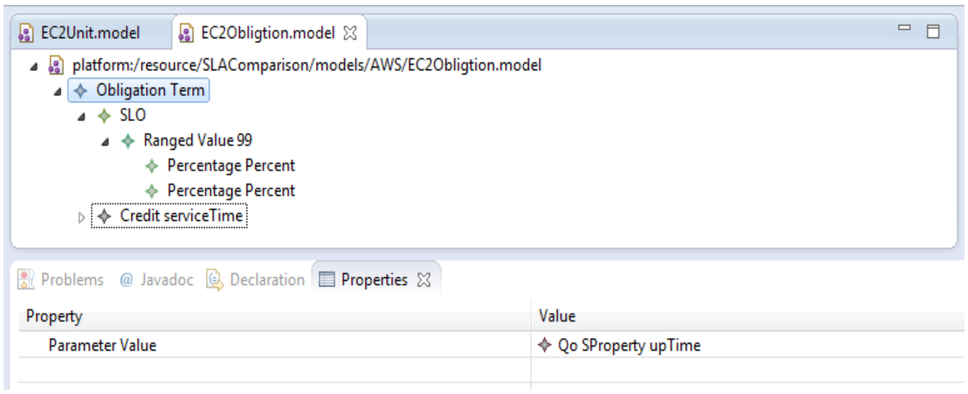


Figure 4.15: Obligation Abstract Syntax

4.3.7.2 Example of Obligation

An example of an Obligation model is illustrated in Figure 4.16. The obligation term has a *parameterValue* named upTime, which is created in the service model (Section 4.3.2.2). For this *QoSProperty* there is an SLO defined, where the threshold level uptime is <99.95% and >99.9% and has a credit value with 10% of the total cost of the service unit.

If this is the case, then there is a credit to be paid to the consumer, as shown in the figure.



```
platform:/resource/SLAComparison/models/AWS/EC2Obligtion.model
  Obligation Term
    SLO
      Ranged Value 99
        Percentage Percent
        Percentage Percent
    Credit serviceTime
```

Property	Value
Parameter Value	Qo SProperty upTime

Figure 4.16: Example of Obligation Model

4.3.8 Price Package

In cloud computing offers, cloud providers usually give a set of cloud service unit prices. Additionally, cloud providers usually provide a calculator for the cloud consumer to calculate the expected cost of consuming cloud services. To support such price estimates, a price package was included, to both compare prices and to calculate the cost of the cloud service units. The proposed metamodel is constructed for both provider and consumer SLAs; thus, this package includes only the prices. The cost (which is the total price for the services used) is calculated separately in Section 6.4.2.

4.3.8.1 Abstract Syntax

The abstract syntax of the price package, which is illustrated in Figure 4.17, consists of:

- Price: This class specifies an attribute *priceValue* which has a Float type and a number of *pricePer* of type *SingleValue* . Moreover, it specifies the *paymentPeriod* of type *PaymentPeriod* (Section 4.3.1.1), *currency* of type *CurrencyType* (Section 4.3.1.1) and may define *extraCharges*.

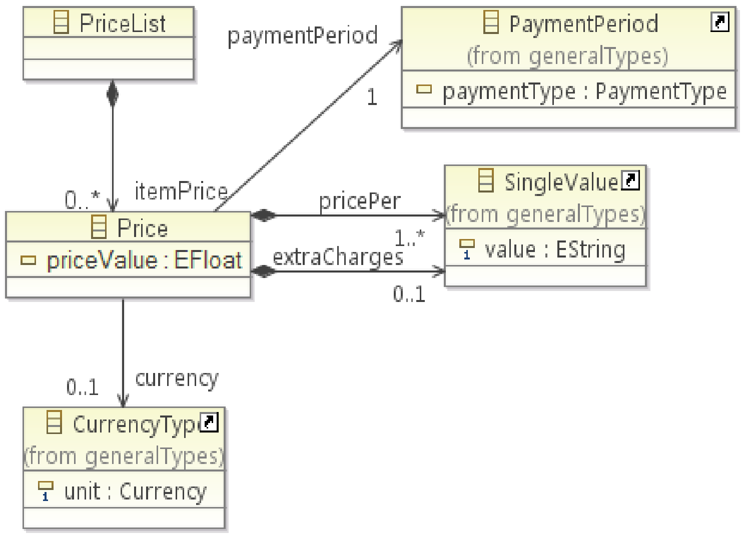


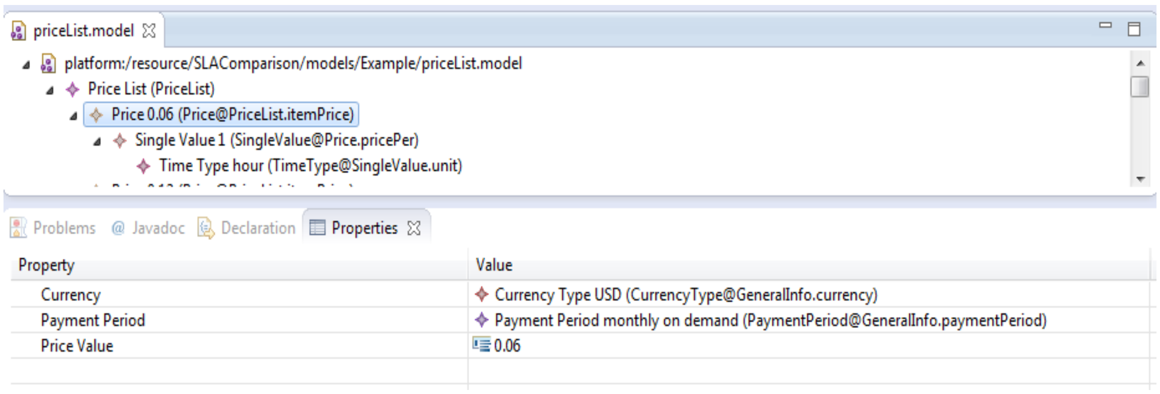
Figure 4.17: Price Abstract Syntax

4.3.8.2 Example of Price Model

The example of a price model in Figure 4.18 indicates a price of \$0.06 for using the service for an hour (*pricePer*) on a monthly payment basis.

4.3.9 SLA Package

Cloud SLAs consist of several components (as explained in Section 2.2.4.1). A predefined cloud SLA, e.g. EC2 or RackSpace, defines for each service an SLA. Each



```
platform:/resource/SLAComparison/models/Example/priceList.model
  Price List (PriceList)
    Price 0.06 (Price@PriceList.itemPrice)
      Single Value 1 (SingleValue@Price.pricePer)
        Time Type hour (TimeType@SingleValue.unit)
```

Property	Value
Currency	Currency Type USD (CurrencyType@GeneralInfo.currency)
Payment Period	Payment Period monthly on demand (PaymentPeriod@GeneralInfo.paymentPeriod)
Price Value	0.06

Figure 4.18: Example of a Price Model

SLA has a definition section where a service and relevant QoS parameters can be defined. It also contains an Obligation section, where it defines an SLO and credits. In the abstract syntax for the SLA package, which is explained in the following section, we define Availability as a QoS parameter; it is the most common QoS concept in cloud SLAs. Two more QoS concepts are defined, Maintainability and Performance, because the public cloud SLAs of GoGrid and RackSpace defined such concepts; their inclusion here is meant to be a proof of concept to show that non-availability concepts can be supported. Other QoS concepts can be added by extending *QoSTerm*. The purpose of constructing this package is to support specifying a cloud SLA consisting of a definition, obligation, QoS concepts and service. In the following section, the abstract syntax is illustrated.

4.3.9.1 Abstract Syntax

Figure 4.19 illustrates the abstract syntax of a SLA which consists of:

- SLA: has a name and a *service* of type *Service* (Section 4.3.2.1). Moreover, each SLA may contain an Obligation and Definition, where different obligation terms and definitions can be defined.
- Definition: contains a number of terms *DefinitionTerm* e.g. QoS parameters. This class specifies definitions of the cloud SLA, which is usually is a part of the cloud SLA (Section 2.2.4.1).
- Obligation: Acts as a container where different *obligationTerms* can be defined (Section 4.3.7). This class specifies the obligations of the cloud provider, which is usually part of the cloud SLA (see Section 2.2.4.1).
- DefinitionTerm: This class has a description attribute and associates a *QoSProperty*. This class specifies the definition term and its description.
- QoSProperty: Specifies a QoS parameter that has a *Value* and associates with a number of *QoSTerm*. This class specifies the quality parameters that can be defined by the cloud provider or consumer. This QoSProperty is used in the ObligationTerm term (see Section 4.3.7.1). This class is used to quantify the QoS concepts [52], i.e. *QoSTerm*.

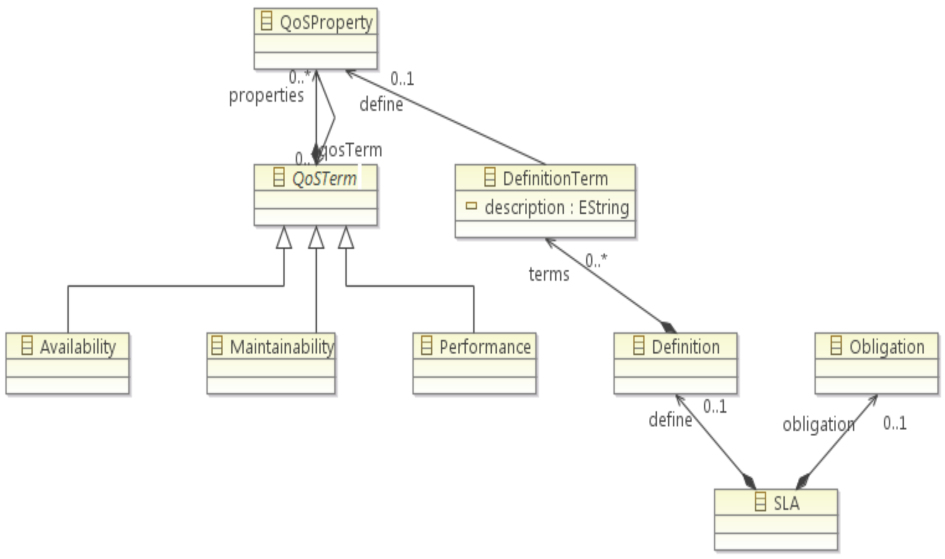


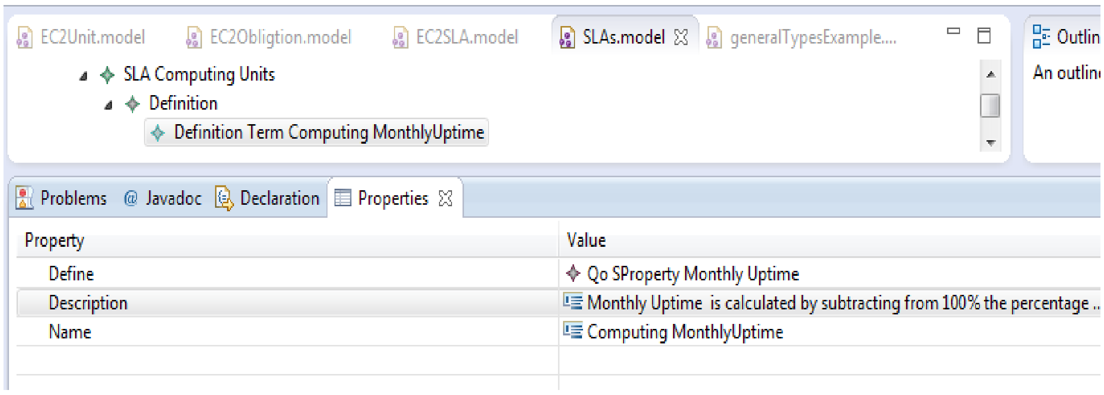
Figure 4.19: SLA Abstract Syntax

- QoSTerm: An abstract class that contains a number of *QoSProperty*. The classes extend the *QoSTerm* are (the QoS concepts): *Availability*, *Maintainability*, *Performance*, *Elasticity*, etc. In this abstract syntax, we define some of these parameters as a proof of concept. We choose Availability because it is common in cloud computing SLAs. The performance parameter is used widely in QoS studies and also in web service SLAs. The performance in the pre-defined cloud SLAs is provided in the GoGrid SLA for the networking service.

- Availability: A class that extends the abstract class *QoSTerm*, where different parameters related to availability can be defined as a *QoSProperty*.

- Performance: A class that extends the abstract class *QoSTerm*, where different parameters related to performance can be defined as a *QoSProperty*.

- Maintainability: A class that extends the abstract class *QoSTerm*, where different parameters related to maintainability can be defined as a *QoSProperty*.



- SLA Computing Units
  - Definition
    - Definition Term Computing MonthlyUptime

Property	Value
Define	Qo SProperty Monthly Uptime
Description	Monthly Uptime is calculated by subtracting from 100% the percentage ..
Name	Computing MonthlyUptime

Figure 4.20: Example of SLA Model

4.3.9.2 Example of SLA Model

The SLA in Figure 4.20 refers to a Computing service created in a Service Model (Section 4.3.2.2). It has the name Computing SLA. This model also defines an obligation and a definition, each of which consists of terms. The obligation defines obligationTerms which is created in Obligation model (Section 4.3.7.2). The definition defines terms; e.g. Computing Monthly Uptime. This term has a text description, e.g. "monthly Uptime is calculated by subtracting from 100% the percentage...", and it refers to the QoSProperty Monthly Uptime created in the Service Model in Section 4.3.2.2.

4.3.10 Contract Package

As mentioned in Section 2.2.4.1, an SLA is part of a contract that includes the parties, definitions of the services and QoS for each service. In our metamodel, the contract defines a number of SLAs and parties. The contract also contains the cloud service specifications and their prices, in the form of a *CloudUnit*. This *CloudUnit* is defined to allow cloud providers or cloud consumers to define their demands and include the costs in SLA comparisons. The pre-defined cloud SLAs define services but do not include the prices and different specifications for these services, which are normally defined separately; e.g. Amazon EC2 defines instance types [7] and instance prices [8]. Each cloud unit is identified by (e.g. Amazon EC2, S3, RackSpace and GoGrid):

1. a set of characteristics (e.g. as mentioned in Section 4.3.4).

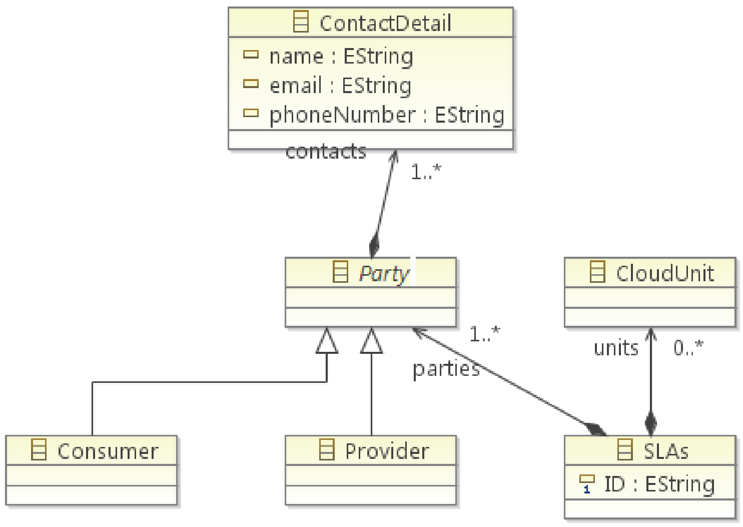


Figure 4.21: Contract Abstract Syntax

2. a geographical location, e.g. different locations are defined for the Amazon EC2, such as the US and EU [8].

3. a price which depends on the characteristics of the cloud unit and geographical location [8].

4.3.10.1 Abstract Syntax

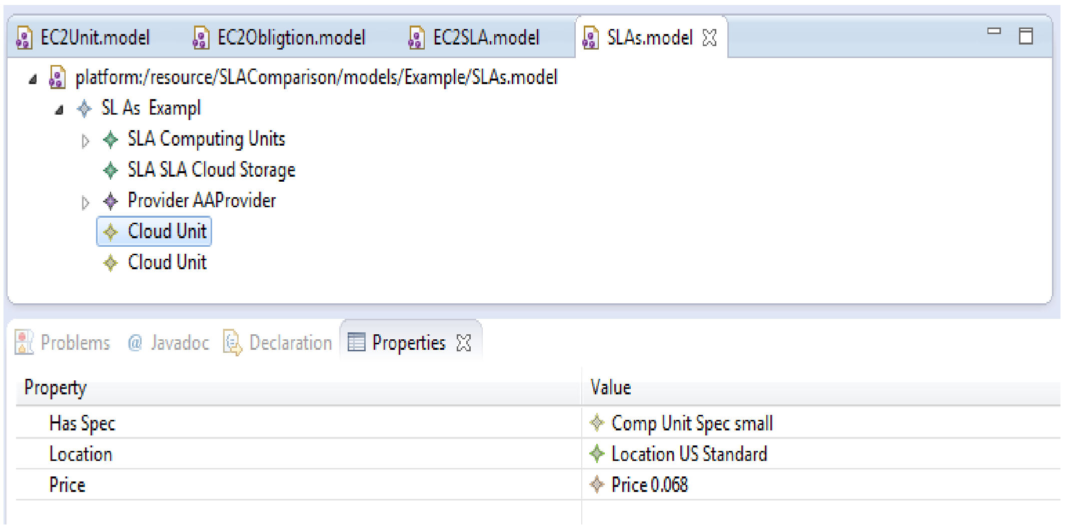
Figure 4.21 illustrates the *Contract* abstract syntax, which defines:

- SLAs: Act as the root of SLA models. It allows a *Party* to define more than one SLA. Moreover it contains a number of *CloudUnit*.

- Party: A Party has a name and contact details and is extended by the Consumer and Provider classes.

- SLA: Contains details about SLA, which is explained in Section 4.3.9.

- CloudUnit: Associates the *Location* (see Section 4.3.1.1), *Price* (see Section 4.3.8) and *CloudUnitSpec* (see Section 4.3.3.1). Thus an SLA party can define a number of cloud units. Figure 4.2 shows that a computing unit named VM1 with price 0.06 USD per hour and located in the US is defined as a cloud unit.



platform:/resource/SLAComparison/models/Example/SLAs.model
SL As Exampl
SLA Computing Units
SLA SLA Cloud Storage
Provider AAProvider
Cloud Unit
Cloud Unit

Property	Value
Has Spec	Comp Unit Spec small
Location	Location US Standard
Price	Price 0.068

Figure 4.22: Example of Contract Model

4.3.10.2 Example of Contract Model

The Contract model example defines different SLA models, e.g. ComputingUnits and CloudStorage SLAs, as shown in Figure 4.22. The model defines a Provider named *AAProvider*. This cloud party can define a number of cloud units. In this figure, we created two cloudUnits. Each CloudUnit refers to a price (which is created in the Price model - Section 4.3.8.2), a location (which is created in the GeneralTypes model - Section 4.3.1.2), a cloud unit (which is created in the ComputingUnit model - Section 4.3.4.2, ta Storage model - Section 4.3.5.2 and a Network model - Section 4.3.6.2). As illustrated in the figure, the computing unit is named “small” and its location is the US and its price is $0.068.

A cloud party can be either a provider or a consumer. Initially, the metamodel was developed for one party, e.g. a cloud provider creates an SLA model for their offers, while the consumer creates an SLA model for their demands. However, an SLA is a contract between two or more parties (as explained in Section 2.1). The notion of a *Contract* makes it possible to define one or more parties. However, we added constraints to restrict the number of consumers and providers as parties in a cloud SLA to one of each. This is illustrated in the EVL constraints in Listing 4.3.

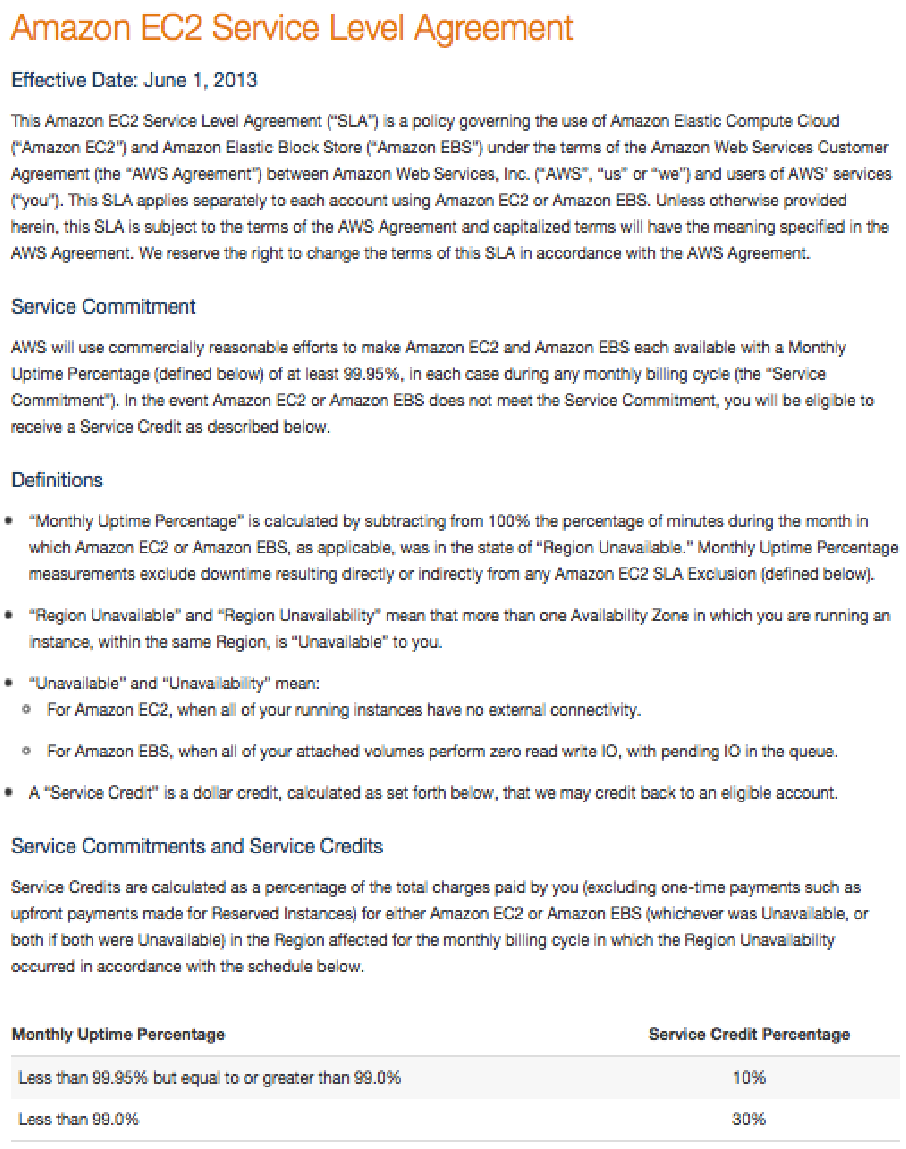
Listing 4.3: The EVL Constraint generated restricting the number of consumers and providers of cloud SLA to one

```
context SLAs!SLAs{
 constraint MoreThanOne {
  check : self.parties.select( e | e.isKindOf(SLAs!Consumer)).
      size() <=1 and self.parties.select(e | e.isKindOf(SLAs!
      Provider)).size() <=1
  message : "SLA must define at most 1 consumer and 1 provider"
 }
}
```

4.4 Early Versions of the Cloud SLA Metamodel

4.4.1 First Version Metamodel for Cloud Computing SLA

Amazon SLA EC2 and S3 have similar structures (Figure 4.23). Each makes use of the following concepts: Definitions, Service Commitments and Service Credits (describes a threshold value of the monthly uptime percentage and the credits if that threshold is exceeded), Credit Request and Payment Procedures (describes how to request the credits) and Parties [4]. Before constructing a cloud SLA metamodel we analysed relevant digital SLAs to determine the accepted ways of specifying SLA concepts and structure. The WSLA (see Section 2.4.5) specifies the following main sections: Parties, Definitions and Obligations. Thus the development of the metamodel SLA is inspired by the WSLA: not only it is similar to the main components in SLA for Amazon SLA EC2 and S3, but it is also objective and precise (see Section 2.4.5).



Amazon EC2 Service Level Agreement

Effective Date: June 1, 2013

This Amazon EC2 Service Level Agreement ("SLA") is a policy governing the use of Amazon Elastic Compute Cloud ("Amazon EC2") and Amazon Elastic Block Store ("Amazon EBS") under the terms of the Amazon Web Services Customer Agreement (the "AWS Agreement") between Amazon Web Services, Inc. ("AWS", "us" or "we") and users of AWS' services ("you"). This SLA applies separately to each account using Amazon EC2 or Amazon EBS. Unless otherwise provided herein, this SLA is subject to the terms of the AWS Agreement and capitalized terms will have the meaning specified in the AWS Agreement. We reserve the right to change the terms of this SLA in accordance with the AWS Agreement.

Service Commitment

AWS will use commercially reasonable efforts to make Amazon EC2 and Amazon EBS each available with a Monthly Uptime Percentage (defined below) of at least 99.95%, in each case during any monthly billing cycle (the "Service Commitment"). In the event Amazon EC2 or Amazon EBS does not meet the Service Commitment, you will be eligible to receive a Service Credit as described below.

Definitions

- "Monthly Uptime Percentage" is calculated by subtracting from 100% the percentage of minutes during the month in which Amazon EC2 or Amazon EBS, as applicable, was in the state of "Region Unavailable." Monthly Uptime Percentage measurements exclude downtime resulting directly or indirectly from any Amazon EC2 SLA Exclusion (defined below).
- "Region Unavailable" and "Region Unavailability" mean that more than one Availability Zone in which you are running an instance, within the same Region, is "Unavailable" to you.
- "Unavailable" and "Unavailability" mean:
  - For Amazon EC2, when all of your running instances have no external connectivity.
  - For Amazon EBS, when all of your attached volumes perform zero read write IO, with pending IO in the queue.
- A "Service Credit" is a dollar credit, calculated as set forth below, that we may credit back to an eligible account.

Service Commitments and Service Credits

Service Credits are calculated as a percentage of the total charges paid by you (excluding one-time payments such as upfront payments made for Reserved Instances) for either Amazon EC2 or Amazon EBS (whichever was Unavailable, or both if both were Unavailable) in the Region affected for the monthly billing cycle in which the Region Unavailability occurred in accordance with the schedule below.

Monthly Uptime Percentage	Service Credit Percentage
Less than 99.95% but equal to or greater than 99.0%	10%
Less than 99.0%	30%

Figure 4.23: A screen shot for EC2 SLA [4]

Our main concern is to find matches between cloud consumer requirements and cloud provider offers based on the QoS parameters. Cloud providers use different terminology in their SLA offers, so comparing the same parameters based on the parameter name is not feasible. For example, two different terms (uptime and availability) are used to express the same concept in cloud SLAs Amazon EC2 and RackSpace. Also, the *Monthly Uptime Percentage* is calculated in terms of the unavailability of the EC2 instances. RackSpace defines the *Monthly Availability* in terms of the unavailability of the server (e.g. server error response or no response). Therefore, our metamodel varies from the WSLA. Figure 4.24 illustrates the first version of our cloud SLA metamodel.

The figure shows that the *SLAContract* defines a number of services and a *serviceObligation*. Moreover, the *ServiceTerm* defines a number of services and may have a *serviceObligation*. Each *Service* defines a number of QoS, which is used to define the QoS concept. Each QoS defines one or more measuredAttribute. Each Metric has attributes: name, type, value and time period. These attributes were defined to be used in the comparison process. Similar to the WSLA, we defined *MeasurementDirectives* and a *Function* that is defined to calculate and exchange metric values.

A model that conforms to this metamodel allows a QoS concept to be defined using different names. Thus it is possible to define the same QoS parameters using different names, e.g. uptime and availability (Amazon EC2 and RackSpace). Thus we refined our metamodel. In the following section we discuss the second stage of constructing a cloud SLA metamodel.

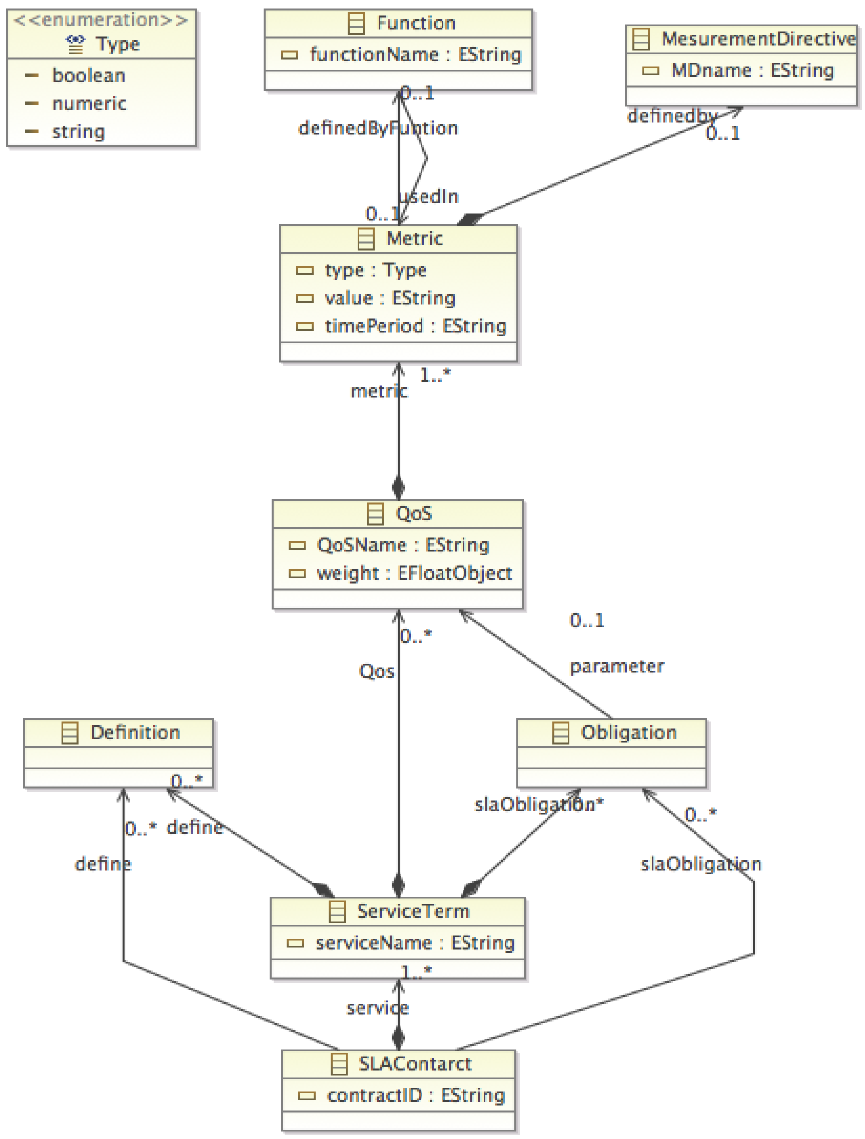


Figure 4.24: First version SLA metamodel

4.4.2 Revised SLA Metamodel

The cloud SLA metamodel was altered frequently during the construction process. We added new components to the metamodel, which required refinements. As discussed in the previous section, the terminology of QoS might have different names but similar meanings. The reason for differences might refer to the providers advertising needs, i.e., for emphasising the cloud provider's unique selling points. We specified the QoS parameters name, so that providers use this specific terminology. Another addition to the first version is the consumer requirements. As illustrated in the previous section we had one metamodel for a cloud SLA. After the first version (Section 4.4.1), the metamodel went through several refinements and testing operations.

For example, the service in metamodel 4.24, can be defined by the user using different names, e.g. the Amazon computing service is called EC2 [3] and the RackSpace computing service is called Cloud servers [21]. When we want to compare, e.g., *QoS* parameters, logically, they have to define the same service. Using different names makes the comparison infeasible. Figure 4.25 shows that a metamodel specifies three classes: *Computing*, *Storage* and *Networking*. For another example, assume that a model defines a QoS parameter name *availability* and another model defines *uptime* as a QoS name. A comparison operation will not compare the two parameters while they may have the same meaning (as discussed in Section 2.2.4.1).

The revised metamodel changed to two metamodels: one for SLA cloud provider offers and the second for cloud consumer requirements. This decision was made for the following reasons:

- As discussed in Section 3.4 Table 3.5 (**S5**), cloud consumers may want to provide their own preferences, which are not included in the cloud provider offers.

- Cloud consumers usually define high level parameters of QoS [174], while cloud providers may provide more details about QoS (in the SLA) to monitor the parameters. This is to avoid penalties in case of SLA violation [76]; for example, high level parameters such as SLA *availability* and low level parameters such as *upTime* and *downTime* [76]. Therefore the cloud provider SLA may include more details than the cloud consumer requirements.

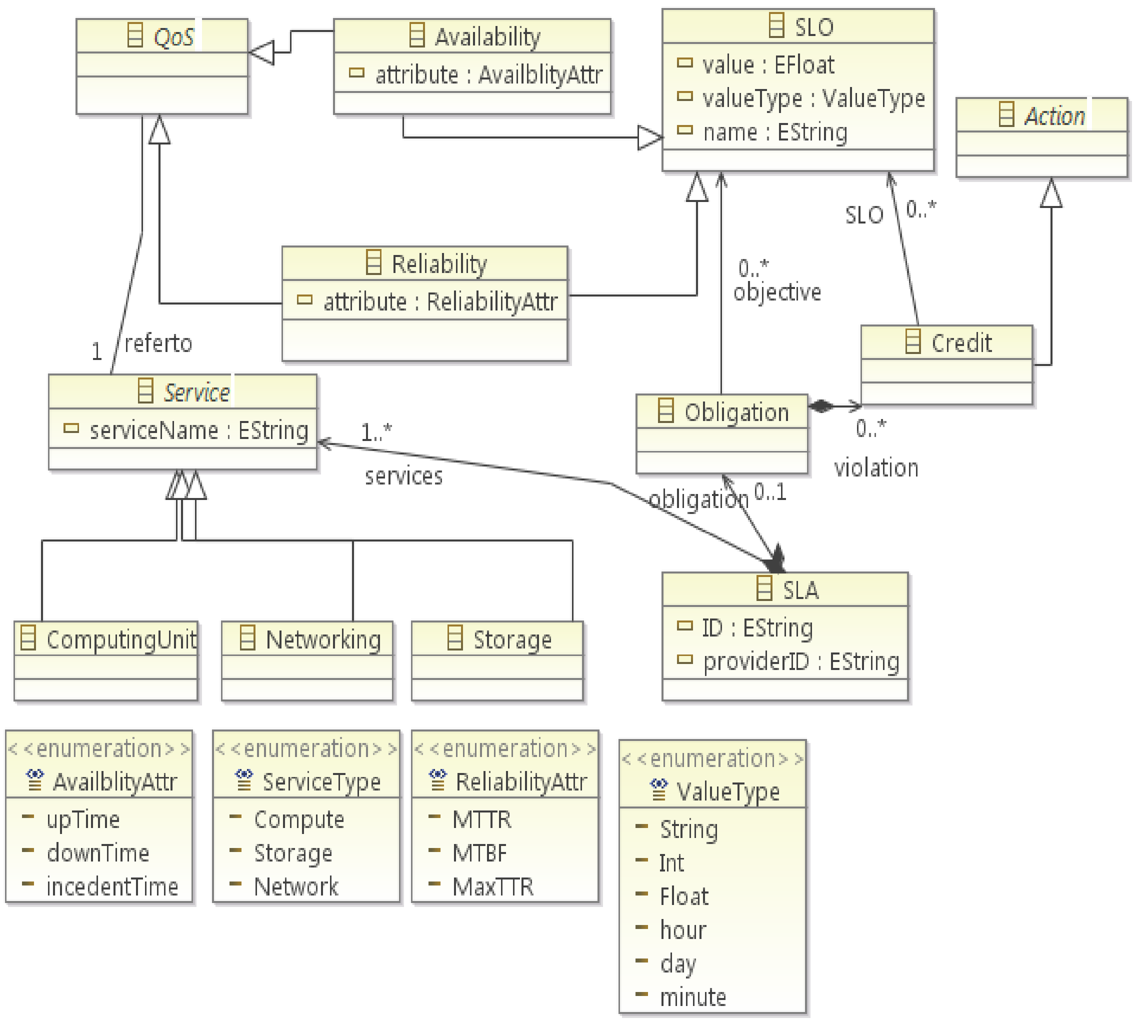


Figure 4.25: A provider SLA metamodel [37].

Figure 4.25 shows the second version of the metamodel for cloud SLA offers. The figure shows that an SLA defines different services. There are different categories of services, which are: Computing units, Storage or Networking, which are presented as subclasses of the Service Class to define the service type. IaaS clouds provide different services: Computing Power, Storage and Networking (Section 2.2.3.2) and the sub classes *ComputingUnit*, *Storage* and *Networking* represent these IaaS services. Furthermore, pre-defined cloud providers provide an SLA for each service, e.g. Amazon (EC2 SLA and S3 SLA) and RackSpce (Cloud Server SLA, Cloud Files SLA and Cloud Load Balancer SLA). The *Service* class refers to the *QoS* class. This QoS is a superclass of Availability and Reliability. These classes are defined to provide the QoS concept. As explained in (Section 2.1.1.1 equation 2.1) availability can be calculated in terms of uptime and downtime, which are described as low level parameters [76]. The figure illustrates that the attribute of *Availability* is called the *attribute* of type *AvailabilityAttr*. *AvailabilityAttr* is an enumeration that defines *upTime*, *downTime* and *incident-Time* (Section 2.1.1.1). Different attributes of *Reliability* include Mean Time TO Repair (MTTR), Mean Time Between Failure (MTBF) [180] and Max Time To Recover (MaxTTR). Thus a cloud consumer and cloud provider define their QoS concepts using specific terminology. Availability and Reliability have attributes: *value*, *valueType* and *name*. The enumerated type *AvailabilityAttr* specifies elements of the availability quality; for example, availability is defined in terms of uptime and downtime [135]. Finally, *Credit* is an action subclass, and an attribute of the *Obligation* class. Cloud providers usually define a credit request in their SLAs as an action. A consumer has to request the credit and provide evidence of the violation, e.g. Amazon EC2 [4].

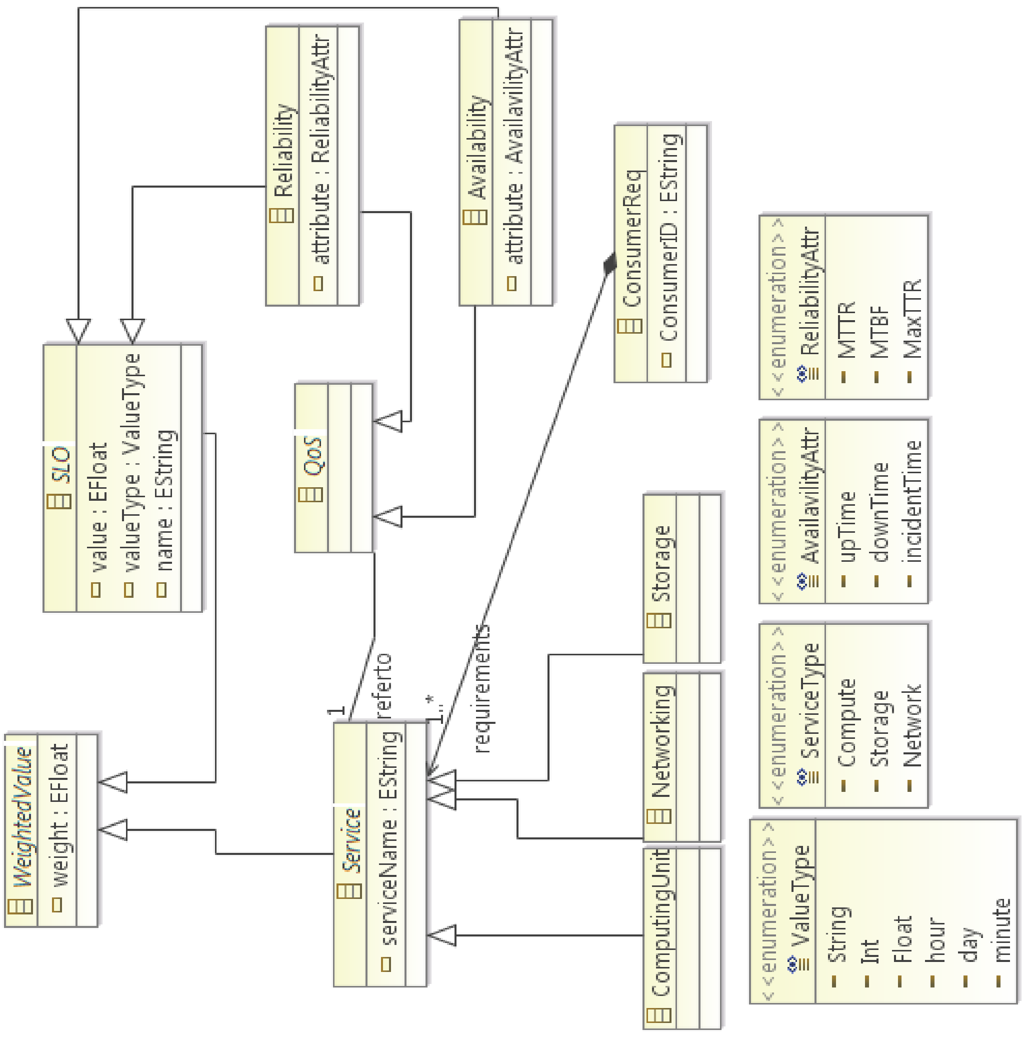


Figure 4.26: A consumer SLA metamodel [37].

Figure 4.26 illustrates the metamodel for consumer requirements. The metamodel is similar to the provider metamodel. The differences are: the *WeightedValue* class (which is not included in the provider metamodel), Obligation, Credits and Action classes (which are not included in the consumer metamodel, largely as a simplification). The WeightedValue class is included in the consumer metamodel to provide a means for consumers to provide their preferences with respect to different cloud attributes, i.e. add weight to the QoS parameters.

4.5 Summary

In this chapter, we described the cloud SLA metamodel. We first provided the general SLA metamodel which is composed of several packages. An example of a cloud SLA model with an object diagram was provided. This chapter presented a detailed description of each package, including the abstract syntax and an example model. As mentioned in Chapter 3, we originally envisioned the need for two metamodels - one for cloud SLA providers and another for consumers - but we ended up with one metamodel. Therefore a description of the early versions of the cloud SLA metamodels showed how the two cloud SLA metamodels for providers and consumers were similar and thus we ended up with one SLA metamodel. The next chapter explains the cloud SLA comparison process.

Chapter 5

A Comparison Process for Cloud SLAs

Chapter 4 introduced a metamodel for cloud SLAs. The metamodel is composed of several packages (such as CloudUnitSpec, Price and Services) and includes features related to QoS and cost. This metamodel was noted as one of the requirements in Table 3.6 in Section 3.4 in order to help to specify and better automate the process of selecting between cloud SLAs. This chapter presents the comparison process for cloud SLAs. The comparison process demonstrates both the core logic for matching SLAs (i.e., those parts that do not vary between different comparison scenarios) and the key variation points for different scenarios (e.g., for different degrees of precision in comparison).

5.1 Introduction

We can use the metamodel previously defined in Chapter 4 as the basis for SLA comparison. SLA comparison can be supported via the use of model comparison (or matching) rules. Comparison and matching is task- and language-specific: for example, there is no unique way to compare UML models - they can be compared in terms of UIDs, concept names, structural similarities, etc. (Section 2.3.3.2 discussed model comparison as a model management task in MDE). Similarly, there is no unique way to compare cloud SLAs; the specific comparison logic that we use should be dictated by the intended use of the result of matching consumer requirements against provider offers (Section 2.4 discusses the matching and similarity between service components and

QoS). In the comparison process that we present in this section we focus on finding matches and similarities between the elements of the cloud SLA models. The overall intention of our approach is to semi-automate the matching process of consumer requirements with cloud provider SLAs, to help consumers to select from amongst different cloud offers.

In our matching approach, we first match an SLA model for consumer requirements and an SLA model for provider offerings; this is the core part of our approach. We then propose three different matching algorithms: Optimal, Approximate and Name-based matching scenarios (as illustrated in use case scenarios in Sections 3.3.2.1- 3.3.2.3). These algorithms serve to provide different types of matching (of different accuracies) to support different scenarios.

This chapter starts with examples to illustrate the matching logic between two SLA models, then discusses the optimal match process in Section 5.4.5. Section 5.4.6 explains the Approximate match algorithm. The Name-based match algorithm is then discussed in Section 5.4.7.

After discussing the algorithms conceptually, we present an implementation based on standard model management tasks and tools, particularly Eclipse EMF and Epsilon. The matching implementation is divided into several ECL tasks. In each section we explain an ECL task, which focuses on matching different parts of the SLA models (Sections 5.5.1- 5.5.7, and 5.5.8). In each section, we first provide the matching implementation of the Optimal match algorithm, and then discuss the variations necessary to support the other matching logics. Finally we describe the metamodel for the results of the matching process (i.e. a trace metamodel). This metamodel – and its conformant models – are used to present the results to a consumer; we describe the uses of these models in the next chapter.

5.2 General Process of Cloud SLA Model Comparison

In this section we describe the general process of comparing cloud SLA models. In this comparison process we focus on finding matching elements in two SLA models. Any matched element is included in the output results. Before delving into the details, it is important to differentiate between the two phrases (*comparison* and *matching*) which will be used throughout this chapter. By comparison, we mean the total process of comparing a service requester model against a number of provider SLA models; while the matching process (whether described as a matching logic or algorithm) is the process of finding matches between two model elements.

Figure 5.1 describes the *comparison* process for comparing a service requester model against n provider SLA models. This comparison process is composed of n matching processes.

The comparison process starts when a service requester (e.g. the consumer) provides a cloud SLA model to be matched against a repository of SLA models (the repository was first discussed in the use case scenarios in Section 3.3) of different providers' offers. Assume a situation as illustrated in Figure 5.1. As such, we have a consumer model as input to the process, along with a repository of *n* SLA models. The input model is compared against *n* SLA provider models stored in the repository. Then, a matching process involving two models is invoked *n* times, with each providing an output model as a result. The output model is called a trace model, as it captures the links between different models.

As illustrated in the figure, each matching process can be composed of several sub-matching processes. Different elements of the cloud SLA, e.g. *QoSProperty* and *CloudUnit* (Section 4.3) can be matched using different matching logics, e.g. Optimal, Approximate and Name-based (see Section 5.4). How the matching is carried out and how these matching logics differ is explained in this chapter, but first we provide a motivating example in Section 5.3 to illustrate the idea of matching SLA elements.

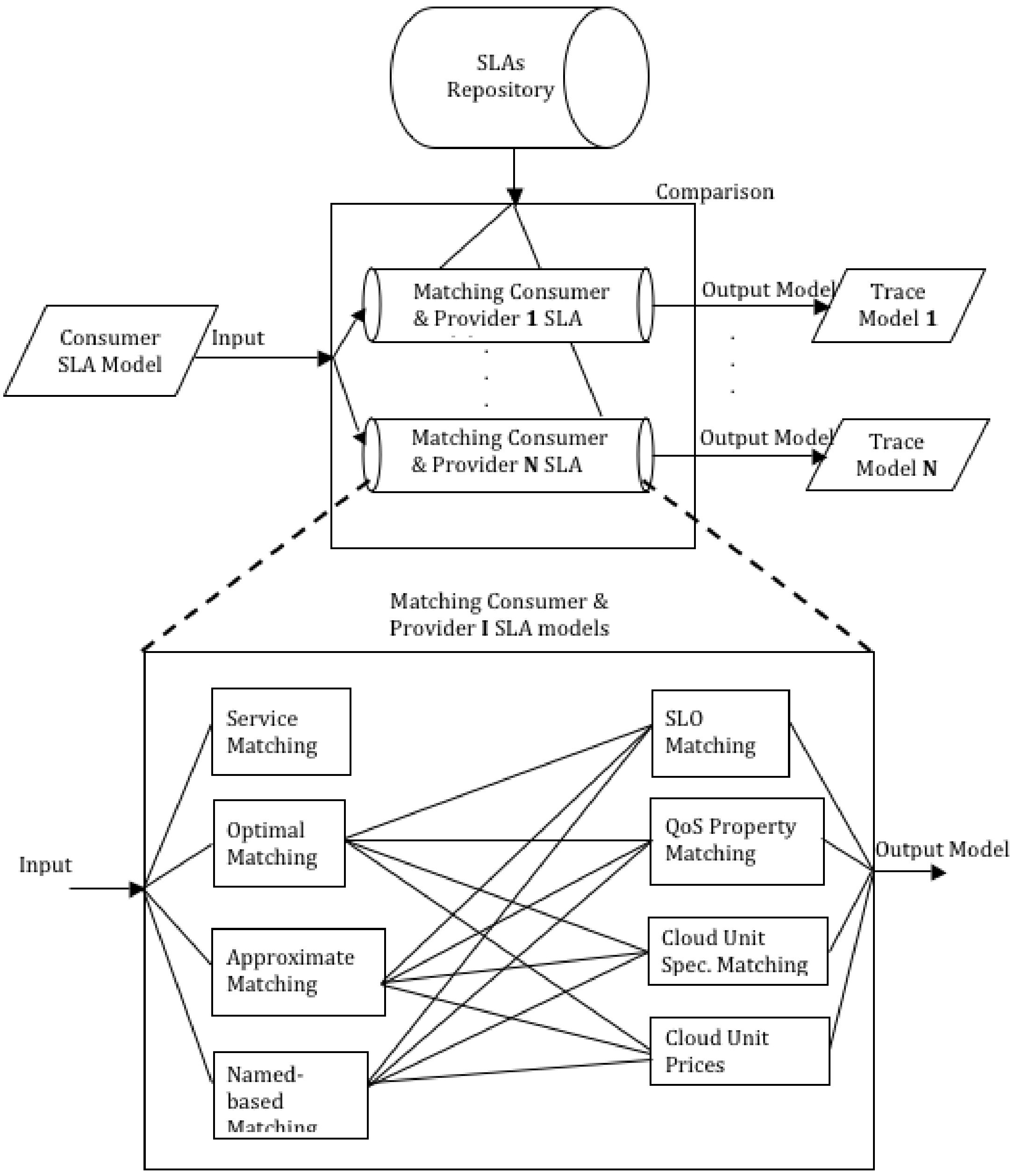


Figure 5.1: Comparison Process Architecture

5.3 Motivating Example

In this section we provide a motivating example to illustrate the matching process of two SLA models. In this section, assume that we have a consumer requirement together with *provider A* and *Provider B* models specified as illustrated in Table 5.1. The model elements in the table represent the elements of cloud SLA models, while the values of the consumer and provider models described in the table can be either elements (which may have attributes and references) or elements' attributes.

In this example, we illustrate which elements of two cloud SLA models can be matched, wherein the matched elements will be included in the output (trace) model. In comparing two models we compare the elements, their attributes and references. To compare these model elements, we have to consider the logic of matching SLA elements. To match two elements we define the left parameter as an element of the consumer model while the right parameter is an element of the provider model.

5.3.1 Which Elements of Cloud SLA Models Can be Matched?

In this example, we explain the matching of cloud consumer's requirements against SLAs offers from two providers (i.e. Provider A and B) (Table 5.1).

- Matching services: to find matching elements between two models (e.g. the consumer and provider models), we first match the *service* elements of the two models. As we see in the table, the consumer model defines two service elements: computing and storage, which match the provider model elements. In the context of the metamodel (see Section 4.3), we match the elements between the consumer model and the provider model.

- Matching QoS concepts: for example, to match Availability (in row 3 of Table 5.1) in the consumer model with Availability in a provider model, a match should be found first between the service, e.g. Computing, in both the consumer and provider models. Therefore, the availability (in row 3 of Table 5.1) of a computing service in both the consumer and providers models is matched against each other. However, the availability of computing in the consumer model is not matched with the availability of the storage service in the provider models (rows 3 and 6 in the table are not matched), because they are defined for different services; namely computing and storage.

- Matching QoS parameters: the *QoS* properties defined within Availability (*QoSTerm* class, see Section 4.3.1.1) can be matched with *QoS* properties defined within Availability in the provider model. The elements defined within the computing service in the consumer model (e.g. availability) are matched only with the elements defined within the same service (i.e. the Availability elements defined for computing) of the provider model (e.g. availability and uptime). In this case we match references, e.g. the *Value* element of both parameters. For example, the *Value* for availability and MTBF in the consumer model are matched with the uptime *Value* defined in the provider model.

- Matching values: to match two values, we have to match their units and then their values.

  1. Matching units: for example, theQoS property availability in the consumer model in row 3 is matched with the uptime defined in the provider A model, but not with the downtime in the provider B model, because the uptime unit is defined as a percentage (i.e. %), while the unit for the downtime is a time unit (i.e. *minute*).
  2. Matching values: the cloud metamodel has two types of value: *SingleValue* and *RangedValue* (Section 4.3). For example, the consumer model defines a response time as a single value which is a time = *2 minutes*, while B defines a range from *70* to *120* seconds. A single value element (*SingleValue* class, see Section 4.3.1.1) and a range value element (*RangedValue* class, see Section 4.3.1.1) can be matched. These two elements can be matched because they define a time using different time units (seconds and minutes). To match these two values, the value and unit of one of the elements should be converted into the other unit of the other element (e.g. 2 minutes should be converted into 120 seconds).

- Matching prices: to match two prices from the consumer and provider models, the two prices should define the same payment type (e.g. the price in the consumer model should match that of a yearly subscription in provider A's model, but not the on-demand payment price). To match two prices they should be the prices of two matched cloud units.

- Matching cloud units: in matching cloud units (*CloudUnit*, see Section 4.3.10), a match between the characteristics of the cloud unit (*CloudUnitSpec* Section 4.3.3.1) of the same type should be carried out, e.g. matching VM1 and VM2 of the consumer model with the VMs in provider A's models. There are four possible combinations in which VMs in the consumer and provider A's models can be matched: VM1 and VM1, VM1 and VM2, VM1 and VM1, and VM2 and VM2. The storage size of the consumer model is compared with that of the provider's models. Computing units are composed of different attributes, which are vCPU, size and storage as illustrated in the example. To match computing units the composite attributes should be matched first.

In the preceding, we discussed which two elements of two SLA models are considered in the matching logic. Now we explore how to match two model elements: when are such model elements alike or the same?

5.3.2 When are Model Elements Alike?

We return to the example illustrated in Table 5.1; there are a number of different elements defined in the consumer model. We first look at provider A and provider B to find the same elements. For example, the consumer model defines its location as the UK; the same location is defined in provider B's model. In this case, the match process returns true for these SLA model elements. Similarly, both the consumer model and provider A's model define a computing service and as such these model elements will be matched.

The availability value is defined in the consumer model as 99.99%, which does not equal the availability value defined in, e.g., provider A's model. In this case, can we say that the availability parameters in both models are not the same? It depends on the consumer to decide the degree of matching that is acceptable for the problem they are trying to solve. For example, if a consumer wants to find the *same or better* values, then, in this example the matching process returns false which means that the availability in both models is not the same. However if the consumer accepts the differences between the values of availability within, e.g. a threshold of 5%, then the matching returns true, which means that the 99.99% and 99.95% values are the same and so, accordingly, the availability in the consumer and provider A's models are the same. We call this matching *approximate* matching.

The response time value in the consumer model is defined as a *SingleValue* (see Section 4.3.1.1), whereas the response time in provider B's model is defined as a *RangedValue* (see Section 4.3.1.1). The matching process returns true (i.e. the value of *SingleValue* is the same as the value of *RangedValue*) if the value of *SingleValue* is within the range of values of *RangedValue*. If both values are defined as a range, a matching process returns true if they define the same or better range of values. For example, the storage size of the consumer and provider A's models is defined as a range from 1 GB to 20 TB, which is an *optimal* match. However, what if one range is a subset of the other range or both ranges intersect at certain values? Again, it is for the consumer to decide the required degree of matching. If, for example, a consumer allows an approximate matching between values, then if the difference between both the lower limits and upper limits is within the approximate value the match is true, and false otherwise. For example, the storage size of the consumer model does not match that of Provider B's model if the approximate value of the difference percentage is 5%, because the difference between the lower limit of the storage size in the consumer model and the lower limit of the storage size in provider B's model exceed the approximate value.

Matching VMs between the consumer and the provider models considers matching the values of the attributes. For the optimal match, VM1 is matched against VM1 and VM2 in provider A's model. The values of the attributes of VM1 are not the same at least and so the matching process returns false, while the matching process returns true for VM2. The consumer may prefer to find approximate matching with a difference of 5% between values. In this example, there are no matches (the matching process returns false) for consumer VM1 and VM2. A consumer may prefer to *relax* the matching degree again to find a similar VM as defined in the consumer model. For example, the name-based match returns true when for matching VM1 of the consumer model with VM1 of the provider A and provider B models, and with VM2 of both providers, while the matching process of VM 2 of the consumer model returns false when matched with all of the VMs in both provider models. This is because the name-based match ignores matching values and checks that the two elements have the same OS (i.e., in this example both VMs have the Linux OS). A consumer can also restrict matching cloud units to match the prices (i.e. using optimal matching to find the lower prices) of those matched VMs. In this case, a matching process returns true when it matches the VM1 of the consumer model and VM1 of provider A's model, while it returns false for the VM2 of the consumer model and VM2 of provides A's model. Another example of a name-based match is matching between MTBF, as defined in the consumer model, and downtime, as defined in provider B's model, where both elements have a time unit. The name-based match ignores the values; the units are similar but still the matching process returns false. MTBF is defined as, the higher the value the better, while, for downtime, the lower the value the better; therefore, the two values are not matched.

In Section 5.4, we discussed different matching logics precisely: Optimal, Approximate and Name-based.

5.3.3 Cloud SLAs Concepts Involved in the Matching Process

As illustrated in Chapter 4, our metamodel is constructed for both provider and consumer SLAs. In Chapter 3, we described different use cases to compare an SLA of a service requester and an SLA of a cloud provider. In this section we provide an example of provider and service requester SLA models to help explain the general matching process, and show the differences between the different matching logics.

The above examples demonstrated that there are different matching algorithms. We break the comparison down into several parts, the first of which is an overview of the key concepts that may be matched during the process. This is specific for cloud SLA matching.

- *Services:* in matching SLAs, we first match the service elements (e.g. computing, storage or networking). Cloud providers usually announce the services and the service unit specifications that are available.
- *Quality concepts:* i.e. elements that conform to classes that inherit the QoSTerm class (see Section 4.3.9), e.g. availability or reliability.

- *Cloud Unit Specifications:* in matching we assume that consumers require a specific unit specification and want to find a match with the provider offers. Usually consumers specify service units with the price that they are willing to pay.
- *QoS Parameters*: i.e. QoSProperty elements (see Section 4.3.9), these parameters define the QoS indicators and their values present at the level of service under question. These values are used as thresholds to monitor the quality of service and any occurrence violations.
- *Obligation:* this represents the degree of commitment with which the service providers will meet their offered and promised quality of service.
- *Prices and cost:* presumably, a consumer may want to compare the costs of different services. We treat the definition and modelling of price and cost as an initialisation step (i.e., prior to full SLA matching): the consumer should specify the quantity and price they are willing to pay for services in which they are interested (Section 6.4). This can be done either using a pre-defined model that captures prices, or interactively during the early stages of the matching process.

These concepts are matched based on model elements.

- Matching model elements. There are different options for matching two elements. It depends on how strict the matching algorithm is. Thus, we have different options for matching logics (i.e. Optimal, Approximate and Name-based, see Section 5.4). For example, two elements can be matched if they have the same name, two elements can be matched if they have the same name, same attributes and attribute values, or two elements can be matched if they have similar references.
- Matching references of two elements. This matching has different options. It is based on the number of references and whether or not the referenced elements are matched.

We specify some of the key options (or variation points) for matching model elements and combine these with the concepts of SLA matching to form the different matching logics (Section 5.4).

5.3.4 Matching patterns

Given each of these different concepts and concerns, we have a number of options available for comparing and matching. Given that there exist many different types of clouds (hybrid, public, private) and cloud consumers (individual end users, small businesses, multi-nationals), we attempt to remain generic, and provide mechanisms to support different types of matching, define the cloud parties' own matching semantics, and configure the degree to which a matching holds. It remains unclear which cloud SLA matching semantics is most suitable in different circumstances. As such, our matching approach allows different experiments to be run, and empirical data to be collected. Given that we aim to treat cloud SLAs – and the comparison process – as models and MDE operations, we have the means to repeat and easily modify such experiments. We have identified three patterns of matching that are applicable to cloud SLAs.

- *optimal value* match encoded in the requirements and provisions (e.g., at least the value of the QoS property requested is provided)
- *approximate value* match (e.g., the requested availability value is within 5% of what is provided); there are a number of variants of this.
- *name-based matching* (described in more detail later)

Other options for matching are possible, such as matching value elements if they have exact "values". For example, x matches y if $(x.value = y.value$ and $x.isPositive = y.isPositive = true)$ or $(x.value = y.value$ and $x.isPositive = y.isPositive = false)$. However, these are arguably special cases, so we focus on precise descriptions of *Optimal value matching*, *Approximate value matching*, and *Name-based matching* as a proof of concept of the different matching methods of cloud SLA models. The difference between optimal and approximate matching is that the former calculates the same or better model elements, including their attribute values. Approximate matching matches the model elements with the percentage differences in their attribute values. On the other hand, name-based matching matches model elements that have some similar attributes values or some similar references. In Section 5.4, we describe the different matching logics we used in our comparison process.

Table 5.1: Consumer requirements and the two providers' (A, B) SLA offers

	Model Elements	Consumer Requirements	Provider A's offer	Provider B's offer
1.	Location	UK	US, Ireland, Sydney and Tokyo	US and UK
	Computing SLA			
2.	Service	Computing	Computing	Computing
3.	Availability (Computing)	availability 99.99%, MTBF = 1 hour	uptime 99.95%	availability 99.95% downtime = 5 minutes
4.	Performance	Response time = 2 minutes	-	Response time = 70 150 seconds
	Storage SLA			
5.	Service	Storage	Storage	Storage
6.	Availability (Storage)	availability 99.9%	availability 99.9%	availability 99.9%
	Cloud Units and Prices			
7.	VM1 price	$262.80 per year (subscription)	$0.070 per hour (on demand) $0.05 per hour (annual subscription)	$0.12 per hour (on demand) $65.7 per month (subscription) $262.80 per year (subscription)
8.	VM1 Spec.	vCPU=1, Mem=4-7GB, Storage=32GB, OS = Linux	vCPU=1, Mem=3.75GB, Storage=4GB, OS = Linux	vCPU=2, Mem=2 GB, Storage=100GB OS = Linux
9.	VM2 price	$0.140 per hour	$0.140 per hour	$0.24 per hour
10.	VM2 Spec.	vCPU=3, Mem=4-7GB, Storage=32GB, OS = Windows	vCPU=2, Mem=7.5GB, Storage=32GB, OS = Linux	vCPU=4, Mem=4GB, Storage=200GB, OS = Linux
11.	Storage price	$60 per GB per month	$0.0300 per GB per month	$0.12 per GB per month
12.	Storage Size	1 GB - 10 TB	1 GB - 20 TB	10 GB - 1 TB

5.4 Matching Logic

One option for comparing cloud SLAs is to attempt to at least match the values held in the models, e.g., the requirements specified by the consumers and providers. We have called this “optimal matching”. An alternative is to *approximately match* the values held in the model. For example, two elements – availability1 = 99.95% and availability2 = 99.9% – are matched if the percentage difference between the two values = 10%. Another possible option is where consumers can accept some similarities (e.g. attributes names) between their requirements and what the providers offer, or are allowed to ignore certain concepts (like credit) and ask for an optimal match between other concepts. Our proposed solution for cloud SLA comparison allows a consumer to customise their matching requirements.

Our focus so far in the matching process has been on QoS parameters; this is largely because these concepts appear most frequently in each of the cloud SLAs we studied (see Section 2.4.3). In SLAs, QoS performance is monitored via these parameters and their given threshold values (SLO, see Section 2.1). Generally, QoS parameters are presented and defined differently, e.g., as service uptime, downtime or restoration time. How do we match QoS parameters in a cloud SLA? In general, we first check if the QoS parameters belong to the same service and represent the same quality (i.e. *QoSTerm*), then we provide different options for matching.

So the question now is: how do we actually carry out a comparison between consumer requirements and cloud provider SLAs? More precisely, what is the structure of the comparison logic? The comparison logic will take into account one (or more) of the matching options described above. This may be helpful when one matching process returns many results between two elements. The consumer has different options for finding a match to their requirements. To simplify the matching logic we break it down into several parts. Depending on the SLA concepts of the elements and the model’s structure for each one, each element may have a different matching algorithm. The matching algorithm of each element type is explained in a different subsection. We first illustrate the general cloud SLA matching algorithm in Section 5.4.1.

5.4.1 Matching the Cloud SLA Model

To illustrate the generic matching, we provide a diagram showing the main components of cloud SLA (see, Figure 5.2). As explained in Section 4.3, an SLA class has one reference to a service; that service is either computing, storage or networking. The generic comparison logic matches the consumer's requests with the provider's offers. We start by matching the service. This ensures that we are matching the same requested and offered services defined in the cloud consumer and provider SLA models. Once we have ensured that the required service matches (i.e. the SLA of the same service), we match the *QoSProperty* defined for this SLA, if required by the consumer. As shown in the figure, the service defines a number of *QoSPropertys*. Also, we can match the the SLO of two SLAs that define the same service. Another component of the cloud SLAs are the cloud units. These cloud units (see Section 4.3.4) are composed of cloud unit specifications and the price of the unit. A match algorithm of this cloud unit is provided in Section 5.5.2.

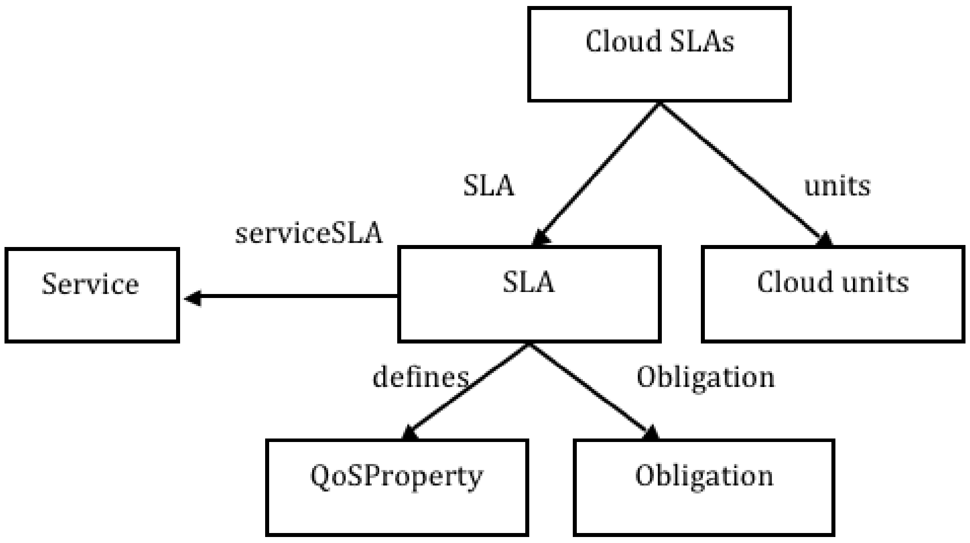


Figure 5.2: General architecture of cloud SLAs

Then, we apply a matching algorithm for the components of the SLA model, i.e. the definition and obligation. These two components define the QoS parameters and the SLO, which define the threshold values of the parameters and credits.

In this general matching process, we match the references of the cloud SLAs. The process of matching the SLA models consists of five main matching tasks as follows:

- Matching SLAs
- Matching units and prices.

Matching SLAs consists of matching composite elements.

- Matching services
- Matching the definition and QoS parameters
- Matching the obligation and SLO

The general structure of the matching logic is illustrated in Listing 5.1 in pseudo code. The providerSLA and serviceReq models are instances of the metamodel described in Section 4.3. Thus, we use metamodel class names in this algorithm.

Listing 5.1: Matching between a service requester SLA model and a provider SLA model

```
for each providerSLA model in a set of provider SLA models{
 p = providerSLA model;
 c = serviceReq model;
 for each element in c {
  for each element  in p {
   If c.SLA.service = p.SLA.service{
    apply matching rule for c.DefinitionTerm and p.DefinitionTerm
       ;
    apply matching rule for c.ObligationTerm and p.ObligationTerm
       ;
   }
   apply matching rule for c.CloudUnit and p.CloudUnit;
  }
 }
  return trace of matching elements     ;
}
```

The comparison logic in Listings 5.1 and 5.2 illustrates the logic of finding matches between the SLA model of a service requester (e.g. a consumer requirement) and a set of SLA models of cloud providers' offers (e.g. provider offers stored in a repository, see Chapter 3). The matching focuses on the *QoSProperty* (see Chapter 4.3.9.1) -

which is part of the service definition (see Section 4.3.9.1) and obligation (see Section 4.3.7) - as well as the cloud units defined in the contract Package (see Section 4.3.10). Listing 5.1 illustrates the comparison logic which finds the matches between two models *serviceReq* and *providerSLA*. It finds matches between the definition terms (*DefinitionTerm* class, see Section 4.3.9.1), obligation terms (*ObligationTerm* class, see Section 4.3.7.1) and cloud units (*CloudUnit* class Section 4.3.10.1). Any two matched elements are stored in the result (trace model, see Section 5.7). As explained in the metamodel (Section 4.3), each provider can define a number of SLAs in one model. Each SLA defines one service and a number of definition terms (including *QoSProperty*) and obligation terms. Therefore, in the algorithm we first match services and then match the *DefinitionTerm* and *ObligationTerm* as they are reference elements of the SLA model.

5.4.2 Matching QoS Properties

In this section, we focus on the *QoSProperty*, which is defined as part of the *DefinitionTerm* in the cloud SLA metamodel. An illustration of the structure of a QoS property is provided in Figure 5.3. The figure shows that a *QoSProperty* has a value and belongs to the *QoSTerm*. To match two QoS parameters (i.e. *QoSProperty*, see Section 4.3.9.1), first, they should describe the same QoS concept (i.e. associate with the same *QoSTerm*, see Section 4.3.9.1). The general matching checks the references of the *QoSProperty*. If they refer to the same elements, a match is found.

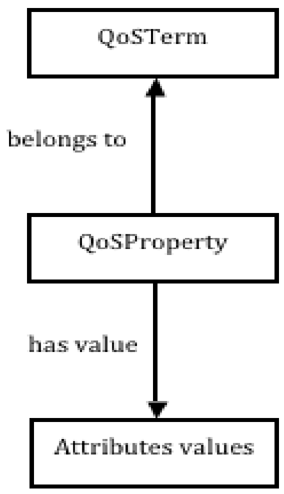


Figure 5.3: General architecture of QoS property specified in cloud SLAs

Listing 5.2 describes the calculation of matches for two QoS parameters *QoSProperty*, which belong to the same service and same *QoSTerm*. In this matching algorithm, we match the consumer and the provider *QoSProperty* element by matching reference elements *Value* and *QoSTerm*. A match is true if both references match. Assume that a consumer defines *downTime* = 1 hour as a *QoSProperty* of an *Availability* concept (i.e. *QoSTerm*) and a provider defines a *response time* = 2 minutes as a *performance* concept. These two parameters can be matched where they define a time unit and the lower their values, the better their quality. However, these two parameters define different concepts.

Listing 5.2: Matching Definition terms and QoSProperty in the service requester SLA model with the provider's SLA model

```
p = providerSLA;
c = serviceReq;
if c.service = p.service{
 apply matching rule for c.QoSProperty  and p.QoSProperty;
 apply matching rule for c.QoSProperty.value  and p.QoSProperty.
     value;
}
```

5.4.3 Matching Obligation term

An obligation is composed of an SLO and a credit (see Section 4.3.7), however we focus in this algorithm on the SLO. The SLO is composed of a reference to *QoSProperty* (which is defined in the definition term, see Section 4.3.9) and a value (representing a threshold value). To match two SLOs, we match the *QoSProperty* (Section 5.4.2) and the value defined in the SLO. The algorithm is illustrated in Listing 5.3. The algorithm applies a matching rule for the SLO elements. This rule matches the SLOs' references. One of the references refers to *QoSProperty* (see Section 5.4.2). The other rule matches the value reference defined in the SLO. A match is found if both rules are true.

Listing 5.3: Matching Obligation terms SLO in the service requester's SLA model and the provider's SLA model

```
p = providerSLA;
c = serviceReq;
if c.service = p.service{
 apply matching rule for c.SLO  and p.SLO;
 apply matching rule for c.SLO.value  and p.SLO.value;
}
```

5.4.4 Matching Cloud Units

Cloud units are part of the cloud's SLA model. Each cloud *SLAs* (see Section 4.3.10) defines a number of cloud units. Each cloud unit consists of the cloud unit specifications, price and location. The general structure of the cloud units is illustrated in Figure 5.4. The main elements in this figure are the cloud unit characteristics (*UnitSpec*, as seen in the figure, specified as a class *CloudUnitSpec* in the SLA metamodel) and the price of the unit (*Price*). The match algorithm matches the unitSpec of the consumer and provider and the related prices of these two units.

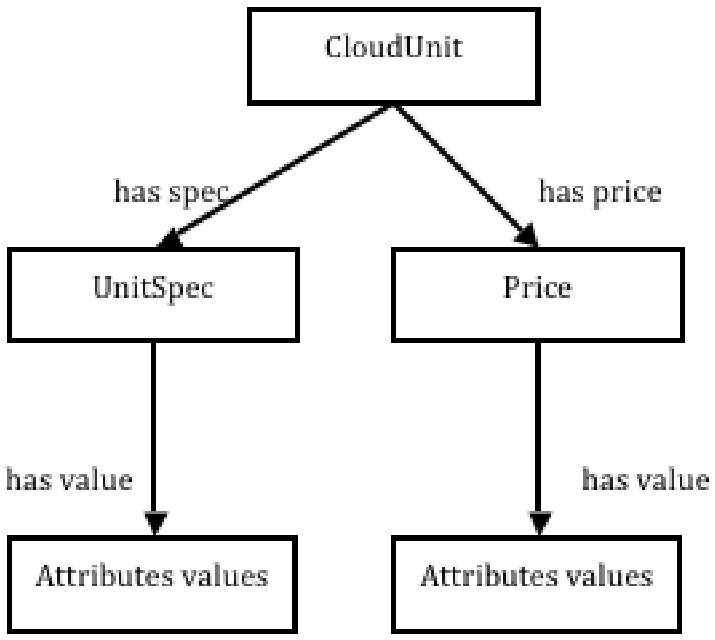


Figure 5.4: General architecture of cloud units specified in cloud SLAs

To match two cloud units, different rules are applied, as illustrated in Listing 5.4: a rule for matching cloud specifications, a rule for matching prices and a rule for matching locations. If all rules return true, then a match for cloud units between the consumer and provider has been found.

Listing 5.4: Matching cloud units SLO in the service requester's SLA model and the provider's SLA model

```
p = providerSLA;
c = serviceReq;
for each c.cloudUnit{
 for each p.cloudUnit{
  apply matching rule for c.cloudUnitSpec  and p.cloudUnitSpec;
  apply matching rule for c.price  and c.price;
  apply matching rule for c.location and c.location;
 }
}
```

The above comparison logic (Listings 5.1- 5.4) is generic in the sense that it can be encoded in different matching libraries and tools. The matching rules defined in these listings define the matching logic of each algorithm. We can instantiate the logic (and rules) further to include optimal matching or name-based matching rules. For example, a consumer can choose an optimal match for *CloudUnit* and a name-based match for *DefinitionTerm* (i.e. *QoSProperty*), where the similarity logic can be defined by the consumer. In the following sections, we discuss the different matching options in detail: optimal value matching (see Section 5.4.5), approximate value matching (see Section 5.4.6) and name-based matching (see Section 5.4.7).

5.4.5 Optimal Value Matching

Optimal value matching is based on the idea that for the two model elements to match, the values of their attributes (of interest to be matched) must be the same or better - we will go into further details on what 'better' means in different scenarios shortly. For example, the *QoSProperty* class has a *Value*; we say that two elements are matched if the attribute (e.g. *Value*) of the offered element is the same as the required attribute value or better. This particular example offers an important example inherent in many cloud SLAs: the QoSProperty defines the direction of the value, which means that the parameter is of better quality if it has a higher value – this pattern/style appears in many cloud SLAs.

The optimal value matching logic matches the values of the attributes of the element *Value*. For example, assume that the required availability is 99.90% and a predefined SLA offers a server uptime of 99.95%. In this case, we can say that a match is found, because both have the same units (e.g., %) and, for both parameters, the higher value the better the offered element (e.g., the offered value 99.95 is greater than the required value 99.90). The logic for matching values is seen in Listing 5.5, where each attribute of the *Value* type of two SLA models (consumer and provider) are checked if they are all equal or better, the match is then true. However, comparing two attributes is not just about checking the values. For example, the *Value* has an attribute: *unit*. In this case, we match the values of similar units. Therefore, the algorithm of the matching *SingleValue* is seen in Listing 5.6.

Listing 5.5: Optimal value matching logic of Value element attributes

```
return (servReq.Value.attribute1 = providerSLA.Value.attribute1)
    and .. and
(servReq.Value.attributeN = providerSLA.Value.attributeN);
```

Listing 5.6: Optimal matching logic of values

```
if (servReq.direction = providerSLA.direction){
 if (servReq.direction = "positive"){
  if (servReq.unit = providerSLA.unit)
   return (servReq.value <= providerSLA.value);
  else if (servReq.unitType = providerSLA.unitType)
   return (convert(servReq.value) <= convert(providerSLA.value));
  else return false;
 }
 else{   //if ( servReq.direction = "negative")
  if (servReq.unit = providerSLA.unit)
   return (servReq.value >= providerSLA.value);
  else if (servReq.unitType = providerSLA.unitType)
   return (convert(servReq.value) >= convert(providerSLA.value));
  else return false;
 }
}
else return false;  // servReq.direction <> providerSLA.direction
```

Listing 5.6 describes the logic of comparing two values of the element *SingleValue* (see Section 4.3.9.1). In this pseudo-code, we match two values *servReq* and *providerSLA*. Each value has a *direction*. The direction presents the parameter tendency, i.e. the higher the value, the better the quality (i.e. direction = *positive*) or the lower the value, the better the quality (i.e. direction = *negative*). We match two values if their directions are the same. Positive values are matched if the offered value (providerSLA.value) is equal to or greater than the required value (*servReq.value*). On the other hand, negative values are matched if the offered value (providerSLA.value) is equal to or less than the required value (*servReq.value*).

To match the two values (i.e. servReq.value = providerSLA.value), for which the consumer has defined the unit, the provider value should define the same unit. If the units are defined as the same type (e.g., a time unit), convert the value and unit into the other unit and match it. This is the case when we match a *SingleValue*. For a range value element (i.e. *RangedValue*), the logic is the same as for a single value, where the minimum and maximum values of both servReq and providerSLA are equal or better. In our metamodel, we defined different types of values, specifically *SingleValue* and *RangedValue*. In *SingleValue*, the type of value can be numeric or string. However, the

RangedValue has a numeric type only.

The case of matching two elements having different value types may occur. The optimal matching logic matches different value types, i.e. *SingleValue* and *RangedValue*. Listing 5.7 illustrates the matching logic we used in our implementation. Two ranged values are matched if the minimum and maximum of the service requester matches (equal or better) the minimum and maximum of the service provider's offer. For single and ranged values, we choose to compare them assuming the possibility of the service requester's fuzzy requirements. They are matched if the value of the *SingleValue* is within the range of values of the *RangedValue*. For example, assume we have 2 single values (x = 10 and y = 10) and two ranges of values (a = [10 .. 20] and b = [15 .. 30]). Matching x and y returns true. Matching a and b also returns true, because the minimum value in a is less than 15 and the same applies to the maximum values. If matching x and a, the value of x (i.e. 10) is within the range of a, while a match not found in case of matching required value b with the offered value y, because b does not include the value of y (i.e. 10) and the required value is higher than the offered value (i.e. minimum value 15 >offered value 10). The consumer and provider may define certain elements, e.g., storage size, and different value, i.e. single and range values. For example, a consumer defines a storage size of 500GB while a provider provides a storage size as a range value [1 - 1000GB]. We wish to provide flexibility in matching values, assuming the imprecise requirements of the consumer. This matching can be used for example, when a consumer requires *VM* with not more than *X* as a price.

Listing 5.7: Matching logic of *SingleValue* and *RangedValue*

```
if (c.direction = p.direction){
 if c.SingleValue and p.SingleValue
  return match(c.SingleValue.value,p.SingleValue.value)
 if c.RangedValue and p.RangedValue{
  return match(c.RangedValue.min,p.RangedValue.min) and match(c.
      RangedValue.max,p.RangedValue.max)
 }
 if c.SingleVlaue and p.RangedValue{
  return match(c.SingleValue.value,p.RangedValue.min) and match(c
      .SingleValue.value,p.RangedValue.max) or (c.SingleValue.
      value >= p.RangedValue.min and c.SingleValue.value <= p.
      RangedValue.max)
 }
 if  c.RangedValue and  p.SingleValue{
  return match(c.RangedValue.min,p.SingleValue.value) and match(c
      .RangedValue.max,p.SingleValue.value) or (c.SingleValue.
      value >= p.RangedValue.min and c.SingleValue.value <= p.
      RangedValue.max)
 }
}
else return false;

operation match(a, b)  : Boolean{
if (a.direction = "positive")
 return (a.value <= b.value);
else(a.direction = "negative")
 return (a.value>=b.value);
else return false;
}
```

5.4.6 Approximate Matching

Assume that in optimal matching (see Section 5.4.5), requires a *VM* with at most *X* price, and the *VM* match is found but not the price. This is because the provided price for the required *VM* is higher than the required price. For example the required price is 0.06$ while the offered is 0.07$. If for example, an approximate value (e.g. 20%) for the price is allowed and provided then a match for this *VM* may be found.

In approximate matching, a match between two different model elements is found if the difference between the two values does not exceed a specified value. The listing in 5.8 illustrates the logic of approximate *SingleValue* matching. The algorithm in Listing 5.8 defines an approximation value = x, then computes the difference between two values of the servReq and ProviderSLA. If this difference is less than or equal to the approximation value, a match is found between these two values. The difference between values can be decided by the consumer. For example, assume that a consumer defines availability as 99.99% and the provider defines availability as 99.0%. The values of the two parameters are unequal; however, using the approximate algorithm returns two parameters as matching if, for example, the consumer defines a difference between values of 10%. This match is used when consumers are not strict about the required values and allow some approximation. As shown in Listing 5.8, *servReq* has an approximation value which is assigned by the consumer. If both elements have the same direction, the algorithm checks the units it calls *approximateMatch* to calculate the difference between two values. If the two elements belong to the same unitType (e.g. time unit minutes and hours), it first convert the values and then calls the approximateMatch. It returns false if the two elements have different units.

Listing 5.8: Approximate matching logic of values

```
x = servReq.approximation; (approximation defined by consumer)
if (servReq.direction = providerSLA.direction){
 if (servReq.unit = providerSLA.unit)
  return approximateMatch (providerSLA.value,servReq.value,x);
 else if (servReq.unitType = providerSLA.unitType)
  return approximateMatch (convert(providerSLA.value),convert(
     servReq.value),x);
 else return false;
}
operation approximateMatch(a,b,x): Boolean{
  return (ApproxDifference(a,b) <= x);
}
```

In range value matching, the match returns true if the differences between the minimum and maximum values of the servReq and providerSLA do not exceed the approximate value (*servReq.approximation*) defined by the servReq. *(*ApproxDifference) calculates the approximate value of difference between two numbers (see Section 5.5.1.1). For example, assume that servReq defines a range value of [15-30] and an approximate value of difference of 10%. The match is true if the minimum and maximum values of providerSLA are within the range [15($\pm15*10\%$) - 30($\pm30*10\%$)].

Listing 5.9 illustrates the matching logic of different Value element types (SinglеValue and RangedValue). This algorithm matches approximately, single and range values together. If the single value is within the range of the ranged value then a match is found. If the single value is outside the range (i.e <min or >max) and the difference between the single value and min/max value is less than the approximate value, then a match is found. For example, assume a consumer defined storage size of 100 GB, an approximate value of 10% and a provider defined storage size of [40 - 90] GB. Although the consumer value is outside the range of the provider values, a match is found. The *approximateMatch* in Listing 5.9 is the same as that illustrated in Listing 5.8. As explained in Section 3.3.2.3, approximation matching may returns matched elements with values that are less or greater than the required values.

Listing 5.9: Approximate matching logic of *SingleValue* and *RangedValue*

```
if (servReq.direction = providerSLA.direction){
x = servReq.approximation; (approximation defined by consumer)
 if servReq.SingleValue and providerSLA.SingleValue
  return approximateMatch(servReq.SingleValue.value,providerSLA.
      SingleValue.value,x);
 if servReq.RangedValue and providerSLA.RangedValue{
  return approximateMatch(servReq.RangedValue.min,providerSLA.
      RangedValue.min,x) and approximateMatch(servReq.RangedValue.
      max,providerSLA.RangedValue.max,x);
 }
 if servReq.SingleVlaue and providerSLA.RangedValue{
  return ((servReq.SingleValue.value >= providerSLA.RangedValue.
      min) and (servReq.SingleValue.value <= providerSLA.
      RangedValue.max)) or (approximateMatch(servReq.SingleValue.
      value,providerSLA.RangedValue.min,x) or approximateMatch(
      servReq.SingleValue.value,providerSLA.RangedValue.max,x));
 }
 if  servReq.RangedValue and  providerSLA.SingleValue{
  return ((servReq.RangedValue.min <= providerSLA.SingleValue.
      value) and (servReq.RangedValue.max >= providerSLA.
      SingleValue.value)) or (approximateMatch(servReq.RangedValue
      .min,providerSLA.SingleValue.value,x) or  approximateMatch(
      servReq.RangedValue.max,providerSLA.SingleValue.value,x));
 }
}
```

5.4.7 Name-based Matching

In this match, we check only the attributes (primitive types) defined in the value type (i.e. *Value*, *SingleValue* and *RangedValue*, see Section 4.3.1.1). In our metamodel, *SingleValue* and *RangedValue* extend the abstract class *Value* (see Section 4.3.1.1).

This class defines the direction of the value, i.e. the higher the value, the better the quality, or the lower the value the better the quality.

In this match, for example, we match the QoSProperty of two elements with the same definition as a unit type attribute, regardless of their values. This match implies fewer restrictions than the other two matching logics (i.e. Optimal and Approximate). This algorithm can be used to find similar elements (e.g. *QoSProperty*) when, for example, consumers cannot find a match using the other two algorithms (i.e. Optimal and Approximate). We try to match all QoSProperty that are used to describe the same service (e.g *computing*) and same QoS (e.g *Availability*). As stated earlier, providers define their SLA in a *platform-specific* way. By using name-based matching, we try to make it possible for consumers to find QoS Properties that define the requested QoS (e.g *Availability*). As illustrated in Listing 5.10, this matching logic matches two elements if they have similar units and the value directions are the same.

For example, assume that the servReq model defines downTime (isPositive=false, value=15, unit=minute, type=float), and the providerSLA model defines MTTR (isPositive=false, value=1, unit=hour, type=float) as a property of QoS *Maintainability*. These two parameters are matched based on name-based matching logic, because both elements define a *false* value for the isPositive attribute, the same unit type (i.e. time unit) and are defined as a property of *Maintainability*.

Listing 5.10: Name-based Matching logic of values

```
if (servReq.direction = providerSLA.direction and servReq.QoSTerm
     = providerSLA.QoSTerm)
 return match(servReq.unit,providerSLA.unit);
else
 return false;
```

The differences between the name-based and the other matching logics are illustrated in Section 5.5, where we explain the implementation of each rule. The following section illustrates the implementation of the different rules for matching logic.

5.5 Implementation of Matching Logic in Epsilon

To investigate the thesis' hypothesis (see Section 1.5), we seek to provide a semi-automatic comparison of cloud SLAs using MDE principles. The logic described in the previous section is generic in the sense that it can be encoded in different matching libraries and tools. As proof of concept, we have implemented the matching logic in Epsilon (see Section 2.3.4.2), using ECL (see Section 2.3.4.4). Epsilon is a set of task-specific languages for managing models [11]. In ECL, a set of match rules with pre and post blocks form an *EclModule*.

In our proof of concept implementation, we create several comparison *EclModules*, each of which is implemented using one of the three comparison logics (i.e. Optimal, Approximate and Name-based Section 5.4). Each task performs a comparison of a set of elements in our model; the same matching logic applies to all of the elements in this module. For example, a module for optimal match values of *QoSProperty*, a module for approximate match values of *QoSProperty* and a module for name-based match of *QoSProperty*. Matching the whole SLA model is performed by a set of EclModules, each of which matches a part of the SLA model. Therefore, different combinations of matching logics can be used to match cloud SLAs. For example, in an *EclModule*, we can *import* a name-based matching module for *QoSProperty*, approximate matching for the cloud units and optimal matching for the prices to find matches between two SLA models.

In this section, we will focus on a general SLA metamodel comparison. In some comparison modules, there may be differences in comparison implementation between different matching logics (i.e. Optimal, Approximate and Name-based) and we will point out these differences, if any, when we explain the rules.

The basic pattern in ECL [11] is to compare models by matching pairs of elements. For this, we define a *matching rule* which has a unique name, *leftParameter* and *rightParameter*. In the *compare* part we define the matching logic for the specific parameters under comparison. The *matches()* operation is used implicitly to determine and invoke the (type-) appropriate matching rule to compare two elements. In the following sections we describe the implemented rules in each *EclModule* we have created. Each ECL rule defines the *left parameter* and *right parameter* to be matched. In our implementation the left parameter represents an element of the consumer model and the right parameter represents an element of a provider model.

5.5.1 Comparison of GeneralTypes

In this matching implementation we define a number of *rules* which take a service requester's (ConsumerSLA in the example) elements as *leftParameters* and a service provider's (ProviderSLA in the example) elements as *rightParameter*s. Different rules are created to match elements of types Value, SingleValue and RangedValue. First the element Value in the rule *matchAbstractValue* is compared to match the isPositive and the type attributes. This rule is extended by other rules: *matchSingleValue*, *matchRangedValue*, *matchSingleAndRanged* and *matchSingleAndRanged*. These rules are:

- compare a single value of ConsumerSLA with a single value of ProviderSLA
- compare a range value of ConsumerSLA with a range value of ProviderSLA
- compare a single value of ConsumerSLA with a range value of ProviderSLA
- compare a range value of ConsumerSLA with a single value of ProviderSLA

This combination of matching rules is defined to make the comparison task generic, and to match different definitions of values between different SLAs. To illustrate this, assume that a service consumer requires a cloud storage as a single value = x GB, and the storage specifications of the provider's offer state that the storage size is a range value equal to [a..b], where $a \leq x \leq b$. The values can be matched because x is defined within the range and both elements define the same unit type (i.e. storage size). If two units define the same unit types but using different units names (e.g. minute and hour), one of the values is converted to the other unit (e.g. 1 hour is converted to 60 minutes), and then the values are matched. These rules are invoked by other rules in different ECL modules to compare the values.

In our metamodel (see Section 4.3), we specified classes to define time units, size units, etc. To match elements that conform to these classes, we define other rules: *matchSizeType*, *matchPercentage*, *matchTimeType*, *matchTimeUnit* and *matchCurrencyType* to compare the units. For example, operation *c.timeperiod.matches(p.timeperiod)* in rule *matchPaymentPeriod* (Listing 5.15) invokes rule *matchTimeUnit*. This ECL task imports an EOL task, which is *ExactValueMatching*. This EOL task consists of a number of operations such as *convertSingleValues* and *convertRangeValues*, to calculate the values in case they have different units. Rule *matchLocation* is included to match the location(s) of the service (if required) by the consumerSLA with the location(s) provided by providerSLA. These rules are defined as an ECL module. The same ECL modules are created for the approximate and name-based matchings.

Listing 5.11 illustrates a match rule for the elements of the type *Value* of the SLA metamodel (see Section 4.3). The rule defines *ConsumerSLA!Value* as a left parameter and *ProviderSLA!Value* as a right parameter. The rule matches two *Value* elements of the consumer and provider models. The two values are matched if they define the same value type e.g. both define a value of the string type or both define a value of numeric type and both define the same value of the *isPositive* attribute.

Listing 5.11: Optimal matching implementation in ECL for the Value element

```
import "OptimalValueMatching.eol";
rule matchAbstractValue
 match c : ConsumerSLA!Value
 with p : ProviderSLA!Value {
 compare {
  if (not c.isPositive.isDefined() and not p.isPositive.isDefined
     () and not c.ValueType.isDefined() and  not p.ValueType.
     isDefined())
   return true;
  else if (c.isPositive.isDefined() and p.isPositive.isDefined()
     and c.valueType.isDefined() and  p.valueType.isDefined())
   return (c.isPositive = p.isPositive and c.valueType = p.
      valueType);
  else if(c.isPositive.isDefined() and p.isPositive.isDefined())
   return (c.isPositive = p.isPositive);
  else if (c.ValueType.isDefined() and p.ValueType.isDefined())
   return (c.valueType = p.valueType);
  else return false;
 }
}
```

The rule defined in Listing 5.12 shows the implementation of matching two single values; this rule can be specialised. In *ECL* the non-lazy rules are evaluated automatically in a top down fashion [11]. We declare this rule (i.e. *matchSingleValue*) as *lazy* so that it is invoked only by the *matches* command. The *SingleValue* class extends the *Value* class. This rule extends the rule explained in Listing 5.11. The rule defines the *SingleValue* of the consumer model as a left parameter and the same element (i.e. *SingleValue*) in the provider model as the right parameter. This rule checks if the left parameter value is equal to or better than the right parameter. If the units of the two values are defined, the rule then calls a *checkPositive* operation to match the two values. The instruction "if (c.unit.EClass.Name = p.unit.EClass.Name)" checks if the two values define different unit names but have the same unit type (e.g. hour and minute different unit names but both define unit type: time); then, to match the two values, the operation *checkPositive* is invoked.

This operation (i.e. *checkPositive*) checks if both elements define *isPositive* as *true* then the rule checks if the left parameter (i.e. required value) is less than or equal to the right parameter (i.e. offered value). If the two elements define a *false* value for the isPositive element then the match rule checks if the left parameter is greater than or equal to the right parameter.

A *convert* operation is called when the left parameter defines a different unit name from the right parameter's unit name. This operation converts a value; for example, 1 hour is converted into 60 minutes, so that the two values can be matched. If both elements define different unit types then the match rule returns false.

Listing 5.12: Optimal matching implementation in ECL for the SingleValue elements

```
@lazy
rule matchSingleValue
 match c : ConsumerSLA!SingleValue
 with p : ProviderSLA!SingleValue
 extends matchAbstractValue {
 compare {
  if (c.unit.isDefined() and p.unit.isDefined())
   if (c.valueType <> "String" and p.valueType <> "String" )
     if (c.unit.EClass.Name = p.unit.EClass.Name)
      return  checkPositive(c,p);
     else return false;
   else return (c.value <= p.value); //return stringMatching
  else return checkPositive(c,p);
  }
}
operation checkPositive(a:ConsumerSLA!SingleValue, b:ProviderSLA!SingleValue): Boolean{
 if (a.unit.isDefined() and b.unit.isDefined()){
  if (a.isPositive.isDefined()){
    if (a.isPositive = true){
     if (a.unit.matches(b.unit))
      return (a.value.asReal() <= b.value.asReal());
     else
      return a.value.asReal() <= convert(a.unit.unit, b.value, b.unit.unit);
    }
    else{
      if (a.unit.matches(b.unit))
       return (a.value.asReal() >= b.value.asReal());
      else
        return a.value.asReal() >= convert(a.unit.unit, b.value, b.unit.unit);
    }
  }
  else{
    if (a.unit.matches(b.unit)){
     return (a.value.asReal() = b.value.asReal());
    }
    else return false;
  }
 }
 else if (not a.unit.isDefined() and not b.unit.isDefined()){
   if (a.isPositive = true)
    return (a.value.asReal() <= b.value.asReal());
   else return (a.value.asReal() >= b.value.asReal());
 }
 else return false;
}
```

The implementation of matching two ranges of values is illustrated in Listing 5.13. Like the rule for *matchSingleValue*, this rule extends the matching rule illustrated in Listing 5.11 and is declared as *lazy*. If two parameters (i.e. left and right) define the same unit names or same unit types, then the checkPositive operation is called. The operation checkPositive is the same as that illustrated in Listing 5.12 and the difference is the ranged values matching. The operation checks the c.min against the p.min and the c.max against the p.max. This operation matches the *c.min* (i.e. the minimum value of a RangedValue element of the consumer model) with the *p.min* (i.e. the minimum value of a RangedValue of the provider model). If both values (min and max) are equal or better in both elements then the match rule returns true; otherwise, it returns false. In other words, the operation checks if the isPositive value is equal to true, then it checks if the c.min and c.max are equal to or less than the p.min and p.max. If the isPositive value is equal to false, then the operation checks if the c.min and c.max are equal to or greater than the p.min and p.max. If the two values have different unit names but define the same unit type (e.g. size or time), the *convert* operation evaluates the value (min or max) in terms of the other unit. A range value defined the values only as numeric (see Section 4.3.1.2).

Listing 5.13: Optimal matching implementation in ECL for the RangedValue elements

```
@lazy
rule matchRangedValue
match c : ConsumerSLA!RangedValue
 with p : ProviderSLA!RangedValue
 extends matchAbstractValue {
 compare {
  if (c.minUnit.isDefined() and p.minUnit.isDefined() and c.
      maxUnit.isDefined() and p.maxUnit.isDefined())
   if (c.minUnit.EClass.Name = p.minUnit.EClass.Name and c.
       maxUnit.EClass.Name = p.maxUnit.EClass.Name )
    return  checkPositive(c,p);
   else return false;
  else return checkPositive(c,p);
 }
}
operation checkPositive(a:ConsumerSLA!RangedValue, b:ProviderSLA!
     RangedValue): Boolean{
 if(a.maxUnit.isDefined() and a.minUnit.isDefined() and b.
      maxUnit.isDefined() and b.minUnit.isDefined()){
  if(a.isPositive.isDefined()){
   if (a.isPositive = true){
     if(a.maxUnit.matches(b.maxUnit) and a.minUnit.matches(b.minUnit))
      return (a.max.asReal() <= b.max.asReal() and a.min.asReal() <= b.min.asReal());
     else
      return(a.max.asReal() <= convert(a.maxUnit.unit,b.max,b.maxUnit.unit) and a.min.asReal() <= convert(a.minUnit.unit, b.min,b.minUnit.unit));
     }
     else{
      if(a.maxUnit.matches(b.maxUnit) and a.minUnit.matches(b.minUnit))
       return (a.max.asReal() >= b.max.asReal() and a.min.asReal() >= b.min.asReal());
      else
       return (a.max.asReal() >= convert(a.maxUnit.unit,b.max,b.maxUnit.unit) and a.min.asReal() >= convert(a.minUnit.unit, b.min,b.minUnit.unit));
     }
   }
   else return false;
  }
  else if(not a.maxUnit.isDefined() and not a.minUnit.isDefined() and not b.maxUnit.isDefined() and not b.minUnit.isDefined()){
   if (a.isPositive = true)
    return (a.max.asReal() <= b.max.asReal() and a.min.asReal() <= b.min.asReal());
   else return (a.max.asReal() >= b.max.asReal() and a.min.asReal() >= b.min.asReal());
  }
  else return false;
}
```

Listing 5.14 defines two rules. The first rule (i.e. *matchSingleAndRanged*) matches the *SingleValue* element as the left parameter and the *RangedValue* element as the right parameter, while the second rule (i.e. *matchRangedAndSingle*) does the opposite. Both rules have the same logic, so we will briefly explain only one of them. In the rule *matchSingleAndRanged*, the consumer element is a single value while the provider element is a range of values. As in Listings 5.12 and 5.13, the match rule executes the checkPositive operation if both elements define the same unit types. The checkPositive operation checks if the left parameter value is less than or equal to the maximum value of the right parameter when the isPositive = true. If isPositive = false then, the operation checks if the left parameter is greater than or equal to the minimum value of the right parameter.

Listing 5.14: Optimal matching implementation in ECL for the SingleValue and RangedValue elements

```
@lazy
rule matchSingleAndRanged
 match c : ConsumerSLA!SingleValue
  with p : ProviderSLA!RangedValue
  extends matchAbstractValue {
  compare{
   if (c.unit.isDefined() and p.minUnit.isDefined() and p.maxUnit
      .isDefined())
    if (c.unit.EClass.Name = p.minUnit.EClass.Name and c.unit.
       EClass.Name = p.maxUnit.EClass.Name )
     return checkPositive(c,p);
    else return false;
   else return checkPositive(c,p);
 }
}
@lazy
rule matchRangedAndSingle
 match c : ConsumerSLA!RangedValue
  with p : ProviderSLA!SingleValue
  extends matchAbstractValue {
  compare{
   if (p.unit.isDefined() and c.minUnit.isDefined() and c.maxUnit
      .isDefined())
    if (p.unit.EClass.Name = c.minUnit.EClass.Name and p.unit.
       EClass.Name = c.maxUnit.EClass.Name )
     return  checkPositive(c,p);
    else return false;
   else return checkPositive(c,p);
 }
}
```

In our metamodel, we specify different unit classes and enumerations such as *Time-Unit*, *TimePeriod* and *UnitType* (see Section 4.3.1.1). A value defined in an SLA model may define a unit; these may be, for example, size or time. When we match values, we must match the units first, to give meaning to the matching process. Therefore, we create rules to match the units, as explained in Listing 5.15. We provide an example of one unit.

Listing 5.15: Optimal matching implementation in ECL for the Unit elements

```
@lazy
rule matchTimeType
 match c : ConsumerSLA!TimeType
 with p : ProviderSLA!TimeType{
  compare : c.unit.name= p.unit.name
 }
@lazy
 rule matchTimeUnit
 match c : ConsumerSLA!TimePeriod
 with p : ProviderSLA!TimePeriod{
  compare : c.interval = p.interval and c.timeunit.matches(p.
      timeunit)
 }
```

5.5.1.1 Modifications for Approximate and Name-based implementation

This section discusses the modifications made in the implementation of the approximate and name-based matching for the GeneralTypes elements, specifically the *SingleValue* and *RangedValue* elements, which were explained in Listings 5.12- 5.14. We created the same rules for the GeneralTypes elements in different ECL modules: one for approximate and one for name-based matching, but some rules differ in terms of their details. The modifications are made for the approximate and name-based matching in rules *matchSingleValue*, *matchRangedValue*, *matchSingleAndRanged* and *matchSingleAndRanged* rules.

The approximate matching implementation invokes an EOL helper function, which is defined as an *approximate* operation to calculate the difference percentage in the numeric values of ConsumerSLA and ProviderSLA. There are many different approximate operations that can be defined; the proof-of-concept in this thesis makes use of formula 5.1 (but other formulae can be used) and two values (*x* and *y*). To illustrate the differences between optimal matching and approximate matching, Listing 5.16 is provided. As illustrated, two EOL modules are imported (named *ApproximateValue-Matching* and *ReturnApproxValueOperation*), which are different from the files imported in the optimal value matching ECL module (Listings 5.12- 5.14). The *ApproximateValueMatching* EOL module calculates the percentage difference between two values (Listing 5.17) and matches it with the approximate value (returned by the operation *returnApproximateValue*). The imported module *ReturnApproxValueOperation* defines the *returnApproximateValue* operation which returns a real value (approximate value) defined in the approximate model (Section 5.6).

In this module, we assign a variable of real type named *approxValue* with the approximate value returned by invoking the *returnApproximateValue* operation. In the rule *matchSingleValue*, an *approximate* operation is invoked when two values are matched. This operation is defined in the imported EOL module. The approximation is calculated as illustrated in equation 5.1. This equation calculates the percent difference as explained in [57].

Listing 5.16: Approximate matching of two SingleValues in ECL

```
import "ApproximateValueMatching.eol";
import "ReturnApproxValueOperation.eol";

.....
rule matchSingleValue
 match c : ConsumerSLA!SingleValue
 with p : ProviderSLA!SingleValue
 extends matchAbstractValue {
 compare {
   var approxValue : Real;
   approxValue = returnApproximateValue(c);
     ....
   if (c.unit.matches(p.unit))
    if (c.valueType <> "String" and p.valueType <> "String" )
     return approximate(c.value.asReal(), p.value.asReal(),
         approxValue);
```

```
      else return approximate(c.value,p.value,approxValue);
     else if (c.unit.EClass.Name = p.unit.EClass.Name)
       return convertSingleValues(c.value,c.unit.unit,p.value, p.
           unit.unit,approxValue);
    else return false;
           ....
  }
```

$$|(x-y)/((x+y)/2)| * 100 \tag{5.1}$$

As part of approximate matching, we need to explain how to match primitive types, e.g. numeric or string values. Listing 5.17 illustrates a part of the EOL module that calculates the percentage difference between two numeric values and an operation that calculates the percentage between two string values. This task also includes other calculations which convert a unit into a different unit, to compare the values. This EOL task is invoked by other ECL tasks to perform the calculations; for example, the EOL task applied to (hour, 1, minute) returns 60 minutes.

For the approximate matching of two strings, a distance algorithm such as [140] can be used. In this implementation and as a proof of concept, we use the Levenshtein Distance algorithm. Then we calculate the percentage distance between two values (one from the consumer model and the other from a provider model), as illustrated in the implementation in Listing 5.17.

Listing 5.17: Calculating difference percentages between two numbers in EOL

```
operation approximate(x:Real,y:Real,approxRate : Real):Boolean{
 return  (approxRate>=((x-y)/((x+y)/2)).abs()*100.0)
}
operation approximate(s1:String,s2:String,approxRate:Real):
    Boolean{
 return (approxRate>=((Simmertic.LevenshteinDistance(s1,s2).
     asReal()/(s1.length.asReal().max(s2.length.asReal()))*100.0))
     )
}
```

Listing 5.18 illustrates the same rules defined in the previous Listings 5.12 and 5.16 that matches the GeneralTypes elements (see Section 4.3.1.1). The Listing illustrates only the rule that matches two single values, to illustrate the changes made for the name-based matching. This rule returns true when the two values have the same units. As we can see, the rule does not check numeric or string values and does not import any EOL files. This rule, like other single value matched rules for optimal and name-based matching extend other rules that match other attributes, i.e. isPositive and valueType. As we explained in Section 5.4.7, in name-based matching, the focus is on matching the type of value rather than the value itself. For example, a consumer requests a response time = 60 seconds and define the isPositive = false, while a provider offer provides a response time = 3 hours with a isPositive = false. This match checks if both elements define the same unit type (i.e. time unit) and have the same isPositive value (i.e. false), then the matching rules returns true.

Listing 5.18: Name-based matching of two SingleValues in ECL

```
rule matchSingleValue
 match c : ConsumerSLA!SingleValue
 with p : ProviderSLA!SingleValue
 extends matchAbstractValue {
 compare {
  if (c.unit.isDefined())
   if (p.unit.isDefined())
    return  (c.unit.matches(p.unit) or (c.unit.EClass.Name = p.
        unit.EClass.Name));
   else return false;
  else return true;
 }
}
}
```

In name-based matching, the value, min and max attributes of *SingleValue* and *RangeValue* are not the main focus; instead, we focus on other attributes (i.e. unit and isPositive), so the rule *matchAbstractValue* remains the same as for the optimal matching. The differences are in the other *value* rules (i.e. *matchSingleValue*, *matchRangedValue*, *matchSingleAndRanged* and *matchSingleAndRanged*); as described in Listing 5.18, these rules match the unit attributes rather than the values attributes (of *SingleValue* and *RangeValue*).

In this implementation we define a helper operation to convert the units and their corresponding values to make them comparable. This is defined because different providers may define different units. For example, the size of Memory can be defined as 0.5 GB (which is equal to 512 MB). This is illustrated in Listing 5.19, which defines the operation: *convert*. This operation converts the *sizeUnitType* and *TimeUnit*, which are provided as a proof of concept that other types such as currency can be converted. This operation is used in both Optimal and Approximate matching, and is precisely defined in the imported EOL module.

Listing 5.19: Converting unit of values in EOL

```
operation convert(inUnit1 : Any, inValue2 : Any, inUnit2: Any) :
    Real{
 var convertedValue: Real = 0.0;
 if (inUnit1.EEnum.Name = "sizeUnitType"){
   ..
 }
 else if(inUnit1.EEnum.Name = "TimeUnit") {
   ..
 }
 return convertedValue;
}
```

The ECL rules explained in this section are invoked by other ECL modules. We kept each different matching logic in a separate ECL module to differentiate clearly between them and reuse them in other files.

5.5.2 Comparison of Cloud Unit Specifications

The cloud SLA metamodel specifies different cloud units: *Computing*, *Storage* and *Networking* (Section 4.3). These cloud units take part of the implementation for matching cloud SLAs, and so are explained in this section.

A comparison module is created to compare cloud units' characteristics. This module consists of rules for matching two cloud unit characteristics (e.g. VM). Again, we created three ECL modules containing the same rules, but each module contains different matching logics. The difference between Optimal, Approximate and Name-based come from the EOL programs that are imported, as explained in the previous section. In this section we illustrate a part of the cloud unit characteristics' optimal matching shown in Listing 5.20.

In this module, we define a rule named *matchCompUnitSpec*, which takes the consumer's *CompUnitSpec* as the *leftParameter* and a provider's *CompUnitSpec* as the *rightParameter* (Listing 5.20). This module imports the other ECL rules explained in Section 5.5.1, to match the values. There are three main rules: *matchCompUnitSpec* (Listing 5.20), *matchStorageSpec* (Listing 5.21) and *matchNetworkUnit* (Listing 5.22).

Listing 5.20: Matching cloud Computing units specifications in ECL

```
import "OptimalGeneralInfoMatchnig.ecl";
...
rule matchCompUnitSpec
 match c : ConsumerSLA!CompUnitSpec
 with p : ProviderSLA!CompUnitSpec{
  compare : (c.cpu.matches(p.cpu) and c.ram.matches(p.ram) and c.
      storageSize.matches(p.storageSize) and c.os = p.os and c.
      isVM =p.isVM)
 }
rule matchCPU
 match c : ConsumerSLA!CPUSpeed
 with p : ProviderSLA!CPUSpeed{
  compare : (c.numberOfCores = p.numberOfCores)
 }
```

Listing 5.21 defines a rule (i.e. *matchStorageSpec*) to match the storage required by the consumer and provided by providers. This rule is declared to be *greedy*, because the elements in this rule conform to an abstract class (which is *StorageSpec*). This rule defines a guard statement. This statement checks if the elements names are equal, because different storage types are specified in the SLA metamodel (see Section 4.3.5.1), which are: *CloudStorage*, *BlockStorage* and *BackUp*. If the elements in the guard statements have the same name (e.g. both are CloudStorage), then the rule proceeds to compare the sizes of the storage. The *matches* operation invokes the rule that matches the size.

Listing 5.21: Matching cloud Storage units specifications in ECL

```
@greedy
rule matchStorageSpec
 match c : ConsumerSLA!StorageSpec
 with p : ProviderSLA!StorageSpec{
  guard : c.EClass.name = p.EClass.name
  compare : (c.size.matches(p.size))
 }
```

The operation *c.cpu.matches(p.cpu)* invokes the matching rule *matchCPU* (Listing 5.20). Rules *matchRequest* and *matchDataTransfere* (Listing 5.22) are defined to match elements, that extends the Networking class (see Section 4.3.6).

Listing 5.22: Matching cloud Networking units' specifications in ECL

```
rule matchNetworkUnit
 match c : ConsumerSLA!NetWorkUnit
 with p : ProviderSLA!NetWorkUnit{
  compare : c.size.matches(p.size) and c.service.matches(p.
      service)
 }
rule matchRequest
 match c : ConsumerSLA!Request
 with p : ProviderSLA!Request
 extends matchNetworkUnit{
  compare : p.requestType.includesAll(c.requestType)
 }
rule matchDataTransfere
 match c : ConsumerSLA!DataTransfertype
 with p : ProviderSLA!DataTransfertype
 extends matchNetworkUnit{
  compare {
   if (c.toService.isDefined() and p.toService.isDefined() and c.
      toLocation.isDefined() and p.isDefined() )
    return (c.transferType.name = p.transferType.name and c.
       toService.matches(p.toService) and c.toLocation.matches(p.
       toLocation));
   else if (c.toService.isDefined() and p.toService.isDefined())
    return (c.transferType.name = p.transferType.name and c.
       toService.matches(p.toService));
   else if (c.toLocation.isDefined() and p.toLocation.isDefined()
      )
    return (c.transferType.name = p.transferType.name and c.
       toLocation.matches(p.toLocation));
    }
  }
```

5.5.3 Comparison of Cloud Prices

Pricing is part of the cloud SLA model; as such, it can be included when matching two SLAs. This section describes the implementation of matching the prices of two cloud units. In the ECL implementation (Listing 5.23), we define a rule named *MatchPrice*. A explained in Section 4.3.8, the price model defines a payment period, value, priced unit (*price per*) and a currency. To match values, we match all of these attributes. As illustrated in Listing 5.23, we first define a *guard* which matches the payment periods. If they match, the *compare* condition is evaluated. This *guard* statement is used to check that the compared prices are for the same payment period, because the prices for on demand or subscription services vary.

Usually, prices in cloud computing are defined per time or size, e.g. VM is priced per hour and Storage is priced per size (GB). The difference in the approximate matching is the *compare* statement. Instead of checking the equality of two values, we check that the differences between two values lies within an accepted range (i.e. an approximate operation, see Section 5.5.1). In the name-based matching, we do not compare the price value, but match the payment periods (i.e. on-demand or subscription) and the price per unit (i.e. time or size). Name-based matching assumes that the service requester is unsure about the price or, for example, want the lowest price for a specific payment type and period (e.g. on demand or subscription).

Listing 5.23: Matching prices in ECL

```
rule MatchPrice
 match c : ConsumerSLA!Price
 with p : ProviderSLA!Price{
  guard : c.paymentPeriod.matches(p.paymentPeriod)
  compare: c.priceValue = p.priceValue and c.pricePer.matches(p.
     pricePer) and   c.currency.matches(p.currency)
 }
```

5.5.4 Comparison of Cloud Unit and Price Implementation

This section describes the implementation of matching the *CloudUnit*, which relates the matching of cloud unit specifications (see Section 5.5.2) and the matching of prices (see Section 5.5.3). In the unit and price comparison implementation in ECL (Listing 5.24), we defined a rule to compare the *CloudUnit* (see Section 4.3.10), which associates the location price and cloud unit specifications. Therefore, we created a rule to match CloudUnits, which invokes the ECL rules explained in Sections 5.5.1- 5.5.3. This rule matches the cloud unit defined by the ConsumerSLA as a combination of invoking three matching rules. Different options are available for this matching rule by mixing different matching logics of the invoked rules. For example, an optimal match can be invoked for both the cloud unit matching and price matching; an approximate match can be invoked for both the cloud unit matching and price matching.

Listing 5.24: Matching cloud unit in ECL

```
rule matchCloudUnitAndPrice
 match cUnit : ConsumerSLA!CloudUnit
 with pUnit : ProviderSLA!CloudUnit{
  compare : cUnit.hasSpec.matches(pUnit.hasSpec) and cUnit.price.
      matches(pUnit.price) and cUnit.location.matches(pUnit.
      location)
 }
```

5.5.5 Comparison of Cloud Service Implementation

In implementing the matching of cloud SLA models, the ECL rules can match any two elements. In the SLA models, these are defined as the left parameter and the right parameter, e.g. availability from the consumer and the provider models. However, these two elements (e.g. availability) may be defined under two different services (e.g. computing and storage). Therefore, we create a rule to match the services and then match other elements which are related (e.g. the QoSProperty) in the model. This section presents the implementation of matching services which is part of the SLA model matching. To match the SLA models, we first match the service defined by the SLA model. We define a rule to match the services defined in two SLA models (e.g.the consumer and provider models). This rule, as illustrated in Listing 5.25, matches the element names, which are: Computing, Storage and Networking (see Section 4.3.2). This rule is imported by other ECL modules that match the SLA (see Section 5.5.8) and QoS (see Section 5.5.6). This rule is the same for the three matching logics.

Listing 5.25: Matching cloud services ECL

```
@greedy
rule matchServices
 match c : ConsumerSLA!Service
 with p : ProviderSLA!Service {
  compare : c.service.EClass.name =  p.service.EClass.name
 }
```

5.5.6 Comparison of Cloud QoS Parameters' Implementation

This section presents the implementation of the matching QoS parameters that are a key concept in cloud SLAs. To match two QoS properties, we need to match the value attributes and the QoS concept that is defined by. In our SLA metamodel (see Section 4.3), the *QoSPropety* class associates a number of *QoSTerm* classes. In the matching implementation (see Listing 5.26), we created a rule named *matchQoSTerm* that matches two *QoSPropety* elements by matching the QoS concepts. Rule *matchQoSTerm* matches two QoS concepts (i.e. *QoSTerm*) and returns true if they define the same QoS concept and the same service. This rule is invoked by another rule named *matchQosProperty*. The rule *matchDefintionTerm* returns true if the *QoSProperties* are matched and the values of the *QoSProperties* elements are matched by invoking two rules: *matchQoSProperty* and matching value rules (Sectionr̃efsub:compGeneral). The changes made in the approximate and name-based matching implementation are EOL programs that are imported, as described in Section 5.5.1.

Listing 5.26: Matching QoS properties ECL

```
....
 rule matchDefinition
  match c: ConsumerSLA!Definition
  with p : ProviderSLA!Definition{
   compare : c.terms.matches(p.terms)
  }
```

```
rule matchDefinitionTerm
 match c : ConsumerSLA!DefinitionTerm
 with p : ProviderSLA!DefinitionTerm {
  guard : c.define.matches(p.define)
  compare : c.define.value.matches(p.define.value)
 }
rule matchQoSTerm
 match c : ConsumerSLA!QoSTerm
 with p :  ProviderSLA!QoSTerm {
  guard : c.belongsTo.EClass.name = p.belongsTo.EClass.name
  compare : (c.EClass.name = p.EClass.name)
 }
rule matchQosProperty
 match  cq : ConsumerSLA!QoSProperty
 with  pq : ProviderSLA!QoSProperty{
  compare : (cq.qosTerm.exists(e | pq.qosTerm.exists(ep | e.
      matches(ep)))
 }
```

5.5.7 Comparison of Cloud Obligation Implementation

This section provides a description of the implementation of matching Obligations in cloud SLA models. Obligations are part of the cloud SLA model (see Section 4.3).

The Obligation is defined in our SLA metamodel as term(s). Therefore we define a rule named *matchObligation* as in Listing 5.27 to compare the obligations. This rule invokes another rule named *matchObTerms* to compare each term. To match SLO, we match the values by invoking value matching (see Section 5.5.1). The difference in implementing approximate or name-based matching is the importing of the ECL task (see Section 5.5.1) for the approximate or name-based matching.

Listing 5.27: Matching obligation terms in ECL

```
...
rule matchObligation
 match c: ConsumerSLA!Obligation
 with p : ProviderSLA!Obligation{
  compare : (c.terms.matches(p.terms))
 }
```

```
rule matchObTerms
 match c : ConsumerSLA!ObligationTerm
 with p : ProviderSLA!ObligationTerm {
 guard : c.parameterValue.EClass.Name = p.parameterValue.EClass.
     Name
  compare : (c.slo.matches(p.slo) and c.violation.matches(p.
      violation))
 }
@lazy
rule matchSLO
 match c : ConsumerSLA!SLO
 with p : ProviderSLA!SLO{
  compare : c.value.matches(p.value)
 }
@lazy
rule matchCredit
 match c : ConsumerSLA!Credit
 with p : ProviderSLA!Credit{
  compare : c.creditValue.matches(p.creditValue) and c.creditType
       = p.creditType
 }
```

5.5.8 Comparison of Cloud SLA Implementation

This implementation explains one way to relate other rules, described earlier (see Sections 5.5.1 and 5.5.4- 5.5.7) to match two SLA models.

In this implementation, we compare two SLA models. We created a rule named *matchSLAs* to match *SLAs* elements, which invokes the *matchSLA* rule (see Listing 5.28). Rule *matchSLA* invokes other rules described earlier (i.e. Sections 5.5.1, 5.5.3 and 5.5.5- 5.5.7). In this implementation, we compare all of the cloud SLA elements. This is illustrated by the operation *import* in the Listing. By running this *ECL* task, two SLA models are compared to match the same elements in both models. The result of the comparison rules is stored in a result model, which is discussed in Section 5.7.

Listing 5.28: Matching two cloud SLA models ECL

```
import "Service.ecl";
import "OptimalObligationMatching.ecl";
import "OptimaltUnitAndPrice.ecl";
import "OptimalMatchQoSProperty.ecl";
import "OptimalGeneralInfoMatchnig.ecl";

...
rule matchSLAs
 match c : ConsumerSLA!SLAs
 with p : ProviderSLA!SLAs {
  compare : c.serviceSLA.matches(p.serviceSLA)
 }
rule matchSLA
 match c : ConsumerSLA!SLA
 with p : ProviderSLA!SLA {
  compare : c.service.matches(p.service)
  do {
   if (c.define.isDefined() and p.define.isDefined())
    c.define.terms.matches(p.define.terms);
   if (c.obligation.isDefined() and p.obligation.isDefined())
    c.obligation.terms.matches(p.obligation.terms);
  }
 }
```

5.6 Approximate Values Model

Sections 5.4.6 and 5.5.1.1 introduced an approximation value. A consumer may need to be able to define different approximation values for the required SLA elements. We support this in our approach by using an *approximation model*. This model contains different approximation values for different SLA model elements. Adding the notion of approximate values to the approximate matching process requires an additional metamodel. The approximate model consists of a number of items, each of which has a value (i.e. approximate value) and refers to an element in the SLA model. These approximate values are used as threshold values for the difference between two element values in cloud SLA model (see Listings 5.16 and 5.17). The abstract syntax is described in the following section.

5.6.1 Abstract Syntax of the Approximation Model

The metamodel of the approximate model is illustrated in Figure 5.5 and consists of:

- *Approx*: a class that has an association with a number of *items* of type *ApproxItem*. This class holds set of items that are assigned an approximate value from a service requester.
- *ApproxItem* class has a string attribute named *name* , a real attribute named *approxValue* and a reference named *itemRef*. The *itemRef* holds the object of SLA model that is assigned the approximate value.

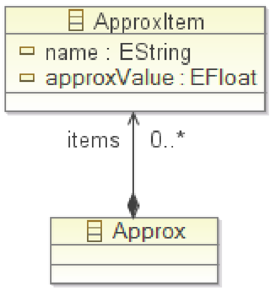
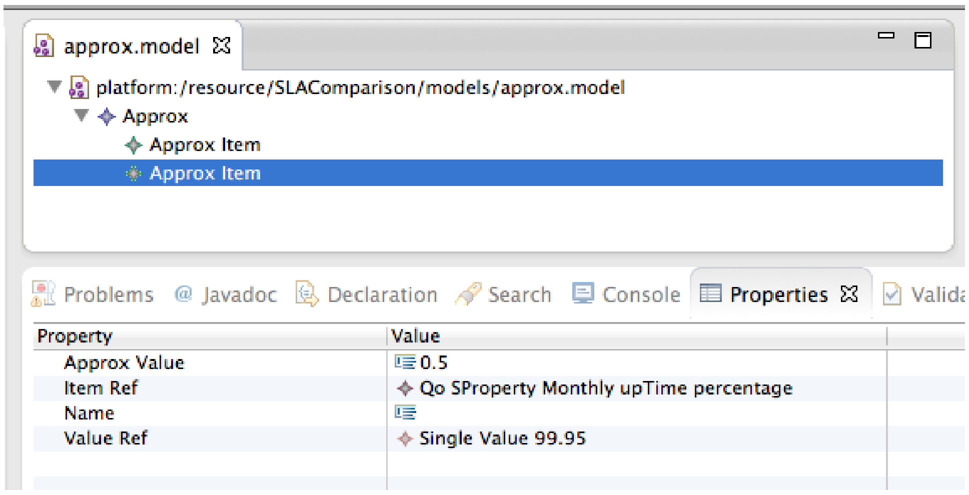


Figure 5.5: Approximate metamodel

5.6.2 Example of Approximate Model

We provide an example of the approximate model to be consistent with the explanation of a metamodel introduced in Chapter 4. As shown in Figure 5.6, a model consists of two *approxItems*. One of the *approxItems* refers to the uptime QoS parameter and its value equals 99.95%. The approximation value of this QoS parameter is assigned to 0.5%. This approximation value is used in the approximate matching process, as explained in Sections 5.4.6 and 5.5.1.1.



platform:/resource/SLAComparison/models/approx.model
- Approx
  - Approx Item
  - Approx Item

Property	Value
Approx Value	0.5
Item Ref	Qo SProperty Monthly upTime percentage
Name	
Value Ref	Single Value 99.95

Figure 5.6: An example of approximate model

5.7 Comparison Results Metamodel

Section 5.2 presented the general architecture of the comparison process: matching two SLA models produces an output model. The output model holds the results of the matching rules, which were previously explained in Section 5.5. Since the objective of this work is to enhance the automation of the cloud SLA selection process, and help cloud consumers to select the appropriate offers from cloud providers, the output of the matching process is presented as a model which holds the matches that were found with SLA provider offers. This resulting model can thereafter be processed and manipulated, e.g., to present the results to consumers in different forms or styles (depending on their needs).

When two elements are matched, the output model holds a reference to both matched elements of a consumer model and a provider model. As explained (in Section 5.5), if the matching rules match a *leftParameter* with a *rightParameter*, then the rule returns *true* or *false*; thus the output results of a match rule holds references to *leftParameter* and *rightParameter*.

The output model thus consists of elements, each of which is created when a match rule returns true. This element holds references to the two matched elements.

The output results (i.e. Trace) metamodel is illustrated in Figure 5.7. The trace class defines a number of *items* of type *TraceItem*. Each item (i.e. *TraceItem*) associates a left and right references. These two references hold matched objects from the two compared models.

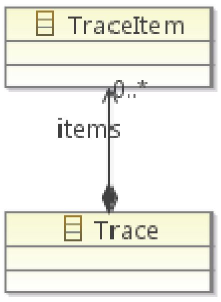


Figure 5.7: Trace metamodel

As presented in Section 5.5.8, in the ECL module, we can define a *post* block which is executed after evaluating the match rules in an ECL module. In the matching rules discussed earlier in this chapter, we included the *post* statements in the ECL module. This *post* block creates the output model.

In the post part of the ECL, in Listing 5.29, we created and externalised a trace model to store the matching elements. The *matchTrace* operation of ECL consists of the number of matches that holds the references to the compared objects [11]. We store the objects that are assigned as the same objects (the match has a *true* value). These statements in the *post* block create elements of the *Trace* metamodel (i.e. *TraceItem*) that holds references of the left and right parameters.

Listing 5.29: Storing the results of the matching rules in ECL

```
post {
 var item : Any;
 var trace = new TraceModel!Trace;
 for (item in matchTrace.matches){
  if (item.matching) {
   var traceItem : new TraceModel!TraceItem;
   raceItem.left = item.left;
   traceItem.right = item.right;
   trace.items.add(traceItem);
  }
 }
}
```

5.8 Summary

In this section, we described the different matching logics of the SLA model comparison. We implemented SLA model comparison that finds the matching elements in the model. Three matching logics were proposed: Optimal, Approximate and Name-based logics. The matching process is implemented using ECL. The implementation of the SLA model comparison is divided into different ECL tasks (modules), each of which match part of the SLA model separately. Each of the compared parts of the SLA is implemented to match the three matching logics, each of which is created in a separate ECL task. To compare two SLA models, the implementation makes it possible to mix the three matching logics with different parts of the SLA elements. A model is created as a result of the comparison task. This model holds the matched elements from the process of matching two SLA models. The comparison tasks may produce multiple results. To make it easier for consumers to compare the matched models, we use a weighting process. The weighting process allows consumers to accumulate the results from the matching process and assign weights to the requirements. This will be discussed in the next chapter.

Chapter 6

A Conceptual Framework for SLA Model Comparison

6.1 Introduction

Chapter 4 discussed different SLA metamodels and models, while Chapter 5 discussed the matching logic for comparing two SLA models. A model that can be used to represent the outcome of the matching process was discussed in Chapter 5. In this chapter we discuss how the results can be presented in ways that can support the consumer (i.e. service requester) in making decisions. As discussed in Chapter 5, the comparison tasks compare a consumer SLA model with a number of provider SLA models. Consumers may have preferences with regard to which SLA elements they want to consider and emphasise in the outcomes of the comparison process. This chapter discusses a so-called consumer preference model and explains how this preference model, and the comparison results (trace models), can be usefully combined in a way that helps consumers when selecting an offer. These models, tasks and overall process can be supported via a workflow of model management tasks, and these are explained in this chapter.

This chapter is organised as follows: first, we explain the conceptual framework for the comparison of cloud SLAs using MDE principles. Then, we explain how the preferences of the consumer/service-requester can be formed as a weight model (Section 6.3). Following this, we introduce a cost model, which can be used to define, for example, the number of cloud units and total cost associated with a requested service of each provider. In addition, we discuss the process of analysing the comparison results, which may require consumer interaction (Section 6.4). The chapter also describes a *decision matrix*, formed from trace models (Section 6.6). The decision matrix is derived to be used in cloud SLA selection problems. After that, the final section discusses the visualisation of these trace results (Section 6.7) to demonstrate proof-of-concept that trace results can be presented in a user-friendly manner.

6.2 Conceptual Architecture of SLA Comparison

As discussed (in Chapter 1), this thesis investigates how to help cloud consumers to make choices regarding cloud SLAs through the semi-automation of the comparison and selection process using MDE principles. In chapter 5, we described the algorithm of matching two models (e.g. consumer and provider) as a major aspect part of the comparison process (Section 5.2). This comparison process matches a consumer SLA model with *n* provider SLA models by repeating the matching process (Chapter 5) *n* times. The outcome of the comparison process is a number of models as described in Section 5.2. Cloud consumers may want to make decisions based upon one of those output models which better matches their requirements. This chapter discusses the outcome of the comparison process and how this can be further managed (using MDE tasks) to help consumers to make decisions. For example, how these outcome models (i.e. trace models Section 5.7), when combined with the cloud consumers preferences, can help them to make better decisions when selecting the cloud provider's offer.

In a case where a selection or decision must be made between multiple attributes (criteria) and multiple alternatives (i.e matched models from different provider offers) (called the MCDM problem) [193, 195], a MCDM matrix is derived to solve this problem. Therefore, in this thesis we propose a MCDM matrix metamodel. Then, the trace model is transformed into the matrix model. The MCDM matrix metamodel is described in Section 6.6. Before we explain the MCDM matrix, we illustrate a conceptual architecture of the framework of the comparison and selection process in Figure 6.1.

In Figure 6.1, the processes are model management tasks, the input of a task is a model and the output is a model. Some tasks may require a consumer interaction which is presented in the figure as dotted arrows. This framework, as shown in the figure, consists of multiple model management tasks, e.g. model comparison and transformation, and a number of models (that conform to different metamodels). The main components are:

- Comparison process: this comparison process includes different matching logics using ECL tasks, which are matching different components of cloud SLA models as describe in Chapter 5.
- Cloud provider SLA models: these models can be instantiated using the cloud SLA metamodel (Chapter 4), these models are assumed to be stored in the SLA repository.
- Cloud consumer SLA model: this model is instantiated by the service requester using the SLA metamodel (Chapter 4).
- Outcome models of the comparison process (trace models): the outcome model of each matching process which conforms to the trace metamodel is described in Section 5.7.
- Create weighting model: this process accepts a consumer mode and preferences for the elements of the cloud SLA from a consumer as a *weight*, to produce a weight model. A detailed description is provided in section 6.3.
- Data Analysis Process: this process is presented in case a consumer interacts with the trace models of the comparison to, for example, add some constraints to the trace model. This is further discussed in Section 6.5. Therefore, the output of this process is like the input model that conforms to the trace metamodel, (i.e. the new trace model- Figure 6.1).
- Create cost: in this simple process, the quantity of each unit defined in the consumer model is requires the price of each unit as defined in the provider model, so we can calculate the expected total cost for each provider according to their requirements. The outcome of this process is a cost model. This is further explained in Section 6.4.

- MCDM matrix: this process forms trace models, a consumer model and cost models as a decision matrix. This matrix is usually formed in such cases to aid in finding a better offer. This is further explained in Section 6.6.
- View matching results: this process presents a way in which the results of the matching process can be visualised. This is explained in Section 6.7.

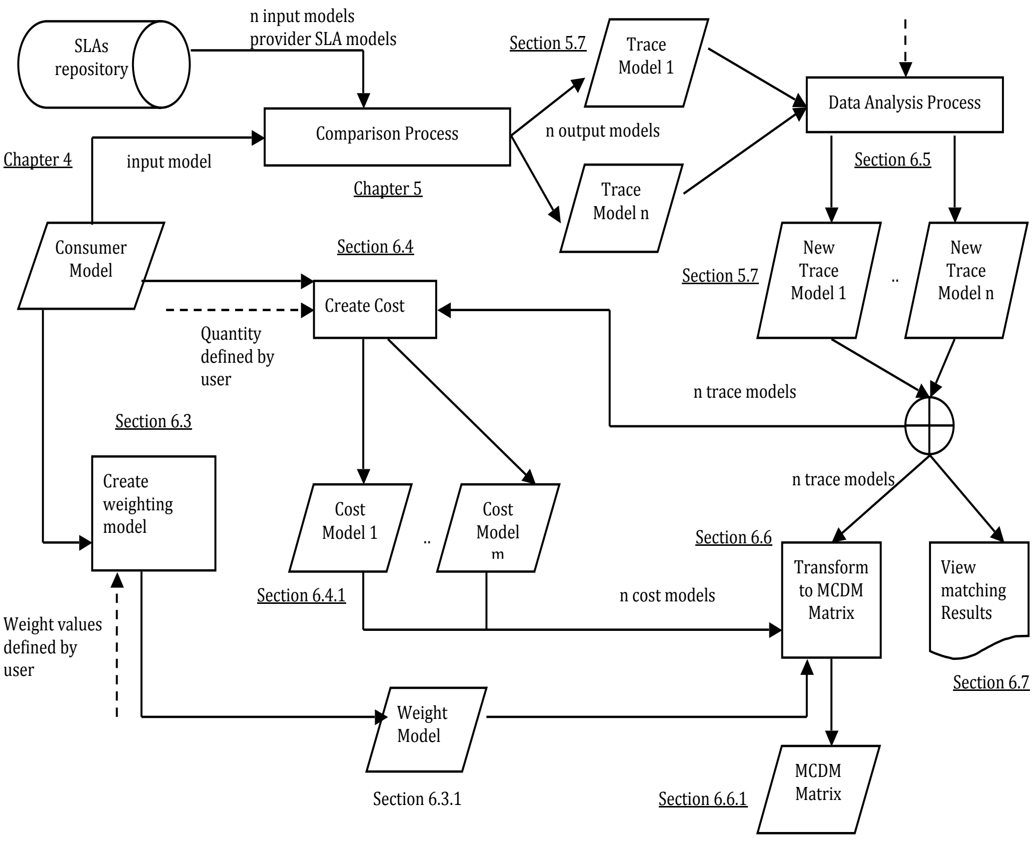


Figure 6.1: Architecture of cloud SLA comparison and selection Process

Now we discuss each component, shown in Figure 6.1 in different sections (except matching cloud provider SLA models and cloud consumer SLA models, which are discussed in Chapter 4. The comparison process and trace models are discussed in Chapter 5). We start with the process of creating a weighting model.

6.3 Create a Weighting Model

A consumer searching for an acceptable cloud SLA may not value all SLA concepts and constructs equally; for example, availability may be markedly more valuable than credit for downtime. To address this, we introduce a weighting model to allow consumers to assign weights to their specific requirements. Adding a notion of weights to the matching process requires an additional metamodel.

This process requires consumer interaction to provide weights for the elements defined in the consumer models. The process assigns elements defined in the consumer model with a weight and forms a weight model. Each weight model consists of a number of items, each of which has a weight and refers to an element in the consumer SLA model. The abstract syntax of a weight model is described in the following section.

6.3.1 Abstract Syntax of Weighting Model

The metamodel consists of (Figure 6.2):

- Weight: a class that has an association with a number of items of type *WeightedItem*. This class holds a set of items that are weighted from a service requester.
- WeightedItem: a class that has a string attribute named *name* and real type attribute named *weight*. It also has a reference to an object named *itemRef*. The weight is used to specify the service requester by adding a weight value to the concepts of the SLA model. The *itemRef* holds the object of the SLA model which can be used for further processing, as illustrated in Section 6.6.

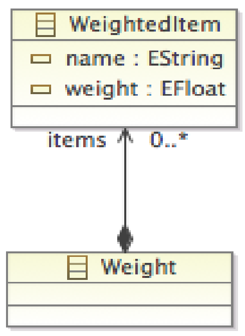
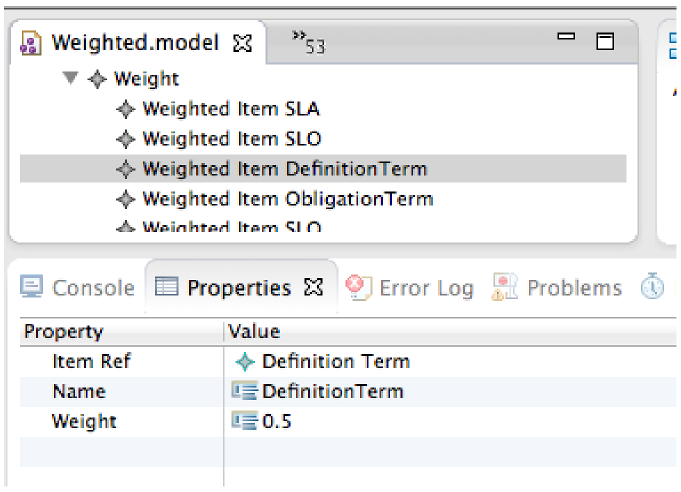


Figure 6.2: A weight metamodel

6.3.2 Example of a Weighting Model

To be consistent with the explanation of metamodels introduced in Chapter 4, we provide an example of the weight model, as shown in Figure 6.3. The weighted item may refer to any class in the SLA model. It is the choice of the service requesters to add weights to the classes of the SLA model to suit their requirements. To create this model, an interaction with a service requester is needed, i.e. consumers should provide weight values for the SLA model elements. Figure 6.3 illustrates the weighting model which consists of a number of *WeightedItems*, presenting SLA model objects: *SLA*, *DefinitionTerm*, *SLO* and *ObligationTerm*. For example, the weighted item *DefinitionTerm* has a weight value = 0.5 and has a reference to *DefinitionTerm*, as defined in the consumer metamodel.



Weighted.model

Weight
- Weighted Item SLA
- Weighted Item SLO
- Weighted Item DefinitionTerm
- Weighted Item ObligationTerm
- Weighted Item SLO

Property	Value
Item Ref	Definition Term
Name	DefinitionTerm
Weight	0.5

Figure 6.3: An example of the weight model

The process of creating a weight model, using elements defined in the consumer model and weight values, is illustrated in Listing 6.1. This process is partially automated created using an EOL module. In this proof of concept study, weight elements are created to act as consumer weights. Each weight element has a weight value. A weight value can be assigned to any element defined in the consumer model. In creating the weighting model process, a number of questions can be raised: Which elements in the consumer SLA should be considered in the weighting process? Should a group of elements, which conform to the same class, have the same weight value? We believe it is the consumer's responsibility to decide which elements are assigned a value. Listing 6.1 describes the algorithm used to create a weight model in pseudo-code.

The inputs of this process are *consumerModel* and cost model which is explained in the next section (6.4), and the outcome is a weighting model.

Listing 6.1: Assigning weights with SLA elements

```
input = consumerModel;
input = costModel;
create new weightModel;
for each element in conusmerModel{
 if (input=hasWeight)
  input weight;
  create a new weightedElement;
  weightedElement.itemRef=element;
  weightedElement.weight=weight;
  add weightdElement to weightModel;
}
```

6.4 Creating and Calculating a Cost Model

As several studies suggest [83, 119], the cost of the service is an essential part of decision-making. A cost model is created when a consumer requires a cost to be a factor in the decision-making process. The cost is the expected cost of consuming cloud services during a specified time period. This cost depends on the prices of the service units that are defined in the provider's SLA offers. It also depends on the quantity of units that are required by a consumer. The quantity is not included in the cloud SLA metamodel. A consumer interaction is expected to provide the quantity. We create a cost model from the consumer requirements (i.e. cloud units) and the quantity of each. In this section, we discuss two processes: firstly, creating a cost model based on the quantities provided by the consumer (Section 6.4.1), then calculating the cost based on the prices in the provider offer (Section 6.4.2).

6.4.1 Cost Model

The cloud SLA metamodel does not specify quantities; thus, we create a simple metamodel that specifies the quantity of each cloud unit as defined in the consumer the metamodel. To create a cost model, we first explain the abstract syntax of the cost model. The abstract syntax is described in the following section.

6.4.1.1 Abstract Syntax of Cost model

The abstract syntax of cost metamodel, as illustrated in Figure 6.4, consists of:

- UnitItemCost: this class specifies an integer attribute *numberOfItems* which is used to define the required quantity of a specific cloud unit. Another attribute specified by this class is the *price* which define the price of the cloud unit. This class specifies a *total* attribute which defines the total cost of a specific cloud unit

- Allunits: this class specifies an attribute *totalCost* which defines the total cost of all units required by the consumer. It specifies a reference *item* to refer to *UnitItemCost*.

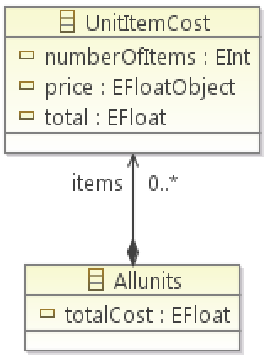


Figure 6.4: A cost metamodel

6.4.1.2 Creating Cost Model

In this study, we assign quantities to create a model which consists of cloud units and their prices from a provider SLA model and the quantity required by consumer. First, we assume that a consumer provides a quantity for each cloud unit as defined in the consumer SLA model; thus a cost model as explained in listing 6.2 is created. This listing creates a cost model using the EOL module. This cost model consists of a number of items. Each item is assigned a number (presents quantity) and a reference to a cloud unit defined in the consumer model. The inputs of this algorithm are *consumerModel* and the quantities required by the consumer (e.g. consumer interaction). The outcome is a cost model holding quantities and refers to the cloud units.

Listing 6.2: Create a cost model for consumer model using EOL

```
new costModel;
input consumerModel;
for (element in consumerModel.cloudUnits){
 new costItem;
 costItem=element;
 costItme.quantity=input quantity;
 add costItem to costModel
}
```

A cost model is created for each provider model matched with the consumer model. As illustrated in Listing 6.3, the inputs of this module are: the *consumerCostModel* (created from Listing 6.2) and a *traceModel*, and the output is a *providerCostModel*. The *item* in the *providerCostModel* is assigned a quantity from the quantity of *element* in *consumerCostModel*. This is done by selecting the *item traceModel* that has a reference to the *element* and assigns the *providerCostModel.item* to the *item.right*.

Listing 6.3: Create cost model for provider offers

```
input consumerCostModel;
input traceModel;
new providerCostModel;
for each element in cosumerCostModel{
 if element.exists in traceModel.item{
  new providerCostModel.item;
  providerCostModel.item.quantity=element.quantity;
  providerCostModel.item=select.traceModel.right(element=
      traceModel.item.left);
 }
}
```

The *traceModel* may contain a unit in the consumer model that is matched with more than one unit from the provider model. In this case, a user interaction is required to eliminate the repeated cloud unit from the matching model, which is discussed in Section 6.5. In the next section, we discuss how the cost may be calculated.

6.4.2 Calculating the Cost

The cost metamodel is created to calculate the cost of the cloud units for each provider. This cost refers to the consumption price of the service in a time period. The metamodel specifies the amount or quantity of service units required. For example, a consumer requires 3 VMs with the same characteristics. Public clouds provide a calculator to calculate the expected cost of the service consumption, e.g. an AWS calculator [3]. The calculator calculates the total cost for a month. In AWS, a consumer provides the expected utilisation of the service unit, i.e. *CompUnitSpec*. The consumer also provides the quantity required of each *CompUnitSpec*. In this implementation, the consumer provides their needs using a cost model.

The service requester provides their required quantity of service units. Then the costs for each provider offer are calculated, based on the quantities provided by a service requester. The costs calculated for the service requester SLA model and other SLA provider offers can be compared.

We explained the abstract syntax of the cost used in the metamodel. In the following, we explain how the cost is calculated.

In calculating the cost of the computing unit, we must consider the *price* and *paymentPeriod* that are provided in the SLA models. Cloud providers, e.g. Amazon EC2 and GoGrid provide the price per hour for the on-demand and subscription period. Usually, a provider calculates the expected cost for a monthly period, e.g an AWS calculator.

Listing 6.4 presents a calculation of the estimated cost for a month period and a 100% utilisation of the cloud computing unit. For the Networking and Storage units, the public cloud offers are given in *GBP* per month. Some providers add extra charges to the service price, so this is added to the total cost. The Storage and Network cloud units are specified by the size, which can be a single value or a ranged value. The algorithm typically performs a computation operation. The input is a *costModel*. The *costModel* is changed by adding a value to the *totalCost* attribute, and a value to the attribute *totalItemCost* for each item involved in this process.

Listing 6.4: Calculate the cost of the cost model using EOL

```
Input costModel;
for each item in costModel{
 if (item is ComputingUnit){
  costModel.totalItemCost=(CalculatePrice(priceValue,pricePer,
     monthperiod)*item.quantity)+extra charge
  }
  else if (item is Storage or item is Networking){
   totalItemCost=calculate(size.value*quantity);
   costModel.totalCost=totalCost+TotalItemCost;
  }
}
costModel.totalCost=totalCost;
```

The next section describes the process of removing duplicate elements from the trace model.

6.5 Analysing Results

An element in the service requester model using matching rules may or may not be included in the trace model. This means that, if there is no match between the service requester element and any of the elements in the provider model, then this element is not included in the trace model as an outcome. If a match is found between an element in a service requester model and another element in the provider model, then this is included in the trace model. The matching rules may also return a match between an element of a service requester model with more than one element in the provider model. In this case, the element in the service requester model is included in the trace models more than once.

This helps us to understand how can we process the outcomes before, e.g. transforming it into the MCDM matrix or cost model. The trace model may contain a repeated element from the consumer model which is matched with more than one element in the provider model. Assuming we have results as in Figure 6.5, the element in consumer model *c1* is matched with elements in the provider models (p1 and p2). Therefore, the trace model contains, in this case, two *traceItem* elements that refer to *c1* in the consumer model. This happens, for example, when the consumer requires VM (OS= Linux, RAM=3.75 GB, number of cores =) price = 0.6$ per hour. The name-based match returns more cloud units that have OS = Linux. In this case, the required VM is matched with more than one VM.

In this case, if there is more than one element in the provider model that matches an element in the service requester model, an interaction with a service requester might be required to eliminate the matches, so that each element in the service requester model will match only one element in the provider model.

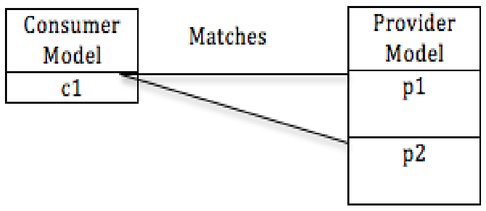


Figure 6.5: Example of repeated elements in the outcome of matching

Human interaction may help to eliminate this, by selecting one element or, alternatively, for example, finding the distances between different matched values. This is a multi-criteria decision analysis [193, 195]. Our proof of concept study illustrates a way to remove the redundant elements from the trace models by using an ETL module, as illustrated in Listing 7.2, as an example. We simply keep the first element found in the trace model and remove any repeated element.

To eliminate the elements of the trace model, we transform the trace model into a new trace model. This is done by transforming the *traceItem* into the new model if the *item.left* is not added to the new trace model. This is illustrated in Listing 7.2. The listing defines two models: *traceModel* as the input model and *NewMatched* as the output model. It simply copies the non-repeated *traceItem.left* to the *newItem* in the *newTraceModel*. Where there are any repeated *traceItem.left*, it is the consumers choice to select one of the *traceItems* to go to the *newTraceModel*, e.g., select the nearest offered item to the required item.

Listing 6.5: Eliminate repeated left items from the trace model

```
input = traceModel;
create newTraceModel;
for(traceItem in traceModel){
 if traceItem.left is not repeated in traceModel.items{
  create newItem;
  newItem=itraceItem;
  add newItem to newTraceModel;
 }
 else if itraceItem.left is repeated in traceModel.items{
  input=select nearest item;
  create newItem;
  newItem=traceItem;
  add newItem to newTraceModel
 }
}
```

Now a trace model can be used to form, for example, a matrix model, which is discussed in the following section (Section 6.6.2).

6.6 Decision Making Matrix

The outcome of the matching process of two SLAs may produce multiple matching elements. Having a set of provider offers matched with the consumer model (multiple trace models) allows the consumer to choose between different matched offers. The selection process may need to take into account consumer preferences (weight model). Selecting between different alternatives (trace models) with multiple criteria (elements in the consumer model) is called an MCDM problem. An MCDM matrix is then derived to solve such a problem. In our implementation, we constructed a metamodel to specify the MCDM matrix. In this section we describe the abstract syntax of this MCDM matrix. The MCDM models are constructed from: a weight model and a number of trace models.

6.6.1 Abstract Syntax of the MCDM Matrix

Figure 6.6 illustrates the abstract syntax of the matrix. We simply try to have the general structure of a matrix, which consists of a number of rows and columns. The metamodel consists of:

- Matrix class: has one or more rows (i.e. associates with *Row* class) and one or more columns (i.e. associates with the *Column* class). It also specifies one or more elements (i.e. associates with the *Element* class).

- Row class: is created to specify the rows of the matrix. Each row defines a string type attribute named *parameterName*. This name defines the parameters (criteria) that are defined in the SLA model (Chapter 4.3) that a service requester

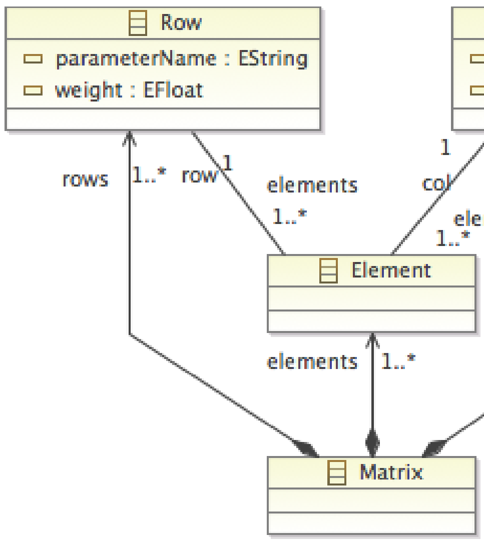


Figure 6.6: MCDM matrix metamodel

wants to include in the decision matrix. This class specifies another real type attribute named *weight*. This weight attribute defines the weights that are specified by the consumer in the weighting model (Section 6.3). It also associates with the *Element* class. The element association defines the data or values in the row.

- Column class: is created to specify the columns in the matrix. Each column specifies string attributes *ID* and *SLAName*. These attributes define the cloud provider id and name. As with the *Row class*, this class associates with the *Element* class, to define the elements of a column.

- Element class: is constructed to define the data of the matrix. Each element is associated with one *Row* class: and one *Column* class. It also has an attributed *eItem* that refers to the object elements of the SLA model.

How can we form the results into a matrix model as explained in this section? This is described in Section 6.6.2. The matrix model is formed from the consumer SLA model elements and the outcomes of the matching processes (number of trace models). Each trace model is an outcome of the matching process between the consumer and provider SLAs models.

6.6.2 Constructing a Matrix Model

A matrix model is formed using different criteria and alternatives. The criteria are the elements of the service requester model and the alternatives are the different provider offers. As we illustrated, the MCDM matrix includes weight values for the criteria. Therefore in constructing the matrix model, the elements in a weighted model (Section 6.3) are used to form the criteria and weight value of the matrix.

We construct a matrix via a number of different steps. First, we transform the elements in the weight model into the matrix with different rows which represent the different criteria of the decision matrix. This is illustrated in Listing 6.6.

A *weightModel* is an input model in this Listing. The algorithm creates a new *matrixModel* and assigns rows to the *items* defined in the *weightModel*. It also assigns a *weight* value to each row, as defined in the *weightModel*.

Listing 6.6: Create matrix rows which present the criteria of the MCDM

```
input=weightModel;
new matrixModel;
 for each element in weightModel{
  create new matrixModel.row
  matrixModel.row.weight=element.weight;
  matrixModel.elements=element;
}
```

The second step is to form the columns into a matrix. Each column presents a service provider offer. In this model, we include the trace model which is generated from the process explained in Section 6.5. The trace model includes items from both the provider and consumer models, so we add the element to the row that presents the same criteria. This is illustrated in Listing 6.7. Two input models are defined for constructing a column in a matrix: a traceModel and a matrixModel (which is the outcome from Listing 6.6 and where the matrix rows were defined). A new column is created (i.e. *matrixModel.column*). We then find the *row.item* in the *matrixModel* that matches with the *item.left* in the *traceModel* and assign this item (i.e. *item.right*) to the *matrixModel.element*. This matrixModel.element is added to the column elements.

Listing 6.7: Eliminate repeated left item of the trace model

```
input1=traceModel;
```

```
input2=matrixModel;
for each provider{
 new matrixModel.column
 matrixModel.column.ID = provider ID;
 for each item in traceModel{
  new matrixModel.elements;
  matrixModel.elements.eItem= item.right;
  row=find(matrixModel.row.item=item.left);
  matrixModel.elements.row=row;
  matrixModel.elements.column=matrixModel.column;
  matrixModel.column.add(matrixModel.element);
 }
}
```

The results may produce empty elements in the matrix. This happens when there are no matches found between a criterion in the service requester model and the provider model. This can be discussed as a multi criteria decision analysis [83, 193, 195].

6.7 View Matching Results

Having a trace model as an output (e.g. trace model and matrix model) model, allows a consumer to view the results. This can be done by, for example, transforming it into documents using the *EGL* model management task. This EGL task may be used to transform SLA models into a HTML code. Another way to view the results is to use *ModelLink*, which is a tool which consists of 2 or 3 EMF tree-base editors [10] to view the links between the consumer, provider and match models.

6.8 Summary

This chapter presented the processes and models involved in cloud computing SLA comparison. It described the general architecture of the comparison process. The main inputs of the process are the service requester SLA model and a set of provider SLA offers. The output of the matching process is a set of trace results. Different MDE model management tasks were performed on those models, such as transformation.

A consumer preferences model and trace models are processed to create an MCDM matrix, which is usually formed in such problems.

Chapter 7

Evaluation

7.1 Introduction

In this thesis, a metamodel for cloud SLA and a semi-automatic comparison process for such SLAs have been proposed. This chapter evaluates the proposed approach, including an evaluation of the hypothesis, which was stated earlier: *Can MDE principles and tools support the precise modelling of cloud computing SLAs in such a way that cloud stakeholders can define their offers and demands? In addition, can the MDE principles and tools enable a semi-automated comparison process for cloud computing SLAs, in order to help cloud stakeholders make better decisions about the appropriateness of offerings from different cloud providers?*

The evaluation is based on three distinct points:

1. Instantiating cloud SLA models for cloud providers and consumers that conform to cloud SLA model described in Chapter 4.

2. Using model comparison (see Chapter 5) to match the instantiated cloud SLAs models by applying different algorithms.

3. Supporting consumers in decision making by applying the different model management tasks (see Chapter 6) based on the outcomes of the model comparison.

The evaluation of the thesis contributions is discussed against the goals and objectives of the thesis. In this thesis, we evaluate our contributions using a case study.

7.2 Case Study

This section describes how we evaluated all of the research contributions in combination, i.e., when they are put together and applied to a new case study. The evaluation helps to explore the strengths and weaknesses of our work. The case study example is built based on other studies in [72, 83, 139, 188]. The values of the consumer requirements are adapted from [83]. The broker service is used for matching consumer requirements with the provider offers [72, 139].

The case study is used to evaluate the hypothesis and the overall methodology that we contribute; as such, it complements the evaluations done in the preceding chapters. To start with, we describe the scenario for the evaluation. A cloud broker service provides a match between cloud consumer requirements and a number of cloud provider SLAs. This service has a repository of SLAs models, cloud providers, cloud consumers (cloud service requesters) and a broker. Cloud providers provide the broker with their SLA offers. A cloud consumer requests an offer based on specific requirements. The broker creates a cloud SLA model expressed in a specific language and finds a match between the cloud provider requirements and cloud offers stored in the repository. In this example, we discuss the cloud broker service components:

- *Cloud SLA and repository*: How can different cloud SLA offers be expressed using the proposed model? Section 7.2.1 describes cloud SLA models using examples of three cloud providers. We describe the offers' details, the concepts used in each offer and how they can be presented using the proposed metamodel, as explained in Section 4. These offers are then stored in the repository.
- *Matching process and matching logics*: the cloud broker then starts a matching process which finds matches between consumer requirements and cloud offers. The cloud brokerage service uses the matching process, which finds matches between cloud SLA models, as explained in Chapter 5.
- *Preparing Results for the selection problem*: the cloud broker provides the results as a decision matrix from which the consumer may select an offer. After the matching process is completed, the results are a number of matching models. These models are processed then organized as an MCDM matrix (Chapter 6).

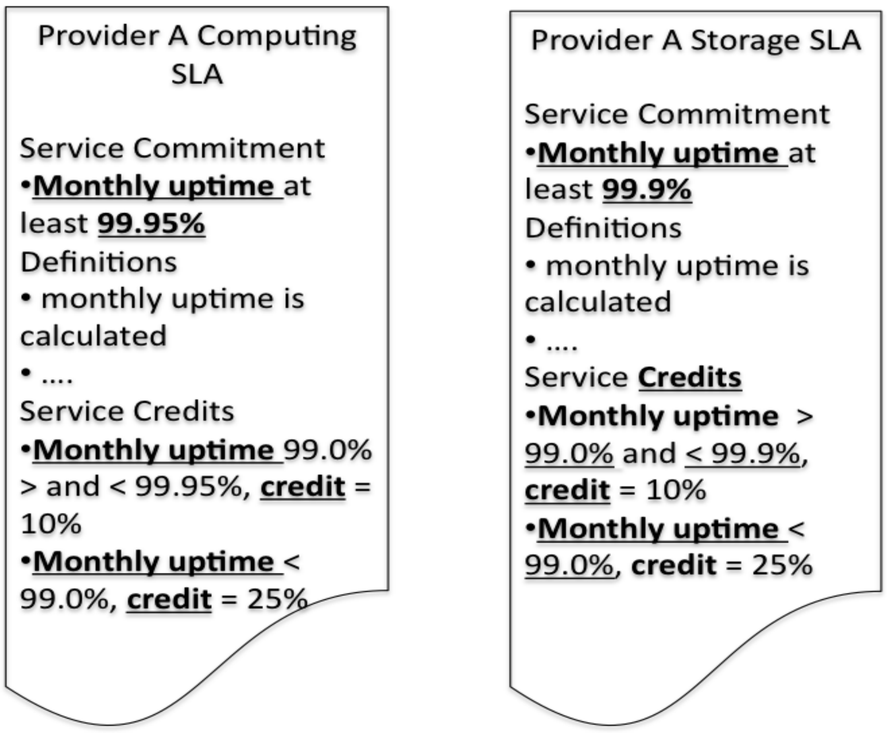
In this example, a consumer may desire to choose a cloud service; they can make use of QoS parameters as the basis for modelling their requirements. These services and QoS parameters are specified in the SLA models of providers. A consumer wishes to find the cloud provider offer that matches their requirements. Therefore, the consumer provides requirements to a cloud broker. A cloud broker is a service that has a repository of cloud SLAs. It stores cloud SLAs in a repository and collects the consumer requirements to find a match with these provider SLAs. A cloud broker finds matches between the consumer SLAs and provider SLAs, by using a common language to *define* both the provider and consumer SLAs.

7.2.1 Creating Cloud SLA models using the Cloud SLA abstract syntax

This section describes how the SLAs of cloud providers and consumers are defined using the proposed cloud SLA abstract syntax (Section 4.3). The purpose of this section is to test the applicability of the proposed metamodel, to define pre-defined cloud SLAs as model instances of this metamodel. In this example, provider SLA models are defined based on three public cloud providers: Amazon AWS, RackSpace and GoGrid. We randomly refer to the cloud providers as provider A, B and C. A consumer requirement is based on the requirements presented in [83] . We provide an example of how the three cloud SLAs look as they are used in the examples. The SLAs of the three cloud providers are as follows:

- The first provider *Provider A* defines two SLAs, one being a service for *Computing* and the other for *Storage*. Both SLAs use specific terminologies: Service Commitment, Definitions and Service Commitment and Credit as sections in the SLAs of provider A. In the Computing SLA, the commitment section defines *Monthly uptime percentage at least 99.95%*. In the Definitions section different terms are provided with descriptions, such as Monthly uptime percentage, describing how the value of this term is calculated. The Service Commitment and Credits defines two conditions (as violations) to request credit. The first is a credit of 10% of a monthly billing period, which is eligible to be requested if the Monthly uptime is less than 99.95% but equal to or greater than 99.0%. The second is a credit of 25%, which is eligible to be requested if the Monthly uptime is

Less than 99.0% of a monthly billing period. The second SLA, which is the SLA for storage, defines within the Commitment section the same parameter with a different value (99.9%). Difference values are given to the same parameter in the Service Commitment and Credits section (Figure 7.1).



Provider A Computing SLA

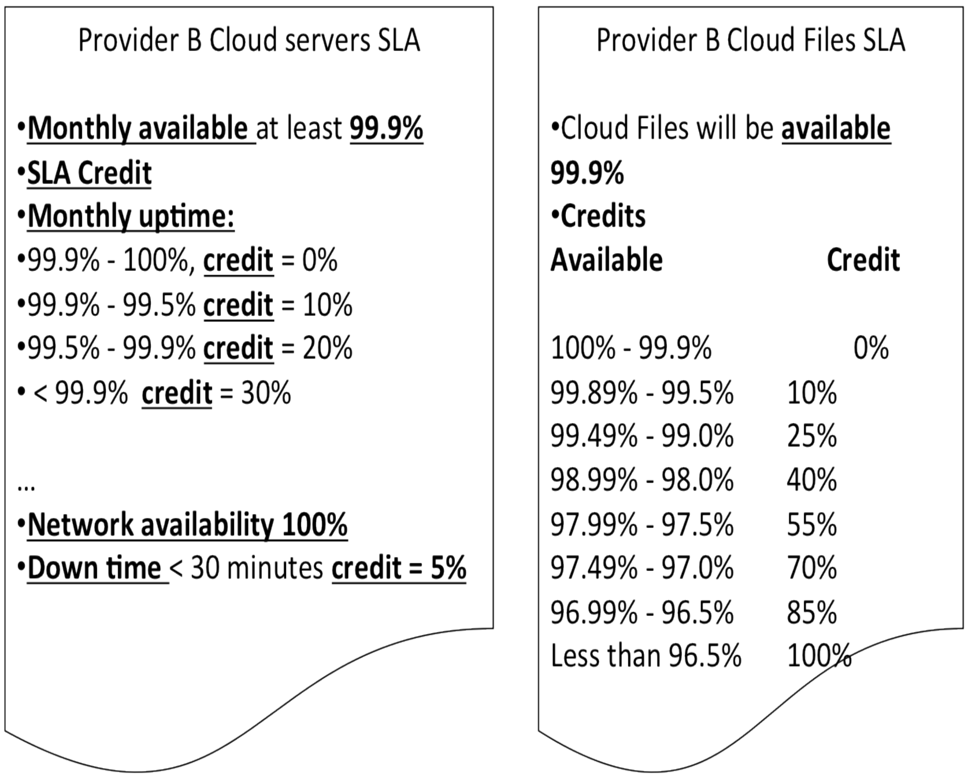
Service Commitment
•**Monthly uptime** at least **99.95%**
Definitions
• monthly uptime is calculated
• ....
Service Credits
•**Monthly uptime** 99.0% > and < 99.95%, **credit** = 10%
•**Monthly uptime** < 99.0%, **credit** = 25%

Provider A Storage SLA

Service Commitment
•**Monthly uptime** at least **99.9%**
Definitions
• monthly uptime is calculated
• ....
Service **Credits**
•**Monthly uptime** > 99.0% and < 99.9%, **credit** = 10%
•**Monthly uptime** < 99.0%, **credit** = 25%

Figure 7.1: Provider A SLA Offers

- The second provider, (i.e. *Provider B*) defines two SLAs, one for computing and the other for storage service. The first SLA defines *Cloud Servers will be monthly available 99.9%*. It defines a number of SLA credits for different values of monthly availability. It defines how monthly availability and credits are calculated. The cloud server host is defined within this SLA and a restoration time within 1 hour is defined. Another parameter, which is *network availability*, is provided as a 100% guarantee. The storage SLA defines: guarantee and credits sections. The guarantee defines *Cloud Files available 99.9%* and some description of what is, for example, unavailability. The credit values are defined in terms of monthly billing periods, as shown in Figure 7.2.



Provider B Cloud servers SLA

- **Monthly available** at least **99.9%**
- **SLA Credit**
- **Monthly uptime:**
- 99.9% - 100%, **credit** = 0%
- 99.9% - 99.5% **credit** = 10%
- 99.5% - 99.9% **credit** = 20%
- < 99.9% **credit** = 30%

...

- **Network availability 100%**
- **Down time** < 30 minutes **credit = 5%**

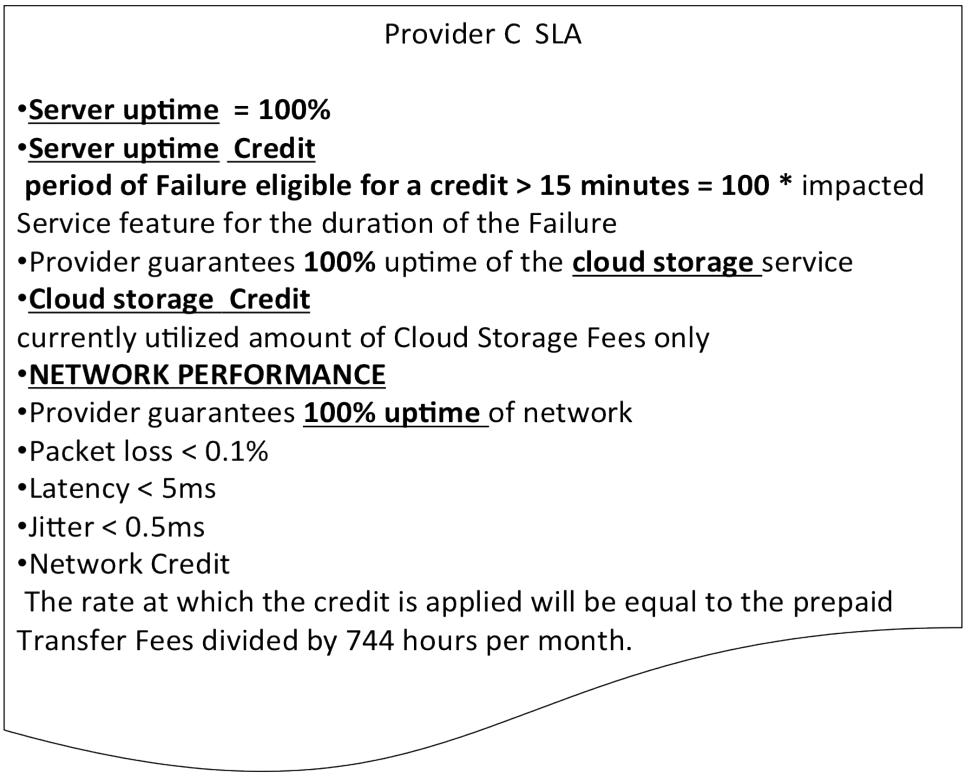
Provider B Cloud Files SLA

- Cloud Files will be **available** **99.9%**
- **Credits**

Available	**Credit**
100% - 99.9%	0%
99.89% - 99.5%	10%
99.49% - 99.0%	25%
98.99% - 98.0%	40%
97.99% - 97.5%	55%
97.49% - 97.0%	70%
96.99% - 96.5%	85%
Less than 96.5%	100%

Figure 7.2: Provider B SLA Offers

- The third provider (i.e. *Provider C*), defines server uptime, cloud storage and network performance in one SLA document. This SLA defines a section named *Server Uptime*. This section defines how *Individual servers will deliver 100% uptime*. The credit values are calculated in terms of *Memory Hours Fees at the time of Failure*. A cloud storage section defines *100% uptime* and credits are calculated in terms of *the impacted Service feature for the duration of the Failure*. This is illustrated in Figure 7.3.


Provider C SLA

•**Server uptime = 100%**
•**Server uptime Credit**
period of Failure eligible for a credit > 15 minutes = 100 * impacted Service feature for the duration of the Failure
•Provider guarantees **100%** uptime of the **cloud storage** service
•**Cloud storage Credit**
currently utilized amount of Cloud Storage Fees only
•**NETWORK PERFORMANCE**
•Provider guarantees **100% uptime** of network
•Packet loss < 0.1%
•Latency < 5ms
•Jitter < 0.5ms
•Network Credit
The rate at which the credit is applied will be equal to the prepaid Transfer Fees divided by 744 hours per month.

Figure 7.3: Provider C SLA Offers

The question is: how can these SLAs of providers be defined using our metamodel? We discuss each provider offer concept (i.e. Providers A, B and C) and match them with the SLA model elements that conform to classes of the proposed syntax for the cloud SLA model.

Let us consider provider A. In its offer, we can construct an SLA model (using the abstract syntax) to capture *two* separate SLAs, where each refers to a specific service element. The party element defines the provider ID and is defined as the provider. The monthly uptime is defined as a *QoSProperty* element. The value of this parameter (99.95%) is defined as a *SingleValue* element. This *QoSProerty* is defined within the SLA that defines computing in the *Service* element. The uptime is defined as an *Availability* element. The credit is defined as a *Credit* element, and the violation value (e.g. Monthly uptime is Less than 99.0%) is defined as an SLO element. This is illustrated in Figure 7.4. The classes and references of the abstract syntax are presented in solid lines, while the dotted shapes present a possible instance of the class. A cloud broker defines also the cloud units for the provider. Therefore, cloud units (i.e. computing and storage) characteristics are shown in Table 7.1, and are also included in the cloud SLA model defined by the cloud broker. This table shows an example of how cloud units are defined (for provider A; providers B and C are similar).

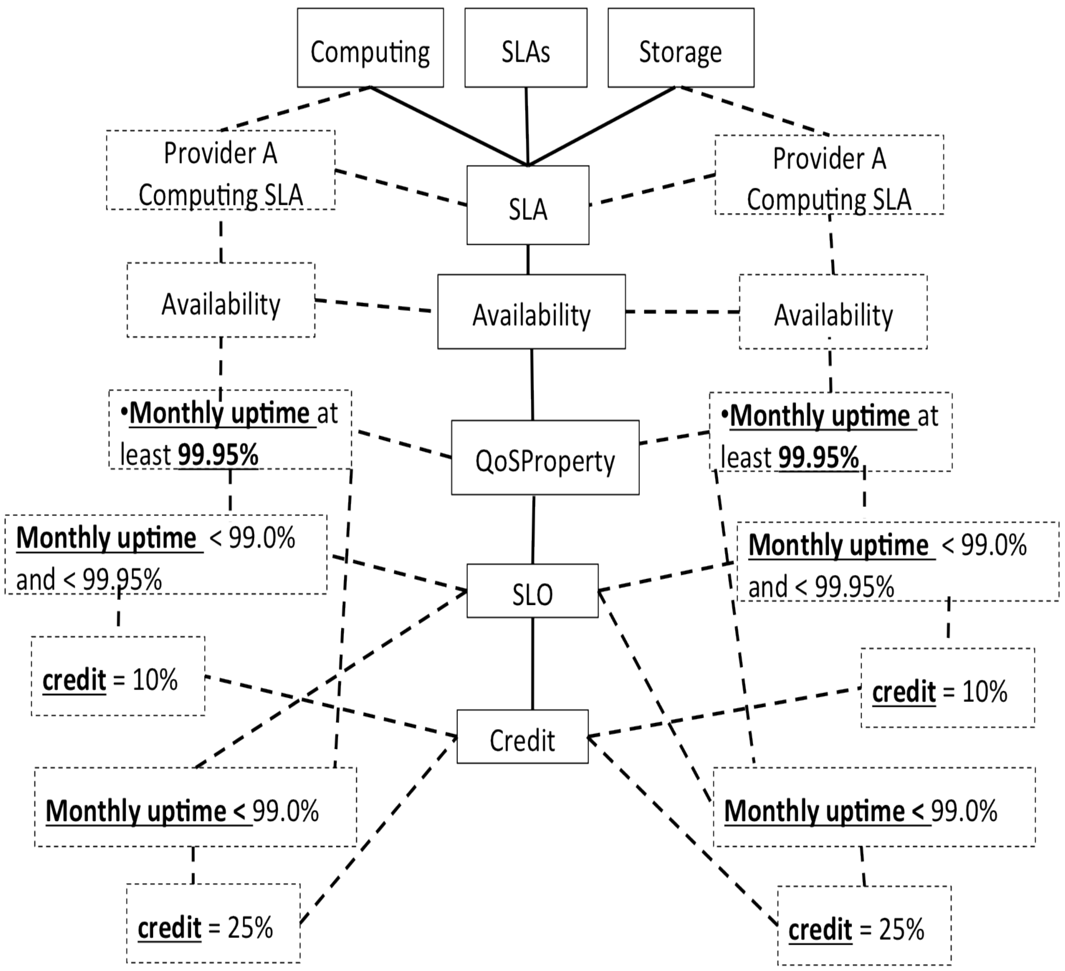


Figure 7.4: Matching cloud SLA with the concepts of the SLA Offer of provider A

The provider B SLA can be defined in the proposed abstract syntax, as illustrated in Figure 7.5. In this figure, a match is performed between cloud *SLA* and Provider B. Each SLA is matched with the SLA element (each element conforms to a service class), monthly available (with a *QoSProperty* of type *Availability*) and restoration time is matched with a *QoSProperty* of *Maintainability* elements. The network parameter is defined as a *QoSProperty* of *Availability* element. One question is within which

Table 7.1: Cloud unit characteristics and prices of Provider A

Computing Units and Prices					
Unit	vCPU	Memory GB	storage GB	OS	**Price**
Small *VM*	1	1.7	160	Linux	$0.06 per hour
Medium *VM*	2	3.75	410	Linux	$0.012 per hour
Large *VM*	4	7.5	2*420	Linux	$0.024 per hour
xLarge *VM*	8	15	2*840	Linux	$0.048 per hour

service this network availability *QoSProperty* is defined. It can be defined within the SLA that defines computing service. In this case, we have two availability parameters that define the cloud server in this service. This is defined in computing SLA, which means this is the availability of the network of computing service. We did not specify network service as a part of computing service, so this network may be defined as a *Networking* parameter or as a *Computing* parameter. In this example, we defined it as a computing parameter.

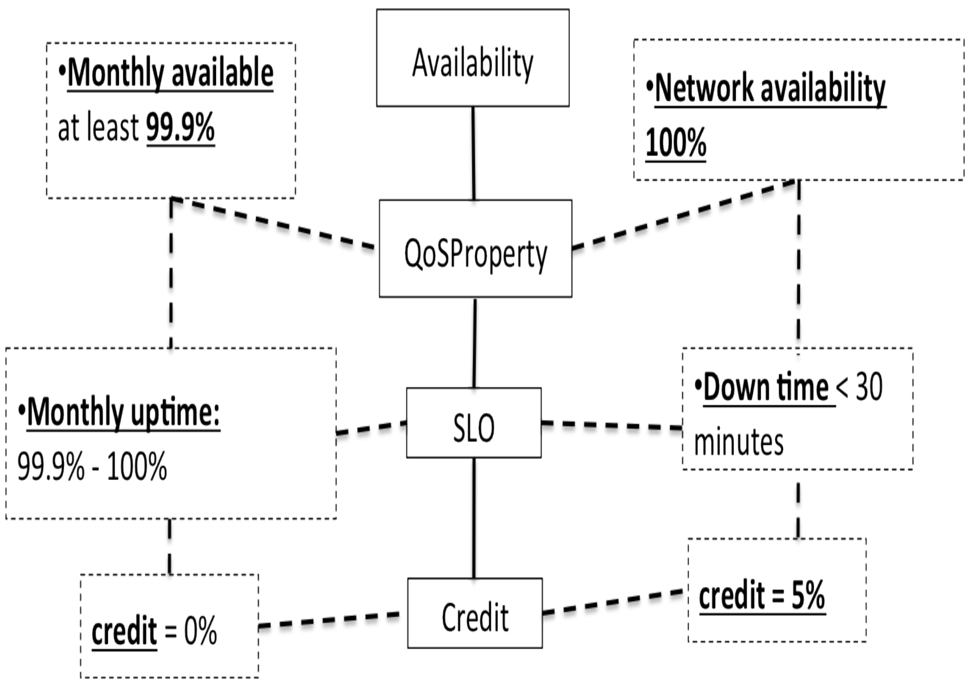

Availability
•Monthly available at least 99.9%
•Network availability 100%
QoSProperty
•Monthly uptime: 99.9% - 100%
SLO
•Down time < 30 minutes
credit = 0%
Credit
credit = 5%

Figure 7.5: Matching cloud SLA with concepts of SLA Offer of provider B

For provider C, an SLA class is matched with the provider SLA, each service in provider C is matched with an SLA class (Figure 7.6). Each SLA class defines a service type: Computing, Networking and Storage. Server uptime is assigned as a *QoSProperty* of type *Availability* in the SLA. This *QoSProperty* is defined as *QoSProperty* of *Computing* service. *Storage* and *Networking* services also define a *QoSProperty* uptime. In addition, network performance is assigned as a *QoSProperty* of type *Performance* in the SLA that defines *Networking*. Figure 7.6 illustrates the SLA concepts for *Computing* and *Networking* services.

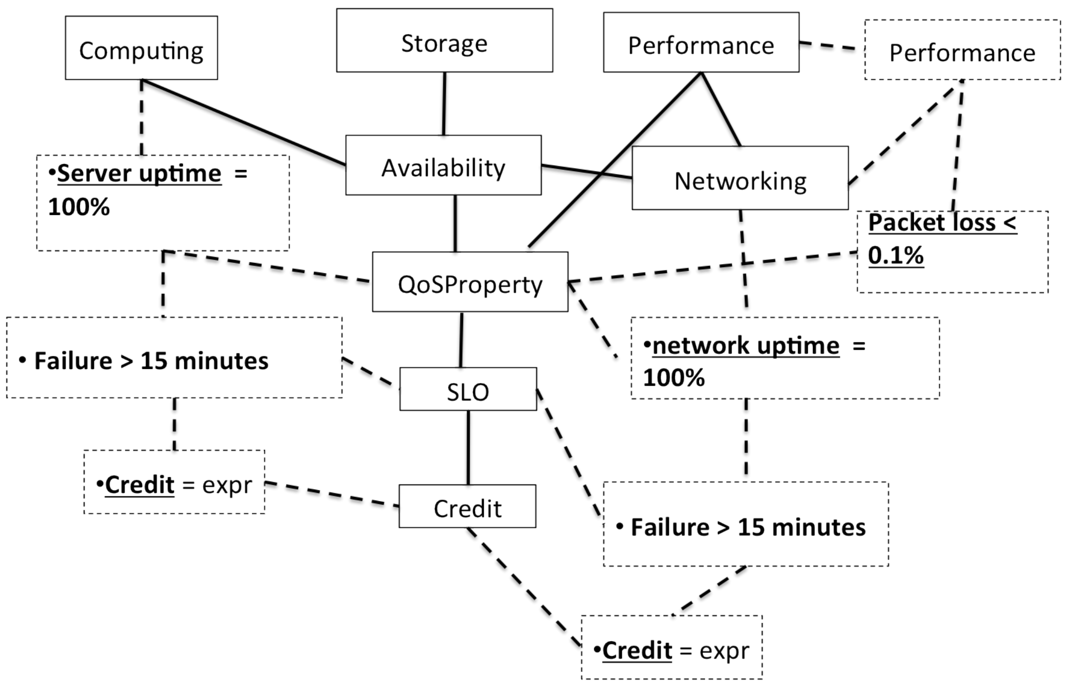
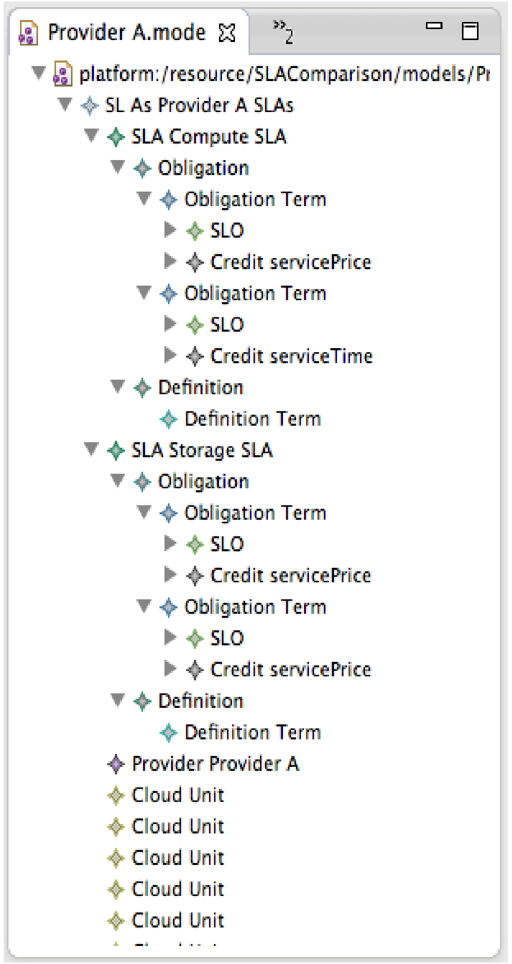


Figure 7.6: Matching cloud SLA with concepts of the SLA Offer of provider C

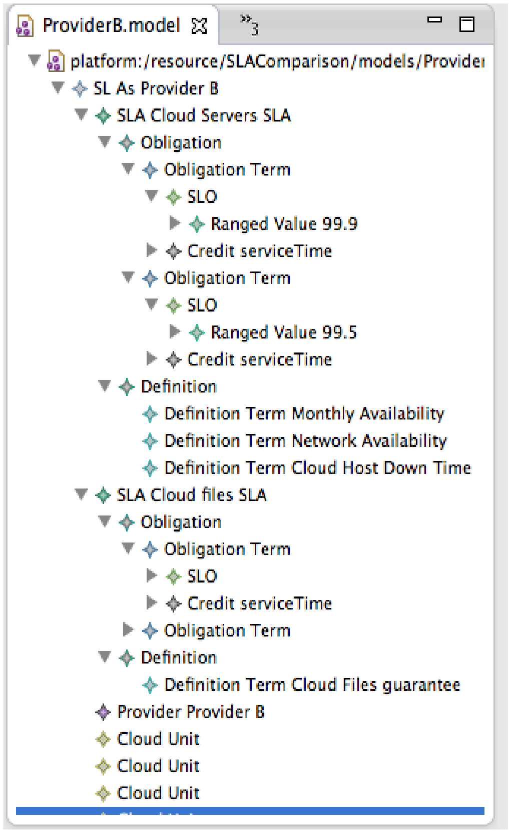
The models from the previous offers are shown in Figure 7.7. The models were created using a tree-based EMF editor. As we can see for the three providers, they have a similar structure. As illustrated, Provider A and B define two SLA elements, one for computing and one for storage. While provider C provides one more SLA element for the networking service, as they define a network performance. As we can see, each SLA has a definition section, which consists of a number of terms. Each term defines a QoS parameter and its value. The SLA section also has an obligation section. Like the definition section, the obligation section defines a number of obligation terms, which consists of an SLO parameter and credits. Each SLO defines a QoS parameter and a threshold value.

The QoS parameters are defined as illustrated in Figure 7.8. This model creates the service type and the QoS concepts that are of interest to the provider. The QoS parameters are defined within each concept. In this figure, we can see the two services created *Computing* and *Storage*. Computing service defines the *Availability* and *Maintainability* concepts. One QoS parameter is defined as *upTime* with its value. This QoS parameter is referred to by the *definition term* illustrated in Figure 7.7.



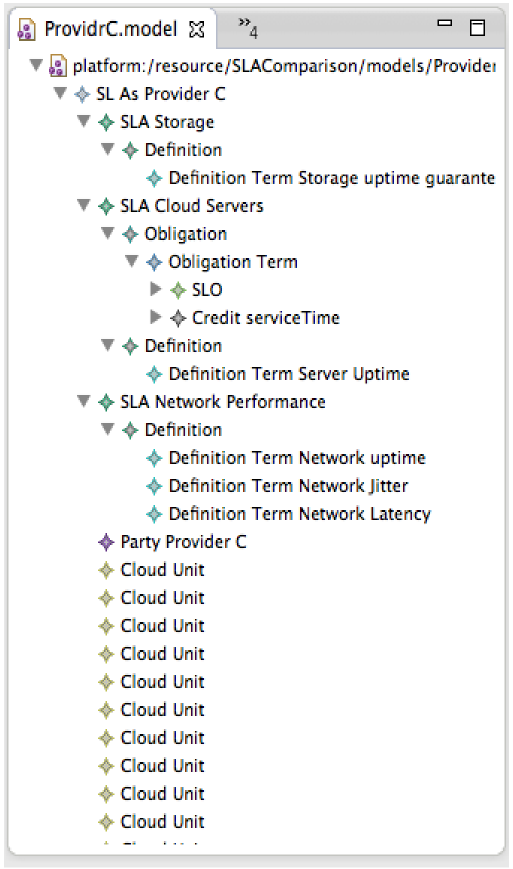
```
Provider A.mode ⊠  »2
▼ platform:/resource/SLAComparison/models/Pr
  ▼ SL As Provider A SLAs
    ▼ SLA Compute SLA
      ▼ Obligation
        ▼ Obligation Term
          ▶ SLO
          ▶ Credit servicePrice
        ▼ Obligation Term
          ▶ SLO
          ▶ Credit serviceTime
      ▼ Definition
          Definition Term
    ▼ SLA Storage SLA
      ▼ Obligation
        ▼ Obligation Term
          ▶ SLO
          ▶ Credit servicePrice
        ▼ Obligation Term
          ▶ SLO
          ▶ Credit servicePrice
      ▼ Definition
          Definition Term
    Provider Provider A
    Cloud Unit
    Cloud Unit
    Cloud Unit
    Cloud Unit
    Cloud Unit
```

(a) Provider A



```
ProviderB.model ⊠  »3
▼ platform:/resource/SLAComparison/models/Provider
  ▼ SL As Provider B
    ▼ SLA Cloud Servers SLA
      ▼ Obligation
        ▼ Obligation Term
          ▼ SLO
            ▶ Ranged Value 99.9
          ▶ Credit serviceTime
        ▼ Obligation Term
          ▼ SLO
            ▶ Ranged Value 99.5
          ▶ Credit serviceTime
      ▼ Definition
          Definition Term Monthly Availability
          Definition Term Network Availability
          Definition Term Cloud Host Down Time
    ▼ SLA Cloud files SLA
      ▼ Obligation
        ▼ Obligation Term
          ▶ SLO
          ▶ Credit serviceTime
        ▶ Obligation Term
      ▼ Definition
          Definition Term Cloud Files guarantee
    Provider Provider B
    Cloud Unit
    Cloud Unit
    Cloud Unit
```

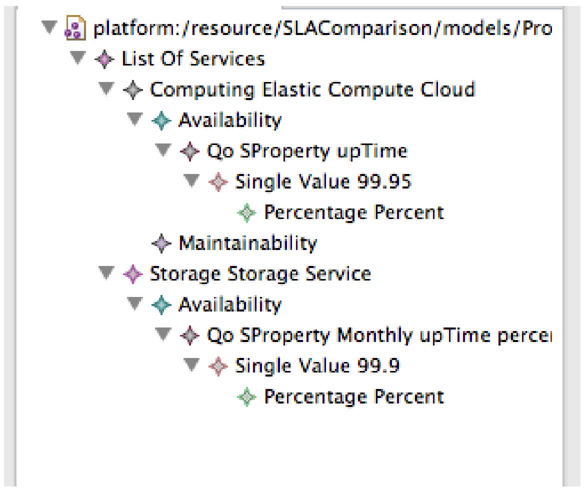
(b) Provider B



```
ProvidrC.model ⊠  »4
▼ platform:/resource/SLAComparison/models/Provider
  ▼ SL As Provider C
    ▼ SLA Storage
      ▼ Definition
          Definition Term Storage uptime guarante
    ▼ SLA Cloud Servers
      ▼ Obligation
        ▼ Obligation Term
          ▶ SLO
          ▶ Credit serviceTime
      ▼ Definition
          Definition Term Server Uptime
    ▼ SLA Network Performance
      ▼ Definition
          Definition Term Network uptime
          Definition Term Network Jitter
          Definition Term Network Latency
    Party Provider C
    Cloud Unit
    Cloud Unit
    Cloud Unit
    Cloud Unit
    Cloud Unit
    Cloud Unit
    Cloud Unit
    Cloud Unit
    Cloud Unit
    Cloud Unit
```

(c) Provider C

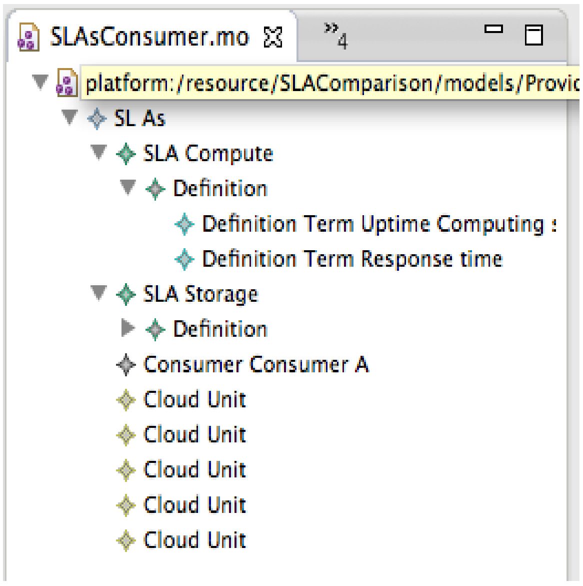
Figure 7.7: Illustration of provider SLAs offers using the cloud SLA metamodel



```
▼ platform:/resource/SLAComparison/models/Pro
  ▼ List Of Services
    ▼ Computing Elastic Compute Cloud
      ▼ Availability
        ▼ Qo SProperty upTime
          ▼ Single Value 99.95
              Percentage Percent
        Maintainability
    ▼ Storage Storage Service
      ▼ Availability
        ▼ Qo SProperty Monthly upTime perce
          ▼ Single Value 99.9
              Percentage Percent
```

Figure 7.8: Illustration of a service model of provider A using the cloud SLA abstract syntax

The main element, which is the *SLAs*, defines a number of cloud units. Each cloud unit defines a specific cloud unit characteristic and its price. For the purpose of this case study, Figure 7.9 illustrates consumer requirements in the form of an SLA model. This consumer model is used to find matches with cloud SLA models, which are explained in Figure 7.7.



```
SLAsConsumer.mo
▼ platform:/resource/SLAComparison/models/Provic
  ▼ SL As
    ▼ SLA Compute
      ▼ Definition
          Definition Term Uptime Computing :
          Definition Term Response time
    ▼ SLA Storage
      ▶ Definition
      Consumer Consumer A
      Cloud Unit
      Cloud Unit
      Cloud Unit
      Cloud Unit
      Cloud Unit
```

Figure 7.9: Example of a consumer requirements using SLA

We have described above the cloud provider SLAs offers and explained how these can be defined using the proposed abstract syntax, as cloud SLA model instances, that can be stored in the repository. A cloud consumer provides an SLA to find a provider offer that matches their requirements. The cloud consumer SLA is defined, as we did with the cloud offers, by using the proposed cloud SLA metamodel. The cloud broker service starts to find a match between the requested and offered SLAs.

7.2.2 Matching the cloud SLA of the consumer and provider

The purpose of this section is to describe the matching logics and the outcome models of the matching process. This process matches cloud consumer requirements with cloud providers SLAs. In this example, a cloud broker has consumer requirements defined as a cloud SLA model and three provider SLA models stored in the repository. The cloud broker service matches the cloud consumer SLA with each provider SLA.

As we described in section 5.4, there are different matching logics – i.e. Optimal, Approximate and Name-Based.

It is up to the service requester to choose the matching logic. For example, if the requester does not provide approximate values the approximate matching is not performed. If the service requester provides values as a lower limit and wants to find better ones, then optimal matching is performed. In this case study, we assume that the broker service performs different matching logics.

In the first match, a matching process matches the cloud SLA concepts and values (Optimal). In this matching a consumer expect the matching process returns the SLA offers with same or better values than the required values. The result of this matching depends on the required values: a match between the required elements and offers may not be found or redundant matches can be found (see Section 7.2.2.1). As a result, a second matching logic (i.e. Approximate) can be deployed to reduce the matching results (see Section 7.2.2.2). A match for the requirements is not yet found: thus a name-based matching logic is applied (see Section 7.2.2.3). This matching returns a number of redundant matching. Therefore, other ways are explored (see Sections 7.2.2.4, 7.2.2.5 and 7.2.3.1). This is explained in the next example, but before that, we explain the outcome model of the comparison (Trace models Section 5.7), which is used throughout this case study.

A consumer can view the result of each matching process; thus there are three trace models which can be viewed by a consumer. This trace model consists of items, and each item has two parameters, i.e. *left* and *right*. The left points to a consumer element model, while the right refers to an element in the provider model. As an MDE concept model can be transformed to text, and to illustrate a perhaps more human-accessible representation of the trace model, this model is transformed into an HTML document, as in Figure 7.10.

We generate simple HTML to represent the trace model, using EGL. This EGL program (as described in Listing 7.1) generates HTML that displays a table of trace items in the trace model. It generates the element name of left and right parameters for each item in a table row. Thus we can view the matched elements of both the provider and consumer models. The operation *printItem* is invoked to print details of each element in the cloud SLA.

Listing 7.1: EGL program to generate an html document for the trace model

```
 [%
var matched : matchTrace!Trace = matchTrace!Trace.allInstances.at
    (0);
%]
   .......
<body>
<h1>  Name-Based units and Approx. prices Matching Results </h>
<table cellspacing="0" id="customers" >
  <tr align="center"><th colspan="2" > Comparison Results</th></
      tr>
  <tr align="center">
    <th colspan="1" > Consumer requirements </th>
    <th colspan="1" align="center"> Provider offers </th>
  </tr>
   [%for (item in matched.items){%]
    <tr>
      <td>[%=printItem(item.left)%]</td> <td>[%=printItem(item.
          right) %]</td>
    </tr>
   [%}%]
</table>
</body>
.....
 [%operation printItem(item: Any){
....
}
```

We will use the generated HTML documents in the following examples to describe the results of different matching logics.

We now provide an example of matching a cloud consumer model (Figure 7.9) with a provider model (Figure 7.7). Different matching logics are applied. The purpose of this example is to show the differences in the trace models using different logics. The matching examples are:

- Matching models using Optimal matching.
- Matching models using Approximate matching.

- Matching models using Name-Based matching.
- Matching models using Name-Based matching for all elements except prices, for which we use Approximate matching.
- Matching models using Approximate matching for all elements except prices, for which we use Name-Based matching.

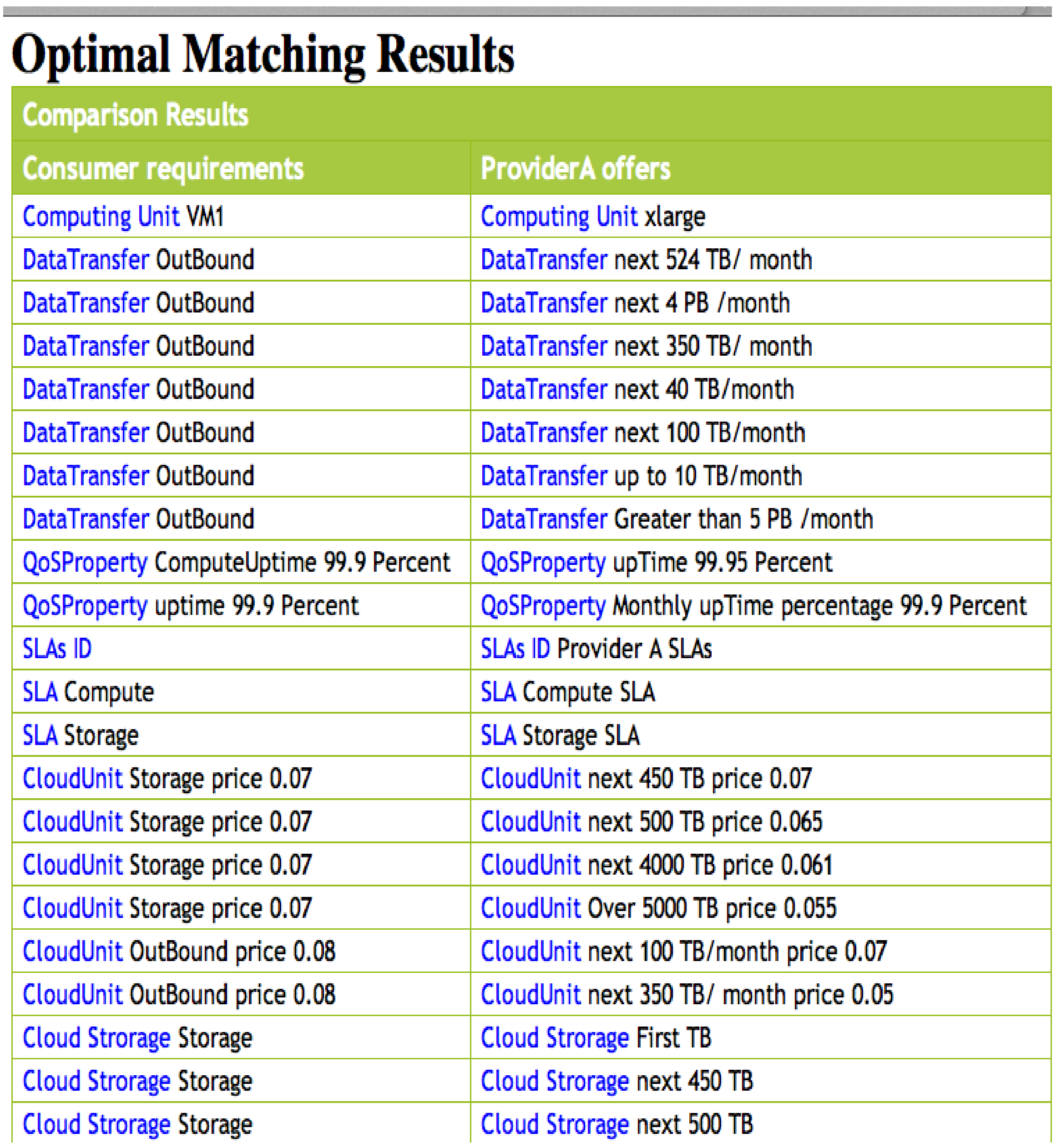
Each matching logic and its results are discussed in different sections. In all examples, the input model for the matching process is a model defined based on consumer requirements, as illustrated in Figure 7.9. We also make use of a provider model (A), as illustrated in Figures 7.7 and 7.8, and the output model is a trace model, which is presented as an html document. In this example, elements of interests that will be matched by the logics are as follows:

- *CloudStorage*: this means a match is found for the storage characteristics (i.e. cloud storage and its size).
- *CompUnitSpec*: this means a match is found for VM characteristics (i.e. CPU, Memory storage and OS).
- *DataTransferType*: this means a match is found for transfer type characteristics.
- *CloudUnit*: this means a match is found for cloud units (e.g. cloud storage, VM and transfer type) and their price.
- *Computing*: this means a match is found for the computing service defined in models.
- *Storage*: this means a match is found for the storage service defined in models.
- *QoSProperty*: this means a match is found for the QoS parameters, which belong to the same QoS concept and service.

7.2.2.1 Matching models using the optimal matching logic

Section 5.4.5 discussed the exact matching logic. In this matching algorithm, we match all attributes in each element, matching the elements of *SingleValue* and *RangedValue*, of the consumer model with all of the attributes in the provider model. These elements define the values of the cloud QoS properties, the cloud unit characteristics and the prices. The result of this matching is illustrated in Figure 7.10. The figure shows the trace model of the optimal matching between the consumer requirements (Figure 7.9) and provider offer (Figure 7.1). This type of matching returns better values (see Section 5.4.5).

The figure shows that two models define the same services (e.g. computing and storage), the same concept (e.g. Availability) and the two *QoSProperty* elements. There is only one match on the *Computing Unit* (i.e. VM1) and no match for VM2. There is more than one match for *Cloud Storage* (i.e. four matches for the cloud storage), *DataTransfer* elements of type *OutBound* and *CloudUnit*. Figure 7.10 shows that *(*CLoudUnit) matches are found for storage and networking (i.e. OutBound) but not computing unit. This match is a result of matching the cloud unit specifications and the price. For the cloud unit of type *Storage*, there is more than one match, since the cloud provider provides for the bigger required storage size less prices per GB.



Optimal Matching Results

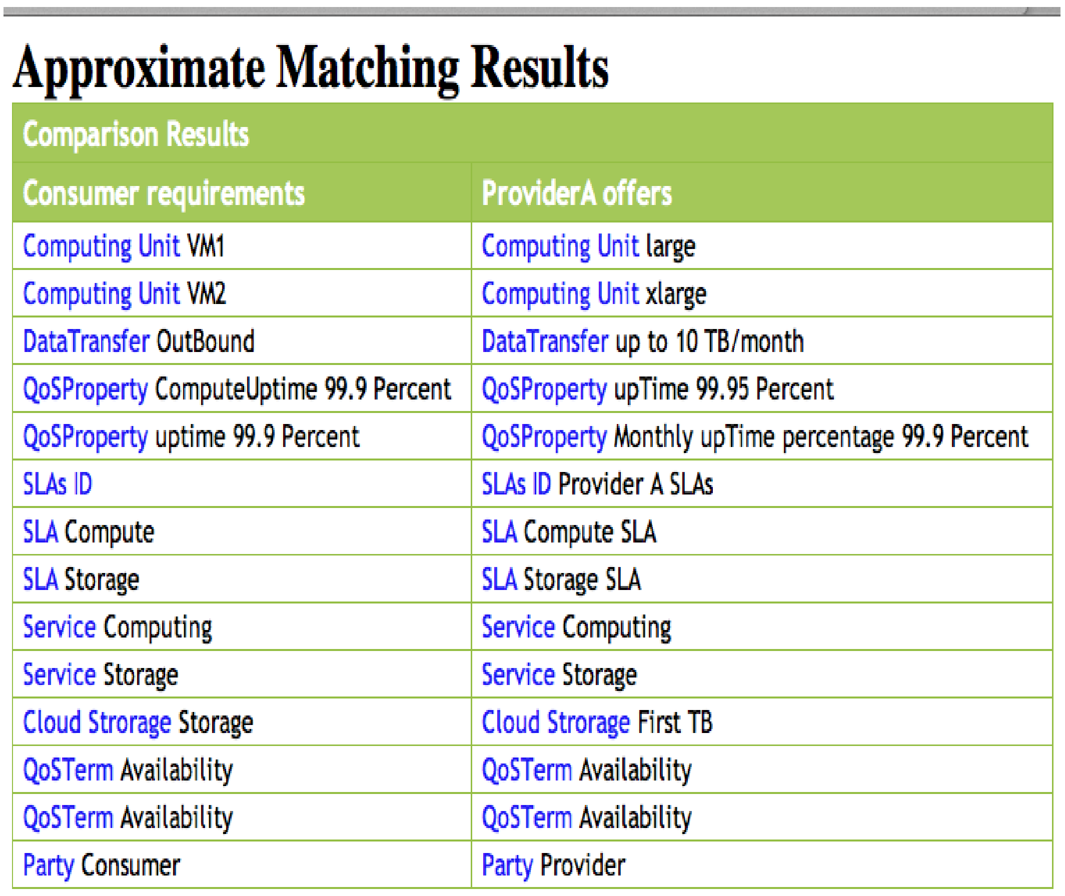
Comparison Results	
Consumer requirements	**ProviderA offers**
Computing Unit VM1	Computing Unit xlarge
DataTransfer OutBound	DataTransfer next 524 TB/ month
DataTransfer OutBound	DataTransfer next 4 PB /month
DataTransfer OutBound	DataTransfer next 350 TB/ month
DataTransfer OutBound	DataTransfer next 40 TB/month
DataTransfer OutBound	DataTransfer next 100 TB/month
DataTransfer OutBound	DataTransfer up to 10 TB/month
DataTransfer OutBound	DataTransfer Greater than 5 PB /month
QoSProperty ComputeUptime 99.9 Percent	QoSProperty upTime 99.95 Percent
QoSProperty uptime 99.9 Percent	QoSProperty Monthly upTime percentage 99.9 Percent
SLAs ID	SLAs ID Provider A SLAs
SLA Compute	SLA Compute SLA
SLA Storage	SLA Storage SLA
CloudUnit Storage price 0.07	CloudUnit next 450 TB price 0.07
CloudUnit Storage price 0.07	CloudUnit next 500 TB price 0.065
CloudUnit Storage price 0.07	CloudUnit next 4000 TB price 0.061
CloudUnit Storage price 0.07	CloudUnit Over 5000 TB price 0.055
CloudUnit OutBound price 0.08	CloudUnit next 100 TB/month price 0.07
CloudUnit OutBound price 0.08	CloudUnit next 350 TB/ month price 0.05
Cloud Strorage Storage	Cloud Strorage First TB
Cloud Strorage Storage	Cloud Strorage next 450 TB
Cloud Strorage Storage	Cloud Strorage next 500 TB

Figure 7.10: A trace model of the Optimal matching between consumer model and provider A model is generated into an HTML document using EGL

7.2.2.2 Matching models using approximate matching logic

This match provides an approximation on the elements of type *SingleValue*, *Ranged-Value* and *Price*, since the previous matching found more than one match for some of the requirements of the consumer. In this match, an approximate value is defined by the consumer for each element to define the difference percentage (see Section 5.4.6). The result of this matching process is shown in Figure 7.14.

The main differences between these results and the results in Figure 7.10, are: a matching for two computing units (i.e. VM1 and VM2), networking units (i.e. Data-Transfer outbound), *CloudUnit* and redundant matches were found. The two computing units, storage unit and networking unit were matched but the prices of these units did not match: thus, there was no match for the *CloudUnit* elements. If the approximation value (i.e. 10%) for the price is increased, this matching logic may return matching elements of type *CloudUnit*.

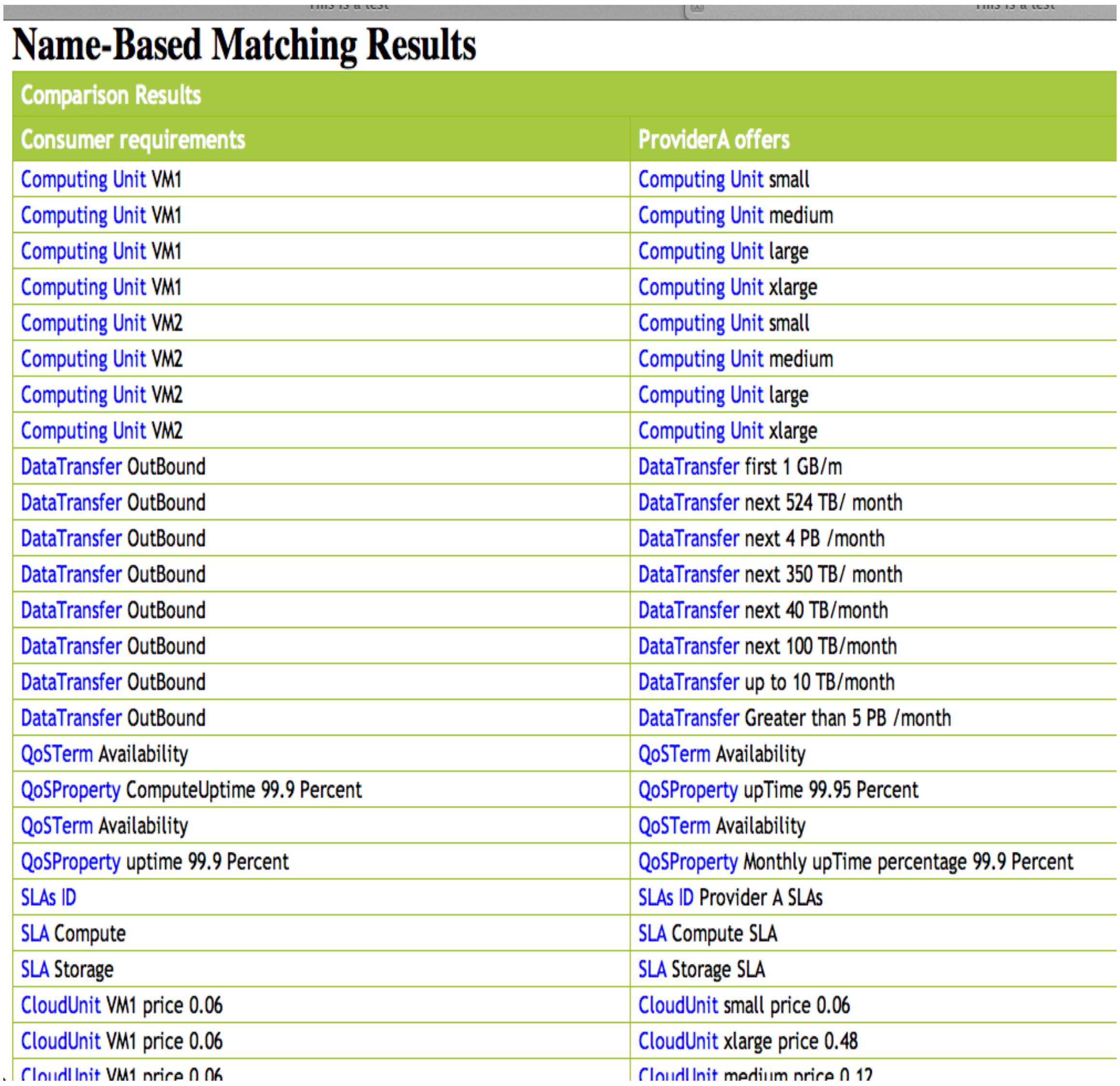

Approximate Matching Results

Comparison Results	
Consumer requirements	ProviderA offers
Computing Unit VM1	Computing Unit large
Computing Unit VM2	Computing Unit xlarge
DataTransfer OutBound	DataTransfer up to 10 TB/month
QoSProperty ComputeUptime 99.9 Percent	QoSProperty upTime 99.95 Percent
QoSProperty uptime 99.9 Percent	QoSProperty Monthly upTime percentage 99.9 Percent
SLAs ID	SLAs ID Provider A SLAs
SLA Compute	SLA Compute SLA
SLA Storage	SLA Storage SLA
Service Computing	Service Computing
Service Storage	Service Storage
Cloud Strorage Storage	Cloud Strorage First TB
QoSTerm Availability	QoSTerm Availability
QoSTerm Availability	QoSTerm Availability
Party Consumer	Party Provider

Figure 7.11: A trace model of the Approximate matching between the consumer model and provider A model is generated into an HTML document using EGL

7.2.2.3 Matching models using name-based matching logic

The last matching methods (i.e. Optimal and Approximate) did not find all of the required cloud units (see Section 7.2.2.1 and 7.2.2.2). Therefore another matching method is performed. This example provides name-based matching between the elements of type *SingleValue* and *RangedValue* and *Price*. This matching logic returns all possible matching combinations, e.g. *VM* is matched in the provider model with all VMs that define the same OS. The results are illustrated in Figure 7.12 which, in this case, do not help a consumer to meet their requirements. Thus we created two more examples to show other options for matching cloud SLAs using MDE principles. These examples are discussed in sections 7.2.2.4 and 7.2.2.5. Section 7.2.3.1 provides an ETL process to eliminate the redundant elements.



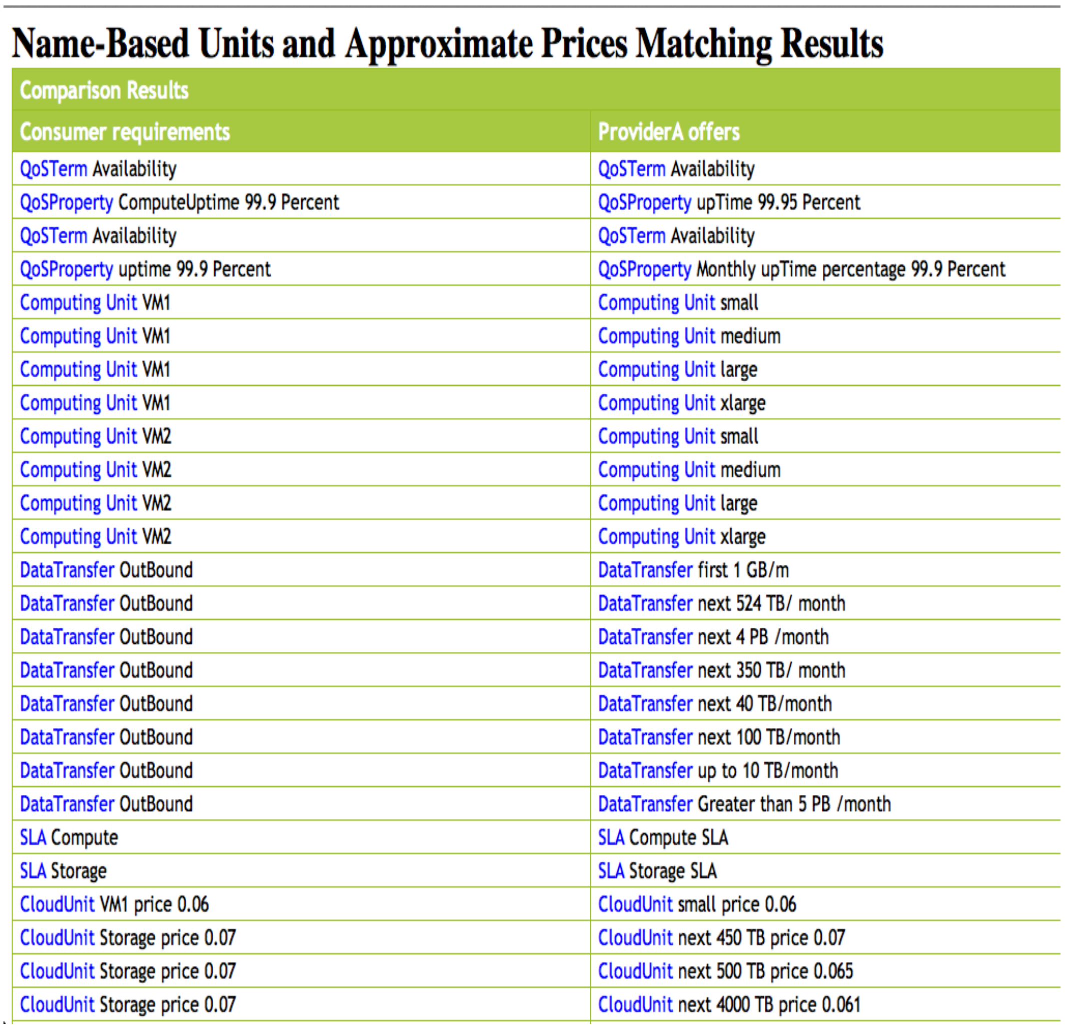
Name-Based Matching Results

Comparison Results	
Consumer requirements	ProviderA offers
Computing Unit VM1	Computing Unit small
Computing Unit VM1	Computing Unit medium
Computing Unit VM1	Computing Unit large
Computing Unit VM1	Computing Unit xlarge
Computing Unit VM2	Computing Unit small
Computing Unit VM2	Computing Unit medium
Computing Unit VM2	Computing Unit large
Computing Unit VM2	Computing Unit xlarge
DataTransfer OutBound	DataTransfer first 1 GB/m
DataTransfer OutBound	DataTransfer next 524 TB/ month
DataTransfer OutBound	DataTransfer next 4 PB /month
DataTransfer OutBound	DataTransfer next 350 TB/ month
DataTransfer OutBound	DataTransfer next 40 TB/month
DataTransfer OutBound	DataTransfer next 100 TB/month
DataTransfer OutBound	DataTransfer up to 10 TB/month
DataTransfer OutBound	DataTransfer Greater than 5 PB /month
QoSTerm Availability	QoSTerm Availability
QoSProperty ComputeUptime 99.9 Percent	QoSProperty upTime 99.95 Percent
QoSTerm Availability	QoSTerm Availability
QoSProperty uptime 99.9 Percent	QoSProperty Monthly upTime percentage 99.9 Percent
SLAs ID	SLAs ID Provider A SLAs
SLA Compute	SLA Compute SLA
SLA Storage	SLA Storage SLA
CloudUnit VM1 price 0.06	CloudUnit small price 0.06
CloudUnit VM1 price 0.06	CloudUnit xlarge price 0.48
CloudUnit VM1 price 0.06	CloudUnit medium price 0.12

Figure 7.12: A trace model of the Name-Based matching between a consumer model and provider A model is transformed into an HTML document using EGL

7.2.2.4 Matching models using name-based matching logic, to match cloud units and approximation to match prices

A constraint is added to the price value where an approximate matching is performed. As we noticed in Section 7.2.2.3 there were redundant elements in the outcome of the name-based matching; thus this constraint is added. This is an example of performing similar matching on the elements of type *CloudUnitSpec* and performing approximation on the price value of type *Price*. One *CloudUnit* element of type computing was found in this example, as matching elements between a consumer and cloud provider. For the other types of the *CloudUnit* elements (i.e. Storage and Networking) redundant matched elements were found. This is shown in Figure 7.14. The results depend on the approximation values and the input models.



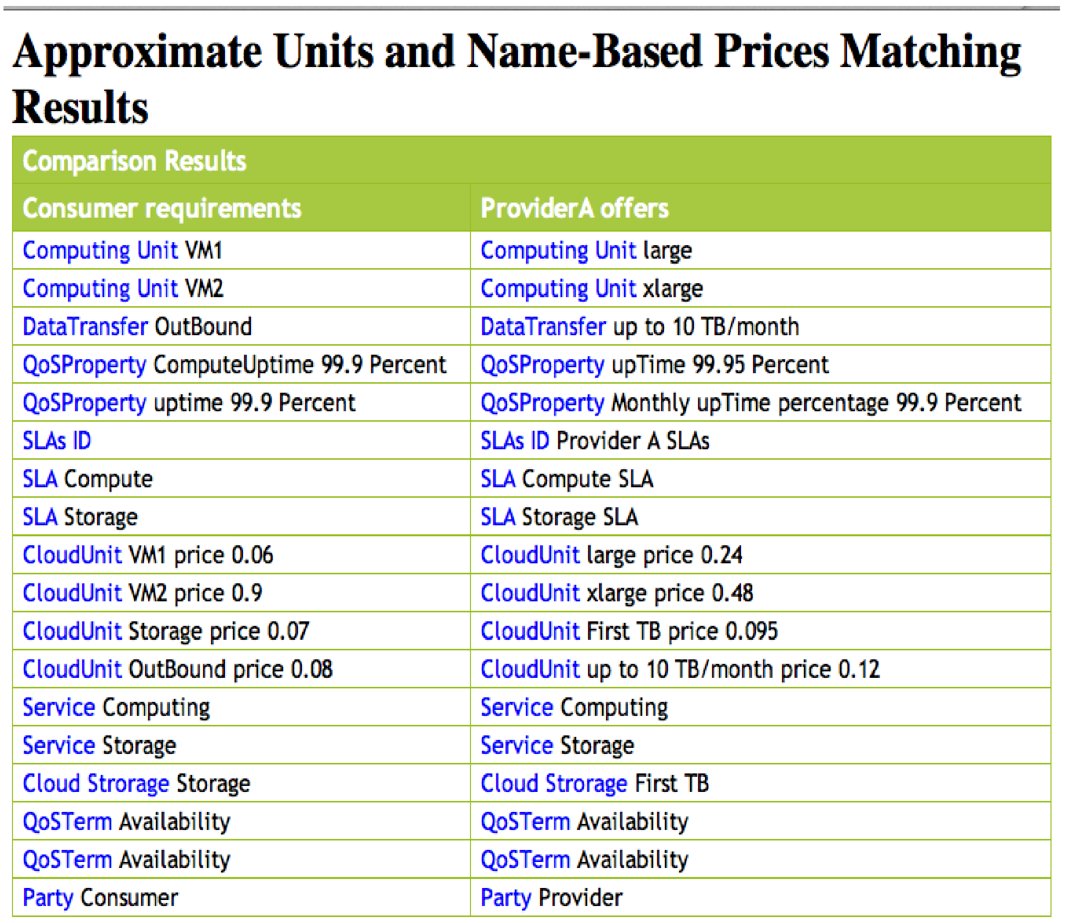
Name-Based Units and Approximate Prices Matching Results

Comparison Results	
Consumer requirements	**ProviderA offers**
QoSTerm Availability	QoSTerm Availability
QoSProperty ComputeUptime 99.9 Percent	QoSProperty upTime 99.95 Percent
QoSTerm Availability	QoSTerm Availability
QoSProperty uptime 99.9 Percent	QoSProperty Monthly upTime percentage 99.9 Percent
Computing Unit VM1	Computing Unit small
Computing Unit VM1	Computing Unit medium
Computing Unit VM1	Computing Unit large
Computing Unit VM1	Computing Unit xlarge
Computing Unit VM2	Computing Unit small
Computing Unit VM2	Computing Unit medium
Computing Unit VM2	Computing Unit large
Computing Unit VM2	Computing Unit xlarge
DataTransfer OutBound	DataTransfer first 1 GB/m
DataTransfer OutBound	DataTransfer next 524 TB/ month
DataTransfer OutBound	DataTransfer next 4 PB /month
DataTransfer OutBound	DataTransfer next 350 TB/ month
DataTransfer OutBound	DataTransfer next 40 TB/month
DataTransfer OutBound	DataTransfer next 100 TB/month
DataTransfer OutBound	DataTransfer up to 10 TB/month
DataTransfer OutBound	DataTransfer Greater than 5 PB /month
SLA Compute	SLA Compute SLA
SLA Storage	SLA Storage SLA
CloudUnit VM1 price 0.06	CloudUnit small price 0.06
CloudUnit Storage price 0.07	CloudUnit next 450 TB price 0.07
CloudUnit Storage price 0.07	CloudUnit next 500 TB price 0.065
CloudUnit Storage price 0.07	CloudUnit next 4000 TB price 0.061

Figure 7.13: A trace model matching similar cloud units and approximate prices is transformed into HTML document using EGL

7.2.2.5 Matching models using approximation, to match cloud units characteristics and name-based matching logic to match prices

From the previous matching, a number of matches were found for requirements (e.g. VM1). This matching adds constraints to the cloud elements of type *CloudUnitSpec* with an approximation. This section provides a matching algorithm that finds cloud units characteristics with a percentage difference in their values, while it also matches similar prices (i.e. on demand prices or subscription prices). From the results, illustrated in Figure 7.14, using similar matching for *Price* maks a difference in the results for the *CloudUnits*, which returns one matching element for each required cloud unit, compared to the output model illustrated in Figure 7.14.



Approximate Units and Name-Based Prices Matching Results

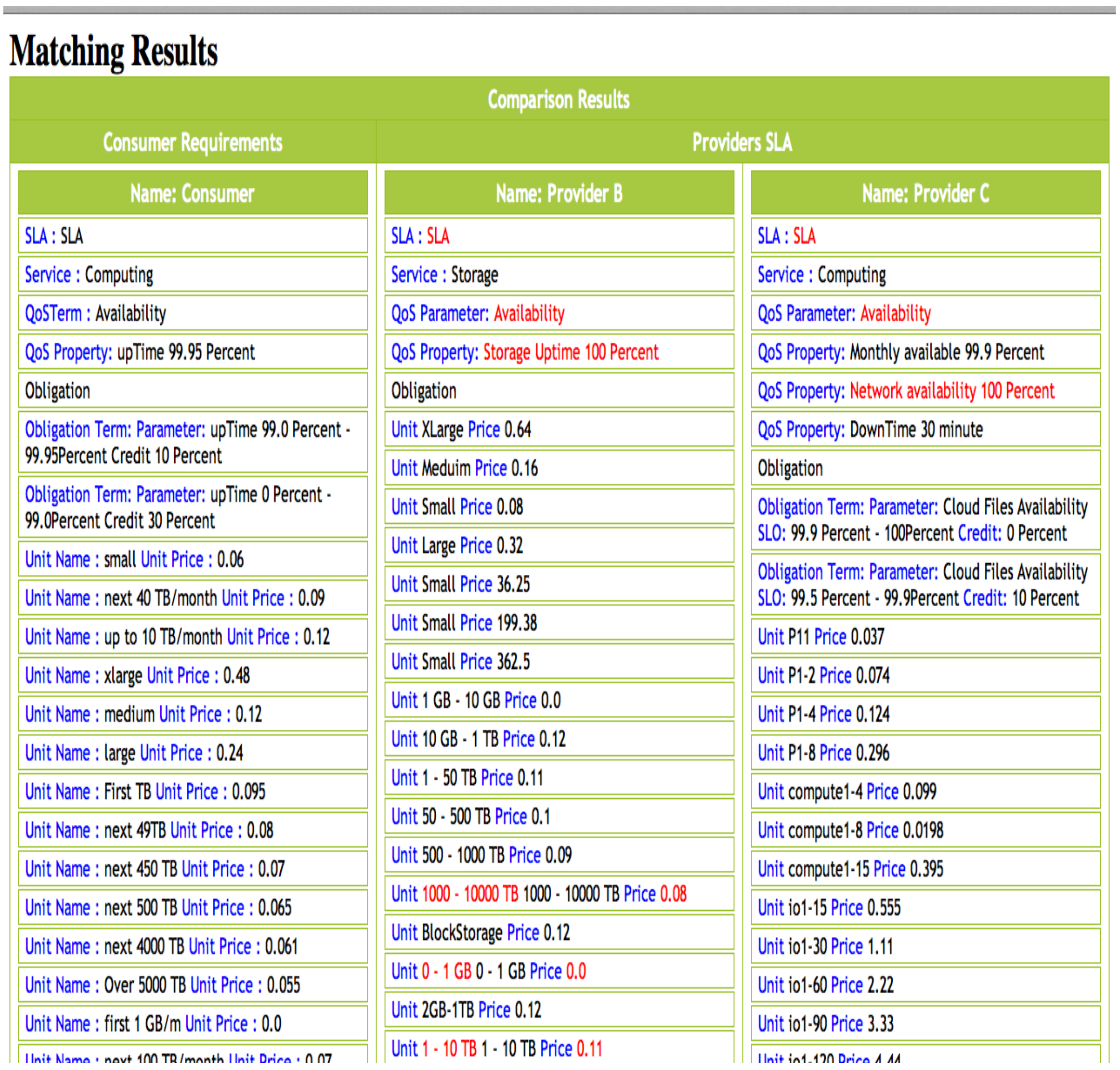
Comparison Results	
Consumer requirements	**ProviderA offers**
Computing Unit VM1	Computing Unit large
Computing Unit VM2	Computing Unit xlarge
DataTransfer OutBound	DataTransfer up to 10 TB/month
QoSProperty ComputeUptime 99.9 Percent	QoSProperty upTime 99.95 Percent
QoSProperty uptime 99.9 Percent	QoSProperty Monthly upTime percentage 99.9 Percent
SLAs ID	SLAs ID Provider A SLAs
SLA Compute	SLA Compute SLA
SLA Storage	SLA Storage SLA
CloudUnit VM1 price 0.06	CloudUnit large price 0.24
CloudUnit VM2 price 0.9	CloudUnit xlarge price 0.48
CloudUnit Storage price 0.07	CloudUnit First TB price 0.095
CloudUnit OutBound price 0.08	CloudUnit up to 10 TB/month price 0.12
Service Computing	Service Computing
Service Storage	Service Storage
Cloud Strorage Storage	Cloud Strorage First TB
QoSTerm Availability	QoSTerm Availability
QoSTerm Availability	QoSTerm Availability
Party Consumer	Party Provider

Figure 7.14: A trace model matching approximate cloud units and similar prices is transformed into an HTML document using EGL

7.2.2.6 Matching cloud providers' SLAs

A cloud brokerage service provides a matching process to match cloud providers' SLA offers. This can help providers to understand how they differ from the competition. Some cloud providers, indeed, provide a comparison of their service offers with other providers offers (i.e. Amazon AWS). It can be used also when a provider compares two different versions of their SLAs. The matching process is the same as the matching discussed in the previous example. However, the matching is performed between two cloud offers.

In this case study, Provider A, for competition reasons, wants to compare its SLA and cloud units with those of Providers B and C. Provider A inspects which elements provide better values in the other two providers. The trace models provides details of the matching points with each provider. However, we provide a simple HTML figure to show the matching points. Figure 7.15 illustrates the optimal matching outcomes between Provider A and Providers B and C. This figure shows that a table of Provider A is compared with providers B and C. In this figure, matched elements are presented as text in *red*.



Matching Results

Comparison Results		
Consumer Requirements	Providers SLA	
Name: Consumer	Name: Provider B	Name: Provider C
SLA : SLA	SLA : SLA	SLA : SLA
Service : Computing	Service : Storage	Service : Computing
QoSTerm : Availability	QoS Parameter: Availability	QoS Parameter: Availability
QoS Property: upTime 99.95 Percent	QoS Property: Storage Uptime 100 Percent	QoS Property: Monthly available 99.9 Percent
Obligation	Obligation	QoS Property: Network availability 100 Percent
Obligation Term: Parameter: upTime 99.0 Percent - 99.95Percent Credit 10 Percent	Unit XLarge Price 0.64	QoS Property: DownTime 30 minute
Obligation Term: Parameter: upTime 0 Percent - 99.0Percent Credit 30 Percent	Unit Meduim Price 0.16	Obligation
Unit Name : small Unit Price : 0.06	Unit Small Price 0.08	Obligation Term: Parameter: Cloud Files Availability SLO: 99.9 Percent - 100Percent Credit: 0 Percent
Unit Name : next 40 TB/month Unit Price : 0.09	Unit Large Price 0.32	Obligation Term: Parameter: Cloud Files Availability SLO: 99.5 Percent - 99.9Percent Credit: 10 Percent
Unit Name : up to 10 TB/month Unit Price : 0.12	Unit Small Price 36.25	Unit P11 Price 0.037
Unit Name : xlarge Unit Price : 0.48	Unit Small Price 199.38	Unit P1-2 Price 0.074
Unit Name : medium Unit Price : 0.12	Unit Small Price 362.5	Unit P1-4 Price 0.124
Unit Name : large Unit Price : 0.24	Unit 1 GB - 10 GB Price 0.0	Unit P1-8 Price 0.296
Unit Name : First TB Unit Price : 0.095	Unit 10 GB - 1 TB Price 0.12	Unit compute1-4 Price 0.099
Unit Name : next 49TB Unit Price : 0.08	Unit 1 - 50 TB Price 0.11	Unit compute1-8 Price 0.0198
Unit Name : next 450 TB Unit Price : 0.07	Unit 50 - 500 TB Price 0.1	Unit compute1-15 Price 0.395
Unit Name : next 500 TB Unit Price : 0.065	Unit 500 - 1000 TB Price 0.09	Unit io1-15 Price 0.555
Unit Name : next 4000 TB Unit Price : 0.061	Unit 1000 - 10000 TB 1000 - 10000 TB Price 0.08	Unit io1-30 Price 1.11
Unit Name : Over 5000 TB Unit Price : 0.055	Unit BlockStorage Price 0.12	Unit io1-60 Price 2.22
Unit Name : first 1 GB/m Unit Price : 0.0	Unit 0 - 1 GB 0 - 1 GB Price 0.0	Unit io1-90 Price 3.33
Unit Name : next 100 TB/month Unit Price : 0.07	Unit 2GB-1TB Price 0.12	Unit io1-120 Price 4.44
	Unit 1 - 10 TB 1 - 10 TB Price 0.11	

Figure 7.15: Example of matching providers offers SLA

7.2.3 Analysing the outcome models of the comparison

In this section, we present how the proposed approach can help to support consumers in making decisions. A number of outcome models - three in our example are available for the consumer to analyse. They can analyse the outcome of matching each provider. As explained in Section 7.2.2, the trace model can be transformed into another model or documents. As we suggested in Section 6.6, a decision matrix can be formed to help consumers. This matrix is used to calculate the score of the alternatives and select the one with the highest score (see Section 6.6). This section describes the different model management tasks that were performed to generate this matrix. First, we have noticed from examples in the last section that there might be missing or redundant values (e.g. when *VM1* from the consumer model is matched with more than *VMs* in a provider model). This is discussed briefly in the following section.

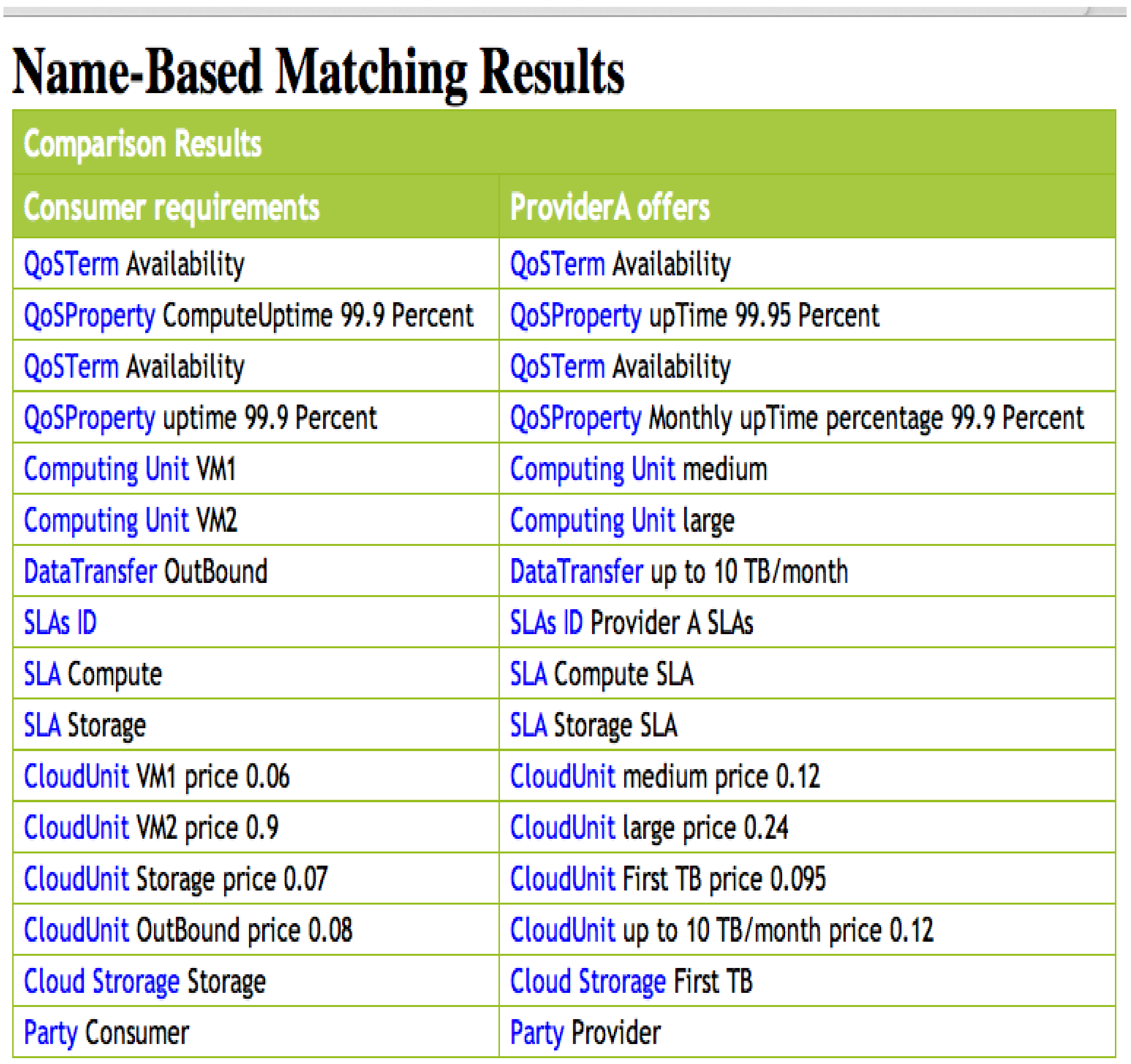
7.2.3.1 Eliminate the redundancy in the outcome model

Based on the results of matching process, some of them (i.e. optimal, name-base matching) returned repeated matches, e.g. *VM1* from the consumer model, is matched with more than one server in the cloud provider model. By using a model management task, e.g. transformation to, for example, remove the redundancy, we generated the following ETL code to eliminate the repeated elements in the outcome (*trace*) model. In this example, we select one of the repeated elements (i.e. the one with the nearest values) and discard all of the others. The Listing 7.2 is an example of this operation.

Listing 7.2: ETL program to eliminate repeated matching

```
pre{
var nMatched : NewMatched!TraceItem ;
var matchedModel : Matched!Trace =  Matched!Trace.allInstances.at
    (0);
var rightSet : Set;
var leftSet : Set;
 ....
}
rule transformTrace
transform m : Matched!Trace
to nm: NewMatched!Trace{
 var nearestItem : Any;
 for (e in m.items){
  nearestItem = e;
  ....
 if (leftSet.size > 0 ){
  for(item in m.items.select(n | n.left = e.left)){
   if(item.right <> nearestItem.right){
    if (not classSet.includes(e.left.EClass.name)){
     if (checkNearest(e,item.right,nearestItem.right) = "nearest"
         )
        nearestItem = nearestItem;
     else nearestItem = item;
     }
    }
   }
   if ( not leftSet.includes(e.left) )
    nm.items.add(nearestItem.equivalent());
   else if (leftSet.includes(e.left)){
    var changedItem : Any;
    changedItem = nm.items.selectOne(n | n.left = e.left);
    nm.items.remove(changedItem);
    nm.items.add(nearestItem.equivalent());
   }
  }
  else nm.items.add(nearestItem.equivalent());
   leftSet.add(e.left);
  }
}
@lazy
rule transformTraceItem
transform m : Matched!TraceItem
to nm : NewMatched!TraceItem{
 nm.left = m.left;
 nm.right = m.right;
}
/* operation checkNearest  */
.......
```

This matching logic is applied to the model illustrated in Figure 7.12. Figure 7.16 shows the name-based matching trace model after applying this ETL task. It is clear from this figure that there is, at most, one matched element for each required element, and any redundancy in Figure 7.12 was eliminated.



Name-Based Matching Results

Comparison Results	
Consumer requirements	ProviderA offers
QoSTerm Availability	QoSTerm Availability
QoSProperty ComputeUptime 99.9 Percent	QoSProperty upTime 99.95 Percent
QoSTerm Availability	QoSTerm Availability
QoSProperty uptime 99.9 Percent	QoSProperty Monthly upTime percentage 99.9 Percent
Computing Unit VM1	Computing Unit medium
Computing Unit VM2	Computing Unit large
DataTransfer OutBound	DataTransfer up to 10 TB/month
SLAs ID	SLAs ID Provider A SLAs
SLA Compute	SLA Compute SLA
SLA Storage	SLA Storage SLA
CloudUnit VM1 price 0.06	CloudUnit medium price 0.12
CloudUnit VM2 price 0.9	CloudUnit large price 0.24
CloudUnit Storage price 0.07	CloudUnit First TB price 0.095
CloudUnit OutBound price 0.08	CloudUnit up to 10 TB/month price 0.12
Cloud Strorage Storage	Cloud Strorage First TB
Party Consumer	Party Provider

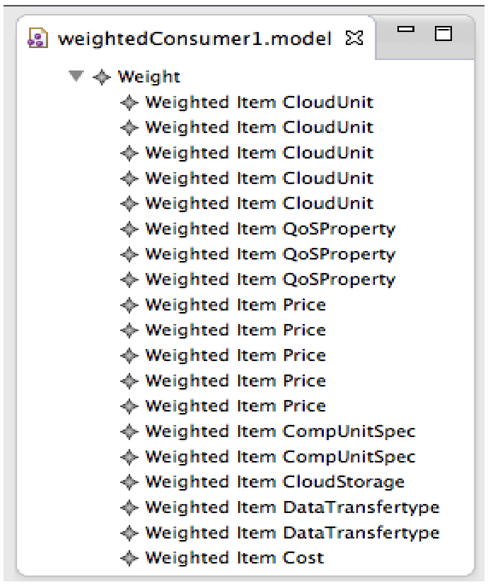
Figure 7.16: A trace model of the Name-Based matching between a consumer model and provider A's model, after eliminating redundant elements, is generated into an html document using SLA.

7.2.3.2 Weighting the requirements

After processing the outcome models, it is time to produce a matrix for the consumer. There are several steps for producing a matrix. First, a weight model is produced, as described in Listing 7.3. This weight model is constructed based on a consumer's requirements. The consumer provides weights to their preferable elements. Then, a model is created to assign weight values to the model elements of the consumer SLA models. In this Listing, we assign weights to the elements of type: *QoSProperty*, *ompUnitSpec*, *CloudUnit*, *Price*, *StorageSpec*, *SLO* and *NetWorkUnit* in the consumer model. The output model of processing this listing is illustrated in Figure 7.17. We can see that a number of cloud SLA model elements are assigned with a weight value. These elements and weights are used to form the matrix, which is described in Section 7.2.3.4. The consumer requires the expected cost to be included in this decision matrix. The cost is discussed in the next section (7.2.3.3).

Listing 7.3: EOL program to eliminate repeated matching

```
var cost : Cost!Allunits = Cost!Allunits.allInstances.at(0);
var m : Sequence := Req.allInstances;
var weight : new weight!Weight;
var weightValue : Real;
var e : Any;
 for (e in m){
  if (e.isKindOf(Req!SLO))
   weight.items.add(createNewItem(e.EClass.name,e,0.6)) ;
  else if (e.isKindOf(Req!QoSProperty))
    weight.items.add(createNewItem(e.EClass.name,e,0.9)) ;
  else if (e.isKindOf(Req!CompUnitSpec))
    weight.items.add(createNewItem(e.EClass.name,e,0.6)) ;
  else if (e.isKindOf(Req!CloudUnit))
    weight.items.add(createNewItem(e.EClass.name,e,0.7)) ;
  else if (e.isKindOf(Req!Price))
    weight.items.add(createNewItem(e.EClass.name,e,0.8)) ;
  else if (e.isKindOf(Req!StorageSpec))
    weight.items.add(createNewItem(e.EClass.name,e,0.6)) ;
  else if (e.isKindOf(Req!NetWorkUnit))
    weight.items.add(createNewItem(e.EClass.name,e,0.6)) ;
 }
 weight.items.add(createNewItem("Cost",cost,0.8));
operation createNewItem(name: Any, itemRef: Any, weightValue:
    Real): Any{
  var item : new weight!WeightedItem;
  item.name = name;
  item.weight = weightValue;
  item.itemRef = itemRef;
  return item;
}
```

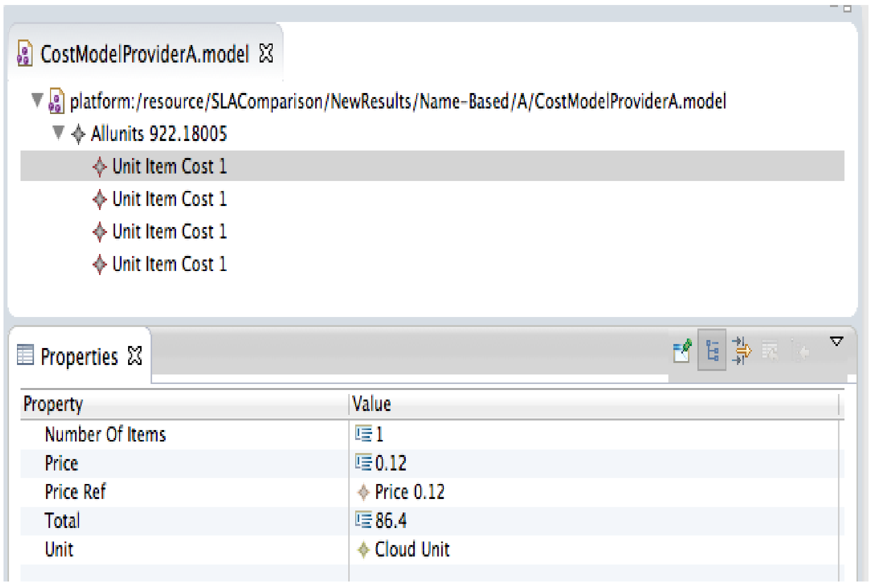

weightedConsumer1.model

- Weight
  - Weighted Item CloudUnit
  - Weighted Item CloudUnit
  - Weighted Item CloudUnit
  - Weighted Item CloudUnit
  - Weighted Item CloudUnit
  - Weighted Item QoSProperty
  - Weighted Item QoSProperty
  - Weighted Item QoSProperty
  - Weighted Item Price
  - Weighted Item Price
  - Weighted Item Price
  - Weighted Item Price
  - Weighted Item Price
  - Weighted Item CompUnitSpec
  - Weighted Item CompUnitSpec
  - Weighted Item CloudStorage
  - Weighted Item DataTransfertype
  - Weighted Item DataTransfertype
  - Weighted Item Cost

Figure 7.17: Weighted elements in a weight model SLA

7.2.3.3 Cost Model

As described in Section 7.2.1 (Figure 7.9), the consumer required two VMs: a storage and data transfer to and from the cloud. There were matches found between consumer VMs and the VMs in the provider models. The storage unit was found in two of the providers' models but none of those providers matched with data transfer "to the cloud". The cost is calculated against the matched units only. VM cost is calculated based on the full utilisation of a month period. These VMs are usually priced per hour. Cloud storage and data transfer costs are calculated based on the size required during a month period. The storage prices provided are per GB. The total cost of consumer requirements is based on the prices provided by Provider A, as illustrated in Figure 7.18. The abstract syntax of this model is described in Section 6.4.



CostModelProviderA.model

```
▼ platform:/resource/SLAComparison/NewResults/Name-Based/A/CostModelProviderA.model
  ▼ Allunits 922.18005
      Unit Item Cost 1
      Unit Item Cost 1
      Unit Item Cost 1
      Unit Item Cost 1
```

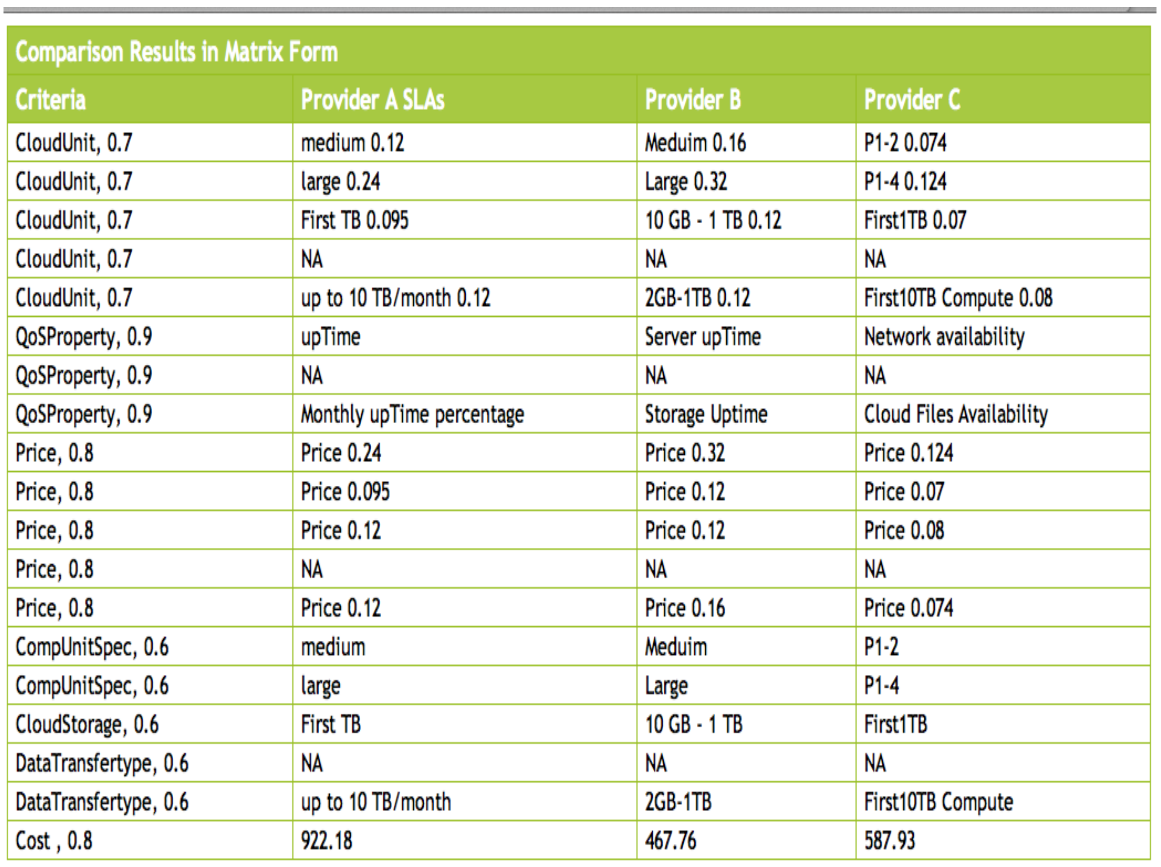
Properties

Property	Value
Number Of Items	1
Price	0.12
Price Ref	Price 0.12
Total	86.4
Unit	Cloud Unit

Figure 7.18: A cost model for matching cloud units of provider ASLA

7.2.3.4 Decision Matrix

There are different multi-criteria decision approaches [93, 145] to selecting cloud provider offers. This work provides the outcome models as an MCDM matrix. The matrix is constructed from the outcome models of the matching process (i.e. trace models produced from Sections 7.2.2 and 7.2.3.1), the weighted model (Section 7.2.3.2) and the cost model (Section 7.2.3.3). Figure 7.19 illustrates a matrix model. As described in the figure, a row in the matrix presents the criteria, while the columns present the cloud provider SLA elements. This can be presented to the consumer, for example, as a HTML document or may take other forms. In this figure, as we can see, there are elements in the consumer model that did not match any of those in the providers' models. This matrix provides information for the consumer about their preferable elements (elements with weights). It is then up to the consumer to decide how to proceed with such information, e.g. to choose one of the MCDM methods [58, 152] to normalise and calculate the values for appropriate offers.



Comparison Results in Matrix Form

Criteria	Provider A SLAs	Provider B	Provider C
CloudUnit, 0.7	medium 0.12	Meduim 0.16	P1-2 0.074
CloudUnit, 0.7	large 0.24	Large 0.32	P1-4 0.124
CloudUnit, 0.7	First TB 0.095	10 GB - 1 TB 0.12	First1TB 0.07
CloudUnit, 0.7	NA	NA	NA
CloudUnit, 0.7	up to 10 TB/month 0.12	2GB-1TB 0.12	First10TB Compute 0.08
QoSProperty, 0.9	upTime	Server upTime	Network availability
QoSProperty, 0.9	NA	NA	NA
QoSProperty, 0.9	Monthly upTime percentage	Storage Uptime	Cloud Files Availability
Price, 0.8	Price 0.24	Price 0.32	Price 0.124
Price, 0.8	Price 0.095	Price 0.12	Price 0.07
Price, 0.8	Price 0.12	Price 0.12	Price 0.08
Price, 0.8	NA	NA	NA
Price, 0.8	Price 0.12	Price 0.16	Price 0.074
CompUnitSpec, 0.6	medium	Meduim	P1-2
CompUnitSpec, 0.6	large	Large	P1-4
CloudStorage, 0.6	First TB	10 GB - 1 TB	First1TB
DataTransfertype, 0.6	NA	NA	NA
DataTransfertype, 0.6	up to 10 TB/month	2GB-1TB	First10TB Compute
Cost , 0.8	922.18	467.76	587.93

Figure 7.19: Comparison results in the matrix model form SLA

One of the most common and simplest methods is the weighted sum [152]. This method has been used to select web services [87, 205]. The used method is explained as follows:

If there are *m* provider services with *n* criteria, the service with the maximum score is selected. The score is calculated as shown in equation 7.1 [177].

$$P(score) = max_i \sum_{j=1}^{n} a_{ij} * w_j \text{ for } i = 1, 2, 3m \tag{7.1}$$

where: P(score) is the alternative (provider SLA), n is the number of criteria, m is the number of the providers SLAs, w_j represents the weight value for the *j-th* criterion and a_{ij} is scaled value for the *i-th* provider SLA and *j-th* criterion. The a_{ij} value is normalized from equation 7.2 [39, 205].

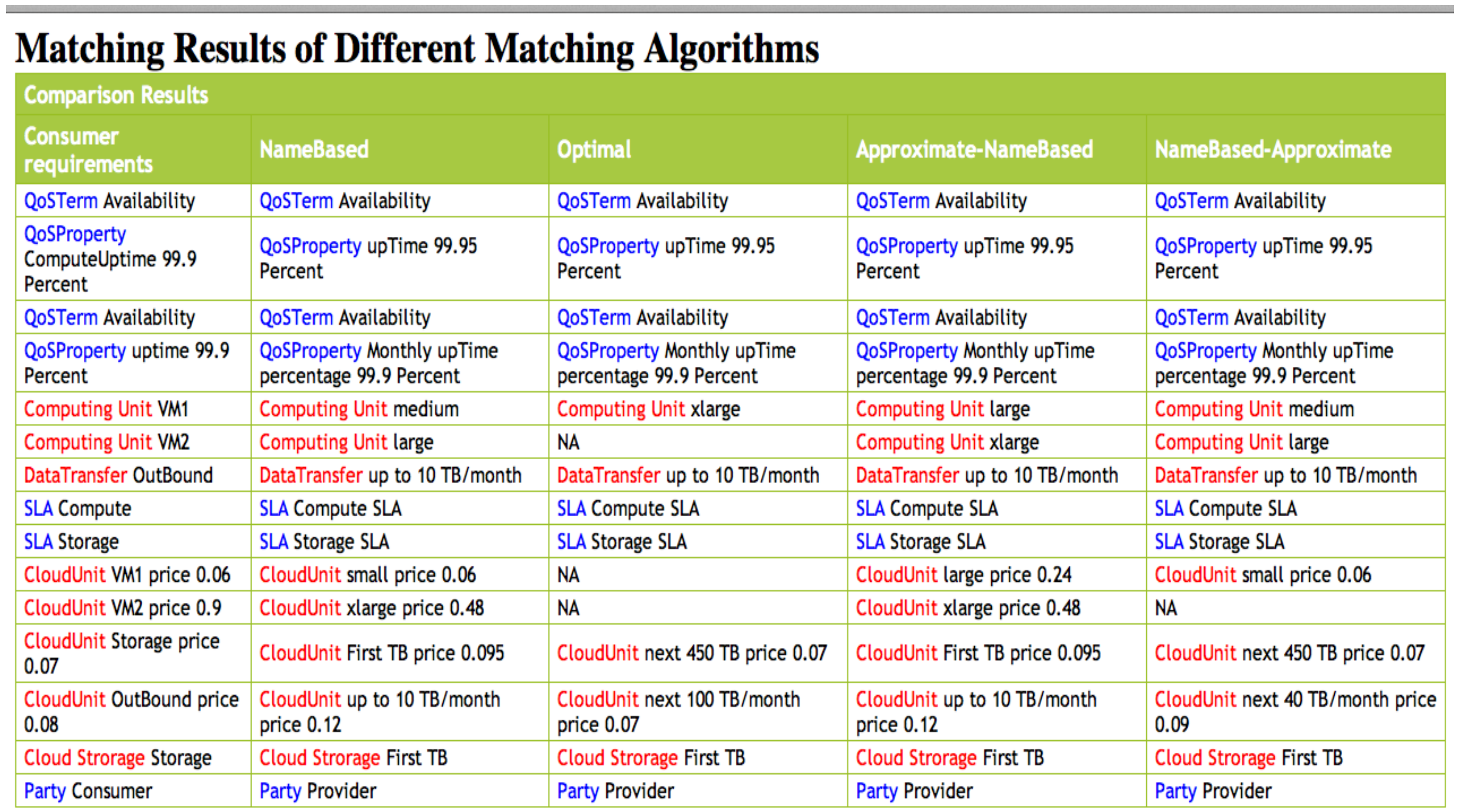
$$a_{ij} = \begin{cases} \dfrac{c_{ij} - c_j^{min}}{c_j^{max} - c_j^{min}} & \text{for criteria more is better and } c_j^{max} - c_j^{min} \neq 0 \\ \dfrac{c_j^{max} - c_{ij}}{c_j^{max} - c_j^{min}} & \text{for criteria more is worse and } c_j^{max} - c_j^{min} \neq 0 \\ 1 & \text{if } c_j^{max} - c_j^{min} = 0 \end{cases} \tag{7.2}$$

The calculations of this MCDM using weighted summation as explained in equations 7.1, 7.2 and the results displayed in Figure 7.19 return values of 0.353, 0.536 and 0.798 for providers A, B, and C respectively. As described in equation 7.1 the best offer has the maximum score. The score of provider C $>$ provider B $>$ provider A, thus provider C has the best offer.

As was discussed in Section 2.4.1.1 different MCDM methods were discussed in the literature such as AHP and TOPSIS that may produce different results. This is not our main concern in this study: however, including different methods as a choice for a consumer might be considered in the future as a potential improvement of this work.

7.2.3.5 Discussing the Outcome of Different Matching Algorithms

This section discusses the outcome models explained in Sections 7.2.2.1 and 7.2.2.3-7.2.2.5. Figure 7.20 illustrates matching the consumer model with provider A using different matching algorithms (i.e. name-based, optimal, approximate-nameBased and nameBased-approximate). The QoSProperty elements in the outcome model of the different matching algorithms were the same. From the figure, we can see that the differences in the results of the matching algorithms were in the *CloudUnits* and *Computing Unit* elements. For example, three matching algorithms return similar outcomes when matching *Computing Unit* VM1, which are: medium and large, while optimal matching returns xlarge.



Matching Results of Different Matching Algorithms

Comparison Results				
Consumer requirements	NameBased	Optimal	Approximate-NameBased	NameBased-Approximate
QoSTerm Availability	QoSTerm Availability	QoSTerm Availability	QoSTerm Availability	QoSTerm Availability
QoSProperty ComputeUptime 99.9 Percent	QoSProperty upTime 99.95 Percent	QoSProperty upTime 99.95 Percent	QoSProperty upTime 99.95 Percent	QoSProperty upTime 99.95 Percent
QoSTerm Availability	QoSTerm Availability	QoSTerm Availability	QoSTerm Availability	QoSTerm Availability
QoSProperty uptime 99.9 Percent	QoSProperty Monthly upTime percentage 99.9 Percent	QoSProperty Monthly upTime percentage 99.9 Percent	QoSProperty Monthly upTime percentage 99.9 Percent	QoSProperty Monthly upTime percentage 99.9 Percent
Computing Unit VM1	Computing Unit medium	Computing Unit xlarge	Computing Unit large	Computing Unit medium
Computing Unit VM2	Computing Unit large	NA	Computing Unit xlarge	Computing Unit large
DataTransfer OutBound	DataTransfer up to 10 TB/month	DataTransfer up to 10 TB/month	DataTransfer up to 10 TB/month	DataTransfer up to 10 TB/month
SLA Compute	SLA Compute SLA	SLA Compute SLA	SLA Compute SLA	SLA Compute SLA
SLA Storage	SLA Storage SLA	SLA Storage SLA	SLA Storage SLA	SLA Storage SLA
CloudUnit VM1 price 0.06	CloudUnit small price 0.06	NA	CloudUnit large price 0.24	CloudUnit small price 0.06
CloudUnit VM2 price 0.9	CloudUnit xlarge price 0.48	NA	CloudUnit xlarge price 0.48	NA
CloudUnit Storage price 0.07	CloudUnit First TB price 0.095	CloudUnit next 450 TB price 0.07	CloudUnit First TB price 0.095	CloudUnit next 450 TB price 0.07
CloudUnit OutBound price 0.08	CloudUnit up to 10 TB/month price 0.12	CloudUnit next 100 TB/month price 0.07	CloudUnit up to 10 TB/month price 0.12	CloudUnit next 40 TB/month price 0.09
Cloud Strorage Storage	Cloud Strorage First TB	Cloud Strorage First TB	Cloud Strorage First TB	Cloud Strorage First TB
Party Consumer	Party Provider	Party Provider	Party Provider	Party Provider

Figure 7.20: Matching results of matching consumer requirements with provider'A SLA using different matching algorithms

Also, optimal matching did not return any matched element with *Computing Unit* VM2 and two *CloudUnits*. This is because these elements contain more than one attributes (e.g. computing unit attributes are: cpu, RAM and storage), which are involved in the matching algorithm. For example, in the optimal matching, *Computing Unit* VM1 is matched with the xlarge one, but no match was found for *CloudUnit* VM1 because the price (which is an attribute of the *CloudUnit* element) did not match.

We cannot objectively decide which algorithm is better in the case of matching *CloudUnit*, this depends on the requester's decision about which unit to choose: the one with the nearest price value or nearest specifications, e.g. *CloudUnit* VM1 matched with small and large for the name-based and approximate-nameBased algorithm.

In this thesis, we do not study the differences between these matching algorithms, but we provide them as available options for consumers and providers. In general, optimal and approximate matching algorithms are more specific than name-based matching; thus, the possibility that name-based matching returns matches between two elements is higher than the other two matching approaches. For consumers who are uncertain (or unspecific) about their required values name-based matching might be appropriate. For consumers with hard requirements, e.g. prices that should not exceed specific values, optimal matching may be used. For consumers with approximate values approximate matching might be more appropriate. However, when we compare *CloudUnits* elements, the results do not show what we expected from optimal and approximate matching. For example, the optimal matching of the *CloudUnits* of type *Cloud Storage* type returns higher size specifications than required, because the price per GB is the same as required. This is because the cloud providers provide cheaper prices per GB for bigger storage capacity. In this case, it might be more appropriate to calculate and compare the cost of the total required storage than comparing prices per GB. In this example, name-based and approximate-nameBased matching is better than the other two matching techniques, because they return the cloud unit that matches the required size and this reflects the real cost of this storage.

7.3 Evaluation of the contributions

In this section, we examine the contributions of this thesis based on the case study.

7.3.1 Requirements of semi-automatic cloud computing SLAs

In Chapter 3, we used requirement engineering to analyse the semi-automatic comparison of cloud SLAs. A number of case studies were defined to determine requirements to semi-automate the comparison of cloud SLAs. The contribution of this chapter is such that a set of requirements based on the MDE principles were provided. To semi-automate the comparison, an approach of metamodel, models, model comparison and model transformation were proposed. Other SLA specifications were developed, but not specified for cloud SLAs. Many of the electronic SLAs were developed to negotiate and monitor QoS parameters, while this study proposes an SLA specification for comparison as a pre-step to negotiation and monitoring. One of the first requirements was to develop a cloud SLA metamodel.

7.3.2 Cloud SLA Metamodelling language

In Chapter 4, a metamodel for cloud SLA was proposed. The purpose of this metamodel is to enable the semi-automation of the cloud SLA comparison. The first version of developing this metamodel was inspired by the existing SLA language, i.e. WSLA [102]. This metamodel was constructed iteratively and incrementally. Different pre-defined cloud provider SLAs were used as example models (e.g. Amazon EC2 SLAs [3], RackSpace [21] and GoGrid [16]).

We used the metamodel to produce example models and used these models in small matching process examples. These matching processes were developed by using Epsilon ECL. In this proof of concept study, and by experimenting with the cloud SLA from different pre-defined cloud provider SLAs and QoS concepts as a main component to the SLA, we determined that this constructed SLA fulfils its requirements. The models that are defined by this SLA are well-defined as they conform to a metamodel.

The constructed models of the public cloud providers pre-defined SLAs are illustrated in Section 7.2.1. The benefit of the cloud SLA is such that different offers can be produced with similar structures and concepts. It also includes the unit specification and prices, which is usually requested by the consumer. Another benefit of this metamodel is that both the consumer and provider can define their models by using its abstract syntax. It provides a communication language between them. The contribution of this SLA is the specification of the cloud QoS concepts, such as *Availability*, where the QoS parameters can be defined within these concepts. Providers can define parameters using their own terminologies, but they have to be specific about to which concept they belong. This cloud SLA is constructed to define IaaS clouds; it specifies only the basic IaaS cloud services. As a result, we take the abstract view that an IaaS cloud service is composed of API, hardware and networking [117].

7.3.3 Evaluation of the comparison logics

As in the examples shown in Section 7.2.2, there are different possibilities available for defining a match between two models. The suitability and appropriateness of a particular logic depends on the user requirements, which are implemented using a set of ECL rules and by using a particular set of input models.

For the cloud computing units, the providers provide a set of units with different characteristics that are priced differently, while the QoS property provided by the providers is less than the cloud units. The cloud computing providers provide a set of units with different characteristics which are priced differently, and the number of QoS parameters provided by the providers is less than the number of cloud units. These two concepts may need different matching logics. We ran a number of examples and test the outcome from the three matching logics. These experiments included the cloud SLAs. We ran an example comparing different cloud offers. The results of each comparison were analysed. By experimenting with the three matching logics on a number of cloud SLA concepts, we determined that this matching process fulfils its requirements.

Further studies and examples are needed to determine the appropriate matching combination that may return reasonable output models, which can help a consumer to make decisions. In the matching logics, we match the concepts of the cloud SLAs domain and also match the values of these concepts. We defined three matching logics, based on matching values; thus we had exact and approximate matches, while similar logic matches only the concepts of the cloud SLAs. The matching process was developed using the ECL.

7.3.4 Evaluation of the Supporting Decisions approach

In Chapter 6, we defined a number of small metamodels (i.e. weight, cost and matrix models) and a set of model management tasks. The purpose of these small metamodels is to define cloud consumer requirements and the expected cost, where they are created separately from the cloud SLAs. A number of tasks were created to manage the outcome models of the matching processes. These outcome models are transformed into a matrix model. We provide, as a proof of concept, that a consumer can analyse the outcome of the matching process. This analysis step is proposed based on the results of a matching process. A matching process finds more than one match for a specific requirement or no matches are found. In the case where no matches are found, a different matching logic can be processed while, in the case of repeated matches, a consumer decides which entry can be removed. These processes are a pre-step to constructing a matrix model.

Studies such as [83, 182], suggested an MCDM approach in such a case, when a selection has to be made between alternatives, based on multiple criteria. This chapter provides the cloud consumer with the matching process as a decision matrix. This section provides the outcome of the processing model in the form of a decision matrix and a weighted sum as the MCDM method for choosing a cloud SLA offer.

A number of experiments were performed to test the outcome models of each process (i.e. weight model, trace model after elimination, cost model and matrix model) and use those outcomes as input models in the other process. We determined, after several experiments, the processes of assigning weight (Section 6.3), creating and calculating a cost model (Section 6.4) and creating a matrix model (Section 6.3.1) to fulfil the requirements.

7.4 Evaluation of the thesis contributions

The contributions of this thesis will now be assessed in terms of the distinct characteristics which were identified in Section 1.5.

1. **Modelling cloud SLA**: The proposed approach provides an abstract syntax for defining the cloud SLA models, taking into account the specification of their characteristics, such as service type, QoS concepts and cloud units (Section 4.3).

This abstract syntax can be used to define consumer and provider models. Cloud providers with different cloud SLA structure models were produced using the abstract syntax. Therefore, we conclude that the proposed approach satisfies modelling cloud SLA.

2. **Comparison process**: The proposed approach provides a comparison process which compares a cloud SLA to a number of cloud provider SLAs. Furthermore, different matching logics were provided which are: exact, approximate and similar matching (Section 5.4). Moreover, different cloud SLA concepts can use different matching logics to find a match with a consumer's requirements.

3. **Comparing cloud providers**: The comparison process was constructed for model comparisons, i.e. cloud consumer models compared with cloud provider models. The constructed metamodel, as described in Section 4.3, is used to define both cloud providers and consumers SLAs. Thus the comparison process provides a comparison for matching providers SLAs when they conform to the same metamodel of consumer SLAs.

4. **Supporting decisions**: The proposed approach provides a number of model management tasks to analyse and transform the models involved in the comparison process. It provides a model for consumer preferences to assign weight values. It also calculates the expected cost for each provider and transforms all these models to one model (i.e. matrix model). This matrix includes information about the matched services, consumer preferences and the expected cost. To this end, a consumer can then decide the approach of processing the MCDM.

5. **Semi-automation**: This approach provides a semi-automatic comparison and selection of cloud SLAs. This is achieved by the matching process and the provision of the MCDM matrix (by using model management tasks, i.e. model comparison and transformation).

7.5 Summary

In this chapter, we provided a case study to evaluate the contribution of this thesis. The case study shows the feasibility of the proof-of-concept approach that we have defined, that can be used to address the hypothesis. First, we implemented our proposed metamodel to define a number of pre-defined cloud provider SLAs. This metamodel was implemented to define the consumer requirements. Then, a process of matching and selecting cloud SLA among provider SLAs was provided. The evaluation outlines the feasibility of applying MDE principles, such as metamodelling and model management tasks, in the process of comparing and selecting cloud SLAs.

Chapter 8

Conclusions and Future Work

8.1 Introduction

In this thesis, we presented a semi-automatic MDE approach for comparing and selecting cloud computing SLAs. An SLA metamodel as proposed for cloud providers offers and consumer requirements. A process composed of two phases for the automated comparison of SLAs was proposed. The first phase is a matching process for matching SLAs one by one. The second process is supporting the consumers' decision.s The input of the first phase are is cloud SLA models. This SLA conforms to the proposed metamodel. The SLA model structure consists of the party, service, obligation and cloud resource units. The SLA defines the QoS terminology as well as the resource units' terminology. The output of the second phase is the MCDM model, where we applied an MCDM approach to select a cloud SLA offer.

This research aimed to investigate the hypothesis described in section 1.5, specifically:

Can MDE principles and tools support the precise modelling of cloud computing SLAs in such a way that cloud stakeholders can define their offers and demands? In addition, can the MDE principles and tools enable a semi-automate comparison process for cloud computing SLAs, in order to help cloud stakeholders to make better decisions about the appropriateness of the offerings from different cloud providers?

The hypothesis detailed above is investigated by defining the research objectives presented in Section 1.6.

1. To provide mechanisms for modelling cloud SLAs, based on MDE principles, techniques and tools (see Chapter 4).
2. To identify and describe precisely different scenarios for comparing cloud SLAs (see Chapter 5).
3. To provide mechanisms for automatically or semi-automatically comparing cloud SLAs, based on MDE principles, techniques and tools (see Chapters 5 and 6).
4. To propose mechanisms for presenting the results of comparing cloud SLAs in machine processable forms (see Sections 5.7 and 6.6).
5. To evaluate the above mechanisms via examples inspired by real SLAs from current cloud providers (see Chapter 7).

The following sections summarise the contributions of this research regarding the thesis hypothesis and research objective. Then, the limitations of the proposed approach and future work are discussed in the following sections.

8.2 Contribution

The contributions of the thesis are summarized in this section.

8.2.1 Cloud SLA Metamodel

The first key contribution of this thesis is a language for cloud SLAs, which is presented in Chapter 4. We observed in the literature that there is no standardised cloud SLA language for cloud computing, and public cloud providers define their SLAs using different terminologies, which makes it difficult for consumers to systematically compare and select between the public cloud SLAs.

The cloud SLA language we present is a metamodel (consisting of abstract syntax and well-formedness constraints) for defining cloud SLAs, specifically for IaaS clouds. The purpose of this abstract syntax is to support the construction of well-defined cloud SLAs models. The abstract syntax includes specifications of QoS concepts (which define a number of QoS parameters, specifications of the obligation terms (i.e. SLO and credits) and cloud resource units. Each cloud unit defines the characteristics of the unit and its price.

The abstract syntax allows both cloud providers and consumers to define a well-structured cloud SLA for use in systematic comparison and selection. This SLA can be then be used as a starting point for cloud negotiations; in particular, the metamodel makes it possible to define different cloud SLA models, which can then be compared semi-automatically.

8.2.2 Comparison Process

One of our research objectives was to semi-automate the comparison process for cloud SLAs. In Chapter 5, we presented such a process. This comparison process includes a matching logic that can be applied to two cloud SLAs. The matching logic is implemented using a model comparison language (ECL). The comparison process can be used to match the concepts of SLA (i.e. matching model elements and properties) as well as matching the values of the SLA concepts. Different matching algorithms were provided, which can be used to support different comparison scenarios and satisfy different customer requirements. One of these algorithms is arguably too restrictive (optimal match), while another is too flexible (name-based match); however, these logics are needed in different scenarios, because clouds are differently defined and specify various QoS and resource units, as well as defining their structures differently. By providing different algorithms, we make it possible for the consumers the possibility to find matches between arbitrary selections of model elements that define their cloud SLA. By using this matching logic, we achieve semi-automation for cloud SLA matching. This matching process matches SLAs one by one, which facilitates, for the providers, the analysis of the similarity and differences between two SLAs.

8.2.3 Methodology for selecting between a set of cloud SLAs

One of our objectives is to help cloud consumers to select a cloud provider, based on cloud SLA comparison. In Chapter 6, we present a methodology to semi-automate the decision-making aspect of this selection process.

This selection process provides a weighting model that allows a cloud consumer to add weight values for the required elements in the cloud SLA model. The weighting model allows the customisation of the comparison process, as it enables a consumer to emphasise those attributes/values that are most important to them. The process make it possible to define a cost model, which can be used as the key criterion for making decisions about which cloud SLAs to select, based on the results of the comparison process. Overall, the contribution of these two models is that they define consumer-specific requirements in models that are separated from the cloud SLA metamodel. This allows us to define one metamodel for consumers and providers, whilst still capturing separate consumer-specific requirements in a model. This helped us simplify the matching process, where the model comparison was constructed based on comparing models that conform to the same metamodel, while still taking into account information from the consumer, which is necessary to tailor the comparison results to his/her own requirements.

The methodology provides a set of model management tasks to create the models (cloud SLA, weighting model, cost model) and also to analyse the results of the matching process. This methodology does not provide a selection of what is the *best* offer but, rather, transforms all of the information – i.e. the outcome models of the matching process, the weighting model and the cost model – into a single model, a *decision matrix*. The decision matrix can then be used by consumers to choose the offer that best matches their requirements. In a sense, our methodology reduces and restricts the search space of cloud SLAs for consumers.

The proposed metamodel is composed of a set of metamodels, e.g. cloud resource units. The cloud resource unit models are defined separately from the cloud SLAs, which can refer to these units. This makes the cloud units modular in nature, and so applicable to other process such as resources provisioning and resource monitoring.

8.2.4 Evaluation Results

Chapter 7 presents an evaluation of the thesis' results, including the thesis' hypothesis. We evaluated the expressiveness of the SLA metamodel in a largely test-based way, by showing how it supports the specification of different pre-defined cloud SLA models from different providers. These models were then used in the examples of the different comparison and matching processes. Then examples of creating weighting, cost and finally matrix models were provided. In Section 7.4, we assessed the feasibility and benefits of the contributions of this thesis.

8.3 Limitations of the cloud SLAs comparison and selection approach

8.3.1 Lack of support for SLA negotiation and monitoring

The purpose of the proposed metamodel is to support consumers in selecting a cloud offer based on cloud providers' predefined SLAs, which can be described as a pre-negotiation step. The specifications of this metamodel did not mention any agreement terms section as, for example, in a WS-Agreement, which defines the parameters under negotiation. Also, this SLA does not specify the service interface or service objective, which makes it possible to monitor the parameters of the agreement.

8.3.2 Lack of support for non-IaaS cloud services

The proposed metamodel focuses on IaaS cloud computing. It specifies IaaS cloud services: Networking, Storage and Computing. Other types of cloud – PaaS and SaaS – may define different service types. Furthermore, the cloud units of such clouds are different to the IaaS cloud units. This cloud SLA metamodel also provides specifications regarding the basic services, while these services can have more components. For example, the computing service is composed of API, hardware and networking.

8.3.3 Lack of support for cloud SLAs not defined using the proposed abstract syntax

This approach supports matching cloud SLAs through model comparison tasks. However, the pre-defined cloud SLA offers are structured differently using different terminology. This approach does not consider SLAs defined using other languages. The approach should facilitate transformation from different cloud offers, e.g. as Amazon EC2 and S3 models to our proposed models.

8.4 Future work

8.4.1 Support for the cloud SLA management life cycle

The proposed approach defines cloud SLA models, for capturing QoS parameters and cloud resource units, which are used to help consumers support their decisions by automating the comparison and selection of a cloud offer, but not the other SLA management process such as negotiation and monitoring. Therefore, automating the negotiation and monitoring of the cloud SLA management process is an interesting area for future work. This can be done, by for example, using MDE techniques to transfer the cloud SLA to one of the reviewed SLAs in Section 2.4.5, e.g. WS-Agreement or SLA*.

8.4.2 Support for non-cloud SLAs models

The proposed approach defines the models to be involved in the comparison process. However, cloud providers define their cloud SLAs using different models (e.g. different terminologies, structures and textual descriptions (documents)). Within the textual descriptions, it is difficult to identify the cloud SLA concepts precisely. Therefore, in order to automate the construction of the cloud SLA models, in this thesis, the cloud pre-defined SLA models were implicitly transformed into instances of the proposed metamodel. One possible approach is to provide explicit transformation rules from pre-defined SLAs into SLA models that conform to the proposed metamodel. This transformation would necessarily provide general rules for critical cases, and thereafter additional rules would be needed to handle unusual or unexpected SLA concepts [67].

8.4.3 Cloud SLAs Matching Patterns

The differences between cloud SLAs raises issues in the matching process. The optimal matching logic was too restrictive to find a match between certain concepts (e.g. Computing units), while the name-based matching logic was too relaxed (it matched all of the requirements with all of the offered computing units). Further analysis might result in the identification of patterns of different matching logics that are specific to the concept of cloud SLAs. This will provide tailored rules that match each concept in cloud SLAs.

8.4.4 Large-scale Experiments

We have conducted small-scale proof-of-concept case studies in this thesis, to demonstrate the key concepts and feasibility of semi-automatically comparing and selecting cloud SLAs. Large-scale experiments need to be carried out to provide quantitative data on the wider feasibility of the proof-of-concept, particularly in terms of its accuracy (e.g., in terms of false positives or false negatives), and scalability.

Acronyms

AHP Analytical Hierarchical Process.

API Application Programming Interface.

AWS Amazon Web Services.

CRM Customer Relationship Management.

ECL Epsilon Comparison Language.

EGL Epsilon Generation Language.

EOL Epsilon Object Language.

ETL Epsilon Transformation Language.

EVL Epsilon Validation Language.

HRM Human Resource Management.

IaaS Infrastructure as a Service.

ITIL IT Infrastructure Library.

KPI Key Performance Indicator.

M2M Model-to-Model.

M2T Model-to-Text.

MCDM Multi-Criteria Decision Making.

MDE Model Driven Engineering.

MTBF	Mean Time Between Failure.
MTTR	Mean Time TO Repair.
PaaS	Platform as a Service.
QoS	Quality of Service.
SaaS	Software as a Service.
SLA	Service Level Agreement.
SLO	Service Level Objective.
T2M	Text-to-Model.
VM	Virtual Machine.
WSLA	Web Service Level Agreement.
XML	Extensible Markup Language.

References

[1] Release: Amazon EC2: on 2006-11-30, . URL http://aws.amazon.com/releasenotes/Amazon-EC2/532. 32, 33

[2] Amazon CloudWatch, . URL http://aws.amazon.com/cloudwatch/. 26

[3] Amazon Elastic Compute Cloud (Amazon EC2), . URL http://aws.amazon.com/ec2/. 26, 32, 33, 58, 88, 125, 190, 235

[4] Amazon EC2 Service Level Agreement, . URL http://aws.amazon.com/ec2-sla/. xii, 7, 34, 36, 66, 72, 112, 121, 122, 127

[5] Amazon Simple Storage Service (Amazon S3), . URL http://aws.amazon.com/s3/. 32, 33, 88

[6] Amazon S3 Service Level Agreement, . URL http://aws.amazon.com/s3-sla/. 34, 36

[7] Amazon EC2 Instances, . URL http://aws.amazon.com/ec2/instance-types/. 105, 106, 118

[8] Amazon EC2 Pricing, . URL http://aws.amazon.com/ec2/pricing/. 94, 106, 110, 118, 119

[9] Eclipse Modeling Framework Project (EMF). URL http://www.eclipse.org/modeling/emf/. 41

[10] Epsilon, . URL http://www.eclipse.org/epsilon/. 43, 44, 197

[11] The epsilon Book, . URL http://www.eclipse.org/epsilon/doc/book/. xiv, 42, 44, 45, 46, 48, 156, 159, 179

[12] What is Force.com? URL http://www.salesforce.com/platform/what/. 30

[13] Google App Engine: Platform as a Service. URL https://developers.google.com/appengine/. 30

[14] Graphical Modeling Project (GMP). URL http://www.eclipse.org/modeling/gmp/. 44

[15] GoGrid, . URL http://www.gogrid.com/. 32, 88

[16] GoGrid SLA, . URL http://www.gogrid.com/legal/sla.php. 7, 34, 36, 235

[17] Itil glossary and abbreviations. URL http://www.itil-officialsite.com/InternationalActivities/TranslatedGlossaries.aspx. 12

[18] A Multi-Criteria Decision-making Model for an i. 5, 50

[19] OMG's MetaObject Facility. URL http://www.omg.org/mof/. 41

[20] Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. URL http://www.omg.org/spec/QVT/1.1/. 41

[21] RackSpace, . URL http://www.rackspace.com/. 7, 32, 33, 88, 99, 125, 235

[22] RackSpace (UK), . URL http://www.rackspace.co.uk/. 99

[23] Rackspace Service Level Agreement, . URL http://www.rackspace.com/cloud/legal/sla/. 34, 36

[24] RackSpace cloud files comparison, . URL http://www.rackspace.co.uk/cloud/files/compare. 75

[25] Cloud Load Balancers Comparison, . URL http://www.rackspace.co.uk/cloud/load-balancers/compare. 75

[26] Sla Management Foundations, . URL http://sla-at-soi.eu/research/focus-areas/sla-management-foundations/. 3, 55, 69

[27] Standard Terms used in SLA SOI, . URL wiki.sla-at-soi.eu. xi, 56, 57

[28] SLA Template, . URL http://www.slatemplate.com/. xi, 14

[29] Unified Modeling Language. URL http://www.uml.org/. 40

[30] Web Service Level Agreements (WSLA) Project. URL http://www.research.ibm.com/wsla/. 59

[31] What Is Google App Engine:. URL http://code.google.com/appengine/docs/whatisgoogleappengine.html. 30

[32] Windows Azure SLA. URL http://www.microsoft.com/windowsazure/sla/. 7, 36

[33] OpenNebula.org Open Source Data Center Virtualization. URL http://www.eucalyptus.com/. 32

[34] Heroku. URL https://www.heroku.com. 30

[35] OpenNebula.org Open Source Data Center Virtualization. URL http://opennebula.org/. 32

[36] Toward dynamic and attribute based publication, discovery and selection for cloud computing. *Future Generation Computer Systems*, 26(7):947 – 970, 2010. 4

[37] F. Al-Kandari and R.F. Paige. Modelling and Comparing Cloud Computing Service Level Agreements. In *Proc. 1st International Workshop on Model-Driven Engineering for High Performance and CLoud computing (MDHPCL)*. ACM Press, 2012. xii, xviii, 126, 128

[38] M. Alhamad, T. Dillon, and E. Chang. Conceptual SLA framework for cloud computing. In *Digital Ecosystems and Technologies (DEST), 2010 4th IEEE International Conference on*, pages 606 –610, april 2010. 4, 7, 27, 54, 55, 63, 88

[39] Jayanath Ananda and Gamini Herath. A critical review of multi-criteria decision making methods with special reference to forest management and planning. *Ecological Economics*, 68(10):2535 – 2548, 2009. ISSN 0921-8009. 232

[40] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web Services Agreement Specification (WS-Agreement). *Global Grid Forum*, 31:1–47, 2007. 16, 60, 69

[41] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53:50–58, April 2010. ISSN 0001-0782. 2, 26, 27

[42] I.Budak Arpinar, Ruoyan Zhang, Boanerges Aleman-Meza, and Angela Maduko. Ontology-driven web services composition platform. *Information Systems and e-Business Management*, 3(2):175–199, 2005. ISSN 1617-9846. 72

[43] Mark L. Badger, Timothy Grance, Robert Patt-Corner, and Jeffrey M. Voas. Cloud computing synopsis and recommendations. Technical report, National Institute of Standards and Technology, May 2012. 32

[44] P. Balakrishnan, S.T. Selvi, and G.R. Britto. GSMA based Automated Negotiation Model for Grid Scheduling. In *Services Computing, 2008. SCC '08. IEEE International Conference on*, volume 2, pages 569–570, 2008. 15

[45] P. Balakrishnan, S. Thamarai Selvi, and G. Rajesh Britto. Service Level Agreement Based Grid Scheduling. In *Web Services, 2008. ICWS '08. IEEE International Conference on*, pages 203–210, 2008. 15

[46] Z. Balfagih and M.F. Hassan. Quality Model for Web services from Multi-stakeholders' Perspective. In *Information Management and Engineering, 2009. ICIME '09. International Conference on*, pages 287–291, 2009. 17, 18

[47] U. Bellur and R. Kulkarni. Improved matchmaking algorithm for semantic web services based on bipartite graph matching. In *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pages 86–93, July 2007. 52, 53

[48] J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. Modeling in the Large and Modeling in the Small. In Uwe Assmann, Mehmet Aksit, and Arend Rensink, editors, *Model Driven Architecture*, volume 3599 of *Lecture Notes in Computer Science*, pages 33–46. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-28240-2. 43

[49] Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, 2005. ISSN 1619-1366. 3, 39, 40

[50] Jean Bézivin. Model Driven Engineering: An Emerging Technical Space. In Ralf Lmmel, Joo Saraiva, and Joost Visser, editors, *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science*, pages 36–64. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-45778-7. 3

[51] X. Blanc, I. Mounier, A. Mougenot, and T. Mens. Detecting model inconsistency through operation-based model construction. In *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, pages 511–520, 2008. 43

[52] Paolo Bocciarelli and Andrea D'Ambrogio. A model-driven method for describing and predicting the reliability of composite services. *Software & Systems Modeling*, 10(2):265–280, 2011. ISSN 1619-1366. 116

[53] Miguel L Bote-Lorenzo, Yannis A Dimitriadis, and Eduardo Gómez-Sánchez. Grid characteristics and uses: a grid definition. In *Grid Computing*, pages 291–298. Springer, 2004. 24

[54] Cédric Brun and Alfonso Pierantonio. Model differences in the eclipse modeling framework. *UPGRADE, The European Journal for the Informatics Professional*, 9(2):29–34, 2008. 42

[55] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599 – 616, 2009. ISSN 0167-739X. 24, 26

[56] Bu-Qing Cao, Bing Li, and Qi-Ming Xia. A Service-Oriented Qos-Assured and Multi-Agent Cloud Computing Architecture. In MartinGilje Jaatun, Gansen Zhao, and Chunming Rong, editors, *Cloud Computing*, volume 5931 of *Lecture Notes in Computer Science*, pages 644–649. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-10664-4. 56

[57] Ed.D. Carl J. Wenning. Percent difference percent error. Illinois State University, Dept of Physics. URL http://www.phy.ilstu.edu/slh/. 165

[58] Prasenjit Chatterjee, Vijay Manikrao Athawale, and Shankar Chakraborty. Materials selection using complex proportional assessment and evaluation of mixed data methods. *Materials & Design*, 32(2):851 – 860, 2011. ISSN 0261-3069. 52, 230

[59] Avraam Chimaris and GeorgeA. Papadopoulos. Implementing QoS Aware Component-Based Applications. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE*, volume 3291 of *Lecture Notes in Computer Science*, pages 1173–1189. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-23662-7. 16

[60] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. A Metamodel Independent Approach to Difference Representation. *Journal of Object Technology*, 6(9):165–185, 2007. 42

[61] Tony Clark, Paul Sammut, and James Willans. Applied metamodelling: a foundation for language driven development. 2008. 40

[62] Mark Claypool, Phong Le, Makoto Wased, and David Brown. Implicit Interest Indicators. In *Proceedings of the 6th International Conference on Intelligent User Interfaces*, IUI '01, pages 33–40, New York, NY, USA, 2001. ACM. ISBN 1-58113-325-1. 54

[63] M. Comuzzi, C. Kotsokalis, G. Spanoudakis, and R. Yahyapour. Establishing and Monitoring SLAs in Complex Service Based Systems. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 783–790, July 2009. 63, 64

[64] Gerard Conway and Edward Curry. Managing Cloud Computing: A Life Cycle Approach. In *2nd International Conference on Cloud Computing and Services Science (CLOSER 2012)*, pages 198–207, Porto, 2012. 27, 34

[65] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006. ISSN 0018-8670. 42

[66] A. Dan, D. Davis, R. Kearney, A. Keller, R. King, D. Kuebler, H. Ludwig, M. Polan, M. Spreitzer, and A. Youssef. Web services on demand: WSLA-driven automated management. *IBM Systems Journal*, 43(1):136–158, 2004. ISSN 0018-8670. 2

[67] Varró, Dániel. Model Transformation by Example. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 410–424. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-45772-5. 245

[68] Amir Vahid Dastjerdi and Rajkumar Buyya. A Taxonomy of QoS Management and Service Selection Methodologies for Cloud Computing. *Cloud Computing: Methodology, Systems, and Applications, L. Wang, R. Ranjan, J. Chen, and B. Benatallah (eds), CRC Press, Boca Raton, FL, USA*, 2011. 15, 16, 76

[69] Martine De Cock, Sam Chung, and Omar Hafeez. Selection of Web Services with Imprecise QoS Constraints. In *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence*, WI 07, pages 535–541, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3026-5. 76, 79

[70] Markus Debusmann and Alexander Keller. SLA-Driven Management of Distributed Systems Using the Common Information Model. In Germän Goldszmidt and Jürgen Schönwälder, editors, *Integrated Network Management VIII*, volume 118 of *IFIP - The International Federation for Information Processing*, pages 563–576. Springer US, 2003. ISBN 978-1-4757-5521-3. 2

[71] T. Dillon, Chen Wu, and E. Chang. Cloud Computing: Issues and Challenges. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pages 27–33, 2010. 3, 27

[72] D.A D'Mello, I Kaur, N. Ram, and V.S. Ananthanarayana. Semantic Web Service Selection Based on Business Offering. In *Computer Modeling and Simulation, 2008. EMS '08. Second UKSIM European Symposium on*, pages 476–481, Sept 2008. 200

[73] AnHai Doan, Jayant Madhavan, Pedro Domingos, and Alon Halevy. Ontology matching: A machine learning approach. In Steffen Staab and Rudi Studer, editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 385–403. Springer Berlin Heidelberg, 2004. ISBN 978-3-662-11957-0. 52

[74] G. Dobson and A. Sanchez-Macian. Towards Unified QoS/SLA Ontologies. In *Services Computing Workshops, 2006. SCW '06. IEEE*, pages 169–174, 2006. 19, 53

[75] G. Dobson, R. Lock, and I. Sommerville. QoSOnt: a QoS ontology for service-centric systems. In *Software Engineering and Advanced Applications, 2005. 31st EUROMICRO Conference on*, pages 80–87, 2005. xi, 19, 20, 22

[76] V.C. Emeakaroha, I. Brandic, M. Maurer, and S. Dustdar. Low level Metrics to High level SLAs - LoM2HiS framework: Bridging the gap between monitored metrics and SLA parameters in cloud environments. In *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, pages 48–54, 2010. 3, 125, 127

[77] Vincent C. Emeakaroha, Marco A.S. Netto, Rodrigo N. Calheiros, Ivona Brandic, Rajkumar Buyya, and César A.F. De Rose. Towards autonomic detection of SLA violations in cloud infrastructures. *Future Generation Computer Systems*, 28(7):1017 – 1029, 2012. ISSN 0167-739X. Special section: Quality of Service in Grid and Cloud Computing. 16

[78] M.C. Eti, S.O.T. Ogaji, and S.D. Probert. Integrating reliability, availability, maintainability and supportability with risk analysis for improved operation of the afam thermal power-station. *Applied Energy*, 84(2):202 – 221, 2007. ISSN 0306-2619. 18

[79] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran. Object storage: the future building block for storage systems. In *Local to Global Data Interoperability - Challenges and Technologies, 2005*, pages 119–123, 2005. 33

[80] Eckhard D Falkenberg, Wolfgang Hesse, Paul Lindgreen, Björn E Nilsson, JL Han Oei, Colette Rolland, Ronald K Stamper, Frans JM Van Assche, Alexander A Verrijn-Stuart, and Klaus Voss. A framework of information system concepts. *The FRISCO report*, 1998. 39

[81] S. Ferretti, V. Ghini, F. Panzieri, M. Pellegrini, and E. Turrini. QoS-Aware Clouds. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 321–328, July 2010. 88

[82] I. Foster, Yong Zhao, I. Raicu, and S. Lu. Cloud Computing and Grid Computing 360-Degree Compared. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1 –10, nov. 2008. 2, 24, 26

[83] S.K. Garg, S. Versteeg, and R. Buyya. SMICloud: A framework for Comparing and Ranking Cloud Services. In *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, pages 210 –218, dec. 2011. xi, 4, 5, 15, 17, 50, 51, 54, 56, 63, 67, 71, 79, 88, 89, 94, 188, 197, 200, 201, 237

[84] T. Gherbi, D. Meslati, and I. Borne. MDE between Promises and Challenges. In *Computer Modelling and Simulation, 2009. UKSIM '09. 11th International Conference on*, pages 152–155, March 2009. 38

[85] M. Godse and S. Mulik. An Approach for Selecting Software-as-a-Service (SaaS) Product. In *Cloud Computing, 2009. CLOUD '09. IEEE International Conference on*, pages 155–158, Sept 2009. 5, 50

[86] T. Grandison, E.M. Maximilien, S. Thorpe, and A. Alba. Towards a Formal Definition of a Computing Cloud. In *Services (SERVICES-1), 2010 6th World Congress on*, pages 191–192, July 2010. 3

[87] Roy Grønmo and Michael C. Jaeger. Model-driven methodology for building qos-optimised web service compositions. In Lea Kutvonen and Nancy Alonistioti, editors, *Distributed Applications and Interoperable Systems*, volume 3543 of *Lecture Notes in Computer Science*, pages 68–82. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-26262-6. 231

[88] Brent Hailpern and Peri Tarr. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, 45:451–462, 2006. 39

[89] Mohammad Hajjat, Xin Sun, Yu-Wei Eric Sung, David Maltz, Sanjay Rao, Kunwadee Sripanidkulchai, and Mohit Tawarmalani. Cloudward Bound: Planning for Beneficial Migration of Enterprise Applications to the Cloud. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 243–254, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0201-2. 27

[90] Seung-Min Han, Mohammad Mehedi Hassan, Chang-Woo Yoon, Hyun-Woo Lee, and Eui-Nam Huh. Efficient Service Recommendation System for Cloud Computing Market. In Dominik Ślezak, Tai-hoon Kim, StephenS. Yau, Osvaldo Gervasi, and Byeong-Ho Kang, editors, *Grid and Distributed Computing*, volume 63 of *Communications in Computer and Information Science*, pages 117–124. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-10548-7. 4, 50, 54

[91] C.N. Höfer and G. Karagiannis. Cloud computing services: taxonomy and comparison. *Journal of Internet Services and Applications*, 2(2):81–94, 2011. ISSN 1867-4828. 30, 31, 32

[92] P. Hofmann and D. Woods. Cloud computing: The limits of public clouds for business applications. *Internet Computing, IEEE*, 14(6):90–93, Nov 2010. ISSN 1089-7801. 29, 34

[93] Ting-Ya Hsieh, Shih-Tong Lu, and Gwo-Hshiung Tzeng. Fuzzy MCDM approach for planning and design tenders selection in public office buildings. *International Journal of Project Management*, 22(7):573 – 584, 2004. ISSN 0263-7863. 50, 230

[94] Angus F.M. Huang, Ci-Wei Lan, and Stephen J.H. Yang. An optimal QoS-based web service selection scheme. *Information Sciences*, 179(19):3309 – 3322, 2009. ISSN 0020-0255. 49, 72, 88

[95] A. Iosup, S. Ostermann, M.N. Yigitbasi, R. Prodan, T. Fahringer, and D. H J Epema. Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing. *Parallel and Distributed Systems, IEEE Transactions on*, 22(6):931–945, June 2011. ISSN 1045-9219. 89

[96] A. Iosup, S. Ostermann, M.N. Yigitbasi, R. Prodan, T. Fahringer, and D.H.J. Epema. Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing. *Parallel and Distributed Systems, IEEE Transactions on*, 22(6):931–945, June 2011. 4

[97] Wayne Jansen and Timothy Grance. Sp 800-144. guidelines on security and privacy in public cloud computing. Technical report, Gaithersburg, MD, United States, 2011. 28

[98] Frédéric Jouault and Ivan Kurtev. Transforming models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, pages 128–138. Springer, 2006. 41

[99] Seny Kamara and Kristin Lauter. Cryptographic Cloud Storage. In Radu Sion, Reza Curtmola, Sven Dietrich, Aggelos Kiayias, JosepM. Miret, Kazue Sako, and Francesc Sebé, editors, *Financial Cryptography and Data Security*, volume 6054 of *Lecture Notes in Computer Science*, pages 136–149. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-14991-7. 28, 29

[100] Jaeyong Kang and Kwang Mong Sim. Towards agents and ontology for cloud service discovery. In *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2011 International Conference on*, pages 483–490. IEEE, 2011. 53

[101] Keven T. Kearney and Francesco Torelli. The SLA Model. In Philipp Wieder, Joe M. Butler, Wolfgang Theilmann, and Ramin Yahyapour, editors, *Service Level Agreements for Cloud Computing*, pages 43–67. Springer New York, 2011. ISBN 978-1-4614-1613-5. 63, 64

[102] Alexander Keller and Heiko Ludwig. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. *Journal of Network and Systems Management*, 11:57–81, 2003. 16, 59, 69, 235

[103] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN 032119442X. 39, 41

[104] Bastian Koller and Lutz Schubert. Towards autonomous SLA management using a proxy-like approach. *Multiagent and Grid Systems*, 3(3):313–325, 2007. 16

[105] Dimitrios Kolovos. *An extensible platform for specification of integrated languages for model management*. PhD thesis, University of York, 2008. 39, 43

[106] Dimitrios S. Kolovos. Establishing Correspondences between Models with the Epsilon Comparison Language. In Richard F. Paige, Alan Hartman, and Arend Rensink, editors, *Model Driven Architecture - Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, pages 146–157. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-02673-7. 42, 45

[107] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Model comparison: a foundation for model composition and model transformation testing. In *Proceedings of the 2006 international workshop on Global integrated model management*, GaMMa '06, pages 13–20, New York, NY, USA, 2006. ACM. ISBN 1-59593-410-3. 42

[108] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. The Epsilon Object Language (EOL). In Arend Rensink and Jos Warmer, editors, *Model Driven Architecture - Foundations and Applications*, volume 4066 of *Lecture Notes in Computer Science*, pages 128–142. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-35909-8. 44

[109] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. The Epsilon Transformation Language. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Theory and Practice of Model Transformations*, volume 5063 of *Lecture Notes in Computer Science*, pages 46–60. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-69926-2. xiv, 41, 42, 46, 47

[110] D.S. Kolovos, D. Di Ruscio, A. Pierantonio, and R.F. Paige. Different models for model matching: An analysis of approaches to support model differencing. In *Comparison and Versioning of Software Models, 2009. CVSM '09. ICSE Workshop on*, pages 1–6, 2009. 43

[111] D. Kondo, B. Javadi, P. Malecot, F. Cappello, and D.P. Anderson. Cost-benefit analysis of Cloud Computing versus desktop grids. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12, May 2009. 89

[112] Daryl Kulak and Eamonn Guiney. *Use Cases: Requirements in Context*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. ISBN 0201657678. xvi, 70, 73, 74

[113] I. Kurtev. *Adaptability of model transformations*. PhD thesis, University of Twente, Enschede, May 2005. 39

[114] Hyun Jung La and Soo Dong Kim. A Systematic Process for Developing High Quality SaaS Cloud Services. In MartinGilje Jaatun, Gansen Zhao, and Chunming Rong, editors, *Cloud Computing*, volume 5931 of *Lecture Notes in Computer Science*, pages 278–289. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-10664-4. 29, 31

[115] Craig Larman and Victor R. Basili. Iterative and Incremental Development: A Brief History. *Computer*, 36(6):47–56, 2003. ISSN 0018-9162. 9

[116] G. Lawton. Developing Software Online With Platform-as-a-Service Technology. *Computer*, 41(6):13–15, 2008. ISSN 0018-9162. 29, 30, 31

[117] Alexander Lenk, Markus Klems, Jens Nimis, Stefan Tai, and Thomas Sandholm. What's Inside the Cloud? An Architectural Map of the Cloud Landscape. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, CLOUD '09, pages 23–31, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3713-9. 236

[118] L. Lewis and P. Ray. Service level management definition, architecture, and research challenges. In *Global Telecommunications Conference, 1999. GLOBECOM '99*, volume 3, pages 1974–1978 vol.3, 1999. 15

[119] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. CloudCmp: comparing public cloud providers. In *Proceedings of the 10th annual conference on Internet measurement*, IMC '10, pages 1–14, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0483-2. 4, 50, 89, 188

[120] Lei Li and Ian Horrocks. A software framework for matchmaking based on semantic web technology. *International Journal of Electronic Commerce*, 8(4): 39–60, 2004. 52

[121] Y. Liu, A.H. Ngu, and L.Z. Zeng. QoS computation and policing in dynamic web service selection. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, WWW Alt. '04, pages 66–73, New York, NY, USA, 2004. ACM. ISBN 1-58113-912-8. 21, 49, 55, 88

[122] Freerk A Lootsma. *Multi-criteria decision analysis via ratio and difference judgement*, volume 29. Springer Science & Business Media, 1999. 51

[123] Heiko Ludwig. Ws-agreement concepts and use–agreement-based service-oriented architectures. Technical report, IBM Research Division, Technical Report, 2006. xi, 61

[124] Heiko Ludwig, Alexander Keller, Asit Dan, Richard King, and Richard Franck. A Service Level Agreement Language for Dynamic Electronic Services. *Electronic Commerce Research*, 3(1-2):43–59, 2003. ISSN 1389-5753. 1, 13

[125] U.S. Manikrao and T. V. Prabhakar. Dynamic selection of web services with recommendation system. In *Next Generation Web Services Practices, 2005. NWeSP 2005. International Conference on*, pages 5 pp.–, Aug 2005. 72

[126] E. Marilly, O. Martinot, S. Betge-Brezetz, and G. Delegue. Requirements for service level agreement management. In *IP Operations and Management, 2002 IEEE Workshop on*, pages 57 – 62, 2002. 15

[127] Sean Marston, Zhi Li, Subhajyoti Bandyopadhyay, Juheng Zhang, and Anand Ghalsasi. Cloud computing – The business perspective. *Decision Support Systems*, 51(1):176 – 189, 2011. ISSN 0167-9236. 28, 29

[128] Benedikt Martens, Frank Teuteberg, and Matthias Gräuler. Design and implementation of a community platform for the evaluation and selection of cloud computing services: a market analysis. In *ECIS*, 2011. 4, 50

[129] Michael Maurer, Vincent C. Emeakaroha, Ivona Brandic, and Jrn Altmann. Cost-benefit analysis of an SLA mapping approach for defining standardized cloud computing goods. *Future Generation Computer Systems*, 28(1):39 – 47, 2012. ISSN 0167-739X. 53

[130] E. Michael Maximilien and Munindar P. Singh. Toward Autonomic Web Services Trust and Selection. In *Proceedings of the 2Nd International Conference on Service Oriented Computing*, ICSOC '04, pages 212–221, New York, NY, USA, 2004. ACM. ISBN 1-58113-871-7. 59

[131] E.M. Maximilien and M.P. Singh. A framework and ontology for dynamic web services selection. *Internet Computing, IEEE*, 8(5):84–93, Sept 2004. ISSN 1089-7801. 19, 21, 22, 49

[132] Oleksiy Mazhelis and Pasi Tyrväinen. Economic aspects of hybrid cloud infrastructure: User organization perspective. *Information Systems Frontiers*, 14(4): 845–869, 2012. ISSN 1387-3326. 29, 31, 32

[133] Peter Mell and Tim Grance. The NIST Definition of Cloud Computing, 2011. http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf. 2, 24, 26, 28, 29, 30, 31

[134] A Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar. End-to-End Support for QoS-Aware Service Selection, Binding, and Mediation in VRESCo. *Services Computing, IEEE Transactions on*, 3(3):193–205, July 2010. ISSN 1939-1374. 76

[135] Anton Michlmayr, Florian Rosenberg, Philipp Leitner, and Schahram Dustdar. Comprehensive QoS monitoring of web services and event-based SLA violation detection. In *Proceedings of the 4th International Workshop on Middleware for Service Oriented Computing*, MWSOC '09, pages 1–6, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-848-3. xi, 17, 18, 127

[136] Parastoo Mohagheghi and Vegard Dehlen. Where Is the Proof? - A Review of Experiences from Applying MDE in Industry. In Ina Schieferdecker and Alan Hartman, editors, *Model Driven Architecture Foundations and Applications*, volume 5095 of *Lecture Notes in Computer Science*, pages 432–443. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-69095-5. 38

[137] Pierre-Alain Muller, Franck Fleurey, Frédéric Fondement, Michel Hassenforder, Rémi Schneckenburger, Sébastien Gérard, and Jean-Marc Jézéquel. Model-Driven Analysis and Synthesis of Concrete Syntax. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 98–110. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-45772-5. xi, xiv, 40

[138] Vinod Muthusamy, Hans-Arno Jacobsen, Tony Chau, Allen Chan, and Phil Coulthard. SLA-driven business Process Management in SOA. In *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '09, pages 86–100, Riverton, NJ, USA, 2009. IBM Corp. 2

[139] S.K. Nair, S. Porwal, T. Dimitrakos, AJ. Ferrer, J. Tordsson, T. Sharif, C. Sheridan, M. Rajarajan, and AU. Khan. Towards Secure Cloud Bursting, Brokerage and Aggregation. In *Web Services (ECOWS), 2010 IEEE 8th European Conference on*, pages 189–196, Dec 2010. 200

[140] Gonzalo Navarro. A Guided Tour to Approximate String Matching. *ACM Comput. Surv.*, 33(1):31–88, March 2001. ISSN 0360-0300. 76, 166

[141] Le Duy Ngan and R. Kanagasabai. OWL-S Based Semantic Cloud Service Broker. In *Web Services (ICWS), 2012 IEEE 19th International Conference on*, pages 560–567, June 2012. 34

[142] Le Duy Ngan and R. Kanagasabai. Owl-s based semantic cloud service broker. In *Web Services (ICWS), 2012 IEEE 19th International Conference on*, pages 560–567, June 2012. doi: 10.1109/ICWS.2012.103. 53

[143] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus Open-Source Cloud-Computing System. In *Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on*, pages 124–131, May 2009. 89

[144] Nicole Oldham, Kunal Verma, Amit Sheth, and Farshad Hakimpour. Semantic WS-agreement partner selection. In *Proceedings of the 15th international conference on World Wide Web*, WWW '06, pages 697–706, New York, NY, USA, 2006. ACM. ISBN 1-59593-323-9. 60

[145] Serafim Opricovic and Gwo-Hshiung Tzeng. Compromise solution by MCDM methods: A comparative analysis of VIKOR and TOPSIS. *European Journal of Operational Research*, 156(2):445 – 455, 2004. ISSN 0377-2217. 230

[146] Richard F. Paige, Phillip J. Brooke, and Jonathan S. Ostroff. Metamodel-based Model Conformance and Multiview Consistency Checking. *ACM Trans. Softw. Eng. Methodol.*, 16(3), July 2007. ISSN 1049-331X. 40

[147] Massimo Paolucci, Takahiro Kawamura, TerryR. Payne, and Katia Sycara. Semantic matching of web services capabilities. In Ian Horrocks and James Hendler, editors, *The Semantic Web - ISWC 2002*, volume 2342 of *Lecture Notes in Computer Science*, pages 333–347. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-43760-4. 52

[148] I.V. Papaioannou, D.T. Tsesmetzis, I.G. Roussaki, and M.E. Anagnostou. A QoS ontology language for Web-services. In *Advanced Information Network-*

ing and Applications, 2006. AINA 2006. 20th International Conference on, volume 1, pages 6 pp.–, 2006. 22

[149] T. Parveen and S. Tilley. When to Migrate Software Testing to the Cloud? In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 424–427, April 2010. 27

[150] Pankesh Patel, Ajith Ranabahu, and Amit Sheth. Service level agreement in cloud computing. *Cloud Workshops at OOPSLA09*, pages 1–10, 2009. xi, 1, 3, 59

[151] Junjie Peng, Xuejun Zhang, Zhou Lei, Bofeng Zhang, Wu Zhang, and Qing Li. Comparison of Several Cloud Computing Platforms. In *Information Science and Engineering (ISISE), 2009 Second International Symposium on*, pages 23–27, 2009. 28

[152] Yi Peng, Gang Kou, Guoxun Wang, and Yong Shi. Famcdm: A fusion approach of MCDM methods to rank multiclass classification algorithms. *Omega*, 39(6): 677 – 689, 2011. ISSN 0305-0483. 52, 230, 231

[153] Alain Pinsonneault and Kenneth L. Kraemer. Survey research methodology in management information systems: An assessment, 1993. URL http://www.escholarship.org/uc/item/6cs4s5f0. 8, 9

[154] Shuping Ran. A model for web services discovery with QoS. *SIGecom Exch.*, 4(1):1–10, March 2003. ISSN 1551-9031. 16, 17

[155] Christoph Redl, Ivan Breskovic, Ivona Brandic, and Schahram Dustdar. Automatic SLA Matching and Provider Selection in Grid and Cloud Computing Markets. In *Proceedings of the 2012 ACM/IEEE 13th International Conference on Grid Computing*, GRID '12, pages 85–94, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4815-9. 5, 53, 54

[156] B.P. Rimal, Eunmi Choi, and I. Lumb. A Taxonomy and Survey of Cloud Computing Systems. In *INC, IMS and IDC, 2009. NCM '09. Fifth International Joint Conference on*, pages 44–51, aug. 2009. 25, 30

[157] Marcel Risch, Ivona Brandic, and Jörn Altmann. Using sla mapping to increase market liquidity. In *Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops*, volume 6275 of *Lecture Notes in Computer Science*, pages 238–247. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-16131-5. 53

[158] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. M. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Caceres, M. Ben-Yehuda, W. Emmerich, and F. Galan. The reservoir model and architecture for open federated cloud computing. *IBM Journal of Research and Development*, 53(4):4:1 –4:11, july 2009. ISSN 0018-8646. 26

[159] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona A.C. Polack. The Epsilon Generation Language. In Ina Schieferdecker and Alan Hartman, editors, *Model Driven Architecture - Foundations and Applications*, volume 5095 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-69095-5. 47

[160] Pawel Rubach and Michael Sobolewski. Dynamic SLA Negotiation in Autonomic Federated Environments. In Robert Meersman, Pilar Herrero, and Tharam Dillon, editors, *On the Move to Meaningful Internet Systems: OTM 2009 Workshops*, volume 5872 of *Lecture Notes in Computer Science*, pages 248–258. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-05289-7. 15

[161] S. Sendall and W. Kozaczynski. Model transformation: the heart and soul of model-driven software development. *Software, IEEE*, 20(5):42–45, Sept 2003. ISSN 0740-7459. 3

[162] M.A. Serhani, R. Dssouli, A. Hafid, and H. Sahraoui. A QoS broker based architecture for efficient web services selection. In *Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference on*, pages 113–120 vol.1, 2005. 21, 49, 55, 88

[163] Dr. N.V. Kalyankar. Shivaji P. Mirashe. Cloud Computing. *JOURNAL OF COMPUTING*, 2:78–82, March 2010. 28, 29

[164] James Skene, D. Davide Lamanna, and Wolfgang Emmerich. Precise Service Level Agreements. In *Proceedings of the 26th International Conference on*

Software Engineering, ICSE 04, pages 179–188, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2163-0. 62

[165] James Skene, D. Davide Lamanna, and Wolfgang Emmerich. Precise Service Level Agreements. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 179–188, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2163-0. URL http://dl.acm.org/citation.cfm?id=998675.999422. xi, 63

[166] Thomas Stahl, Markus Völter, Jorn Bettin, Arno Haase, Simon Helsen, and Krzysztof Czarnecki. *Model-Driven Software Development Technology, Engineering, Management*. John Wiley & Sons, 2006. 41

[167] Vladimir Stantchev and Christian Schrpfer. Negotiating and Enforcing QoS and SLAs in Grid and Cloud Computing. In Nabil Abdennadher and Dana Petcu, editors, *Advances in Grid and Pervasive Computing*, volume 5529 of *Lecture Notes in Computer Science*, pages 25–35. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-01670-7. 16

[168] Anthony M. Starfield, Karl Smith, and Andrew L. Bleloch. *How to Model It: Problem Solving for the Computer Age*. McGraw-Hill, Inc., New York, NY, USA, 1993. ISBN 0070058970. 39

[169] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008. ISBN 9780132702218. 43

[170] S. Sundareswaran, A. Squicciarini, and D. Lin. A Brokerage-Based Approach for Cloud Service Selection. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 558–565, June 2012. 54

[171] A.G. Sutcliffe, N. A M Maiden, S. Minocha, and D. Manuel. Supporting scenario-based requirements engineering. *Software Engineering, IEEE Transactions on*, 24(12):1072–1088, Dec 1998. ISSN 0098-5589. 70

[172] L. Taher and H. El Khatib. A framework and qos matchmaking algorithm for dynamic web services selection. In *In Proceedings of the 2 nd International Conference on Innovations in Information Technology (IIT05*, 2005. 19

[173] Dave Thomas. Enabling Application Agility–Software as a Service, Cloud Computing and Dynamic Languages. *Journal of Object Technology*, 7(4):29–32, 2008. 30

[174] Emir Toktar, Guy Pujolle, Edgard Jamhour, ManoelC. Penna, and Mauro Fonseca. An XML Model for SLA Definition with Key Indicators. In Deep Medhi, JosMarcos Nogueira, Tom Pfeifer, and S.Felix Wu, editors, *IP Operations and Management*, volume 4786 of *Lecture Notes in Computer Science*, pages 196–199. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-75852-5. 125

[175] Vuong Xuan Tran and H. Tsuji. A survey and analysis on semantics in QoS for web services. In *Advanced Information Networking and Applications, 2009. AINA '09. International Conference on*, pages 379–385, May 2009. doi: 10.1109/AINA.2009.43. 22

[176] Vuong Xuan Tran, Hidekazu Tsuji, and Ryosuke Masuda. A new QoS ontology and its QoS-based ranking algorithm for web services. *Simulation Modelling Practice and Theory*, 17(8):1378 – 1398, 2009. ISSN 1569-190X. Dependable Service-Orientated Computing Systems. 19, 76, 88

[177] Evangelos Triantaphyllou. Multi-criteria decision making methods. In *Multi-criteria Decision Making Methods: A Comparative Study*, volume 44 of *Applied Optimization*, pages 5–21. Springer US, 2000. ISBN 978-1-4419-4838-0. doi: 10.1007/978-1-4757-3157-6_2. URL http://dx.doi.org/10.1007/978-1-4757-3157-6_2. 51, 231

[178] Evangelos Triantaphyllou. *Multi-criteria decision making methods: a comparative study*, volume 44. Springer Science & Business Media, 2013. xi, 52

[179] Evangelos Triantaphyllou and Khalid Baig. The impact of aggregating benefit and cost criteria in four mcda methods. *Engineering Management, IEEE Transactions on*, 52(2):213–226, May 2005. 51

[180] D.T. Tsesmetzis, IG. Roussaki, IV. Papaioannou, and M.E. Anagnostou. QoS awareness support in Web-Service semantics. In *Telecommunications, 2006. AICT-ICIW '06. International Conference on Internet and Web Applications and Services/Advanced International Conference on*, pages 128–128, Feb 2006. 127

[181] T. Unger, F. Leymann, S. Mauchart, and T. Scheibler. Aggregation of service level agreements in the context of business processes. In *Enterprise Distributed Object Computing Conference, 2008. EDOC '08. 12th International IEEE*, pages 43–52, Sept 2008. 2

[182] Z. ur Rehman, F.K. Hussain, and O.K. Hussain. Towards Multi-criteria Cloud Service Selection. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2011 Fifth International Conference on*, pages 44–48, June 2011. 5, 237

[183] Z. ur Rehman, O.K. Hussain, and F.K. Hussain. IaaS Cloud Selection using MCDM Methods. In *e-Business Engineering (ICEBE), 2012 IEEE Ninth International Conference on*, pages 246–251, 2012. 4, 5, 50, 51, 54, 67, 71

[184] MIKE USCHOLD. Knowledge level modelling: concepts and terminology. *The Knowledge Engineering Review*, 13:5–29, 2 1998. ISSN 1469-8005. 39

[185] A. van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009. ISBN 9780470012703. 70, 72

[186] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39:50–55, 2008. 2, 24, 26

[187] Luis M. Vaquero, Luis Rodero-Merino, and Rajkumar Buyya. Dynamically scaling applications in the cloud. *SIGCOMM Comput. Commun. Rev.*, 41:45–52, 2011. ISSN 0146-4833. 24

[188] Salvatore Venticinque, Rocco Aversa, Beniamino Di Martino, Massimilano Rak, and Dana Petcu. A Cloud Agency for SLA Negotiation and Management. In MarioR. Guarracino, Frédéric Vivien, JesperLarsson Träff, Mario Cannatoro, Marco Danelutto, Anders Hast, Francesca Perla, Andreas Knüpfer, Beniamino Di Martino, and Michael Alexander, editors, *Euro-Par 2010 Parallel Processing Workshops*, volume 6586 of *Lecture Notes in Computer Science*, pages 587–594. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-21877-4. 200

[189] Dinesh C. Verma. Supporting Service Level Agreements on IP Networks. Macmillan Technical Publishing, 1999. 1, 13

[190] Eelco Visser. WebDSL: A Case Study in Domain-Specific Language Engineering. In Ralf Lmmel, Joost Visser, and Joo Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II*, volume 5235 of *Lecture Notes in Computer Science*, pages 291–373. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-88642-6. 30

[191] Jeffrey Voas and Jia Zhang. Cloud computing: new wine or just a new bottle? *IT professional*, 11(2):15–17, 2009. xi, 23, 24

[192] Lizhe Wang, Gregor Laszewski, Andrew Younge, Xi He, Marcel Kunze, Jie Tao, and Cheng Fu. Cloud Computing: a Perspective Study. *New Generation Computing*, 28(2):137–146, 2010. ISSN 0288-3635. 24

[193] Ping Wang. QoS-aware web services selection with intuitionistic fuzzy set under consumer's vague perception. *Expert Systems with Applications*, 36(3, Part 1): 4460 – 4466, 2009. ISSN 0957-4174. 21, 78, 182, 193, 197

[194] Xia Wang, Tomas Vitvar, Mick Kerrigan, and Ioan Toma. A QoS-Aware Selection Model for Semantic Web Services. In Asit Dan and Winfried Lamersdorf, editors, *Service-Oriented Computing - ICSOC 2006*, volume 4294 of *Lecture Notes in Computer Science*, pages 390–401. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-68147-2. 16, 19, 21, 22, 53, 54, 55, 88

[195] Xiaoting Wang and Evangelos Triantaphyllou. Ranking irregularities when evaluating alternatives by using some ELECTRE methods. *Omega*, 36(1):45 – 63, 2008. ISSN 0305-0483. Special Issue Section: Papers presented at the INFORMS conference, Atlanta, 2003. 21, 182, 193, 197

[196] Christof Weinhardt, Arun Anandasivam, Benjamin Blau, Nikolay Borissov, Thomas Meinl, Wibke Michalk, and Jochen Stößer. Cloud Computing- A Classification, Business Models, and Research Directions. *Business & Information Systems Engineering*, 1(5):391–399, 2009. 27

[197] L. Wu and R. Buyya. Service Level Agreement (SLA) in Utility Computing Systems. *ArXiv e-prints*, October 2010. 3, 7, 15, 54

[198] Edward Wustenhoff and Sun BluePrints. Service level agreement in the data center. *Sun Microsystems Professional Series*, 2002. 2

[199] Jianhong Xu, Weizhi Gong, and Ye Wang. A cloud service discovery approach based on FCA. In *Cloud Computing and Intelligent Systems (CCIS), 2012 IEEE 2nd International Conference on*, volume 03, pages 1357–1361, Oct 2012. 4, 5, 50

[200] Xun Xu. From cloud computing to cloud manufacturing. *Robotics and Computer-Integrated Manufacturing*, 28(1):75 – 86, 2012. ISSN 0736-5845. 29

[201] Jiann Liang Yang, Huan Neng Chiu, Gwo-Hshiung Tzeng, and Ruey Huei Yeh. Vendor selection by integrated fuzzy MCDM techniques with independent and interdependent relationships. *Information Sciences*, 178(21):4166 – 4183, 2008. ISSN 0020-0255. 71

[202] Viktor Yarmolenko and Rizos Sakellariou. Towards increased expressiveness in service level agreements. *Concurrency and Computation: Practice and Experience*. 53

[203] L. Youseff, M. Butrico, and D. Da Silva. Toward a Unified Ontology of Cloud Computing. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1 –10, nov. 2008. 30

[204] Hong Qing Yu and Stephan Reiff-Marganiec. Non-functional property based service selection: A survey and classification of approaches. In *Non Functional Properties and Service Level Agreements in Service Oriented Computing Workshop co-located with The 6th IEEE European Conference on Web Services*, 2008. 49, 50

[205] Liangzhao Zeng, B. Benatallah, A.H.H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. Qos-aware middleware for web services composition. *Software En-*

gineering, IEEE Transactions on, 30(5):311–327, May 2004. ISSN 0098-5589. 231, 232

[206] M. Zhang, R. Ranjan, A. Haller, D. Georgakopoulos, M. Menzel, and S. Nepal. An ontology-based system for cloud infrastructure services' discovery. In *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2012 8th International Conference on*, pages 524–530, Oct 2012. 34, 101

[207] Chen Zhou, Liang-Tien Chia, and Bu-Sung Lee. Semantics in service discovery and QoS measurement. *IT Professional*, 7(2):29–34, Mar 2005. ISSN 1520-9202. 19

[208] Dimitrios Zissis and Dimitrios Lekkas. Addressing cloud computing security issues. *Future Generation Computer Systems*, 28(3):583 – 592, 2012. ISSN 0167-739X. 27