A Methodology for Automated Service Level Agreement Compliance Prediction

Thesis by
Rouaa Yassin Kassab

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

School of Computing Science
Newcastle University
Newcastle upon Tyne, UK

(Submitted 3 June 2013)

Declaration

I declare that this thesis is my own work unless otherwise stated. No part of this thesis has previously been submitted for a degree or any other qualification at Newcastle University or any other institution.

Rouaa Yassin Kassab

Acknowledgements

In the name of Allah, the Most Gracious and the Most Merciful.
Alhamdulillah, all praises go to Allah on whom ultimately we depend for sustenance and guidance. Alhamdulillah for all the strengths and blessing He gave me in completing this thesis.

I would like to thank all the people who have provided me with valuable advice and support throughout my doctoral study.

First and foremost, my utmost gratitude goes to my supervisor, Professor Aad van Moorsel who continuously guide me, support me, and encourage me throughout the course of my research and writing of this thesis. He much importantly taught me to be a confident self-dependent researcher. His patience and understanding on different occasions eased all difficulties led to the completion of this research work. Special thanks for Dr. Graham Morgan and Dr. Carlos Molina-Jimenez for being members of my supervisory committee.

I would like to thank Tishreen University and the Ministry of Higher Education in Syria for supporting me financially throughout my PhD study.

I can never express enough gratitude for the support I received from my parents. Without their continuous emotional support and encouragement, I would never have been able to complete my works. They were the place to reveal my fears without fearing any blame especially my mother who passed away before I complete my PhD.

I owe my sincere gratitude to my wonderful husband, Siddek, for being patient, supportive and understanding during the long period of my work. My thanks for my little girl, Nour, who always succeed to bring the smile to my face, and my unborn baby, who was accompanying me during my viva time. You have all been my source of joy.

My heartfelt thanks goes to my parents in law, my brothers, my sisters and all the family members for their encouragement, support and for taking care of my daughter during my writing up stage.

To my close friends, thank you for being supportive, helpful and caring friends during the hard times.

Last but not least, thanks to everybody in the School of Computing Science who have directly and indirectly helped me to complete the thesis.

Abstract

Service Level Agreement (SLA) specification languages express monitorable contracts between service providers and consumers. It is of interest to determine if predictive models can be derived for SLAs expressed in such languages, if possible in a fashion that is as automated as possible. Assuming that the service developer or user uses some SLA specification languages during the service development or deployment process, the Service level agreement Compliance Prediction (SlaCP) methodology is proposed as a general engineering methodology for predicting SLA compliance. This methodology helps contractual parties to assess the probability of SLA compliance, as automatically as is feasible, by mapping an existing SLA on a stochastic model of the service and using existing numerical solution algorithms or discrete event simulation to solve the model. The SlaCP methodology is generic, but the methodology is mostly described, in this thesis, assuming the use of the Web Service Level Agreement (WSLA) and the Stochastic Discrete Event Systems (SDES) formalism. The approach taken in this methodology is firstly to associate formal semantics with WSLA elements in order to be understood mathematically precise. Then, a five-step mapping process between the source and the target formalisms is conducted. These steps include: mapping into model primitives, reward metrics, expressions for functions of these metrics, the time at which the prediction occurs, and the ultimate probability of SLA compliance. The proposed methodology is implemented in a software tool that automates most of its steps using Möbius and SPNP. The methodology is evaluated using a case study which shows the methodology's feasibility and limitations in both theoretical and practical terms.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Background and Motivation

Over recent years, computer and Internet technologies have been incorporated into many everyday activities such as aircraft control, shopping, banking and so on [2]. This rapid growth in interconnected computer networks has allowed companies to offer their services electronically [3]. A number of paradigms have been developed to support this [4], including Web Services [5], Cloud Computing [6], Utility Computing [7] and Service Oriented Computing [3]. The concept on which these paradigms rely is that of building distributed applications using electronic services; this has resulted in loosely coupled, dynamic and inexpensive applications [8].

A service is a software system used to perform a specific task for its customers using request-response messages [4]. A service customer may choose a specific service from among similar ones that offer the same business. For this reason, it is a challenge for a service provider to maintain the running of the service at an adequate level in order to keep attracting potential customers [2, 9, 10]. Customers' interest regarding the level of service offered may vary and this can be related to different *dependability*, *performance* and *performability* metrics such as response time, availability, throughput, reliability, exception handling, and security [2, 10, 11]. In this context, and in order to give customers the ability to choose which service is best suited to them, the term *Quality of Service* (QoS) has evolved to denote the quality of the non-functional properties of a service [10]. Service providers and customers choose QoS metrics and specify guarantees of their values over a certain period of time; these are called *Service Level Objectives* (SLOs) [10, 12]. Owing to their importance in attracting customers, SLOs have become a crucial part of a larger legal document called a *Service Level Agreement* (SLA)[10].

SLAs were first developed in the 1980s by telecommunications firms and their importance was strengthened later by the Grid computing community [13]. The most important reason why an SLA is used in service provision is to clarify and formalise the relationship between the contractual parties regarding the overall quality of the service offered [12, 14, 15]. Previously, the process of writing and editing SLA contracts was carried out manually using natural language, which made this process both difficult and time-consuming [16]. For this reason, and to create and specify easily an SLA structure using templates that provide a certain amount of automation, different formal specification languages have been introduced. Examples of these languages include WSLA [17], WS-Agreement [1], SLAng [18] and NextGRID [19]. These languages aim to facilitate the construction of different SLA elements, define their contents, and monitor their compliance [20]. Although each SLA language has its own syntax and semantics, they have in common declarations of several pieces of information. This may include information regarding the contractual parties, the definition of the service, the specification of the set of QoS metrics (such as availability) for a specific object of this service, their SLOs (e.g. service availability is more than 90% in each business day), and finally the penalties in case of breach of contract [21].

After agreeing in an SLA on the level at which a service should be delivered, it becomes essential for the provider to implement techniques that provide some assessment of these service metrics during a service's design, its implementation, its deployment, and during its running [2]. This is important because a service provider may be at risk accepting an SLA that the service's infrastructure is not able to fulfil. Costly penalty payments, and adjustments to the contract or the underlying system, may occur as a consequence [22]. Similarly, service customers are also interested in such assessments because they are keen for guarantees that the level of service they receive is that to which they agreed [9].

SLA compliance assessment techniques aim either to monitor or predict SLA compliance, or a combination of both. SLA compliance monitoring implies checking service performance during run-time against the agreed SLA. In the case of any deviation, the provider is prompted to take corrective actions. Although this monitoring informs the provider of weak points in the SLA or in the underlying service, it neither precludes errors nor allows enough time to adopt any necessary modifications [23]. On the other hand, predicting SLA compliance can be used to verify in advance whether the service's performance conforms to an SLA, either at the design stage or when deploying the service in the real world. In the latter case, the prediction can be carried out either off-line or during runtime. Predicting SLA compliance

beforehand gives the provider enough time to make any necessary adjustments in order to improve the service [24]. Also, early prediction helps in determining which SLO threshold the service can maintain by evaluating different ones and choosing the optimal threshold from among them.

Many approaches have been devised for the purpose of SLA compliance assessment, such as measuring the real service or a prototype thereof. This is often not practical since such an approach cannot be used unless the service is being operated in real, but controlled life. Furthermore, it may require years to obtain enough and appropriate measurements of specific events (e.g. the expected failures in a system) [2]. Another and more flexible way of providing such assessments is to construct a model that captures the service's characteristics using either *Discrete Event Simulation* (DES) or *analytic model solutions*, and then perform assessments [2].

Assessment approaches using modelling techniques give the provider the ability to predict if the service will be able to conform to an SLA when a new contract is established with new users, or when the service parameters are modified according to special circumstances [25]. In adopting a model-based prediction approach, stochastic models have been used by some researchers because they better capture the nondeterministic nature of service dynamics on the web [26]. Solving these models analytically, or using simulation, allows for predicting the expected values of those QoS metrics that are not available before a service is deployed. These predicted values can then be used to determine SLA compliance. The approach in this thesis adopts the use of stochastic models for predicting SLA compliance.

1.2 Research Problem

A number of aspects regarding model-based SLA compliance prediction, representing some research perspectives which motivate this work, have been recognised.

Firstly, *current research does not typically try to utilise existing SLA contracts.* Most such research uses model-based metric definitions as an SLA and then checks compliance. Software engineers have certain reasons for using SLA contracts instead of the metrics used in mathematical modelling frameworks. In such cases, a service contract already exists and the provider wants to check the probability of its fulfilment. Hence, using an SLA as a starting point in the prediction process is desirable.

Secondly, *in current research, the scope of QoS metrics that are predicted by stochastic models focus mostly on a limited set of QoS metrics.* The metrics that are used in prediction, such as response time and availability, are usually basic metrics:

i.e. there is no consideration of more complex metrics. Thus, when using existing SLA definitions, there is a need to provide a prediction mechanism which is tailored to the actual metrics specified in the SLA, such as maximum response time from a number of service invocations and sequences of successful invocations.

Thirdly, *a generalised stochastic model that is able to predict SLA compliance is lacking.* Usually, a researcher uses a specific type of stochastic model such as Process Algebra or Stochastic Petri Nets, to predict SLA compliance. This unique stochastic model of the service puts a usage restriction on non-specialists or other researchers who might be unaware of this particular type of modelling language and the tools related to it. For example, Franken, in [27], chose a Stochastic Petri Net (SPN) to model the stochastic processes under consideration and used an SLA language that defines metrics which fit with this model.

Fourthly, *the lack of a formal relationship between an SLA and a model may be readily recognised.* Most of the research that has been carried out into SLA prediction has used mainly two approaches. In the first, the prediction starts from an ad-hoc SLO expression or a stand-alone QoS metric which might not specify exactly how the QoS metric is assembled and computed. In this approach, there is no consideration of the SLA specification from which these expressions or metrics are taken. The assignment of the QoS metrics to the service model is carried out manually according to the modeller's perception; this may not be sufficiently precise. In the second approach, the process starts the other way round by defining rewards of interest in the model and then building an SLA with QoS metrics in a way that fits with the model's fragment description. For example, Suto et al. in [28] built an SLA that suits the definition of the system model and then predicts its fulfilment. This approach requires engineers or modellers to write and understand the sophisticated metrics associated with a stochastic process. Hence, the automated conversion of a metric from an existing SLA into a stochastic model is of interest.

Fifthly, *a software tool that automates SLA compliance prediction is not presented in the literature as an all-in-one software package starting from using an SLA as the primary input and ending with compliance probability as an output.* One reason for this might be the lack of understanding of how to relate the QoS metrics of an SLA to the model of the service. Such a tool would help service providers, SLA engineers, or other users if they have only a basic knowledge of all the aspects related to model-based SLA compliance prediction.

The aim of this thesis is to find a solution that investigates all of the aforementioned issues.

1.3 Research Hypothesis and Questions

In model-based SLA compliance prediction, an obstacle for a service provider or an SLA engineer is to produce an adequate stochastic model of the service [25]. This is because the service model has to capture the service behaviour and has to reflect the correct QoS metrics indicated by the SLA. If this is done correctly, the service model can be evaluated and compared to thresholds implied in the SLOs to predict SLA compliance. In line with this, the research hypothesis is:

Hypothesis: *"The process of model-based SLA compliance prediction can be automated using an existing SLA document as the only input."*

If this hypothesis is fulfilled by a new SLA compliance prediction engineering methodology, it means that this methodology will be able to obtain a service's stochastic model and its QoS metrics from an SLA specification in an automated way to support the SLA compliance prediction process. Furthermore, if this hypothesis is valid, it can be utilised in a supported software tool that will automate the methodology. This tool will allow an SLA engineer or a service provider who has a limited knowledge of stochastic modelling analysis to perform a prediction without having to gain a thorough understanding of model-related metrics and analysis tools. Furthermore, it will support them in parameterising SLAs more effectively by informing them how different levels of service performance may affect SLA compliance.

Bearing in mind the hypothesis mentioned above, a number of research questions have been formulated. The research in this thesis is a hybrid of both theoretical concepts and practical implementation, and the research questions are related to both theoretical and practical aspects of the research. For the theoretical part, the main question that is identified is as follows:

- **Question 1**: *"Can an existing SLA be mapped theoretically to metrics of a stochastic model in an automated fashion?"* This question implies several sub-questions:

    1. Are all SLA elements useful for prediction-related mapping or are some of them monitoring-related only? What are these elements?
    2. Does an SLA provide any information that helps in automatically creating a complete service model or a part of it? If yes, what are these elements? Can other supporting documents enhance this automatic model creation?
    3. Assuming that such a service model is available, how do the prediction-related SLA elements correspond to the service model? In other words,

are they mapped on the model primitives or are they captured by a function over the results of solving this model?

4. Given that the mapping from SLA into a stochastic model is feasible, to what extent can the mapping process be automated?

- **Question 2**: For the practical part of the research, the following question is formulated: *"Is the theoretical mapping applicable in a real example scenario?"* This implies the following sub-questions:

1. Is the methodology, which exploits the research hypothesis, applicable before or after deploying the service in the real world? Is it useful for providers and customers?
2. Assuming that the methodology is applicable before deploying the service, how will a user be able to obtain the necessary information for parameterising the model (e.g. delay time)? How can the initial state of the model, which is necessary for solving it, be determined (e.g. does this depend on simulation results or historical data)?
3. Does the type of model affect the usage of this mapping? In other words, is the type of stochastic model (i.e. closed or open, steady state or transient) important for mapping validity?
4. Is there any difference in prediction if a service is composite? Can a service model for a composite service still be generated and used by the methodology assuming the hypothesis?
5. Can an all-in-one software tool automate all aspects of the methodology?

1.4 Research Aim, Objectives and Challenges

Following the research problem, the hypothesis and the questions presented in the previous two sections, the aim of the research conducted in this thesis is:

Aim: "*To propose a new engineering methodology that helps SLA contractual parties to predict automatically, as much as is possible, if an SLA can be fulfilled by the service, when designing, deploying, or using the service.*"

The main idea in accomplishing this aim is to adopt a model-based approach as a means to predict the unknown values of QoS metrics. In other words, one of the things the research seeks "*to use an existing SLA language as a specification of the metrics of a predictive discrete-event stochastic model*". This implies the following objectives:

1. To create a general methodology that is able to predict the compliance probability of a predefined SLA. This involves the following:
   (a) To map theoretically existing SLA contracts on the metrics of a discrete-event stochastic model, as much as possible in an automated fashion.
   (b) To use this mapping as the basis for producing a model in order to predict values of the QoS metrics in the SLA.
   (c) To use these values to compute the ultimate QoS metrics used by the SLO, then to compare them with specified thresholds and hence predict the SLA compliance probability.
2. To implement the methodology for a specific type of SLA and stochastic model in order to check the methodology's feasibility.
3. To construct a software tool that automates this methodology as much as possible to support the methodology and to investigate its applicability in practice.
4. To evaluate the research's validity and the degree to which the research questions have been addressed through a detailed case study.

The model-based SLA compliance prediction approach, based on mapping SLAs into a stochastic process, is not trivial owing to a number of challenges.

Firstly, *SLAs are not written for the purpose of model-based prediction*; they are defined to be monitorable. As an example, SLAs do not typically define steady-state metrics but rather functions over periodically monitored variables. The modelling and solving of such metrics is typically more involved than solving steady-state metrics because these SLA metrics are difficult to solve analytically.

Secondly, *not all the information which is required to evaluate compliance with an SLA is available before deploying the service.* In an SLA, QoS metric values are provided by measuring or intercepting service resources while the service is running. Hence, before deploying a service in the real world, only an estimation of metrics can be derived through a correct, well-defined model of this service.

Thirdly, *an SLA does not provide information about the system's dynamics.* In the approach taken in this thesis, the service model relies on the SLA. However, there is not always a clear connection between an SLA, the system dynamics and the underlying business process. SLA supporting documents such as service description or work-flow documents can contain some information about service behaviour which one can try to exploit.

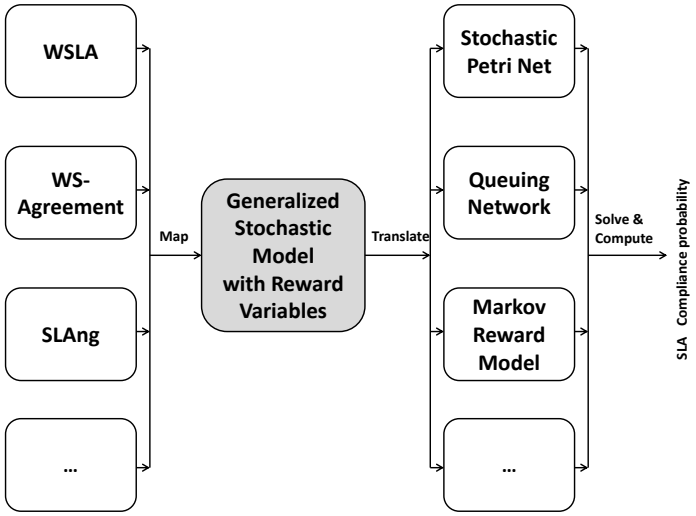

Figure 1.1: The approach of the general SLA compliance prediction: mapping any SLA contract to any stochastic model

Fourthly, *SLA elements are not mathematically defined*. The semantics of SLA elements and metrics are usually defined in a natural language, which makes it difficult for the QoS metrics to be understood precisely. This lack of precision might result in different perspectives of a single metric being adopted by a service provider and a customer. It is therefore necessary to be more formal about the SLA semantics.

1.5 Research Approach

The research work in this thesis begins by investigating SLA compliance prediction techniques in the context of a service-based environment. In addition, related work carried out by other researchers working in similar research areas is studied. Based on this, a novel methodology is developed for automated SLA compliance prediction. This is based on using an SLA as input for generating a stochastic model in order to determine compliance probability. For this reason, the approach taken supports the creation of a stochastic model by trying to prove that it can both be built and then enhanced by using SLAs in a structured way. This automatically translates SLA elements into stochastic model primitives in order to produce the service model (or part of it). In addition, it automatically translates the definitions of QoS metrics, along with the temporal constraints defined in the SLAs, into high-level model description reward variables. It then adds them to the service model to produce the desired model.

The approach taken to address the first main research objective is that the

methodology should be general. This means, as depicted in Figure 1.1, the proposed methodology has to map an SLA written in any SLA language (such as WSLA, WS-Agreement, SLAng, etc.) to an intermediate and generalised stochastic model with reward variables. This can then be translated into any stochastic model preferred by the user (e.g. Stochastic Petri Net (SPN), Queuing Network (QN), Markov Reward Model, etc.) in order to solve the model, perform any required computation, and then produce the SLA compliance probability.

The abstract derivation of the general SLA compliance prediction methodology proved to be complex. For this reason and, to address the second research objective, the approach here is to implement this methodology for a specific SLA language, namely the Web Service Level Agreement language (WSLA) [17], and a specific generalised stochastic modelling formalism, namely the Stochastic Discrete Event System (SDES) formalism developed in [29]. WSLA is chosen because it is, unlike the rest of the SLA languages, powerful enough to define explicitly different QoS metrics based on a constructive ontology (hierarchical QoS metrics). Thus it is suitable for the aim of this thesis. Another reason for choosing WSLA is that it is common, widely used and flexible in that an SLA engineer can extend new metric types that suit a domain-specific environment. The stochastic modelling formalism chosen is SDES as it supports a wide range of stochastic modelling formalisms in such a way that a translation into any of them from SDES is not difficult.

After implementing the methodology for WSLA and SDES, and to address the third research objective, the approach taken is to construct a software tool that will automate the proposed methodology by adopting a special type of SDES, namely the Stochastic Petri Net (SPN) [30], and tools that will solve them, namely SPNP [31] and Möbius [32]. Finally, to address the last research objective, the approach taken to validate the theoretical methodology and its practical implementation is carried out based on a detailed case study of a stock quote service. A number of evaluation questions are formulated to evaluate these in terms of automation, applicability, generality, and user support.

1.6 Contributions of the Thesis

The main contribution of this thesis is:

"A new engineering methodology that helps SLA contractual parties to predict automatically, as much as is possible, if an SLA can be fulfilled by the service, when designing, deploying, or using the service."

The principal contributions, which involve conducting, completing, and demon-

strating this novel methodology, are four aspects:

1. *A generalised SLA Compliance Prediction (SlaCP) methodology for predicting SLA compliance probability.* This includes its design, in addition to the architectural design of a software tool that automates it. The SlaCP methodology facilitates the process of predicting the compliance of predefined SLAs. It allows the user to create the model of the service, solve it, and perform the necessary computations to produce the compliance probability in a partially automated manner. Model-based SLA compliance prediction, evolved from mapping SLAs on a stochastic model, has not yet been considered in the literature, and certainly not in the fashion pursued in this thesis. Most work has used model-based metric definitions as SLAs which need extra manual effort from service engineers or modellers in order for them to comprehend and write adequate metrics related to a stochastic model. However, the work in this thesis starts from an SLA specified by an engineer to provide him/her with the metrics that are a closest fit with the intended meaning of the SLA. SlaCP methodology also differs from other works in that they do not provide an all-in one methodology that uses an existing SLA as an input to predict compliance automatically. In addition, they do not provide a generic methodology that can be used for different SLAs and stochastic models; finally they do not consider the architectural design required to build a tool that automates it. The value of such a methodology is to help a user who is not expert in model-based evaluation and analysis tools to predict SLA compliance as automatically as possible.

2. *An implementation of the SlaCP methodology using WSLA and SDES (called WslaCP methodology).* This contribution includes two inputs:

   (a) *A mathematical representation of WSLA contracts.* The structure and semantics of all the WSLA elements constituting the SLO are formalised. A precise and formal interpretation is given of the basic measured QoS metrics, the time instances/intervals, and the functions of the composite metrics; this is accomplished by associating them with mathematical terms. A mathematical representation of SLAs is available in the literature [33] but not in the fashion considered here. The work in this thesis not only formalises the main elements of an SLA, it also formalises how different QoS metrics are composed according to the WSLA ontology. In addition, the specific interpretation and semantics of the different WSLA elements used in composing the desired QoS metrics are determined in

detail. Defining the semantics of WSLA elements is important because some of its terms are described vaguely, being defined using natural descriptive language only (e.g. Span, Gauge, etc.). This thesis provides a mathematical interpretation, more rigours than the descriptive one, so that the semantics of WSLA elements can be easily and fully understood.

(b) *A theoretical mapping of WSLA on SDES.* This mapping consists of:

- Systematic translation of service operations, on which QoS metrics are defined, onto SDES state variables/actions.
- Systematic translation of the measurable QoS metrics, indicated in a WSLA specification, onto SDES reward variables.
- Systematic translation of the time instants and intervals, at which metrics are measured, onto a set of observation intervals for the reward variables.
- Systematic translation of the WSLA functions, used by composite QoS metrics, onto functions associated with a mathematical semantic tailored to the model's stochastic nature; this specifies further the reward variables in SDES.
- Finally, a systematic translation of the SLO onto an evaluation function that allows SLA compliance probability to be evaluated: i.e., a determination of whether the agreed service level can be met.

This contribution is important because it facilitates the understanding of the abstract SlaCP methodology. It also reflects the feasibility of providing a formal methodology for mapping from a WSLA to a general-purpose stochastic model. Predicting SLA compliance is possible using the model emerges from the mapping. Hence, by solving this model the provider is able to produce the values needed to evaluate the SLA compliance before deploying the service, thus avoiding penalties.

3. *An implementation of the architectural design of a software tool to support and automate the WslaCP methodology.* In order for the methodology to be more useful and user-friendly, it is employed in a software tool that facilitates and automates most of its different steps. This makes two contributions:

(a) *SDESSch*: This is an intermediate XML language that expresses the mapped elements from the WSLA's mathematical representation into SDES in a high level, machine-readable format. This language is independent of the SLA being used and the stochastic modelling formalisms

being utilised. Creating this language is necessary to achieve a higher level of automation and modularity for the tool across different modelling formalisms by performing a simple translation into them.

(b) *An all-in-one software tool that automates the WslaCP methodology as much as possible*: The general architectural design of the tool is implemented in a software tool that is developed using Java and which is augmented with both the Möbius and SPNP tools. The construction of this tool illustrates the viability of the methodology in practical terms and adds more value to it. In addition, it is a step towards helping users to check automatically the probability of SLA compliance for different SLO thresholds and for different service parameters. This is because this tool is the first that can automatically derive QoS metrics and SLOs definitions from an existing SLA, map them as a stochastic model and its rewards, take their expected values, and then predict SLA compliance.

4. *An evaluation of the proposed methodology and the tool through a detailed case study.* The case study utilised in this research facilitates the demonstration of the applicability of both the methodology and the tool, the degree to which they can achieve their desired objectives, and areas of enhancement. The value of such an evaluation is that it leads to a new contribution, namely, the use of a WSDL file in the automatic creation of the service model. This contribution proves that using other supporting documents of a service, such as WSDL, can provide a more complete service model and hence increase the level of automation of the methodology.

1.7 Thesis Outline

The rest of the thesis is organised as follows:

Chapter 2 provides a relevant literature review and offers background information that allows the reader to understand the topics and related work in the area of SLA compliance prediction. The chapter focuses on SLA content and some of the languages used to build it. It also explains SLA compliance management and the different approaches to do this, as well as shedding light both on the stochastic models used in the thesis and on the SDES formalism. It presents performance, dependability and performability models with the types of metric defined in them. Finally, some tools that are used for stochastic modelling and solving are presented.

Chapter 3 demonstrates the design of the SlaCP methodology and then presents it from two perspectives: those of the user and tool designer. It also gives an outline of the implementation of this methodology, called WslaCP, for WSLA and SDES in particular, as well as how the different steps in this outline are described in subsequent chapters.

Chapter 4 introduces the mathematical representation of WSLA contracts in addition to the precise semantics of WSLA elements. Formalising an SLA and defining the semantics of QoS metrics is an introductory step in implementing the methodology and for mapping the WSLA contract to the SDES model.

Chapter 5 empirically describes the theoretical mapping from WSLA contracts to the generalised Stochastic Discrete Event System, SDES. A discussion about the feasibility of this mapping is also provided.

Chapter 6 proposes the architectural design for developing a software tool that is able to automate the methodology that is presented in Chapters 4 and 5. It also describes its implementation and the intermediate language used to aid the automation and modularity of the tool.

Chapter 7 evaluates the described methodology and tool through a case study in terms of its automation, feasibility, generality, and usefulness for the user. It also shows how using WSDL can support the automated model creation.

Chapter 8 concludes the thesis by answering the research questions, providing reflections on the whole thesis, and summarising the research's contributions. It also suggests some possible extensions and future work for such research.

1.8 Publication History

The thesis contains some parts that have been published in or that are related to peer-reviewed publications written by the author. These publications are as follows:

1. "Rouaa Yassin Kassab and Aad van Moorsel. Mapping WSLA on reward constructs in Möbius. In *24th UK Performance Engineering Workshop*, pages 137-147, 2008."

   The idea of mapping an SLA into a model was first introduced in this publication at the UKPEW 2008 workshop [34]. It illustrated the feasibility of this approach through an ad hoc mapping from a WSLA to Möbius rewards.

2. "Rouaa Yassin Kassab and Aad van Moorsel. Formal mapping of WSLA contracts on stochastic models. In *8th European Performance Engineering*

Workshop - EPEW 2011, volume 6977 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2011."

This covered an extended and more generic version of a WSLA compliance prediction approach [15]. This version, albeit in a somewhat different format, is presented in more detail in Chapters 3, 4 and 5.

3. "Simon Edward Parkin, Rouaa Yassin Kassab, and Aad van Moorsel. The Impact of Unavailability on the Effectiveness of Enterprise Information Security Technologies. *In Proceedings of ISAS'2008*. pp.43-58."

   I was a co-author of this publication [35] in which the contribution was in the development of a SAN model of the USB access control. Although it is not included in the thesis, this work helped in developing a better understanding of the construction of the stochastic model and the usage of the Möbius tool.

In addition to peer-reviewed papers, a number of technical reports have been written and published in the Computing Science School Technical Reports Series.

1. "Rouaa Yassin Kassab and Aad van Moorsel. *Predicting Compliance of WSLA Contracts Using Automated Model Creation*. School of Computing Science. 2010. School of Computing Science Technical Report Series 1204."

   This was a preliminary version of the design of a software tool that exploited the early version of the WSLA compliance prediction methodology.

2. "Rouaa Yassin Kassab and Aad van Moorsel. *Formal Mapping of WSLA Contracts on Stochastic Models*. School of Computing Science. 2011. School of Computing Science Technical Report Series 1245."

   This was also a preliminary version of the generalised theoretical WSLA compliance prediction. The paper was later enhanced and published in EPEW 2011 [15].

Chapter 2

Background and Literature Review

This chapter provides background information and a literature review related to the research conducted in this thesis to give the context for the problem of SLA compliance prediction. The background gives information about the theoretical and practical basis of the study of SLA prediction methodology, including service-oriented computing, SLA specifications and their related QoS metrics. The relevant literature is reviewed including: presenting different perspectives of the motivations behind performing SLA compliance prediction (such as maximising revenue, increasing customer satisfaction, minimising SLO violation, or raising the alarm regarding the probability of performance degradation); describing different SLA compliance prediction techniques used by researchers (such as measurement, simulation or performance models); presenting approaches to and issues related to QoS mapping and adding performance attributes in order to produce analytic models.

The remainder of the chapter is structured as follows: Section 2.1 gives a brief description of service-oriented computing and web services. In Section 2.2, the importance of SLAs, along with the content and languages used to create them are presented. Section 2.3 introduces SLA compliance management using both monitoring and prediction approaches, the different perspectives of performing SLA compliance prediction, and the methods used for this purpose. This section also illustrates the motivation behind the choice of using model-based prediction in this thesis. Section 2.4 presents the aspects utilised in designing the methodology and Section 2.5 explores stochastic modelling formalisms, and then describes those that are relevant to this thesis. Section 2.6 describes the types of attribute, the reward models, and the techniques used for analysing them. It also outlines some of the tools used to create such models. Finally, Section 2.7 concludes this chapter.

2.1 Service Oriented Computing and Web Services

The rationale for companies' tendencies to use service-based applications is that they help to build dynamic, easily configurable and low cost software applications in a way that increases their business efficiency [36]. The service-oriented computing approach and the associated Service Oriented Architecture (SOA) paradigm were developed to employ services as the key elements in building distributed applications [3]. The architecture of the applications developed using service-oriented computing depends on the existence of three roles: a provider, a customer and a registry that is used by the provider to advertise his/her service [36].

A web service is an implementation of the Service Oriented Architecture approach [3]. It is defined as an interface with a set of operations that can be invoked through the Web using XML messages [37]. The web service model also includes three roles that interact with each other using **Publish**, **Find** and **Bind** operations as depicted in Figure 2.1.

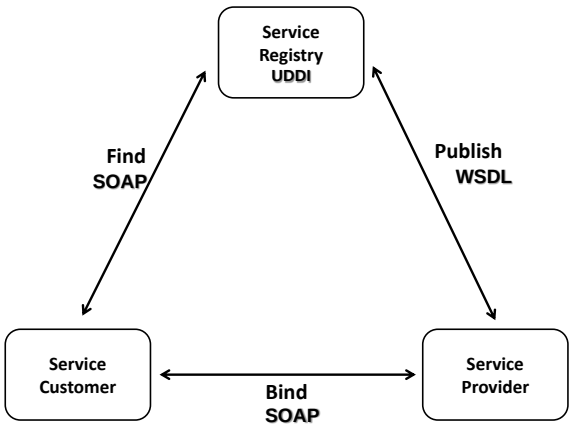

Figure 2.1: The web service model

The service provider advertises his/her service by **Publish**-ing its description in a registry. The description is written using the Web Services Description Language (WSDL)[1], while the registry of this service description is the Universal Description, Discovery, and Integration (UDDI) directory[2]. Service customers can access the service description registry using the **Find** operation. Finally, to invoke the desired service, the customer has to **Bind** to the service that exists on the provider's side.

[1] www.w3.org/TR/wsdl.html
[2] www.uddi.org

Registry access and service invocation are done through the Simple Object Access Protocol (SOAP)[1] [37, 38].

2.2 Service Level Agreement

In the interconnected world of electronic services, the quality of the offered service has been used by customers as a factor to distinguish between providers that offer the same service [9]. To express a service's offering, Service Level Agreements (SLAs) were developed. These help in organising the relationship between the service providers and their potential customers, whether they are consumers or other businesses [39]. The SLA is a document that includes information regarding the definition of the contractual parties of a particular service and their roles, the description of the specific QoS promises offered by the service provider for different sets of businesses and customers, the charges the customer has to pay for using the service, and finally the provider's obligation in case of failing to satisfy its pledges [39, 40].

SLAs have been used in a wide range of areas like e-commerce and outsourcing between organisations [41]. Using SLAs, customers gain more confidence about the service they desire to use because they have clearly defined expectations to receive the service for which they pay. For this reason, customers have become keener to negotiate an SLA that increases efficiency [9]. Providers, likewise, have become more eager to propose a reasonable offering in their SLA that better suits their real resource capacity in order to avoid incurring any penalties. Furthermore, using SLAs compels providers to control and monitor their services more efficiently to avoid any breach of contract that may lead to financial loss [9].

Contract breaching of an SLA occurs when a service provider is not able to fulfil the Service Level Objectives specified in the service SLA. An SLO is defined through thresholds that should be maintained for the desired service properties over a certain validity period. A simple example of an SLO is *'continuous down time is less than 8 minutes in a business day'*.

In this section, the QoS metrics used within an SLA are described along with their categorisation. Then, some of the languages which are used in SLA specification are described.

[1] www.w3.org/TR/soap/

2.2.1 QoS Metrics Related to an SLA and their Categorisation

A service has a set of functional and non-functional properties; the non-functional ones are restrictions on the functionality of the service and are referred to as QoS attributes [42, 43]. A service can be assigned different QoS attributes which are usually defined for the operations of the service. However, any service object can be assigned a QoS, such as interfaces, attributes, operation parameters and results [44]. Each QoS attribute is measured by a metric and is thus referred to as a QoS metric. A metric is usually used to describe the unit, the type of value that it can take, and the time required to measure this attribute. For this reason, a QoS metric can be considered as one evaluation of the QoS attribute [44].

In terms of the SLA, QoS metrics are its primary components and are related to the non-functional attributes of the specified service [5, 21]. The service customer and provider usually negotiate the desired quality of service metrics to be included in their SLA. In addition, they agree to set the level at which the service has to offer these metrics [45]. A QoS value is a positive or negative number [46], while the QoS level specifies the upper or lower limit a QoS value can reach.

To be able to understand the semantics of the QoS metrics in a particular SLA, the class under which these metrics may be categorised has to be addressed. This

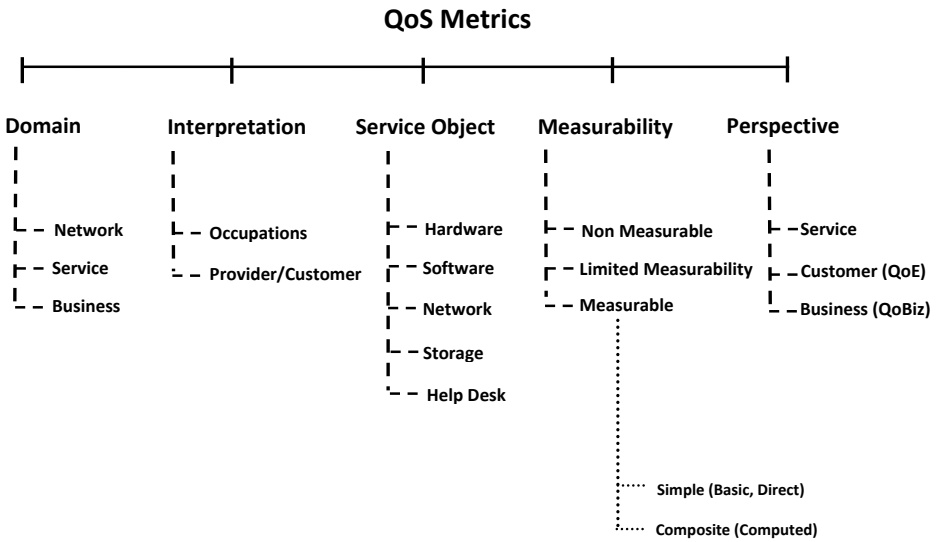

Figure 2.2: Summary of QoS metrics' classification

is also useful in assigning the monitoring semantics of an SLA contract when it is presented later in Section 4.5. A summary of all QoS classifications and sub-classifications are depicted in Figure 2.2. These classifications are described in what follows according to their occurrence in the figure moving from left to right.

The choice of QoS metrics in a particular SLA may vary according to the **Domain** in which they are used, such as **Network** management, **Service** management and **Business** management [21]. For example, in the network management domain, a typical QoS metric is '*bandwidth*', which indicates the network capacity [47]. On the other hand, a typical QoS in the business management domain may be '*revenue*' and '*cost*', indicating the outcome and price of the service [48]. Despite this variety in QoS metric types used in different domains, the most commonly used ones are those relating to service performance (e.g. '*response time*') or to reliability (e.g. '*availability*') [21]. Additional QoS metrics can include '*serviceability*', '*security*' and '*scalability*' [5, 21, 45, 49]. The aforementioned QoS metrics can be defined in the web service domain, which is the focus of this thesis, as follows [5, 50]:

- *Availability*: Indicates the probability that the service is up and working.
- *Throughput*: Indicates the number of requests the service receives during a specific period of time.
- *Scalability*: Indicates the ability of a web service to handle arriving requests even under different workloads.
- *Security*: Indicates the type of mechanism used to authenticate and authorise customers, which is critical to protect their privacy.
- *Response time*: Indicates the time taken to respond to a request once it is received by the service [51].

As the type of QoS metric used within an SLA differs from one service domain to another, the **Interpretation** of a single QoS metric varies accordingly [21]. Each **Occupation** may have its own interpretation of a certain QoS metric, just as the service **Provider** and **Customer** may have different perspectives [21]. An example of this diverse understanding of a QoS metric is that of '*availability*'. From the provider's perspective, availability means that the hardware is not down (an aspect of infrastructure). However, from the customer's point of view, it could be the ability to serve a request (a service application aspect) or what is called '*successability*' [1] [21].

[1] Successability is the number of response messages received divided by the number of request messages sent [52]

Another categorisation of QoS related metrics can be according to the **Service Object** they are defined for. This might be related to physical **Hardware**, application **Software**, the communication **Network** and its infrastructure, data **Storage** repository or service **Help Desk** or to combination of them [21]. A typical example of a QoS metric related to storage is *'bytes per second'* which reflects the reading and writing speed as specified in [21].

In addition to distinguishing QoS metrics according to their related domain, interpretation, and object type, they can also be classified into three categories according to their **Measurability** [21]. The first category is the **Measurable** metrics; these, as their name suggests, can be measured automatically from the underlying service, for example *'queue size'*. A metric that cannot be automatically measured is called a **Limited Measurability** metric. This metric is related to customer opinion, for example, the *'degree of customer satisfaction'*, and can be collected using a questionnaire only. Finally, the metric that does not belong to the previous two types is a **Non-Measurable** metric such as *'staff quality'* [21].

Measurable QoS metrics are the most desirable metrics since they can be quantified and evaluated [53]. These metrics can also be categorised according to simplicity into **Simple** and **Composite** metrics. The simple (basic or direct) metrics are obtained directly from the service by probing or instrumentation using measurement directives such as throughput of *'customer arrival'*. However, composite or computed metrics are derived by applying a function to a set of simple metric values such as *'maximum throughput of customers'* [17, 21].

Another way of categorising QoS metrics is according to the **Perspective** of the **Service**, **Customer**, and **Business** [53]. In terms of service, QoS metrics are related to the IT infrastructure of the service from the provider's point of view. This perspective is related to the service itself and does not consider the customer's or business's point of view. An example of such a metric is service *'throughput'*. The customer perspective, on the other hand, is usually referred to as the Quality of Experience (QoE) and is related to the degree of customer satisfaction with the service; this could involve subjective factors. An example of a QoE metric is the *'response time'*, as recognised by customers, from sending a request until receiving the response (i.e. including network delays). The last perspective is that of the business, which is referred to as Quality of Business (QoBiz) metrics. These metrics convey how the service provider or customer looks at the service based on the monetary value of the service's properties. An example of a QoBiz metric is the provider estimated financial loss per lost customer.

2.2.2 SLA Specification Languages

Until a few years ago, SLA contracts were mostly written using natural expressions; examination of compliance to the agreement also had to be done manually [54]. One attempt to facilitate this process by using SLA templates was limited and unable to specify different service levels for different customers [54]. For this reason, it has become a necessity to automate the procedure through which different SLAs are flexibly described, provisioned and observed [54].

Several SLA specification languages have been developed by researchers within the service provision community to address the previous aspects [54]. Their aim is to simplify the contractual process for the parties involved and to minimise the time and cost included in this process [54]. Foremost among the SLA languages are the Web Service Level Agreement (WSLA) [17] framework and the Web Service Agreement Specification (WS-Agreement) [1]. The Service Level Agreement Language (SLAng) [18] is also another attempt. All of these languages define the most important aspects of SLAs, typically focusing on the technical aspects of the service [20].

In the following subsections, WSLA, WS-Agreement and SLAng are described in detail. Then, a comparison of these languages in terms of the requirements of the proposed methodology is presented.

2.2.2.1 Web Service Level Agreement (WSLA)

The Web Service Level Agreement (WSLA) contract is an XML-based document. The main strength of this contract is its flexibility as it allows the contractual parties to define their desired QoS metrics [17]. This flexibility is due to WSLA's constructive ontology that facilitates the construction of QoS metrics in a hierarchical way [10]. This ontology defines the desired QoS metric by allowing a set of *well-defined terms* (i.e. measured QoS metrics) to be composed using different *composing operators* to produce *new terms* (i.e. composite QoS metrics) [10]. The *well-defined terms*, *composing operators*, and the ultimate *new terms* are referred to in WSLA as `MeasurementDirective`(s), `Function`(s), and `SLAParameter`(s) respectively. For example, to define the new term *'average of service response time'* in WSLA's constructive ontology, the well-defined term (i.e. `MeasurementDirective`), *'response time'*, needs to be specified first, then the composition operators (i.e. `Function`(s)), *'series of response time values'* and *'average'*, need to be specified to create the desired new term (i.e. `SLAParameter`).

WSLA describes all the aspects that are contracted between the signatory parties, the service supplier and the service consumer, regarding a specific service. The

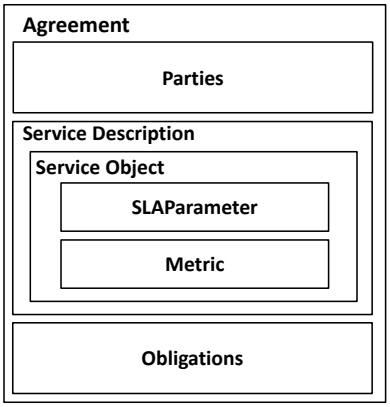

Figure 2.3: WSLA agreement structure

negotiation process for establishing such a document could be accomplished either online or off-line through the use of a WSLA template that comprises most of the defined and the agreed upon information needed to create the SLA contract [17].

WSLA represents, in its agreement, all the information that is normally contained within an SLA document. This information is situated, as depicted in Figure 2.3, in the following three main parts:

- **Parties**: contains information about the parties engaged in the SLA contract. These are the signatory parties, the service provider and the service consumer, in addition to the supporting parties that may be involved in measuring, monitoring or managing particular parts of the contract [17, 34].
- **Service Description**: contains an explanation of the **Service Object** on which the QoS metrics are defined, its **SLAParameter**(s), and what **Metric**(s) are used to compute their values. A metric will be measured from a source by identifying a measurement directive, if it is basic. However, in the case of a composite metric, its value will be computed using a function that takes other metrics or constants as its operands. The SLA parameters are one of the important parts of an SLA because they correlate the metrics to a particular consumer with specific accepted values [17, 54, 34].
- **Obligations**: contains Service Level Objectives the service provider is obliged to maintain. These are the agreed values of SLA parameters during a specific duration and the actions to be taken in the case of a contract violation [17, 34].

The basis of the WSLA language is designed to be thin; its rules and its standard extensions provide the most common requirements for the service providers [17].

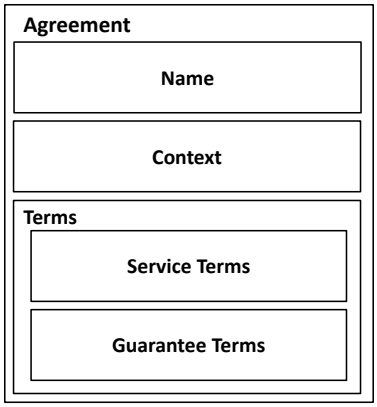

Figure 2.4: Agreement structure of WS-Agreement, as specified by Andrieux et al.[1]

However, to cover the complexity of real systems, WSLA, by the use of XML schema, can be extended to create new types which express domain specific concepts. By using this, new service descriptions, functions, and measurement directives can be derived and used flexibly in describing different ranges of SLAs [17, 34].

One of the main strengths of WSLA in addition to its constructive ontology, is its ability to provide management information (relating SLA to a monetary value) and management actions (a warning in case of contract contravention). Furthermore, since SLA parameter descriptions are separated from the SLOs and those of the parties, a third party monitoring agent can be incorporated without breaching privacy. A WSLA shortcoming can be that it requires measurements that are available from monitoring agents whose role is not defined in the language specification [55]. In addition, its semantics are not formally specified.

2.2.2.2 Web Service Agreement Specification (WS-Agreement)

WS-Agreement is an SLA specification presented by the Grid Resource Allocation and Agreement Protocol Working Group of the Compute Area of the Open Grid Forum [1]. It is an XML-based language and web service protocol for: firstly, promoting through templates a set of possible accepted agreement offers from agreement responders (which could be the service provider or consumer); secondly, generating a proposed agreement offer that the agreement initiator is keen to establish based on one of these templates; thirdly, negotiating this agreement according to specific constraints; fourthly, creating an agreement between the service provider and customer with all conditions and restrictions, and then finally observing its fulfilment [1]. The WS-Agreement is structured in three parts [1] as follows:

1. *Agreement Schema* (agreement creation offer schema): This is used by the agreement initiator to create an offer according to a specific template. The agreement creation offer and the agreement are structurally the same. The agreement offer, as specified in Figure 2.4, contains the agreement **Name**, the **Context** (this includes the involved parties and the agreement life span), and the **Terms**, which is the most important part of an agreement offer. Within the *Terms*, each term contains at least one **Service Term** and zero or more **Guarantee Terms** which could be combined using logical operators. These are defined as follows:

    - Service term is used to describe the service to be offered and consists of three parts: *Service Description terms* (SDTs) for describing the service functionality, *Service Reference terms* to identify a service, and *Service Property terms* to identify quantifiable measures (e.g. response time) that are used to define service level objectives.
    - Guarantee terms identify the agreed quality level of the offered service that is specified in the service term. The Guarantee term consists of many parts such as the *service scope* which names the services for which this guarantee is valid; the *qualifying conditions* that make the guarantee obligatory; the *service level objectives*, which define the required quality of the service; and the *business value* list that includes the SLO's importance and the agreed penalties and rewards.

2. *Agreement Template Schema*: This is used by the agreement responder to promote acceptable agreement offers.

3. *Set of ports type and operations*: These are used for organising and administrating the agreement life-cycle operations, such as accepting or rejecting the offer, or identifying a type of a document for monitoring the agreement states.

One of the main strengths of WS-Agreement is that it is powerful in terms of extending new elements that are domain-specific [56]. In addition, it contains more information about the service functional's properties than WSLA does. Also, business values related to QoS metrics can be specified even if an accounting procedure is not supported [57]. WS-Agreement does not support a constructive ontology to define QoS metrics which is necessary to understand their exact definitions. Also its semantics are not defined precisely. However, WS-Agreement has recently been extended with SWAPS Extension [58] to overcome this semantic ambiguity.

2.2.2.3 Service Level Agreement Language (SLAng)

The Service Level Agreement Language (SLAng) is a language that describes a domain-specific SLA between a service provider and a consumer. It is one of the deliverables of the Trusted and Quality of Service Aware Provision of Application Services project. Its syntax is defined by an XML schema that allows it to be combined with existing service description languages (e.g. WSDL) or other technologies to allow a complete business clarification [18].

The semantics of SLAng are described by producing a UML model of the language rules which is then used in the service's behavioural model (including models for the involved parties). By producing this abstract syntax, the SLA constraints, which are described using OCL (Object Constraint Language, which is part of the UML standard), comprise the semantics of SLAng [55].

The specification of SLAng is derived from a reference model of a distributed system architecture. This reference model consists of three tiers: *Application*, *Middle*, and *Underlying resources* which contain an *Application*, an *Application Service*, *Container*, *Storage* and *Network* providers [55].

An SLAng contract consists of a specification of the involved parties, contract information (contract lifetime), and a Service Level Specification (SLS) which specifies the QoS metrics and their related values, the provider roles, the user roles, and their shared roles [55].

SLAng differs from the other SLA specifications in many aspects [55] such as:

- Unlike most SLA specifications that concentrate on SLAs for web services only, SLAng is used for a wider range of Internet services (such as the application service provision, Internet service provision, and storage service provision).

- SLAng represents service and client behaviour in a formal semantic. This decreases vagueness in the meaning of the language and minimises the need to re-check it to eliminate any contradiction or flaw. Moreover, it makes SLAng more user-friendly and simplifies SLA negotiation.

SLAng however suffers from the absence of certain characteristics. For example, it does not contain a description pertaining to the actions the obliged party will take or the charges that this party will incur if a contract is breached. In addition, unlike WSLA, its structure does not allow the submission of only a part of the SLA to a monitoring party. This causes information about the parties and the service to be revealed to such a party. Furthermore, SLAng is unable to identify new parameter types according to existing ones [55].

Table 2.1: Comparison of SLA languages

	QoS Constructive Ontology	Semantic Description	Functional Properties
WSLA	Yes	Natural language	Contains reference to WSDL
WS-Agreement	No	Natural language	Defined in Service Description Terms, SDTs
SLAng	No	Formal using UML and OCL	Contains reference to WSDL and BPEL

2.2.2.4 Comparison of SLA Languages

The main differences, with regard to the methodology proposed in this thesis, between the three SLA languages described in the previous subsections, are summarised in Table 2.1. In this table, it is clear that the most vital difference between WSLA and the other languages is its constructive ontology that allows hierarchical QoS metrics to be defined. SLAng's core difference from the others is that it is the only language whose elements are formally defined using UML and OCL. Finally, an important point regarding WS-Agreement is that some of the functional properties of the service are defined within it, whereas the others only contain references to separate documents that may contain a service description document like WSDL, or a service business process like BPEL[1].

2.3 SLA Compliance Management

SLA management in general can be related to different steps in the SLA life cycle, including its discovery, negotiation, establishment, violation (compliance issue), termination and enforcement [13]. This section concentrates on SLA compliance management as it leads to the topic of SLA compliance prediction which is the focus of this thesis. In the following sub-sections, SLA management is defined and the motivation for using is described, along with where in the service SLA management is conducted. The types of SLA management are described thereafter.

2.3.1 What is SLA Management?

SLA management, as stated by Sahai et al. in [59], implies using techniques for continuously monitoring, enforcing and optimising an SLA. The notion of Service

[1]BPEL is the Business Process Execution Language that describes business processes.

Level Management, as defined by Buco, was developed to describe the continuous and accurate evaluation of SLA contracts in a way that aims at making the right management decision [60].

SLA management can be considered as referring to SLA monitoring and prediction during service runtime only. Design-time prediction of an SLA contract is considered in the literature but under the performance prediction modelling area, as in the work of Rathfelder in [61]. SLA design-time prediction was not considered as part of SLA management until the creation of the SLA@SOI framework [1]. In this framework, the prediction of the performance and reliability of the QoS properties of a service was included as part of the SLA management.

2.3.2 The Motivation for SLA Compliance Management

SLA compliance management and the performance prediction of a service are motivated primarily by business considerations, where minimising SLA violations is essential so that costly penalties can be avoided [22]. Another motivation for maintaining the QoS metric levels defined in an SLA is to increase customer satisfaction so that customers will re-use a service continuously [24]. The latter motivation can also be considered to have a business value.

A wide range of studies have offered a variety of techniques to perform SLA management depending on the previous motivations. This is done by utilising an SLA or its QoS metrics as a basis for enhancing the service infrastructure or for assisting any modifications to be made to SLA thresholds in a way that will accomplish the desired business objectives [62].

An example of a work that considers an SLA for delivering value to business metrics is the work of Bartolini [62] who introduced a Management by Contract (MbC) concept as a new paradigm for IT Management. He proposed a way of analysing contractual relationships in order to better inform IT-related decisions by creating a utility function that computes the business loss/gain in cases of breaching or fulfilling an SLA. Using this function, the provider can consider what is the optimal choice between these cases. In a later work [63], Bartolini proposed the term MBO (IT service Management driven by Business Objectives). He built a decision support tool and an MBO business objectives information model for incident management to prioritise incidents, which are variations of the standard operation of a service that causes a drop in the QoS. Prioritising incidents are based on their financial impact on the business objectives and they provide stability by quickly

[1] http://sla-at-soi.eu/

re-establishing the degraded service.

Another work that depends on an SLA to enhance business metrics is the work by Sauve [64] who introduced an objective model to choose the optimal SLO according to business perspectives. This optimal SLO was used later in building an SLA that would minimise both the cost of system design and business loss.

An on-line management and dynamic resource allocation according to business driven optimisation is also considered in the literature. For example, the work of [65] used SLA thresholds as part of an algorithm to check any deviation in service levels. This was done by running multiple simulations to allow a resource manager to choose the optimal resource usage in a virtual environment in such a way that the SLA could be fulfilled. Also, the work by [66] introduced a load-balancing solution to distribute the incoming requests of a web server among a class of web services when a heavy work load was detected. It predicted the maximum accepted request the server could take and then distributed them on the services in a way that satisfied QoS guarantees in the service SLA.

2.3.3 Where is SLA Compliance Management Conducted?

The existing frameworks for SLA management consider QoS metrics either at the service level (i.e. application level), as in service throughput, or at the infrastructure level, as in server properties; the latter is the one most often considered in the literature [67]. However, some runtime monitoring and prediction techniques consider QoS deviation at both service and infrastructure levels, as in EVEREST+ [68], where a general framework was described for detecting SLA violations at different levels using statistical model-based prediction techniques.

SLA management can also be accomplished on the service side, the customer side, or at specific points through the network. For example, QoS metrics such as *processing time* can be measured on the service side, while *total round-trip* must be measured on the customer side [69]. Many researchers are keen in server side management only, using probing requests to the service, such as the work in [70]. Others focus on client side management only, like the counters of the Windows Communication Foundation[1] that are measured continuously to return server side metrics. Little research, such as the work in [12, 71], considers both service and customer sides.

[1] http://msdn.microsoft.com/en-us/library/ms735098.aspx

Table 2.2: Comparison of SLA managements types

	Runtime monitoring	**Runtime Prediction**	**Off-line Prediction**
Carried out	Service runtime	Service runtime	Service design time, deployment time, negotiation phase, or off-line modes
Violation Detection	After occurring	Before occurring	Design or deployment time
Advantage	Notifying of service weakness	Reducing the negative consequence by corrective actions	Change service infrastructure, SLO threshold.

2.3.4 Types of SLA Compliance Management

There are three types of SLA compliance management that are considered in the literature: SLA runtime monitoring, SLA runtime prediction and SLA off-line prediction. The main differences between these types are concerned with when they are carried out, when SLA violations are detected, and what are their advantages. These are summarised in Table 2.2. The three types of SLA management, along with the three comparison criteria, are described in more detail in what follows.

SLA runtime monitoring: This is carried out while the service is running and can detect SLA violation only after it occurs [24]. Although this detection is useful in notifying the provider of a specific weakness in the service, it cannot preclude them; neither can it give the provider enough time to consider any changes [24]. Examples of monitoring SLA compliance are in the works proposed in [12, 54, 69, 71, 72]. These works are not described here as they are out of the scope of this thesis.

SLA runtime prediction: To avoid paying penalties and to have consistent satisfaction for their potential customers, service providers are keen to predict SLA violations before they have occurred [24]. This early prediction of SLA violation is useful because it gives the providers a considerable amount of time to take any corrective action that may reduce or eliminate the effect of this violation [24]. For this reason, SLA runtime prediction is carried out while the service is running, to predict any SLA violation before it occurs.

SLA runtime prediction may be performed using the following models: (1) *Historical Models* that depend on QoS values and are based on how the service ran in the past to predict prospective ones; (2) *Observation Models* that depend on the

present infrastructure model to estimate the potential QoS values; (3) *Predictive Models* that depend on past service usage and past infrastructure observation to predict prospective QoS values [73].

On-line predictive models, which are used for the sake of runtime prediction, often include machine learning regression capabilities based on historical data training which allow the model to predict SLA deviation [24]. For example, the work in [24] proposed the notation of check points to perform the prediction at specific points during the service runtime. Two types of data facts (which are previously measured values of a typical QoS metric) and estimates (which are values that are not yet available) are used as input to a predictive model to produce a numeric estimation of SLO.

According to previous studies, SLA runtime prediction is useful when the service runs for a long time and enough historical data is available for the provider.

SLA off-line prediction: This may be carried out throughout the service design time (before implementation), in the service deployment time, or during off-line modes of the service. It can also be performed in the SLA negotiation phase. It is used during these times to predict the probability of SLA violation or the service's ability to conform to a pre-defined SLA or QoS level.

SLA off-line prediction has the same advantages as SLA runtime prediction but it allows more time to consider any changes before implementing the service; this can also save money. Hence, this early prediction of SLA compliance will allow the provider to consider changes to the IT infrastructure of the service in a way that will help it to conform to the SLA, or to change SLO thresholds that contain the agreed QoS levels to conform to the service infrastructure capability [25].

SLA compliance prediction needs data that are available only during runtime (e.g. rates of incoming requests). Thus, obtaining historical data measurements, estimations, or probability of their values has to be provided to be used as input for predictive models [24]. This can be seen in the work of [74].

2.4 Areas Related to SLA Compliance Prediction

To the best of our knowledge, this work is novel in using this kind of approach. Few related works in exactly the same area exist like the works related to using Stochastic Probes for the specification and evaluation of the performance queries [75, 76]. In addition, this thesis includes ideas and aspects that expand into different existing research areas. This includes using a model-based approach to predict

the QoS metrics of an SLA, mapping between source and target formalisms, and finally transforming design-oriented models into analysis-oriented ones. These related works are described in the following sub-sections.

2.4.1 Using Stochastic Probes for Performance Queries Specification and Evaluation

Similar to the approach taken in this thesis is the work in [75]. This work present a new methodology to define performance measures on stochastic models. This approach makes use of FPS, a unified Functional Performance Specification Language, to define passage time, transient and steady state performance queries. It then makes use of a generalisation of stochastic probes, a formalism for describing stochastic process algebra models, to produce the values of these queries. This approach can be used to predict the probability of meeting different SLO levels defined in an SLA. The similarities between the approach taken in this thesis and [75] laid in the specification of the performance queries in an intermediate language that could be integrated in a stochastic model to produce their value. In addition, both approaches separate the definition of the performance query from the stochastic model. The work in this thesis is different since it does consider an automatic mapping from a given SLA into a generalised stochastic model.

2.4.2 Model-Based Evaluation

In the following subsections, the motivation for using a model-based evaluation in general is described. Then, using such evaluation for predicting SLA compliance is presented, together with some related works that have adopted this usage.

2.4.2.1 Why Use Model-Based Evaluation?

Many techniques have been presented for the purpose of predicting and evaluating the different attributes of a system. These techniques are summarised in Figure 2.5 and are described in what follows.

Guesses from **Similar Systems** can be derived to predict estimations about these values; measurements of a **Real System or its Prototype**, in addition, can be utilised. This may be not practical, however, since such techniques cannot be used unless the system is deployed. Furthermore, there may be a need in many situations to wait for a long period of time to derive measures of specific events (e.g. the expected failures in a system). Another way to evaluate a system is

to construct a **Model** that captures its characteristics [2]. Models, describe the temporal characteristics of the system and are used in complex computer systems to predict their attributes using simulation or numerical solvers [77]. These models are also used in software engineering to evaluate the characteristics of software during its development [78].

The models used in system evaluation can be **Simulation** based (Discrete Event Simulation DES) or **Analytic** models [2]. Simulation is widely used and can convey system attributes correctly. In spite of the availability of several tools that assist the user in designing and executing a simulation, it is very expensive because a long execution time is required to produce precise results and it is difficult to simulate rare events. On the other hand, analytic models that describe important aspects of system behaviour in an abstract way are less time consuming and more cost effective in representing a wide range of system characteristics for analysis [2]. There is a broad variety of analytical models, often supported with software tools to create and solve them. These models have potential power and limitations related to their simplicity, the level of precision with regard to the results produced, and the existence of software supported tools.

Analytical models can be categorised into two different kinds: **Non State Based Models** (such as *fault trees* and *reliability graphs* for system dependability and *directed acyclic task precedence graphs* for system performance), and **State Space Models** (such as *Markov Chains* and *Stochastic Petri Nets*) [2]. Non state based models are simple, accurate and have effective solution techniques but they cannot represent characteristics such as system component dependency, concurrency and synchronisation; each model can represent either performance or dependabil-

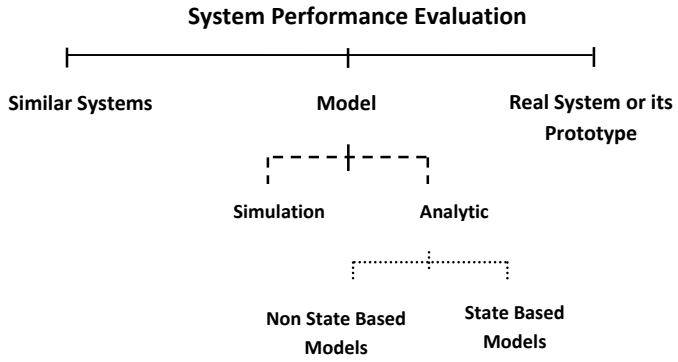

Figure 2.5: Techniques for evaluating system attributes

ity. State space models, on the other hand, eliminate these shortcomings; they are flexible and can model dependability, performance and performability [2]. However, they have the problem of state-space explosion.

2.4.2.2 Using Model-Based Evaluation in Predicting SLA Compliance

Analytical models in a service domain context, for which the SLAs are defined, can be used to predict QoS metric values and hence SLA compliance. Using these models for predicting the values of QoS metrics is employed in the literature. However, non-deterministic QoS metrics like *Security* are not considered useful metrics for model based prediction [24]. Model-based evaluation of an SLA contract is sometimes referred to in the literature as '*QoS prediction*'. Works in this domain consider QoS metrics out of the context of their SLAs, such as the work of [79] which used a predictive queuing network model to change dynamically the system parameters in a way that satisfied a set of stand-alone QoS requirements.

Due to the nature of services and the network that is connecting them, using a stochastic model to represent the service's analytical model is more natural [78]. The popular stochastic models which are used in model-based prediction are Stochastic Petri Nets, Queuing Network, and Stochastic Process Algebra [78].

An example of using stochastic models for SLA compliance prediction is the work pursued by Teixeira [26]. In this research, the author proposed a new methodology that predicts any deviation in SLA thresholds, for a response time metric in a SOA-based system, using stochastic models. He created an analytic model that implements different SOA features and then accompanied it with a failure model that is able to figure out if the result from the analytic model fails to comply with a pre-defined SLA. This is done using two types of transition, the former is connected to the analytic model and represents the completion of a request, while the latter is connected to the failure model and its firing time represents the agreed response time. When a new request arrives to the service, a token is sent both to the analytic model and to a place in a failure model; the modeller waits to see which transition fires first. If the analytic model completes first, the SLA is not violated and the token is removed from the place of the failure model. If the opposite is true, the SLA is considered to be violated. The model was simulated by TimeNet 4 to help in deciding the optimal SLA when the workload is known and vice versa. Doing this helps in assigning a workload limit at which the response time threshold can be satisfied. Comparing to the proposed methodology, the model in this work cannot predict any QoS metric in an SLA; rather, it is mainly used for predicting response time compliance according to changes in workload. In addition, the model deals

with the services as black boxes, which is not always effective since some QoSs can be related to a specific component in the service, not the service as a whole. Finally, the mapping of the response time into the model is not automated.

The same author suggested a similar mechanism for SLA planning in a data base [80]. In this work, the author considered a stochastic Petri Net model that is simulated to predict how the resource consumption and performance of a database can differ according to changing workloads. In this work, no mapping from the SLA was considered. Instead, different SLA clauses of response time under different arrival rates were assumed in order to choose one which was typical.

Model-based evaluation for the management of a cluster-based web service was also considered in [81]. In this research, the author constructed a queuing model which predicts the response time of a request under various resource allocations to inform a business-based utility function. In this work, there is no direct usage of an SLA.

2.4.3 Mapping between Source and Target Formalisms

In the methodology proposed in this thesis, the source and target formalisms are an SLA specification and a stochastic model respectively. In the literature, there are no directly related works regarding mapping SLAs to stochastic models. This is because SLAs were mostly mapped to and represented by models like the Unified Modelling Language UML to formalise the SLA structure precisely, such as in the work of [82]. Although UML models are precise and accurate, they are not a mathematically based method that will allow the modeller to carry out performance analysis which is needed for the work in this thesis. Hence, this mapping cannot be exploited.

Another mapping mechanism that is considered in the literature is mapping QoS metrics between different levels of the service: i.e. application or infrastructure levels. An example of this mapping is the work of [83] which mapped network performance QoS metrics from a Web service layer to the underlying network layer. Also, the work of [47] performed a mapping between the QoS metrics specified in SLAs and the network performance metrics. Although this mapping is useful for understanding the related service object that a QoS metric is defined for, the fine-grained mapping details may not be important in the prediction context; this mapping might be more suitable for use while monitoring.

Other works in the literature that concerned the mapping of QoS metrics actually focused on representing these metrics as the metrics of stochastic models in an ad hoc manner, rather than carrying out a one-to-one mapping as in the work of [26]. This is

because, in this research, QoS metrics were considered without their SLAs; instead, standalone QoS expressions were mostly used. Furthermore, the representation was accomplished manually and did not obey the exact temporal constraints through which the QoS value had to be maintained; only average or percentile values were considered. In addition, since the QoS metrics were considered outside the context of an SLA, there was no support for automatic mapping between them and the stochastic models. Therefore, and given the previously mentioned studies, the work in this thesis can be considered as bridging the gap between the SLA contract, stochastic models and the mapping between them.

2.4.4 Transferring a Design-Oriented Model to an Analysis-Oriented Model

The central idea in this thesis is based on using an SLA as the basis to create a generalised stochastic model, and then adding reward variables to it in order to produce a reward model. This is transformed into a specific modelling formalism to be solved so the expected values of QoS metrics can be obtained. This approach has similarities with the well-known approach of adding performance attributes to a system design model in order to generate an analytical predictive model. The similarity lies in transferring the SLA, which is a non-modelling and non-predictable oriented document, to an analytic model that represents it in order to assist in deciding the probability of compliance.

An example of transferring a design-oriented model into an analytic one is the work by Petriu [84] who created a unified intermediate meta-model that used the UML Profile for Schedulability, Performance and Time (SPT) [85] to produce an analytic model. The SPT UML extended UML, which was used to model the system's structure, by adding the ability to model time and performance related factors.

Other studies in the same area mapped a UML model to a specific stochastic model for the sake of performance analysis. For example, [86] mapped a UML model to a Stochastic Automata Network, while the work in [87] translated this model into a Stochastic Process Algebra model. A list of works related to the direct translation from a design model to an analytical one can be found in the survey paper in [78].

Another study that is much closer to the idea of the work presented in this thesis, which is using an intermediate generalised model, is the research in [88]. In this work, the author proposed the idea of mapping between a non-analysis oriented model and an analysis oriented one to allow for the early prediction of a system's performance. This was achieved by offering an intermediate language called '*KLAPER*', defined us-

ing Meta-Object Facility, to bridge the gap between these two models. This centred language reduced the cost of mapping N design models to M analysis models from $N.M$ into $N + M$ transformations to and from the KLAPER language. The author also implemented a tool that automatically maps [89] a UML model, representing the design-oriented model, to a Layered Queuing Network model, representing the analysis-oriented model.

Additional work which used a modular solution for mapping between design and analysis models is the work in [90]. In this work, the author created a transformation method based on an Intermediate Model (IM) to perform mapping at a meta-model level. This method used an annotated UML model, in its XML format, as an input and produced an analytic model, in its XML format, as an output by the use of a graph transformation method. The author implemented this method using LQN as a target model and the transformation techniques using XML algebra.

Since the work in this thesis considers using a stochastic model as a target formalism when mapping from an SLA, an overview of these stochastic models is presented in the next section.

2.5 Stochastic Modelling Formalisms

The likelihood of the occurrence of great many daily events is probabilistic making them described as stochastic processes. For this reason, these events are modelled using stochastic models which depend on the probabilistic theory. The *Stochastic Process* is a mathematical representation of the system with probabilistic or random characteristics. It models the system behaviour as a function of time's which could be continuous or discrete [30].

To specify the stochastic process formally, several definitions need to be introduced first. These are as follows. The *Random Experiment* is an experiment that may have one or more potential results (e.g. students' marks). The *Sample Space* of this experiment is a set of all potential results which could be finite or infinite (e.g. positive integers between 0 and 100). If a single result is obtained from the sample space (e.g. a student's mark is 62) then it is called a *Sample Point*. The *Random Variable* is a function identified over the sample space of an experiment and it gives a real number to each result from the sample space. An example of this is the random variable *Pass* which gives 0 (failed) for students whose marks are under 50 and 1 (passed) otherwise [91].

Given the aforementioned definitions, the *Stochastic Process* (or the *Random Process*) $\{X_t, t \in T\}$ is defined as a set of random variables sorted by a parameter,

from an indexed set, T, which mostly represents a time, t. Hence, X_t is said to be the current state of the system at time, t, and the *State Space*, S, of this process is the set of all the random variable values [91].

The state space, S, of a stochastic process can be *Discrete*, if the states can be counted by positive integers, or *Continuous* in the opposite case. Accordingly, the stochastic process is said to be a *Discrete-State Stochastic Process* (or a *Chain*) if its state space is discrete (e.g. the number of job arrivals), or a *Continuous-State Stochastic Process* if its state space is continuous (e.g. the waiting time of jobs to be served). Also, the index set, T, can be discrete if the process is examined in specific time instants, or continuous if the process is examined during an interval of time. Consequently, the stochastic process could be a *Discrete-Time Stochastic Process* if its time parameter is discrete (e.g. every hour of the day), or a *Continuous-Time Stochastic Process* if its time parameter is continuous (e.g. during the whole day). Bearing these types in mind, the stochastic process could be one of four types depending on the combination between state types and time types. These types are: (1) *Discrete Time Discrete State Space Stochastic Process* (e.g. the number of customers waiting in a shop every hour of the day); (2) *Continuous Time Discrete State Space Stochastic Process* (e.g. the number of customers waiting in a shop at any time of the day); (3) *Discrete Time Continuous State Space Stochastic Process* (e.g. the waiting time of customers arriving every hour of the day); and finally (4) *Continuous Time Continuous State Space Stochastic Process* (e.g. the waiting time of customers arriving at any time of the day) [91, 92, 93].

Most concrete stochastic processes show some kind of dependence between the states that have previously occurred, the current state and the future state. For example, the total gain of a person after n coin flips depends on the gain at the end of the $(n-1)$-th flip. However, the more complicated this dependency becomes, the more difficult the analysis of such systems. For this reason, processes with a dependence of the first-order are desirable [92]. There is a set of stochastic processes that exploits this specific kind of dependency of system states: this is called the Markov property. *Markov Property* or the *Memoryless Property* considers that the future state depends only on the present state; it is independent of the previous states or the time spent in the current state. In other words, the firing rate of system activities is exponentially distributed [30, 91]. The *Markov Process* is a stochastic process that satisfies the *Markov Property*, while in the *Semi Markov Process*, the sojourn time of the current state influences the next state: i.e. the memoryless property of the state's sojourn time is not valid [93].

As previously mentioned, the stochastic process describes the behaviour of the

system with states and activities that change them [94]. To facilitate the application of an analytical and numerical solution, these stochastic models often use a Markov or Semi-Markov chain which describes the system at a low level, providing all the states and transitions that the system may go through. However, high-level modelling formalisms, such as Stochastic Petri Nets (SPNs), are often used because of to the complexity of giving a full representation of every state and transition in a concrete system. These formalisms are then automatically transformed into the core Markov or Semi-Markov chain [28, 95].

In the next sub-sections, two modelling paradigms that are used to describe stochastic systems are reviewed. Markov and Semi-Markov models are not described because they are not used in this thesis. Only the paradigms that have been used in this work are described; these are the Stochastic Discrete Event System (SDES) and the Stochastic Petri Net (SPN).

2.5.1 Stochastic Discrete Event System (SDES)

In this thesis, the main concern relates to one important type of stochastic process, which is the stochastic discrete event system [96]. Stochastic discrete event systems can be in one state for some time before moving to another when an event occurs [96]. An example of such a system is a queuing system where a state is represented by the number of customers in a queue. This is changed once a new customer enters or an existing customer leaves [96].

One of the abstract paradigms used to describe stochastic discrete event systems is the Stochastic Discrete Event System (SDES) formalism. It is a general formalism in which well-known formalisms, such as Stochastic Petri Nets and Queuing Networks, can be expressed.

Definition 1 *A stochastic discrete event system is a tuple, SDES=*(SV, A, S, RV) *[29], where, SV is a set of state variables, A is a set of actions, S is a sort function* $S : SV \rightarrow \mathbb{S}$*, that gives all possible values of a state variable* $sv \in SV$ *(where* $\mathbb{S}$ *is the set of all possible sorts), and RV is a set of reward variables.*

An SDES is characterised by its state $\sigma \in \Sigma$, $\Sigma = \prod_{sv \in SV} S(sv)$, where Σ is the set of all theoretically possible SDES states (not all of them are necessarily reachable). An SDES moves between its reachable states through the execution of its actions. For the purpose of this thesis, the reward variable, $rv \in RV$, needs to be explained further:

Definition 2 *An SDES reward variable* $rv \in RV$ *is a tuple,* $rv = (rv_{rate}, rv_{imp}, rv_{int}, rv_{avg})$*, where,*

- $rv_{rate} : \Sigma \to \mathbb{R}$*: is a rate reward function that specifies the reward obtained while the system is in a specific state.*
- $rv_{imp} : A \to \mathbb{R}$*: is an impulse reward function that specifies the reward obtained when a specific action fires.*
- $rv_{int} = [lo, hi]$*: is an observation interval under consideration specified by the boundaries* $lo, hi \in \mathbb{R}^{0+} \cup \{\infty\}$ *and* $lo \leq hi$*. Hence* $lo = hi$ *implies an instant of time measure and* $lo < hi$ *an interval of time measure [97].*
- $rv_{avg} \in \mathbb{B}$ *is a boolean value specifying if the measures should be computed as an average over time (*$rv_{avg} = TRUE$*) or accumulated (*$rv_{avg} = FALSE$*).*

An SDES model is represented as a stochastic process, $SProc = \{\sigma(t), A(t), t \in \mathbb{R}^{0+}\}$, where $\sigma(t) \in \Sigma$ denotes the state at time t, and $A(t) \subset A$ is a set of actions executed at time t. Hence, the reward variable value at time instant t can be defined as follows:

$$R(t) = rv_{rate}(\sigma(t)) + \sum_{a \in A(t)} rv_{imp}(a)$$

In [29], this is written as:

$$rv = \begin{cases} \lim\limits_{t \to lo} R(t), & \text{if } lo = hi \text{ and } \neg rv_{avg} \\ \lim\limits_{x \to lo, y \to hi} \int_x^y R(t)\, dt, & \text{if } lo < hi \text{ and } \neg rv_{avg} \end{cases}$$

2.5.2 Stochastic Petri Nets

Petri Nets (PNs) are widely used for modelling systems with simultaneous and chronological transitions in order to obtain qualitative measures. They are also very effective in representing system concurrency and synchronisation. A Petri Net consists of a set of places, transitions, and arcs that connect them and its state is specified by the number of tokens stored in each place. Most Petri Net models are basic and have no time specification related to the activities or places of models [30, 98].

To allow the extraction of quantitative and time-related performance results, Stochastic Petri Nets (SPNs) were introduced by assigning exponentially distributed random functions to the delay of the Petri Net transitions. This exponential distribution allows the state of the modelled system to be changed in a probabilistic

manner. This permits the estimation of more cumulative performance results from the steady state distribution such as the average delay. Furthermore, the exponential distribution allows SPNs to resemble Continuous Time Markov Chains because both of them use the memoryless property of transition firing [30, 98].

SPNs offer a desirable combination of the graph modelling and probabilistic modelling which allow a system's behaviour to be analysed. This is due to an SPN's ability to give a visual description of a system process that is automatically transformed into the underlying Markov Chain model for performance analysis [98]. To extract performance measures, reward variables are defined at the network level. However, to be able to solve these models numerically in order to obtain performance results, they should be converted first to their equivalent underlying state-level stochastic process with the corresponding rewards specified at the state level [99].

One of the shortcomings of an SPN is its ability to model, mostly, small-sized systems. SPN graphs become very complicated when the system size expands, making the number of Markov states explode dramatically. Another weakness of SPNs is the need to associate an exponential distribution to each transition; this may not be desirable for transitions with low impact on the model. Removing the time from these kinds of transition may lead to a smaller number of Markov states and thus simplify performance extraction [98].

To overcome the aforementioned weakness of SPN, Generalized Stochastic Petri Nets (GSPNs) were created by introducing two kinds of transition: *Timed* with an exponentially-distributed delay function, and *Immediate* with zero time delay. Hence, a delay function is only related to timed transitions; it can be fixed or dependent on a place marking. Immediate transitions have a firing priority over timed transitions if they are both enabled. When multiple immediate transitions are enabled, they fire according to a probability distribution function [98].

GSPNs exploit all the characteristics of SPNs, from the accurate description of system operations to their correspondence to the Markovian models. However, its smaller reachability set (which is the set of all the marking that can be reached from the initial marking through launching a series of transitions) reduces the chaos of performance analysis much more than an SPN does [98].

In addition to places, timed/immediate transitions, and directed arcs, GSPNs also use inhibitor arcs. The inhibitor arc enables a transition to fire in such a way that is opposite to that of normal arcs. The transition cannot fire if the input place which is connected to the inhibitor arc contains tokens. This additional type of arc allows for a more flexible description of the system graph and reduces its size [98].

In GSPN, the reachable markings are divided into *Vanishing* and *Tangible*. The

Vanishing markings are the markings that enable one or more immediate transitions, thus producing firing delay equal to zero. However, *Tangible* markings only fire timed transitions yielding a time delay [100].

A GSPN model has different extensions. In the following subsections two of these extensions that are used in this thesis are described. These are the Stochastic Activity Network (SAN) and the Stochastic Reward Net (SRN).

2.5.2.1 Stochastic Activity Network

The Stochastic Activity Network (SAN) is one of the Petri Net's stochastic extensions; it allows the integration of time in the system model. It is also extended from the activity networks (which are non-probabilistic models) with added probabilistic nature. SAN models system behaviour at the network level and is used for performability analysis. Furthermore, it is similar to the discrete state Markov process and is employed in three modelling tools which are METASAN, UltraSAN and Möbius [101]. These tools simplify model construction and solving with an equipped analytic solver or simulation [102]. A SAN consists of many components connected by arcs which can be graphically depicted. These components are as follows [101]:

- Places: *Network marking* is defined as a vector; each of its elements represents the number of tokens in a specific place in the set of network places. Places are symbolised as circles with small dots representing the tokens.

- Activities: These can be either *timed* or *instantaneous* with a non-zero set of *cases*. *Timed activities* denote those activities whose delays influence the system's functionality. Each timed activity is assigned a time distribution function to specify the delay period. *Instantaneous* activities indicate those activities that have a tiny or negligible delay time. *Cases* are coupled with an activity which has a case distribution function to indicate which case will be selected. Activities are depicted as hollow ovals for timed activities and solid bars for instantaneous ones. In situations where cases are used, small circles are shown on the output side of the activity.

- Input gates: These have a bounded set of inputs, each of which is linked to a single place, while only one output is connected to a single activity. Each input gate has an enabling predicate to define the pre-conditions of an activity firing, and an input function to define the change in marking after the completion of an activity. An input gate is depicted as a triangle with its head pointing to the left.

- Output gates: These have only one input, which is connected to a single activity, while they are linked to a number of output places. Each output gate has an output function to specify the marking change in the network after an activity fires. An output gate is depicted as a triangle with its head pointing to the right.

SAN allows reward variables to be defined on the net level rather than on the state level, allowing for more natural definitions of the performance variables [97].

The SAN model has many strengths. It is simple and its primitives can be easily understood and graphically constructed. Also, it is based on an underlying mathematical core which allows it to be exposed by verification and analytic tools. Furthermore, it is very flexible due to its ability to describe the input/output functions of the input/output gates using the C programming language. Moreover, the SAN model is computational, allowing the functional examination of the model using SAN's nondeterministic settings (for the correctness of the model structure) and operational assessment using SAN's stochastic settings (performance analysis). Finally, the SAN model can be solved analytically or by using simulations; a Markov (or non-Markov) model can be derived for it [103].

The SAN formalism also has many shortcomings. Because of its simple primitives, it is relatively hard to construct a complex system using them, and there is no way to construct the system hierarchically from other sub-models. Furthermore, there is no technique to begin modelling the system using abstract primitives which could be substituted later with more comprehensive ones. Lastly, as in PN, SAN also has a rapidly growing reachable graph that makes its analysis very complex. For this reason, extensions for SAN have been developed which maintain the power of SAN and add some other desirable capabilities. Examples of these extensions are Hierarchical Stochastic Activity Networks (HSANs), Colored Stochastic Activity Networks (CSANs), and Object Stochastic Activity Networks (OSANs) [103].

2.5.2.2 Stochastic Reward Network

The Stochastic Reward Network (SRN) is a stochastic extension of GSPN allowing reward functions to be defined at the net level [104]. The SRN model has places, transitions and arcs that connect them. These transitions can fire either after a delay drawn from an exponential distribution or a zero time which result in a timed transition firing or an immediate transition firing respectively [31]. An enabling function can be assigned to a transition to enable/disable it; in addition, a transition priority relation can be specified to prioritise the firing of multiple enabled

transitions.

Other new features that have been added to the SRN model include: advanced guard functions, marking dependency, and priority specification. Marking dependency on a single or multiple places can be included in the firing rate, arc cardinality, or the reward function [104]. For example, the cardinality of an arc can be the number of tokens in an input place. If this number is zero, then the transition connected to it will be disabled and the arc is recognized as disappeared [104].

The SRN model has many strengths. Using this model, complex systems can be modelled in more compact way due to its marking dependency feature. In addition, since reward functions and firing rates are defined on the net level; they can be changed easily leaving the actual model structure intact [105]. Furthermore, SRN models have underlying Markov Reward Models (MRMs), allowing them to be analysed in order to derive performability characteristics. Moreover, an SRN model can be built and analysed using tools such as SPNP [31] and SHARPE [106]. These tools have been developed to simplify a model's construction and solving with an equipped analytic solver or simulation. The SRN model also has many shortcomings. The most important one is that it only defines transitions which are immediate or exponentially distributed [31]. This prevents a modeller from building a model with an underlying Semi-Markov Process.

Given that there are many formalisms that can be used to model a stochastic system, different performance, dependability, or performability attributes can also be specified on these models. Furthermore, different techniques can be adopted to derive measures of these attributes. Model attributes and their solving techniques are part of the work undertaken in this thesis. For this reason, they are reviewed in the next section.

2.6 Performance, Dependability, and Performability Models

In this section, the types of attribute that can be defined in a system model are presented, together with their classifications. Also, the reward models that are used in this thesis are defined. Then, the methods that are used for analysing a model are presented, along with the software tools that help in building and solving these models.

2.6.1 Attributes of a Model with their Classifications

The importance of identifying the behaviour of a stochastic system increases the need to develop techniques for assessing its performance, dependability, and performability attributes [94]. Definitions of these attributes, and an example of each of them, are presented in Table 2.3 and are described in what follows.

Table 2.3: Performance, dependability and performability attributes

System Attributes	Description	Example
Performance Attributes	It is related to system behaviours in a failure-free system	*Throughput*
Dependability Attributes	It is related to the infrastructure modification due to temporal deficiency	*Availability*
Performability Attributes	It is related to system behaviour with an existence of a failure	*Average response time given a fault presence*

Performance Attributes describe measures relating to a user's work flow and the behaviours of other users or systems in the case of a failure-free system. On the other hand, **Dependability Attributes** describe measures concerning underlying infrastructure modifications due to a temporary or permanent deficiency [107]. Given these two kinds of measure, dependability modelling and performance modelling techniques do exist where a pure *'Dependability Model'* usually models a system's behaviour related to up/down, fail/repair and reconfiguration characteristics in order to derive measures related to *Availability*, *Safety* and *Reliability* [2]. However, a pure *'Performance Model'* usually models a system's behaviour under a breakdown-free assumption to derive performance measures related to *Response Time*, *Throughput*, and *Utilization* [2].

In recent years, *'Performability Model'* has been presented as a term to describe the integration between system performance and system dependability assessments which largely reflect the effectiveness of a system [107]. This is because a pure performance or dependability model cannot reflect the behaviour of the actual system and hence can produce unrealistic results. For this reason, **Performability Attributes** are usually related to a system's behaviour when there is a failure to derive measures such as average response time given the presence of a fault [2].

A model of a system can be queried for two different kinds of performance query and the distinction is made depending on the type of result that is derived from these queries. Table 2.4 provides a comparison of the two types of performance query that can be produced from stochastic models. This is described in what follows:

Table 2.4: A comparison between types of performance query

Performance Queries	Answer	Language used To Express them	Verified
Performance Requirements	Boolean answer Yes/No	CSL, CSRL, Performance Trees	Model checker such as PRISM, PIPE2 tool
Performance Measures	Quantitative answer	Tools based reward representation, Performance Trees	Analysing tools such as SPNP, PIPE2 tool

Performance Requirements give a Boolean answer to logical performance questions such as *"s the response time less than 15 seconds for 80 % of the requests?"* and **Performance Measures** give a quantitative answer to questions such as *"What is the average down time of the system in a steady state?"* [28].

Performance requirement questions can be identified using several formalisms such as CSL (Continuous Stochastic Logic) [108], CSRL (Continuous Stochastic Reward Logic) [109], aCSL (Action-based Continuous Stochastic Logic), eCSL (Extended Continuous Stochastic Logic). Performance measures, however, can be identified using a tool-based reward representation of an analysis framework such as SPNP (Stochastic Petri Net Package), SHARPE or Möbius. Both of these can be specified by Performance Trees [28, 95].

To resolve performance queries, model checking tools, such as PRISM or MRMC (Markov Reward Model Checker), are used to give results for performance requirements, while analysis tools, such as SHARPE or Möbius, give quantitative performance measures [28].

It should be noted that the Performance Tree formalism can represent the two types of system performance query. This has many advantages over Stochastic Logic which was the primary method used to specify performance questions. The first strength of a Performance Tree lies in its simplicity and this is because it contains many operational and value nodes that can be gathered visually in a hierarchical tree which means that difficult performance queries can be easily constructed. It is also extensible either by identifying new nodes or using a macro to create a representation of a complex operation from existing nodes [95]. Furthermore, it expresses a wider range of queries than other formalisms. In addition, it is general because it is specified using abstract states which allow it to be independent from the core system modelling formalism. The greatest important feature of a Performance Tree is its ability to give logical answers to the performance questions, in addition to providing quantitative results with regard to the system performance measures [28, 95]. Performance queries formulated using Performance Trees can be created

and evaluated using the PIPE2 tool [110].

Performance queries can also be classified according to the time they are define for [28, 111] as follows.

Transient Queries: are used to derive the probability, at an instant of time t, that a system occupies a state, or a set of states [28]. An example of this is: *"What is the probability that the system is in an up state at time instant 12?"*. Transient queries are normally used to derive a probability, a state, or true/false answer [28, 111].

Steady-State Queries: are used to derive the probability, in the long run, that a system resides in a state, or a set of states [28, 111]. An example of this is: *"What is the steady-state probability that the system is staying in an up state?"*. Steady-State queries are normally used to derive a probability, a state, or an average action rate [28, 111].

Passage Time Queries: are used to derive the time taken to reach a specific state starting from a predefined one [28, 111]. An example of this is: *"What is the average time until the system reaches a state Exit, given that it started from a state Enter?"* These queries are mainly use for computing response time measures.

2.6.2 Reward Models

The model attributes, described in Section 2.6.1, are reflected in the model using the notion of *reward* [94]. These reward-based attributes depend on using *Reward Models* which are structured from a *stochastic process*, a *reward structure*, and a *performance variable* applied on that structure, as stated in [94]. These are discussed in what follows in more detail:

The *Stochastic Process:* This describes the system's behaviour. It is already defined in Section 2.5.

The *Reward Structure:* After building the model of the system, a reward structure (or what is called a *Reward Function*) is used to define the reward attributes of interest in the state space of this process. This is achieved by applying two functions: the *Rate* function, which gives the accumulated rewards depending on the time spent in a specific state of the system, and the *Impulse* function, which gives the accumulated rewards when a specific action fires [94]. Rewards can be understood as cost; thus, attributes such as accumulated rewards under a specific threshold can be identified.

The Performance Variable: Since the reward structure (reward function) does not identify the period of time within which the reward will be computed, the performance variable (or *Reward Variable*) is used to identify it. This variable could

be an *Instant of Time*, *Interval of Time* or *Timed-Averaged Interval of Time* [94].

The *Instant of Time* reward variable is used to derive its value at the instant of time t. This time, t, can go to infinity which results in two types of the *Instant of Time* reward variable [97]. The *Interval of Time* reward variable is used to derive the value accumulated during an interval of time $[t, a]$. In this time interval, either t or a can go to infinity. This results in three types of *Interval of Time* reward variable. Finally, the *Timed-Averaged Interval of Time* reward variable is used to derive its accumulated value averaged during an interval of time $[t, a]$. In this time interval, either t or a can go to infinity as in the previous case.

As an example of a reward function, assume there is a printer with two states (*up* and *down*) and two transitions ($fail$, $repair$) and the modeller wants to count the number of times the printer fails, and the time spent in the down state. To do so, a reward (real number) is associated to the system state or transition, according to what is required. For the first attribute, a reward of 1 can be associated with the transition $fail$, and each time it fires, an accumulation will be computed, thus counting the failure times. In the second example, the state *up* can be equipped with reward of 1 so it is accumulated whenever the system spends time in this state.

Some stochastic models, such as the Markov Reward Model (MRM) and Stochastic Reward Nets (SRN), are equipped with the ability to define rewards. This is accomplished by adding rewards to the equivalent non-reward models which are the *Continuous Time Markov Chain* and the *Generalized Stochastic Petri Net* (GSPN) respectively. These reward models are better suited to system description and can be used to represent measures of new types [112].

2.6.3 Methods of Model Analysis

Many techniques can be used to solve a reward model, employing either a *simulation* or a *numerical solver*. These are described in the following sub-sections.

2.6.3.1 Numerical Solver

A numerical solver of a model depends on its underlying state space, or what is called a reachability graph, in order to derive the value of the model attributes [32]. Numerical solvers are chosen according to the type of model and the type of attribute. The type of model is determined by the firing distribution of its actions, while the type of attribute can be determined as: (1) solved as in a steady state or transient; (2) defined at a specific instant or during an interval of time; or (3) measured using the mean or the variance [32].

The results obtained from a numerical solver are exact. However, numerical solvers are not typically applicable for non-Markovian models. Only models with an immediate or exponential firing distribution of its actions can be solved [113]. Another problem regarding the numerical solvers concerns state space explosion; because of this, these are only useful for small-sized models.

2.6.3.2 Simulation

A simulation of a model depends on producing a set of possible trajectories this model can pass through and then deriving model attributes from statistics applied to them [114]. A single trajectory reflects a single possible behaviour of the system; hence, multiple trajectories are required to reflect true system behaviour.

A simulation can be a discrete event or a continuous state. Discrete event simulation is used for systems in which states are changed according to discrete time instants. There are two types of discrete event simulation: *terminating* and *steady state* simulations. The former is used to solve the model for transient measures, including instant of time or interval of time measures, while the latter is used for steady state measures solved to infinity [114].

For transient analysis, discrete event simulation uses the independent replication technique while for steady-state analysis, batch means are used. Estimations of the statistics obtained from using these techniques include the mean, variance, and distributions for specific confidence intervals [32], where the simulation runs as many batches/replicas as necessary until reward variable results converge to a pre-defined confidence interval width.

Using simulation entails utilising a smaller memory space than with a numerical solver since no reachability graphs are generated [113]. However, the results from simulation are not as accurate as with a numerical solver as they are estimated within specific confidence bounds. In addition, simulation is time consuming.

2.6.4 Software Tools for Building and Solving Models

Several software tools have been introduced to help in carrying out all the steps from building the model, assigning attributes of interest, then solving the model. Since this thesis concentrates on stochastic models, and SPN in particular, the review that follows relates to some of the tools that solve SPN models. However, the largest proportion of the section below is devoted to the first two tools under review as they are the ones used for the tool implementation in Chapter 6. It should be noted that a comparison of the tools is not conducted in this section. Instead, Section

6.3.2.2, and Table 6.2 in particular, compares them in the light of the requirements of the proposed WslaCP tool. The reason for deferring the comparison until Section 6.3.2.2 is that the WslaCP tool requirements are only specified in Sections 6.2.3.4 and 6.3.2.1.

2.6.4.1 Möbius

The Möbius tool is used in modelling the performance of a wide range of discrete state computing systems. It is a framework that comprises multiple modelling formalisms (SAN, Bucket and Balls, PEPA, Fault tree) and multiple solution methods (simulation, numerical solvers). Many of these methods are independent of the modelling formalism being used and hence these can be employed in combination with each other [102, 115].

Möbius allows a single model to be built using multiple modelling formalisms [115]. Once a model has been constructed using supported modelling formalism components, it is converted into a model that is specified using Möbius framework components. This allows the tool to be extended by adding new modelling or solution formalisms [102]. The different parts of the model communicate using the Abstract Functional Interface (AFI) which is a group of C++ functions that allow interaction between the different models and solvers [115].

To measure the attributes of a given system using the SAN formalism in Möbius, these steps should be followed [32]:

- Building an *atomic model* that depicts all the relevant system states (using tokens in the simple or extended places); the state changes that occur through actions (using timed or instantaneous activities), which fire according to a distribution rate (deterministic, exponential, etc.); and the input/output gates that may be used to define the enabling predicate of an activity or the marking change in a place.

- Creating the *composite model* if necessary. When the atomic model is a part of a larger model, the modeller can compose different atomic models using one of the composition formalisms, such as the Replicate/Join composition formalism, graph composition formalisms, or synchronisation on actions.

- Building the *reward model* by associating the rewards of interest (as performance variables) from which metrics will be computed with the atomic or the composed model. There are two kinds of reward: rate rewards, which represent the time spent in each state (place), and impulse rewards, which count

activity completions. Each reward variable has a reward function that computes its value, and a time that specifies when the reward function should be evaluated. The reward variables are defined on the net level.

- Specifying *studies* on the model. Sometimes, *global variables* can be used when constructing the atomic, composed, or reward models. These variables are assigned to state markings, activity rates, functions, etc. without assigning any value to them. The model cannot be solved unless each of these variables receives a value; this is called an *experiment*. The set of all experiments for a specific model is called a *study*.

- Solving the model by either by *discrete event simulation* or *analytical numerical solvers* in order to derive transient or steady state measures. The result may be the mean, the variance, or the distribution of the reward variables. In the case of using a numerical solver, the *State Space generator* should be used first to produce the state space for the underlying Markov Process of the modelled system whose activities are exponentially distributed [115].

- Creating a *connected model* for a set of reward models and their equivalent solvers when the input of one of them depends on the result from the preceding model.

Each of the aforementioned models and each modelling or solving formalism has its own separate editor interface in the Möbius tool. When new modelling, compositions, rewards, solving or connecting formalisms are added to the tool, a new editor will be integrated without any changes being made to the remaining editors [102].

2.6.4.2 SPNP

The Stochastic Petri Net Package (SPNP) is a modelling tool that is used for analysing the performance, dependability and performability of the system model. It is used for building and solving Stochastic Petri Net (SPN) Reward Models, especially the Stochastic Reward Net (SRN) with the underlying Markov Reward Models (MRM).

SPNP allows reward rates to be defined at the net level. It can be used to obtain transient, steady-state, cumulative, and time-averaged measures using an analytic model or discrete event simulation. Non-Markovian SPN models can be also defined using SPNP; however, they can only be solved using discrete event simulation.

SPNP has a graphical and a textual input which uses iSPN and SCPL respectively. CSPL is a language which is a subset of the C programming language with extra constructs for defining the model primitives [116]. The iSPN interface has a set of GUIs to facilitate creating and solving the model. Some of these GUIs are:

- *Petri Net editor*: to build the SRN model graphically.
- *Function definition GUI*: to create the reward, guard, distribution, arc cardinality, and probability functions.
- *Environment GUI*: to choose the solver type, whether it is numerical or simulation. From the same GUI, the analysis option (i.e. steady state or transient) can be specified, in addition to all parameters required by the solver.
- *Analysis frame*: to define the time used to solve the reward variables. From the same GUI, the solver can be run and the results are depicted.
- *Animation GUI*: the iSPN also has an animation GUI that allows the modeller to visualise how the tokens are moved in the model.

2.6.4.3 PIPE

The Platform Independent Petri net Editor (PIPE) [110] is an open source Petri net modelling tool that was extended to allow users not only to model a system using a GSPN formalism, but also to identify queries on it and solve them [117]. After building the model and identifying the performance query of interest using the PIPE front-end user interfaces, they are both converted into XML files and are then sent to the *Analysis Server* for assessment. A single query may consist of many sub-queries that need to be evaluated before the main query can be assessed. For this reason, the query is decomposed into its sub-queries according to their dependencies. The analysing server controls many distributed analysing tools, and is used to allocate each derived query to a suitable analysing tool after transforming the XML files to an input type appropriate for that tool. These tools are: DNAmaca, which is dedicated to solving queries of steady state probability, the mean rate of a transition firing, or statistical analysis; SMARTA, which is used to calculate passage time density and distribution; and MOMA which is used to compute the raw moments. The analysing server then collects the results of the performance queries, gathers them in an ordered manner, and then sends the result back to the client [117]. PIPE is the first tool that provides an embedded performance tree [95] editor.

2.6.4.4 SHARPE

The Symbolic Hierarchical Automated Reliability and Performance Evaluator tool (SHARPE) is a modelling tool that is used for analysing the performance, dependability and performability of a system model [106]. SHARPE is used for building models using different formalisms, including the Generalized Stochastic Petri Net (GSPN). It also models Markov and Semi-Markov chains in very compact way because it allows for a hierarchical composition of the model using different formalisms.

SHARPE has both graphical and textual inputs. The textual input is based on SHARPE's own language that follows MRM enumeration [118]. Reward rates are inserted at a state level by enumerating each state transition and the reward given for each of them [118]. SHARPE can solve the model using only an analytic-numeric solver that solves either state space or non-state space models [113].

As in SPNP, SHARPE also has a Graphical User Interface that allows the modeller to create models easily. It also allows the user to plot the generated results or export them into Excel spreadsheets.

2.6.4.5 GreatSPN

The GRaphical Editor and Analyser for Timed and Stochastic Petri Nets (GreatSPN) is a software tool used for building, validating and analysing the system model [119]. It runs on the Unix operating system. The model is built using either a Generalized Stochastic Petri Net (GSPN) and its coloured extensions, or a Stochastic Well-formed Net (SWN) [120].

GreatSPN version 1.3 has a non-event driven based simulation and was originally mainly devoted to steady state analysing [121]. This was solved in later versions and a simulation based on the Natural Regeneration method has since been utilised [119]. Steady state performance measures can be obtained using batch means simulation [122].

GreatSPN has no common rate and impulse reward definition. Instead, the user can define performance results (or performance indices) which have limited expressive power [31].

2.7 Conclusion

This chapter has provided background information regarding important aspects related to the contribution made by this thesis. These aspects include: Service Oriented Computing and Web Services, SLA compliance management and its types,

stochastic modelling and stochastic Petri Nets, reward models and their metrics, analysis techniques and tools. Furthermore, the chapter examines some related works in the literature that: firstly, adopt a model-based approach for predicting SLA compliance; secondly, examine mapping between source and target formalisms; and finally, transfer a design-oriented model into an analysis-oriented one. The information in this chapter forms the foundation of the proposed methodology described in the next chapter.

Chapter 3

SlaCP Methodology for SLA Compliance Prediction

This chapter describes SlaCP, the novel engineering methodology proposed in this thesis to predict SLA compliance probability. The value of this methodology is to help its users, who may have a basic knowledge of model-based analysis and methodologies, to predict the compliance of their existing SLA contract without having to gain a thorough understanding of reward metrics and analysis tools. Another advantage of this methodology is that it can help SLA engineers to assess how parameterising the service model might affect SLA compliance probability, thus allowing them to take better decisions regarding the choice of SLA thresholds or the service's infrastructure.

The main contribution of this chapter is illustrating the design of the SlaCP methodology to allow a clear understanding of its constituting phases that are used to address the aim of the methodology. The chapter introduces this design of the methodology in its generic form from the perspectives of both users and tool designers. The latter view helps in setting the guidelines to design and develop a software tool with SLA compliance prediction capability in a way that automatically addresses the view of the former. The generic design with its first perspective represents the theoretical basis of the methodology while the second perspective represents its practical aspect. The chapter also briefly outlines the application of this methodology for a particular SLA and model specification to demonstrate its implementation. This outline forms the basis for the detailed description of the methodology implementation which is presented in Chapters 4 and 5.

The remainder of this chapter is structured as follows: Section 3.1 provides a preliminary outline of the proposed methodology while Section 3.2 describes the design of methodology in its generic form, then it presents this design from two

perspectives: a user utilising this methodology, and a tool designer employing it in a software tool in order to automate it. Section 3.3 establishes an outline of how this methodology is implemented for a specific type of SLA and a particular unified abstract stochastic model. Finally, Section 3.4 concludes the chapter.

3.1 SlaCP Methodology: Preliminary Information

Before describing the proposed SlaCP methodology, it is necessary to present the information that will help in its design. For this reason, in this section, the users the methodology is intended for are identified, and the requirements along with the characteristics that should be considered for inclusion in the proposed methodology are described. Finally, the methodology's assumptions are presented.

3.1.1 The Targeted Users of the Methodology

Predicting SLA compliance can influence the selection of SLO thresholds, the choice of QoS metrics, the adoption of an alternative infrastructure or different design of the service, and so on. Any professional who is directly responsible for making a decision regarding any of the aforementioned issues is a potential user of the proposed methodology. This might be an SLA engineer, service provider, service engineer, or a modeller. This choice of users can be justified for the following reason.

It is a complicated task for an SLA engineer if he/she is the person responsible for dealing with the challenges related to model-based SLA compliance prediction while, at the same time, he/she is trying to design the SLA that includes an appropriate SLO threshold. This is because an SLA engineer might be unfamiliar with the method used to create an adequate model or to insert correctly the QoS metrics required by an SLA. Because of this difficulty, this task is delegated most often to a modeller. The modeller has to carry out extensive work with regard to this task each time an SLA or any service parameter has been changed. For this reason, the methodology proposed in this thesis is intended to be used primarily by an SLA engineer (SLA designer) in addition to the service provider. This is because they are the ones who are directly responsible for making the final decisions about different SLO thresholds and different infrastructure design choices. A modeller can also use it to reduce the efforts involved in SLA compliance prediction.

3.1.2 Requirements of the Methodology

The requirements for the methodology to perform the SLA compliance prediction lie in two areas: theoretical feasibility to demonstrate that it is possible in principle, and practical feasibility to demonstrate its automation in particular so that the methodology is helpful in the real world. These are described in what follows.

1. Theoretical feasibility: To discover if the methodology is possible in principle, its theoretical basis has to be described; this will illustrate the proposed methodology's ability to produce the probability of SLA compliance. Describing this theoretical basis will help in demonstrating the feasibility of the new methodology, as well as aiding in the recognition of those aspects that could be automated. This theoretical description of the methodology has to be presented in terms of its design, and from the point of view of a user of this design. This will help a reader to understand the details that are either hidden or shown to users of the methodology.

2. Practical feasibility: The theoretical basis of this methodology has to be utilised in a way that is most useful for its users. For this reason, and to address the practical feasibility of the methodology regarding automation in particular, a design for a software tool that employs the theoretical basis needs to be created. Moreover, presenting the methodology from the point of view of a tool designer who utilises this methodology will help in implementing a tool that automates the process of SLA compliance prediction. Using such a tool allows for minimum interaction from a user. To guarantee such minimum interference, the tool has to employ automatic mapping of the SLA elements onto the metrics of a stochastic model of the underlying service.

According to the previously mentioned requirements, the SlaCP engineering methodology proposed in this thesis need to be presented theoretically, by giving its design with its user perspective, and practically by considering it from the point of view of a tool designer.

3.1.3 Characteristics of the Methodology

Model-based SLA compliance prediction has many challenges: these include defining an appropriate service model and choosing the desired QoS metrics to be predicted [25]. Knowing these metrics, there is then the challenge of clearly understanding their semantics and expressing them in such a way that will fit with the stochastic model. The degree of accurate SLA compliance probability which is predicted

using models depends heavily on overcoming the aforementioned challenges. Given these challenges, and the gaps in the current research on SLA compliance prediction presented in Chapter 1, four characteristics have to be considered while designing the SlaCP methodology. These characteristics are first outlined in what follows and then the motivation for considering them is described.

1. Considering the SLA used by service engineers as a starting point.
2. Considering different types of QoS metrics to achieve as wide a usability as possible for the methodology.
3. Considering a generalised stochastic model to achieve a higher level of flexibility.
4. Considering automation to achieve minimum user interaction.

The motivation for the first characteristic, (i.e. *considering the SLA used by service engineers as a starting point*) is that SLA engineers may have already created their SLA using a particular SLA specification; thus, there is a need to use this SLA without forcing them to use another SLA specification that is specific to the methodology. In short, there is a need to start with what the SLA engineers have already achieved. In addition, using an SLA as a starting point allows a formal relationship to be formed between the SLA of a service and its model so the translation between them can be accomplished automatically. This will help in deciding the exact QoS metrics in the SLA that need to be predicted without relying on the modeller's perception to do so. It will also help in predicting the compliance of legacy contracts with their exact QoS metrics, as well as the precise temporal constraints. Finally, using an SLA as the starting point of the methodology also helps in automating it as the necessary information can be extracted automatically without human interference. (This characteristic addresses the first and fourth research problems described in Section 1.2).

Regarding the second characteristic (*the consideration of different types of QoS metrics to achieve as wide a usability as possible for the methodology*), this methodology must consider a wider range of QoS metrics whether they are basic or composite. This is useful for addressing different metrics types defined inside the SLA. (This characteristic addresses the second problem described in Section 1.2)

Regarding the third characteristic (*considering a generalised stochastic model to achieve a higher level of flexibility*), the proposed methodology has to consider a generalised stochastic modelling formalism rather than a specific one. This is to avoid putting constraints on a user who may lack awareness of a specific type

of modelling formalism. Using a generalised model, a simple translation can be conducted to produce a model according to the desired modelling formalism. (This characteristic addresses the third problem described in Section 1.2)

Finally, the rationale for the fourth characteristic (*considering automation to achieve minimum user interaction*), is that automating the prediction methodology is one characteristic that makes it appealing to use, as set in the second requirement of Section 3.1.2. This is because automation will allow the extraction of the relevant information which is necessary to perform the prediction process without the need to obtain and assign it manually. Hence, an all-in-one tool that takes the SLA as an input and produces its satisfaction probability can be achieved. (This characteristic addresses the fifth problem described in Section 1.2)

3.1.4 Assumptions of the Methodology

The assumptions which were made before designing the methodology relate to the SLA contract and the stochastic model. Regarding the former, the SLA contract is supposed to be syntactically valid and pre-defined earlier; for the latter, the stochastic behavioural model of the service is assumed to be complete and ready after the mapping process has been completed and before it has been solved. This can result from automatic mapping (which is part of the methodology), or as a result of a user building it manually.

3.2 SlaCP Methodology

In this section, the design of the SLA Compliance Prediction (SlaCP) methodology is described. Then it is presented from the perspectives of its users (a theoretical view) and from the perspective of a tool designer (a practical view). This allows a reader to understand the phases that the methodology passes through, as well as which of them can be automated (i.e. hidden from users) and which cannot (i.e. they require user interaction). The largest share of this section is devoted to the methodology design because it forms the basis of the content of Chapters 4 and 5, while the user and the tool designer's perspectives are only outlined in this section. This is because, extensive details regarding the tool's design are offered in Chapter 6 in order to link with its implementation which is described in the same chapter.

3.2.1 The Design of the SlaCP Methodology

The design of the SlaCP methodology is given in Figure 3.1. In this figure, seven consecutive phases, depicted as rounded gray rectangles, must be employed to achieve the purpose of the proposed methodology. In the top left corner of this figure, an **SLA Contract**, represented as folded document, is proposed to be used as input to this design. The seven methodology phases of the SlaCP design flow from left to right; these are explained in what follows:

Phase 1: SLA Interpretation: The SLA contract created for a particular service has to be used as an input to this phase. This SLA should be parsed to obtain the information that is useful only for SLA compliance prediction. This information concerns the SLO's definition and the different QoS metrics that pertain to this SLO. The parser should ignore any information that is not relevant, such as the definitions of related parties and their roles, corrective actions in the case of violating an SLA, and measurement mechanisms that are used to retrieve measured

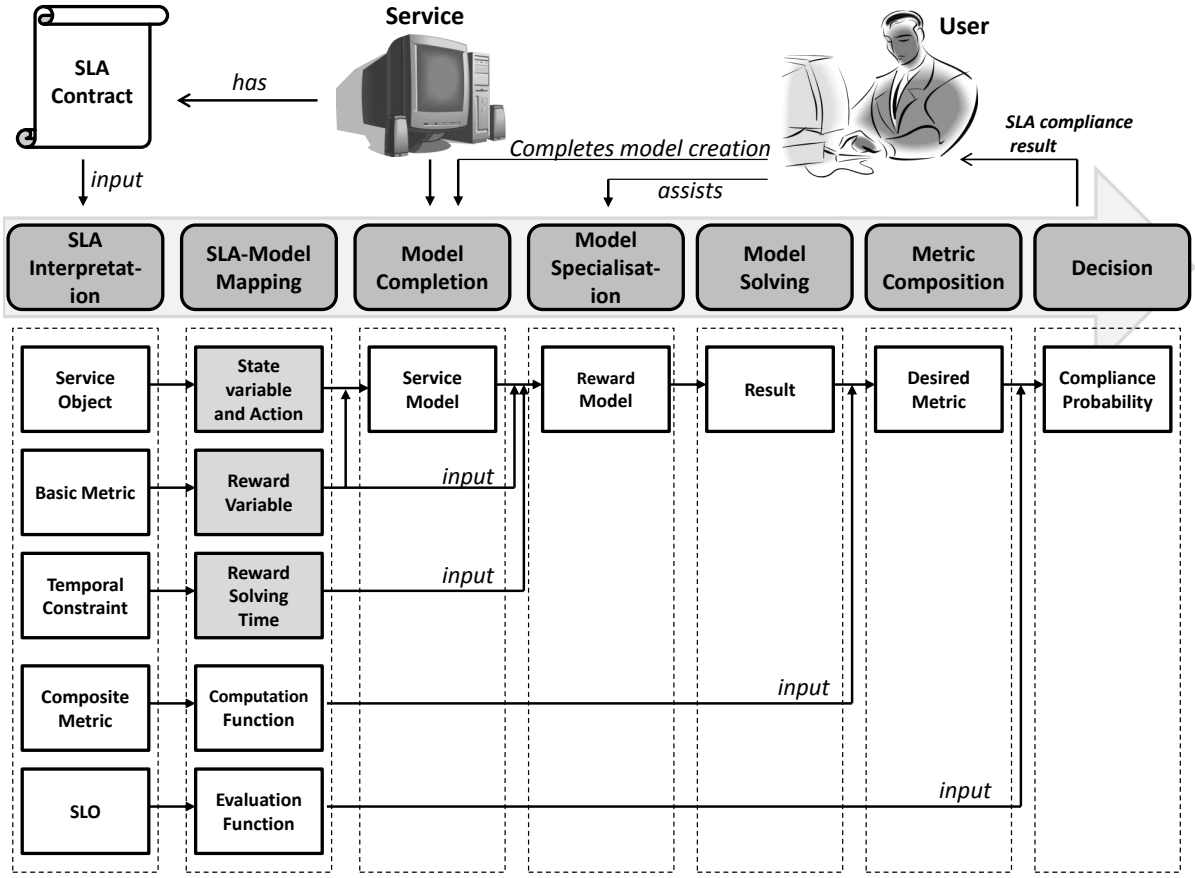

Figure 3.1: The design of the SlaCP methodology

metrics while monitoring the running service.

The SLA Interpretation phase should have five outcomes: a set of service objects, basic metrics, temporal constraints, composite metrics, and the SLO. These outcomes, represented as white rectangles in Figure 3.1, are described in more detail below:

1. **Service Object**: In each SLA, the desired QoS metrics are defined for a particular object of the service (e.g. an operation). Obtaining the object's name is necessary in order to distinguish the QoS metrics defined for it from the ones that are defined for other objects i.e. several QoS metrics that have exactly the same semantics might be defined for different service objects in the same SLA. For example, in one SLA, QoS metrics that represent the throughput of incoming requests and also out-going responses might be defined. Furthermore, having a set of service objects should help in realising some state variables/actions the stochastic model of the specified service has to include in the second phase.

2. **Basic Metric** (measured metrics): The basic metrics are usually read directly from a URI (or other measurement directives) because they are measured during service runtime. Since the values of these metrics are not known before the service's deployment, and cannot be measured during an off-line mode of a service, the value of them has to be predicted. Basic metrics are the primary unit for producing the value of the desired metric used by the SLO; hence, their value has to be obtained first. Obtaining the type of basic metric is vital if they are to be mapped correctly in the second phase. One example of a type of a basic metric is response time or throughput.

3. **Temporal Constraint**: A temporal constraint in an SLA can refer to two types of time constraint. The first is related to the period when a specific SLO is valid. The other refers to the time intervals during a period when basic metrics have to be measured to obtain composite metrics (in case the QoS metric required by an SLO is a composite one). The total number of intervals in a time period determines how many instances of the basic metric have to be obtained before computing one instance of the composite metric. For this reason, and to produce accurate results for the composite metrics, the SLA has to be examined to extract these time intervals and periods. An example of a time constraint is specifying a minute interval in a week period to check an operation's response time in order to produce its average response time later.

4. **Composite Metric**: The QoS metrics used within an SLA can be composite. This means that extra computation is carried out on different instances of a basic metric to produce the composite metric required. This phase has to investigate what types of computation and functions are performed on a basic metric in order to derive the composite one. Extracting this information is necessary to produce the exact value of the desired QoS metric so that it can be compared to the SLO threshold. An example of the types of function used in a composite metric are Maximum and Average.

5. **SLO**: This SLO threshold is the numeric value that specifies the limit that the desired QoS metric should maintain during a specific time period. The phase has to obtain this value along with the binary relation ($=$,$\leq$,$\geq$,$>$,$<$) through which the comparison is defined. An example of this is an SLO that must maintain, at most, a three-second response time through the day (i.e. less than 3 seconds).

After extracting all the aforementioned outputs from an SLA, there is sometimes a need to formalise them and define their semantics mathematically. The reason for formalising and mathematically interpreting SLA elements is that some SLAs describe their elements in a textual, descriptive format. According to this, the precise meaning and interpretation of the semantics can be misleading and may differ between service provider and customer. To avoid any ambiguity in an SLA's syntax and semantics, this phase should assign a mathematical representation of each of the extracted SLA elements. In doing this, the mapping also becomes more precise since it depends on a rigid mathematical basis.

Phase 2: SLA-Model Mapping: This is the core of the SlaCP methodology. The purpose of this phase is to draw a correlation between the SLA of a service and its stochastic model in a way that a result for SLA compliance prediction can be derived. This has to be done by taking the outputs of the SLA Interpretation phase and mapping them to related fragments of the stochastic model. Nevertheless, some outputs cannot be mapped on the actual model. Instead, they must be mapped on the outputs of solving the model; this is described later in Section 5.2.4. The rationale for this mapping, is to use the stochastic model, which represents the SLA's content, to predict the information that is unknown.

The SLA-Model Mapping phase must has five outcomes corresponding to SLA Interpretation phase outcomes; these are, respectively, sets of: state variables and actions, impulse/rate reward variables, reward solving time instants/intervals, com-

Table 3.1: An example of the outcomes of the SLA-Model Mapping phase

SLA Elements from Phase 1	**Example**	**SLA-Model Mapping phase outcomes**
Service Object	getQuote	An action, a, connected to an input state variable, sv.
Basic Metric	Throughput	An interval-of-time impulse reward variable, R^a_{imp}, that gives the number of firing of an action, a: $R^a_{imp}(t) = \begin{cases} 1 & \text{if } a \text{ fires} \\ 0 & otherwise \end{cases}$
Temporal constraint	Each hour in a day	Accumulated impulse reward assessed during each hour for 24 hours, $T = \{1 \dots 24\}$: $Acc_{R^a_{imp}}(t) = \Sigma_{t \in T} R^a_{imp}(t)$
Composite Metric	Maximum	Apply Maximum function on solver output: $max = Max\{Acc_{R^a_{imp}}(t_1) \dots Acc_{R^a_{imp}}(t_n)\}_{t_1,t_n \in T}$
SLO threshold	At most 100	A function to compare the result to the specified threshold: $f(max, <, 100) : max < 100$

putation functions of the model's output, and evaluation functions with SLO thresholds. Recalling Figure 3.1, the light gray rectangles below the SLA-Model Mapping phase represent the elements that should be mapped on the stochastic model itself, while the white rectangles represent the ones should be mapped to the outcomes of the solved model. To illustrate the mapping, Table 3.1 provides an example of the outcomes of this phase given the outcomes of phase 1. This example is related to a stock quote service whose provider promises in its SLA that the maximum throughput a *getQuote* operation is able to deal with is, at most, 100 requests each hour in the day. The phase outcomes, along with an illustrative example, are described in more detail as follows:

1. A **Service Object** has to be mapped as an **Action** of a stochastic model with an input **State Variable**. In the example provided, the mapping of a service object (i.e. a *getQuote* operation) must be as a state variable, sv, connected to an action, a. The reason for mapping the service object, as described earlier, is that the service object usually needs time to serve the request and this is reflected in the firing time of an action. Also, since the requests might arrive at a rate higher than the servicing rate, a state variable has to be included to reflect such queuing behaviour.

2. A **Basic Metric** refers to a service QoS attribute whose value is unknown before deploying the service. For this reason, it has to be mapped as a **Reward Variable** of a stochastic model (as in the performance modelling prediction approach). Hence, this phase should relate the semantics of basic metrics in

an SLA to equivalent reward variables represented in the format of a general stochastic model. For each reward variable, this phase has to specify according to the definition of each basic metric, the reward variable type, impulse or rate; the time to be taken (i.e. instant or interval); and the reward function that utilises the state variable or the action generated from the mapping of the service object. However, the utilisation of these generated model primitives might be inappropriate for the user, as will be described in Section 5.2.2, and different stochastic model primitives are then preferred. For this reason, the reward functions may be considered only as templates that provide an abstraction of either a state variable or action. Hence, the actual assignment to a concrete model primitive has to be the responsibility of the Model Specialisation phase which is described later. In the example in Table 3.1, the basic metric, throughput, is considered as an impulse reward variable with a reward function that assigns a value of 1 for each firing of an action, a. If a different action is required, this can be changed later. This reward variable should be considered as an interval of time variable in order to keep track of each incrementation in request numbers. The mapping of different basic metrics is described in Chapter 5.

3. A **Temporal Constraint** is used to sample the value of a basic metric at different intervals within a period, and is mapped as the **Reward Solving Time** which is the time required for solving a reward variable. The time interval is mapped as the instant/interval at which basic metrics are sampled, while the time period is mapped as the total time during which these samples are taken. In the example in Table 3.1, the time is mapped as solving the reward variable during the interval of one hour for 24 hours. The temporal constraint related to the SLO validity period is usually the same as the one used for evaluating the basic metrics. For this reason, it is not considered here. This is described in Section 5.2.5.

4. A **Composite Metric** is based on sampling the values of a basic metric over a period of time; these are then either aggregated or have another function applied to them. For this reason, this metric should be mapped as a **Computation Function**. These mathematical functions should not be mapped on the model itself but on the results of solving the reward variable at the specified time which represents their input. Hence, these functions should be tailored to the nature of the output results (expected or distribution values) rather than the nature of the monitoring results (integer or float values). In

Table 3.1, the maximum function is mapped as a function that computes the maximum but to a set of reward variable results.

5. An **SLO** threshold and its binary relation have to be mapped as an **Evaluation Function** with a mathematical value and an arithmetic relation. This function has to compare the result obtained from the previous step (i.e. the outcome of the desired composite metric that is used within an SLO) to the specified value according to the arithmetic relation. The result of this comparison should determine the probability to conform to the specified threshold. In the example in Table 3.1, the SLO threshold is mapped as a function which takes the result of the maximum function to determine if this is less than 100.

Phase 3: Model Completion: The SlaCP methodology adopts the use of a stochastic model for predicting the SLA compliance. Hence, in this phase, a stochastic model has to be created for the specified service. Since one of the characteristics of the methodology is that it should be automated as much as is possible, the **Service Model**, that describes its behaviour, should be produced as an outcome of this phase and its creation has to be aided automatically. However, it is not actually possible to derive a complete model automatically from an SLA alone. This is because, as stated in Section 1.4, the SLA does not provide any information about the system's dynamics. Furthermore, the mapping from phase 2 only includes a small set of action/state variables; this does not help in creating a proper model. Accordingly, as indicated in Figure 3.1, a **User** has to complete this model, at any abstraction level, independently based on his/her knowledge of the service; alternatively, he/she should use another means of automatic model creation in addition to an SLA. An example of the latter is using service description documents to help in creating a service model automatically. The remaining issue related to the Model Completion phase is parameterising the model. Parameterising the stochastic model (for aspects such as delay functions and the model's initial state) can be achieved through similar implemented services or from the service's historical data.

One of the characteristics considered in the methodology is that the service model should be general. This can be satisfied by the following:

1. The model is not type-specific: In essence, a user can choose any appropriate stochastic modelling formalism to model his/her service. However, in order to be general and modular, this service model has to be specified in an abstract stochastic modelling formalism, such as SDES, that can be converted later to any specific one (such as Stochastic Petri Net) with minor translation.

2. The model is not service-layer specific (i.e. not QoS-type specific): The service has many different layers (e.g. software, hardware or network) and can be looked at, in terms of its level of detail, in depth or abstractly (i.e. as relations between services in a composite service, or as relations among the infrastructure components of a specific service). Each abstraction level and each layer has specific types of QoS metric that can be defined. For generality, the model's abstraction level has not been standardised in this methodology; the same is true for the layers. Using a specific level of abstraction makes the methodology inapplicable for several types of QoS metric. For example, in the literature, the stochastic model of a service is targeted mostly either at a specific layer of service infrastructure, such as software, or at a very high level through which only the interactions between different services are modelled, as in an SOA architecture [26]. In the latter, each service's internal behaviour is treated as a black box while modelling the communication between services is the primary concern for the stochastic model; only response time related metrics can be predicted from these very abstract models. In the proposed methodology, the choice is left to the user to decide the level of detail his/her model has to take into account. In this way, the model is not specific to one layer of the service related infrastructure; it can reflect the software, hardware or network layer, or any combination of them, that serves the purpose of the SLA.

3. The model is not single-service-specific: In the methodology proposed in this thesis, the aim is to model the behaviour of a single service. However, it is not obligatory and the model can represent multiple services with the interaction between them.

Phase 4: Model Specialisation: The outcome of the Model Completion phase is a generalised stochastic model of the service. Since the user can choose a specific formalism in which to model the service, the generalised model has to be translated accordingly into this formalism. In addition, the reward variables generated from step 2 of phase 2 have to be translated to suit the input language of the specified model. The service model, along with the reward variables, produces a **Reward Model** that is able to generate the value of these reward variables. Also, in this phase, the time periods and intervals for solving the reward variables has to be translated so that the solver can solve these reward variables according to them. Hence, the Model Specialisation phase translates the generalised service model, the general reward variable specification, and the reward solving time into fragments

that are compatible with the stochastic model and the solver chosen by the user.

In this phase, and as specified in the SLA-Model Mapping phase, the actual connection between the reward function templates and the primitives of the actual service model has to be specified. For example, if the basic metrics relate to the throughput of a service object, and an impulse reward variable is generated for the action representing this object, a link between the abstraction of this action and its actual reference in the model has to be assisted by the user since this cannot be completed automatically. If the desired action is the one generated automatically by the SLA-Model Mapping phase, the user has to confirm this; otherwise the desired action has to be chosen. Making the required reference allows the template to be updated with a concrete reward function so that it is comprehensible to the model solver in the next step. If this connection is not valid, the solver will not be able to produce a meaningful result.

Phase 5: Model Solving: The outcome of the four consecutive phases presented earlier is a reward model that has to be solved analytically or by using simulation to produce the required results. During this phase, the expected **Result** of the reward variables at the specified time instants or intervals have to be produced and are then available for further processing. In the tool representation of the proposed methodology in Chapter 6, the design choices available for choosing the solver are described.

Phase 6: Metric Composition: During this phase, and to produce the value of the desired composite QoS metric, the remaining specified computations (derived from step 4 of phase 2) have to be applied as functions of the solver results. Consequently, a set of functions can be applied to produce the **Desired Metric**. A set of the functions that are used to further define a QoS metric, and how to accommodate the expected values of the solver output, are described in Chapter 5.

Phase 7: Decision: After obtaining the value of the desired metric, the evaluation function taken from step 5 of phase 2 has to be applied to compare this value with the SLO threshold. The result of this comparison is the **Compliance Probability** of the SLA contract for this particular SLO.

3.2.2 User's Perspective of the Methodology

The user's view concerning the design of the SlaCP methodology depends on the ability of the methodology to automate the aforementioned seven phases. From the

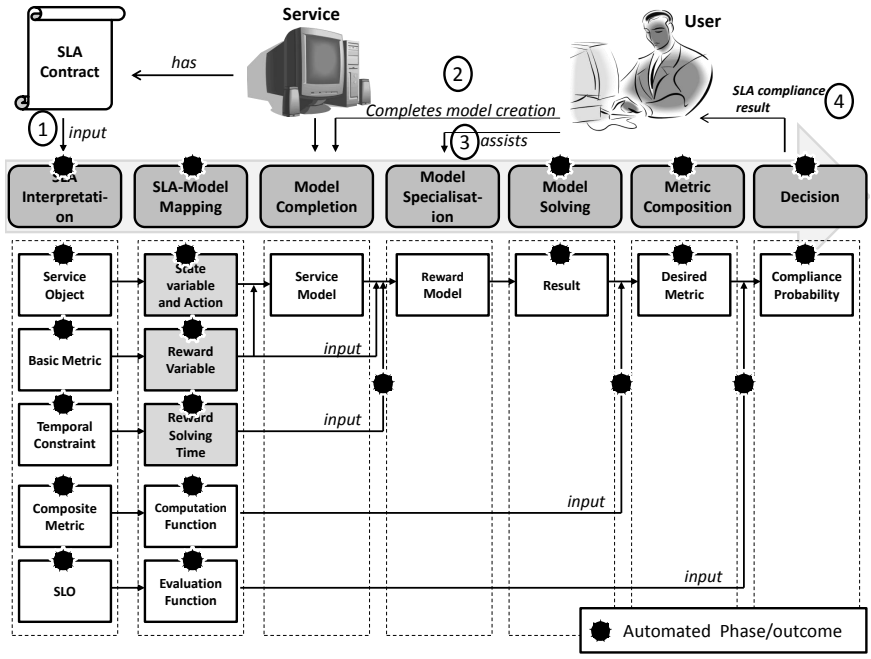

Figure 3.2: The proposed methodology from a user perspective

user's point of view, automating a phase allows it to be concealed and seen as a black box. The rationale for describing this perspective is to help the tool designer who is exploiting this methodology to understand the aspects that should be automated.

Depending on the design of the SlaCP methodology described earlier, Figure 3.2 depicts the phases with their outcomes that has to be automated. The automation of a phase or its outcomes are illustrated in this figure as a black gear, while the un-automated aspects of the methodology or the ones that the user can see are numbered. According to this figure, the methodology has to be shortened for the user into four steps. These are as follows:

1. The user has to supply the SLA document.
2. The user has to complete the generated model creation. This is part of the Model Completion phase.
3. The user has to assist the correlation between each abstract primitive in the reward function to the relevant primitive of the model that represents it. This is a part of the Model Specialisation phase.
4. The result regarding SLA compliance probability is shown to the user.

To avoid repetition, the reasons why the rest of the SlaCP phases can be automated is presented in Chapter 6 where the tool architecture design is described.

3.2.3 Tool Designer's Perspective

The theoretical design of the SlaCP methodology proposed in Section 3.2.1 should make it possible to map a given SLA of a service into a stochastic model. The outcome of this methodology has to be a prediction of the probability of SLA compliance; this is derived from solving the model after mapping (or what is called a reward model). It is appealing to a user if this methodology can be supported by a software tool which will automate the phases where possible. This software tool, called SlaCP tool, has to consider the theoretical phases of the methodology design so that a user's perspective can be adopted. The tool designer has to represent each phase as an engine that automates its functionality. In addition, the designer has to make use of a set of existing and novel techniques to increase the modularity and automation of the tool that exploits this methodology.

To achieve a better flow of information, the description of the tool designer's perspective (i.e., the SlaCP tool architectural design) is represented in Chapter 6 where the implementation of this tool design is also described.

3.3 SlaCP Implementation for WSLA Contracts and SDES Models

The SlaCP methodology explained in the previous section describes a generic method for SLA compliance prediction; it can be applied to any SLA contract and any general stochastic model formalism. To facilitate a better understanding of the different phases and perspectives of both the SlaCP methodology (that is explained in this chapter) and the tool (that is explained in Chapter 7), an implementation of these is performed. The implementation has been carried out for a specific type of SLA contract, namely WSLA [17], and a specific generalised stochastic modelling formalism, namely SDES [29]; this is called WSLA Compliance Prediction or WslaCP. The Web Service Level Agreement language (WSLA) is chosen because it is already published and is publicly available; it is also powerful enough to define SLA documents in several domains (such as web services). It also defines QoS metrics clearly and explicitly using constructive ontology which is important for the proposed methodology. Finally, it is XML based so it is accessible and extensible. Similarly, the Stochastic Discrete Event System (SDES) formalism is a general purpose stochastic process formalism that includes a variety of modelling formalisms; the mapping to SDES therefore directly translates to formalisms that have extensive tool support.

The implementation of the SlaCP methodology is referred to as the WslaCP

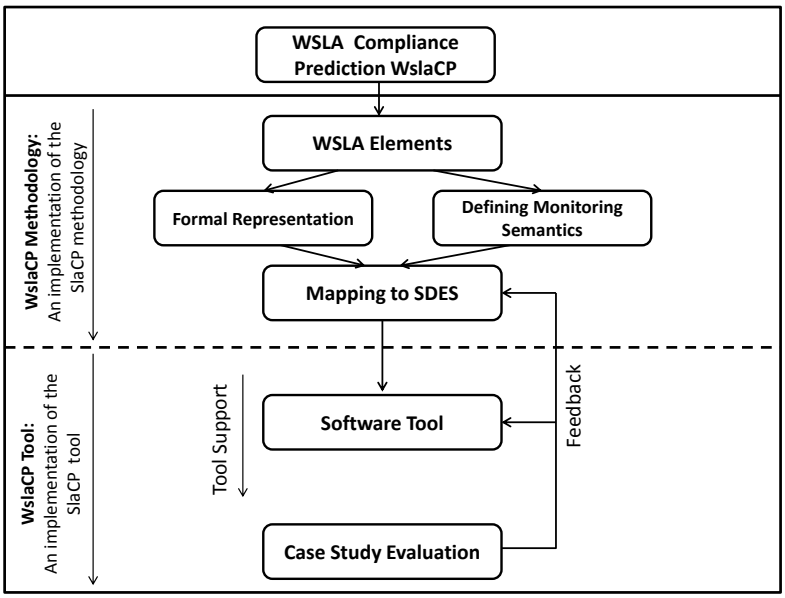

Figure 3.3: The WslaCP methodology diagram

methodology, while the implementation of the SlaCP tool design is referred to as the WslaCP tool. These are briefly outlined in the following two subsections.

3.3.1 WslaCP Methodology: An Implementation of the SlaCP Methodology

The WslaCP methodology implements the seven phases described for the SlaCP methodology. In this implementation, there is no consideration of a new algorithm to implement the Model Solving phase; it is assumed to be predefined.

The implemented WslaCP methodology is presented in two parts: WSLA elements and mapping to SDES, as depicted in the middle part of the WslaCP methodology diagram in Figure 3.3; each has a separate chapter devoted to it. The rationale for this is to achieve a better flow of related information. These parts are as follows:

1. **WSLA elements**: This part represents the SLA Interpretation phase. It is concerned with the **Formal Representation** of WSLA elements, first, to define accurately the structure of the document. Second, it concerns **Defining Monitoring Semantics** to WSLA elements since WSLA is an XML based document whose elements are described using a verbal description. The semantics of WSLA elements in this part are precisely described so they can be mapped correctly. This part is described in detail in Chapter 4.

2. **Mapping to SDES**: This part completes the mapping process for the ultimate SLA compliance prediction. It represents the remaining six phases of SlaCP. Completing the interpretation of WSLA's behavioural semantics, the actual theoretical mapping of the WSLA contract to a stochastic model is carried out. The mapping to SDES consists of five steps; these are outlined in what follows [1]:

   - **Step 1**: `Operation`(s) Mapping maps service operations specified in a WSLA document into SDES primitives.
   - **Step 2**: `MeasurementDirective`(s) Mapping maps all the basic metrics specified in a WSLA document into SDES reward variables.
   - **Step 3**: `Schedule` Mapping maps the time specified in a WSLA document to a set of observation intervals of the reward variable.
   - **Step 4**: `Function`(s) Mapping maps the composite metrics into functions of the results gained from solving SDES.
   - **Step 5**: `ServiceLevelObjective` Mapping maps the SLO onto an evaluation function that produces the SLA compliance probability.

   As described in the Model Specialisation phase of SlaCP, this mapping must be carried out with the help of a user. Accordingly, the user's interaction is necessary in steps 1 and 2 in order to provide suitable information for completing the model creation and assigning the rewards. The aforementioned five steps in this part are described in detail in Chapter 5.

3.3.2 WslaCP Tool: An Implementation of the SlaCP Tool

The architectural design of the tool for the SlaCP methodology (or what is called the SlaCP tool) is implemented in a **Software Tool**, the WslaCP tool, that employs WSLA and a Stochastic Petri Net (SPN) model [30]; this is depicted in the lower part of Figure 3.3. The WslaCP tool is built using Java language and was augmented with an SPNP modelling tool and Möbius for the solution of SPN models. The rationale for using these tools in particular stems from their ability to describe the SPN models, handling and solving them not only through a GUI but also using its input language. This flexibility in expressing the SPN models allows the tool easily to access the file that contains the model description in order to extract any information

[1] It should be noted that some keywords related to WSLA might be unfamiliar to the reader at this stage (although these are described in the background chapter). However, each step, along with any WSLA-specific keywords related to it, is described in Chapter 5.

necessary to assist the user more effectively, to complete the definitions of the reward variables, and then to run the solver to solve the model. This implementation is described further in Chapter 6.

To evaluate the WslaCP methodology and its software tool, a **Case Study Evaluation** is conducted in Chapter 7 to assess, provide feedback and then amend where necessary the proposed methodology and tool, as depicted in the rounded rectangle at the bottom of Figure 3.3. This case study guided the approach taken in this thesis, making it more robust and enabling it to be more automated. It also highlighted the weaknesses, strengths and areas for enhancement.

3.4 Conclusion

This chapter described the SlaCP methodology and set the requirements and the characteristics that need to be included in its design. This theoretical design, across seven phases, should allow the user (possibly an SLA engineer, service designer, service provider, or a modeller) to predict the compliance probability of a given SLA in a largely automated way. The automated steps of the methodology are highlighted by presenting the methodology from a user perspective where he/she is responsible for entering the SLA, completing the service model, and assigning the right primitives to the reward function. An outline of the methodology is also briefly presented from the perspective of a tool designer who automates its phases. Due to the complexity of the abstract derivation of this methodology, and to help in understanding it, the methodology and its tool are implemented using WSLA and SDES. In this chapter only an outline is given for the WslaCP methodology and tool. This is discussed in detail in the forthcoming chapter.

Chapter 4

A Formal Representation of WSLA

This chapter covers the first fold of the WslaCP methodology that implements the SLA Interpretation phase of the SlaCP methodology. It addresses the problem of the lack of formal, precise and unambiguous semantics of some WSLA elements. The contribution of this chapter is firstly, to provide a formal characterisation of WSLA elements in order to clarify their structure and relationships between them; secondly, to give a mathematical definition of WSLA elements whose semantics are not rigorously defined. This is done to avoid any misinterpretation of their meaning between service providers and customers. Formalising and mathematically defining the semantics of WSLA elements provides a firm basis for mapping on the SDES models.

The remainder of the chapter is structured as follows. Section 4.1 introduces the problem of semantic precision in WSLA specifications while Section 4.2 provides the solution requirements to solve this problem. Section 4.3 specifies the foundation of the formal representation. This includes defining the WSLA prediction-related elements to be formalized and the exact XPath locations for them. In addition, the non-prediction related elements are defined. In Section 4.4, a formal definition of the main elements of WSLA is provided using tuples with the mathematical representation of each component in each tuple. Section 4.5 then defines the mathematical semantics for WSLA elements that are not precisely defined. These semantics will be addressed from a monitoring perspective to distinguish this from the prediction perspective that is of a stochastic nature. Related work is provided in Section 4.6. Finally, the chapter is briefly summarised in Section 4.7.

4.1 Introduction

WSLA has the flexibility to define the desired QoS metrics due to its constructive ontology [10, 17]. This ontology, as described in Section 2.2.2.1, allows a set of measured QoS metrics to be composed using different operators in order to produce the desired composite metrics. The measured metrics, operators of the composite metrics, and the final QoS metric are presented in WSLA using `MeasurementDirective`(s), `Function`(s), and `SLAParameter`(s)[1] respectively.

Although WSLA defines a wide range of standard `Function`(s) for simplifying the composition process, it does not provide precise mathematical semantics for them. These functions are defined in a textual description, as appeared in [17], making it difficult sometimes for the reader to understand their exact meaning. An example of such a function is the `SPAN` function. This function is defined in WSLA as returning "the number of sequential occurrences of a particular value in a time series or queue, backwards from the most recent entry" [17]. This means that this function gives for a specific position in the time series, the maximum length of an uninterrupted sequence of a specific value ending at that position. This function is not intuitive: i.e. it is hard to understand even after describing it in different informal ways (unlike functions such as `Max` or `Mean`). If the function description is put into a mathematical formula, its semantics will be more precise.

Semantic ambiguity is also associated with `MeasurementDirective`(s). Measurements are related to a service object attribute, but some of their semantics are not intuitive, like `Gauge` which intercepts the current value of an entity; hence their semantics need to be defined. In addition, other commonly-used measurements may hold multiple semantics according to a user's perspective (i.e., service providers and customers). Therefore, they have to be assigned a unified semantic in a way that allows providers and customers to share the same perspective. An example of a measurement with multiple semantic perspectives is `ResponseTime`. From a provider perspective, it may be considered as the time taken to complete the job and send the response back, starting from the instant of receiving the request. However, from a customer perspective, response time may be considered as the time that elapses from sending the request from the terminal until the response is received. The difference in perception could be explained by the fact that the former perspective does not consider network delay, while the latter does. Given this example implies that the difference in a single measurement's interpretation maybe due to the fact that it can be related to one or more of the service objects or

[1] WSLA elements are referred to using Courier font.

layers, including hardware, software, network, storage and help desk [21]. Response time, for example, can be related to all the aforementioned service objects. Since a measurement can comprise different service objects, and given the diversity in understanding its exact semantics [45], clarifying its definition is important to avoid any confusion.

4.2 Representation Requirements

The requirements identifying the aspects of the problem that have to be tackled, and recalling the first fold of Section 3.3.1, are defined as:

- Formal Representation of WSLA Elements: To describe formally main elements of WSLA and their sub-elements which are important for prediction only. This requirement is particularly helpful in clarifying the structure and the dependencies between WSLA elements so they can be mapped appropriately. (cf. Section 4.4.)
- Defining the monitoring semantics of WSLA elements: To provide a mathematical representation that conveys the exact meaning of WSLA elements in cases of monitoring. Monitoring semantics are those the customer is interested in when agreeing to SLAs and hence he/she needs to understand them clearly. This requirement helps in addressing the issue of semantic ambiguity and aids in providing an exact mapping later on. (cf. Section 4.5.)

4.3 Representation Foundation

To address the aforementioned requirements, this section firstly presents the main WSLA XML elements that are required to be formalised along with their relationships. For simplicity, the XML attributes and sub-elements they require are described when providing the formal representation in Section 4.4. This section also presents the non-prediction related elements. Secondly, to clarify the location of the prediction-related elements inside a WSLA document, the exact XPath location for each of them is provided. An augmented WSLA contract of a stock quote service adapted from [17] is used as a running example throughout this chapter to illustrate the required elements and the different aspects of the formal representation.

4.3.1 WSLA Elements and their Relationships

In this section, the WSLA contract is not formalised as a whole; only the elements necessary for SLA prediction are considered. These are the agreed service level objective (`ServiceLevelObjective`), the desired QoS metric (`SLAParameter`), and the elements constituting them. The excluded information are regarding the involved parties, the action guarantees in cases of SLO violation, and any element or attribute that does not matter to the prediction process. The prediction-related and non-related elements are described in Sections 4.3.1.1 and 4.3.1.2 respectively.

4.3.1.1 WSLA Prediction-Related Elements

To clarify the elements required for the WslaCP, Figure 4.1 is constructed to depict the entity-relationship diagram of the main elements to be formalised, their relationships, and the WSLA section that each of these elements belongs to.

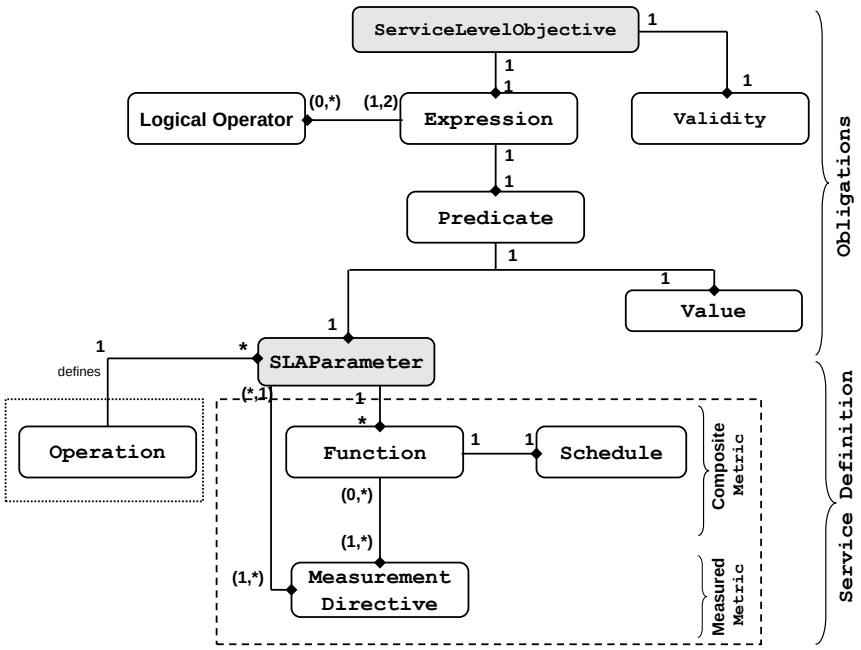

Figure 4.1: The Entity- Relationship diagram for the required WSLA elements in WslaCP methodology

The `Obligations` part of Figure 4.1 shows that a `ServiceLevelObjective` is defined by a single logical `Expression` that has to be valid during a specific `Validity` period. This `Expression` defines a single `Predicate` which performs a comparison of a specific type (such as `Less`, `Greater`, `Equal`, `LessEqual`, `GreaterEqual`[1])

[1]There is another `Predicate` element called (`Violation`) which is used for triggering correction

between a single SLAParameter and a particular threshold Value. Many logical operators (such as And, Or, Not, Implies) can be used to express an SLO with nested expressions.

Listing 4.1: An example of a Service Level Objective for a stock quote service.

```
 1:<Obligations>
 2:  <ServiceLevelObjective name="ContinuousDowntimeSLO">
 3:    <Validity>
 4:        <Start>2001-11-30T14:00:00.000-05:00</Start>
 5:        <End>2001-12-31T14:00:00.000-05:00</End>
 6:    </Validity>
 7:    <Expression>
 8:        <Predicate xsi:type="Less">
 9:          <SLAParameter>CurrentDownTime</SLAParameter>
10:          <Value>10</Value>
11:        </Predicate>
12:    </Expression>
13:  </ServiceLevelObjective>
14:</Obligations>
```

Listing 4.1 shows, through an SLO of a stock quote service, the representation of the Obligations' elements in a WSLA context. In descriptive terms, this SLO states that the system will not go down for 10 minutes or more in a row throughout the last month of the year 2001. In WSLA terms, an SLO called ContinuousDownTimeSLO (line 2) specifies an SLAParameter called CurrentDownTime (line 9) to be Less (line 8) than a Value of 10 (line 10) during a Validity period of a month Start-ed at the end of November (line 4) and End-ed at the last day of December 2001 (line 5). The reason for specifying the threshold value in minutes is that the CurrentDownTime unit is defined as minutes (see line 10 in Listing 4.2).

The Service Definition part of Figure 4.1 is divided into two segments. The one encompassed in the dashed rectangle shows the elements constituted in defining the SLAParameter referred to in the Obligations section. The second segment encompassed in a dotted rectangle shows the Operation (it represents a service object) for which this SLAParameter is defined.

For the former segment, and as described in the introduction, WSLA allows the contractual parties to choose the way the desired SLAParameter is measured and computed. This is done using MeasurementDirective or Function elements. A MeasurementDirective provides raw data obtained by intercepting or probing a particular service object. If these data are insufficient to express an SLAParameter, further manipulations are performed using Function(s). Many functions can be applied as needed to express the desired SLAParameter. A Function may use a

actions while monitoring. This is not used in the ServiceLevelObjective context. For this reason it has been ignored here.

`Schedule` if it is intended to construct a time series. This schedule defines a period in order to specify the time during which the values are collected (day, month, etc.), and an interval to specify the instances when a new value is added (minute, hour, etc.). `MeasurementDirective`(s) and `Function`(s) are defined in WSLA inside a measured and composite `Metric` respectively. However, in this thesis, and as shown in Figure 4.1, `Metric` usage is ignored in the formal representation of WSLA elements because it is used in WSLA to hold the value of a measurement or a function for re-usability. This is not important for this SLA prediction approach. Hence, excluding `Metric` usage is helpful for the sake of simplicity; it also avoids overlapping in an `SLAParameter` definition.

For the latter segment, WSLA defines at least one service object in its service definition section. A service object in WSLA may represent, for example, a WSDL/SOAP operation, a business process, a web hosting service, an on-line storage service [17], an interface, attribute, operation parameter, or operation result [44]. The `SLAParameter` is defined for one of these objects which is mainly an `Operation`.

Listing 4.2: An example of an SLAParameter for a stock quote service

```
<ServiceDefinition name="DemoService">
  <Schedule name="availabilityschedule">
    <Period>
      <Start>2001-11-30T14:00:00.000-05:00</Start>
      <End>2001-12-31T14:00:00.000-05:00</End>
    </Period>
    <Interval> <Minutes>1</Minutes> </Interval>
  </Schedule>

  <Operation name="GetQuote" xsi:type="WSDLSOAPOperationDescriptionType">
    <SLAParameter name="CurrentDownTime" type="long" unit="minutes">
      <Metric>CurrentDownTime</Metric>
    </SLAParameter>

    <Metric name="CurrentDownTime" type="long" unit="minutes">
      <Function xsi:type="Span" resultType="double">
        <Metric>StatusTimeSeries</Metric>
        <Value> <LongScalar>0</LongScalar> </Value>
      </Function>
    </Metric>

    <Metric name="StatusTimeSeries" type="TS" unit="">
      <Function xsi:type="TSConstructor" resultType="TS">
        <Schedule>availabilityschedule</Schedule>
        <Metric>MeasuredStatus</Metric>
        <Window>1440</Window>
      </Function>
    </Metric>

    <Metric name="MeasuredStatus" type="integer" unit="">
     <MeasurementDirective xsi:type="StatusRequest"  resultType="integer">
```

```
28:        <RequestURI>http://ym.com/StatusRequest/GetQuote</RequestURI>
29:      </MeasurementDirective>
30:    </Metric>

31:   </Operation>
32:</ServiceDefinition>
```

As an example of an `SLAParameter` definition in a WSLA context, Listing 4.2 specifies how `CurrentDownTime` (line 10) of a `GetQuote` operation (line 9) is computed. In descriptive terms, the status of the service is probed periodically each minute for a month and stored into a time series. This time series is checked each minute to compute the length of consecutively occurring down status backwards from the current minute. In WSLA terms, the `StatusRequest` measurement (line 27) in the `MeasuredStatus` metric (line 26) checks if the service is up or down and returns 1 or 0 respectively. The result from this metric is used as input to the `StatusTimeSeries` metric (line 19) that uses a `TSConstructor` function (line 20) to construct a series of `MeasuredStatus` values (line 22). To determine when these values are collected, the `availabilityschedule` (line 2) is used by the `StatusTimeSeries` metric to take input from the `MeasuredStatus` metric at each minute (line 7) for a month (lines 4 and 5). A `Window` of 1440 elements (line 23) of the time series is saved by this metric to allow enough values to be available for doing computation at any time instant. In turn, the `CurrentDownTime` metric (line 13) applies a `Span` function (line 14) on the produced series to give, for a specific position in that time series, the maximum length of an uninterrupted sequence of a value (0 in this example) ending at that position. Finally, the `CurrentDownTime` metric output (line 11) is used as the `SLAParameter` value.

All the aforementioned elements define the SLO in a constructive and detailed way and constitute the information necessary for the WslaCP methodology. These are the `ServiceLevelObjective` with its `Expression`, `Predicate` and `Validity` period, and the `SLAParameter` with its `Function`, `MeasurementDirective`, `Schedule` and the `Operation` that it is defined for. The sub-elements and attributes related to the previous elements are specified in Section 4.4.

4.3.1.2 WSLA Non-Prediction Related Elements

The rest of the WSLA elements are ignored. Examples of these elements and the reason for ignoring them are described in what follows. Given that the proposed methodology is not intended to react to any low SLA compliance probability, the `ActionGuarantee` element and the notification mechanisms needed in case of SLO violation (defined in the `Obligation` section) are ignored. Since the methodology is

mainly used before service deployment, there is no need to include information about the contractual parties (`ServiceProvider` and `ServiceConsumer`) since they may be not yet exist. Also, since the methodology is not intended for monitoring, there is no need to include information about the supporting parties (`SupportingParty`) and their roles in monitoring the service. Hence, the `Parties` section with its elements is ignored. Furthermore, as the methodology is used for prediction only, all the information related to the monitoring nature of the WSLA contract are omitted. An example of this information that is defined in the `ServiceDefinition` section is the `communication` elements that describe which party is able to see `SLAParameter` values and how these values are transferred to the required parties (using `Pull` and `Push` mechanisms). Finally, as the methodology assumes that the WSLA document is predefined and valid, information regarding the data `type`, `resultType` and `unit` of different WSLA elements is ignored. The reason behind this is that a validity check of WSLA syntax is not required to prove that the data types of the elements constituting an `SLAParameter` complement with each other.

4.3.2 XPath Location for WSLA elements

To be able to provide a precise representation of WSLA prediction-related elements, there is a need to identify the actual places of the elements inside the WSLA document. Table 4.1 provides the exact location path of the required elements using the XPath 2.0 query language [123]. These location paths will be employed in the parsing algorithm needed when implementing the Interpretation phase of the methodology.

Table 4.1 contains three columns representing WSLA elements and attributes, their notation in the equivalent formal representation, and their location path inside WSLA. The formal representations of WSLA elements with their notation are described in the forthcoming section. The location paths are not explained as they are self-described. However, the main constructs used by XPath to define these paths are specified in what follows.

XPath is used to identify elements in an XML document. It uses location path expressions to reach a specific node or set of nodes [123]. A node, used in this work, is an element, an attribute, or a text node, reached using `nodename`, `@nodename` and `text()` constructs, respectively. A path expression consists of a set of steps separated by a slash `/`, to represent a parent-child relation, or double slash `//` to represent an ancestor-descendant relation. The latter relation means that the nodes matching the selection are returned starting from the ancestor node, regardless of

Table 4.1: Formal elements and associated WSLA location using XPath 2.0

WSLA Element/attribute		Formal Element Notation		WSLA Location Path Expressions
ServiceLevel-Objective		*slo*		/SLA/Obligations/ServiceLevelObjective/@name
ServiceLevelObjective	SLAParameter	*slo*	*slap*	//ServiceLevelObjective[@name='*slo*']//Expression/Predicate/SLAParameter/text()
	Predicate type		*c*	//ServiceLevelObjective[@name='*slo*']//Expression/Predicate/SLAParameter[text()='*slap*']/../@xsi:type
	Value		*v*	//ServiceLevelObjective[@name='*slo*']//Expression/Predicate/SLAParameter[text()= '*slap*']/../Value/text()
	Start		*vs*	//ServiceLevelObjective[@name='*slo*']/Validity/Start/text()
	End		*ve*	//ServiceLevelObjective[@name='*slo*']/Validity/End/text()
	Expression		*expr*	//ServiceLevelObjective[@name='*slo*']//Expression/node()
	And, OR, Not, Implies		*lo*	//ServiceLevelObjective[@name='*slo*']/Expression//Expression/../fn:name()
	Predicate		*pre*	//ServiceLevelObjective[@name='*slo*']//Expression/Predicate/node()
SLAParameter	Measurement-Directive	*slap*	*m*	/SLA/ServiceDefinition/Operation/Metric/MeasurementDirective/@xsi:type
	Function		$F_{m,i}$	//Metric[Function/Metric/text()=//Metric/MeasurementDirective[@xsi:type='m']/../@name | //Metric/Function[@xsi:type='$F_{m,i-1}$']/../@name]/Function/@type
	Schedule		*sch*	//Metric[Function/Metric[text()=//Metric/MeasurementDirective[@type='*m*']/../@name]]/Function[@type='TSConstructor']/Schedule/text()
Schedule	Start	*sch*	*s*	/SLA/ServiceDefinition/Schedule[@name='*sch*'] /Period/Start/text()
	End		*e*	/SLA/ServiceDefinition/Schedule[@name='*sch*'] /Period/End/text()
	Interval		*i*	/SLA/ServiceDefinition/Schedule[@name='*sch*'] /Interval/node()/text()
Operation		*op*		/SLA/ServiceDefinition/Operation/Metirc[MeasurementDirective[@xsi:type='*m*']]/../@name
RequestURI MeasurementURI		*uri*		//Operation/Metirc/MeasurementDirective[@xsi:type='*m*']/node()/text()
Window		*w*		//Function[@xsi:type='TSConstructor']/Window /text()
Value		v_f		//Function/Value/node()/text()
Element		e_f		//Function[@type='TSSelect']/Element/text()
Digit		d_f		//Function[@type='Round']/Digit/text()

where these nodes are. Predicates (inserted into square brackets `[]`) are used to search for a node with a particular index or that matches a particular value. Also two dots (`..`) are used to select the parent of the current node. The function

`fn:name()` is used to return the name of the node. Finally, `node()` is a wild card used to return all the child nodes of the current node.

All the XPath expressions in Table 4.1 were validated using Altova XMLSpy software [124] to check their correctness. This was done using the WSLA example in Listings 4.1 and 4.2.

4.4 Formal Representation of WSLA Elements

In this section, the structure of the WSLA's core elements, depicted in Figure 4.1, is formalised. These elements are represented using mathematical constructs rather than XML tags. The XML sub-elements and attributes related to these elements and needed for prediction are also formalised. In the following subsections, the formal representations of `ServiceLevelObjective`, `SLAParameter`, `MeasurementDirective`, `Schedule`, and `Function` are described. Then a formal representation of the common order in which `MeasurementDirective`, `Schedule`, and `Function` elements are aggregated to define the `SLAParameter` is defined.

4.4.1 Service Level Objective

`ServiceLevelObjective` is the key element in WSLA contracts, defining the ultimate goal. As described in the flow of Figure 4.1, an SLO defines a logical `Expression` that could be nested into another by using a logical operator. If the `Expression` is evaluated to true through the specified `Validity` period, then the SLO is met; otherwise it is violated.

The simplest SLO in WSLA has a single expression that defines one `Predicate`; this compares one predefined `SLAParameter` with a specific `Value`. An example of an SLO with a simple expression is the example in Listing 4.1. Here the SLO consists of one expression (line 7) that has one `SLAParameter` called `CurrentDownTime` (in line 9) compared using `Less` (in line 8) to a value of `10` (in line 10).

An example of an SLO with nested expressions is the example in Listing 4.3 which is taken from [17].

Listing 4.3: An example of a Service Level Objective with nested expressions

```
1:<ServiceLevelObjective name="ConditionalSLOForTransactionRate">
2: <Validity>
3:  <Start>2001-11-30T14:00:00.000-05:00</Start>
4:  <End>2001-12-31T14:00:00.000-05:00</End>
5: </Validity>
6: <Expression>
7:  <Implies>
```

```
 8:    <Expression>
 9:     <Predicate xsi:type="Less">
10:      <SLAParameter>OverloadPercentage</SLAParameter>
11:      <Value>0.3</Value>
12:     </Predicate>
13:    </Expression>
14:    <Expression>
15:     <Predicate xsi:type="Greater">
16:      <SLAParameter>TransactionRate</SLAParameter>
17:      <Value>1000</Value>
18:     </Predicate>
19:    </Expression>
20:   </Implies>
21: </Expression>
22:</ServiceLevelObjective>
```

The SLO in this listing consists of three expressions (lines 6, 8, 14): one for `OverloadPercentage` to be `Less` than `0.3` (lines 10, 9, 11 respectively), the other for `TransactionRate` to be `Greater` than `100` (lines 16, 15, 17 respectively), and the last (the topmost expression) is the expression that combines these two expressions together using the `Implies` (line 7) logical operator (this operator is equivalent to '*giving that*').

In the following subsections, a distinction is made between formalising an SLO depending on expression types (simple or nested) because the paper describing this formalisation [15] was limited to the first type. Nested type is supported later.

4.4.1.1 Service Level Objective with a Simple Expression

The `ServiceLevelObjective` with a simple `Expression` does not include logical operators in its definition. This means `Expression` is a `Predicate` that compares an `SLAParameter` with some threshold `Value` for a specific `Validity` period. This consideration allows the following formal definition, where the `Expression` is not represented, since it does not matter to the mathematical semantics of this case.

Definition 3 *A WSLA* `ServiceLevelObjective` *with a single expression can be denoted by a tuple* $slo = (slap, c, v, vs, ve)$*, where:*

- $slap \in SLAP$*: is the desired* `SLAParameter` *from the set of all* `SLAParameter`*(s) (SLAP) defined in a WSLA document. slap is defined in Definition 7.*
- $c \in C$*: is the comparison type, where* $C = \{=, <, >, \leq, \geq\}$*.*
- $v \in \mathbb{R}$*: is the value the slap is compared to.*
- $vs, ve \in \mathbb{R}_{\geq 0}, vs \leq ve$*: is the start and end of the validity period.*

Given the example in Listing 4.1, the *slo* named `ContinuousDowntimeSLO` can be written as:

$$slo = (slap, <, 10, 0, 31),$$

where, the *slap* named `CurrentDownTime` is specified in Section 4.4.2. For simplicity, *vs*,*ve* are represented by 0 and the difference between the start and end dates which is 31 days. The value of *ve* can be represented using a different time unit other than days (such as second, minutes, etc.).

4.4.1.2 Service Level Objective with Nested Expressions

A `ServiceLevelObjective` with nested `Expression`(s) (as in Listing 4.3) may use a logical operator such as `Or`, `And`, `Not`, or `Implies` to combine multiple expressions. In this case, the topmost expression that encompasses all other expressions should be valid during the specified `Validity` period. This consideration allows the following definition.

Definition 4 *A WSLA* `ServiceLevelObjective` *with nested expressions can be denoted by a tuple* $slo = (expr, vs, ve)$*, where:*

- $expr \in Expr$*: is the topmost expression that consists of at least two expressions from the set of all expressions, Expr, defined in a WSLA document. expr is defined in detail in Definition 5.*
- $vs, ve \in \mathbb{R}_{\geq 0}, vs \leq ve$*: is the start and end of the validity period through which the expr should be valid.*

The topmost expression, $expr \in Expr$, contains all the nested expressions defined for an *slo*. These expressions are combined using a unary or binary logical operator. In WSLA, a unary operator is applied on a single expression (such as the negate operator `Not`) to produce another expression, while a binary operator is used to aggregate two expressions into a new one (such as `And`). In WSLA, applying any logical operator will result in a new expression. Therefore, the topmost *expr* has at least two expressions, one for a unary operator and the other for a single predicate. It will have at least three expressions with two predicates if a binary operator is used. Given what was described previously, *expr* is defined as the following.

Definition 5 *A WSLA* `Expression` *of an SLO can be denoted by a tuple* $expr = (LO, Pre)$*, where:*

- LO*: is a set of logical operators, i.e.* $LO \subseteq \{\vee, \wedge, \neg, \implies\}$.

- *Pre: is a non-empty set of predicates, where a predicate pre ∈ Pre is defined in detail in Definition 6.*

The way the two sets are combined to create a logical expression is complicated to write in mathematical terms. Instead, an algorithm that conveys this is implemented in the tool. The same problem was identified in [33] where the service level objective cannot be defined formally in an ideal way.

The set *LO* in this definition can be empty. In this case the expression is simple and is equal to a predicate. (This is the case in Definition 3.)

Each `predicate` compares an `SLAParameter` with some threshold `Value`. Hence, $pre \in Pre$ is defined as:

Definition 6 *A WSLA* `Predicate` *can be denoted by a tuple* $pre = (slap, c, v)$, *where the components in this tuple follow the same definition as in Definition 3.*

As an example of the previous two definitions, and given the SLO with nested expressions presented in Listing 4.3, the following can be obtained:

$$LO = \{ \implies \}, Pre = \{pre_1, pre_2\} : expr = pre_1 \implies pre_2,$$

where: $pre_1 = (slap_1, <, 0.3)$, and $pre_2 = (slap_2, >, 1000)$ and the definition of $slap_1, slap_2$ (named as `OverloadPercentage` and `TransactionRate` respectively) are not presented in this example.

In this section, the formal definition of *slo* is specified. This was done for *slo* with both simple and nested expressions. The rest of the definitions in the following sections can be applied to *slo* with simple or nested expressions.

4.4.2 SLAParameter

An *slo* refers to *slap* in its definition. What *slap* means exactly is represented in the WSLA `ServiceDefinition` section. The next step is to define the `SLAParameter`, *slap*, formally. As depicted in Figure 4.1, the most commonly used case for defining the exact *slap* is by collecting `MeasurementDirective`(s) at regular intervals of a `Schedule` and then by applying a set of `Function`(s) over them. This allows the definition of *slap* as:

Definition 7 *A WSLA* `SLAParameter` *is a tuple* $slap = (M, Sch, F)$, *where:*

- *M: is a non-empty set of* $|M|$ *elements. Each* $m \in M$ *specifies a* `Measurement-Directive` *that is used to define this slap.* $m \in M$ *is defined in detail in Definition 8.*

- *Sch: is a set of* $|Sch|$ *elements. Each* $sch \in Sch$ *specifies a* `schedule` *used by a WSLA function to collect measurements or function values periodically.* $sch \in Sch$ *is defined in detail in Definition 9*

- *F: is the set of all* `Function`*(s) defined for a specific slap. For each* $m \in M$*, a set of functions* $F_m = \{F_{m,1}, \ldots, F_{m,|F_m|}\}$ *is defined to identify the ultimate slap; each refers to a WSLA* `Function`*. This set can be empty if the slap represents the value of a single* $m \in M$*. The set* F_m *is defined further in Definition 10.*

Given the example in Listing 4.1, the *slap* named `CurrentDownTime` can be written as:

$slap = (M, Sch, F)$, where:

m=(($StatusRequest$, $GetQuote$, "$http://ym.com/StatusRequest/GetQuote$"))

$sch = \{0 \ldots 44640\}$

$F_m = \{F_{m,1}, F_{m,2}\}$, where:

$F_{m,1} = (TSConstructor, m, \{0 \ldots 44640\}, 1440)$

$F_{m,2} = (Span, F_{m,1}, 0)$

This can be written as:

$slap$=(($StatusRequest$, $GetQuote$, "$http://ym.com/StatusRequest/GetQuote$"), $\{0 \ldots 44640\}, \{(TSConstructor, m, \{0 \ldots 44640\}, 1440), (Span, F_{m,1}, 0)\})$[1],

where the tuple assigned to $m \in M$, named `StatusRequest`, is discussed in Section 4.4.3. Also, the list of values assigned to *sch*, named `availabiltyschedule`, is discussed in Section 4.4.4. The same is true for the set F that is described in Section 4.4.5.

4.4.3 Measurement Directives

The `MeasurementDirective`(s) are the actual metrics constituting the *slap*. These can be one of seven types, namely `Status`, `StatusRequest`, `Counter`, `Gauge`, `ResponseTime`, `DownTime`, and `InvocationCount`. These types can be extended to add measurements of a domain-specific type. Each *slap* consists of at least one measurement taken from measuring or intercepting the service [17].

Listing 4.4: General structure of the MeasurementDirective element in WSLA

```
<MeasurementDirective xsi:type="wsla:Measurement_Type" resultType="result_Type">
   <?additional tags specifying URI name?>
</MeasurementDirective>
```

[1]The formal representations of WSLA functions and measurements are referred to by emphasizing them to differentiate them from the XML tags.

Listing 4.4 provides the generic structure of a `MeasurementDirective`. The values of the attributes depend on the measurement type; all other tags remain the same. The measurement type is specified in the `type` attribute, which affects the type of result specified in the `resultType` attribute. The structure also contains an element that refers to the URI, from where this measurement value can be retrieved.

Formally, the set $\mathcal{M}$ can be defined as the set of all `MeasurmentDirective` types. Hence, the set M that is defined for a particular *slap* is:

$$slap = (M, F, Sch), M = \{m : m \in \mathcal{M}\} \text{ and } M \neq \emptyset$$

Since each *slap* is defined for a particular `Operation`, m should refer to it because m is used to measure this operation. Accordingly, m is defined as:

Definition 8 *A WSLA* `MeasurementDirective` *is a tuple* $m = (m_{name}, op, uri)$*, where:*

- m_{name}*: specifies the label, or name, of this measurement.*
- $op \in OP$*: is a string that specifies the operation that the measurement is defined for from the set of all service operations* OP*.*
- uri*: is a string that specifies the place from which to read the measurement's value during the runtime.*

Given the example in Listing 4.4, m, named `StatusRequest`, can be written as:

$$m = (StatusRequest, GetQuote, "http://ym.com/StatusRequest/GetQuote"),$$

where *StatusRequest* is defined in Section 4.5.1.

4.4.4 Schedules

A WSLA `Schedule` can be used inside many elements in a WSLA document. For this reason, it is defined separately in the `ServiceDefinition` section allowing multiple WSLA elements to refer to it.

In the `ServiceDefinition` section, a `Schedule` is used mostly by a time series function (called `TSConstructor`) to create a time series of either measurement's values or the values of another function used within a specified *slap* (see line 22 of Listing 4.2). A `Schedule` defines a `Period` (line 3 of Listing 4.2) during which the

values have to be collected. It is characterised by a `Start` and `End` time[1]. It also specifies an `Interval` (line 7 in Listing 4.2) between consecutive retrievals of new values. An `Interval` element contains sub-elements representing `Milliseconds`, `Seconds`, `Minutes`, and `Hours`. Any combination of these sub-elements can be used to specify the required interval. Formally, the schedule can be specified as:

Definition 9 *A WSLA* `Schedule` *is a tuple* $sch = (s, e, i)$*, where this in turn defines a set of time points* $\{t_1, \ldots, t_k\}$*, where:*

- $t_1 = s$ *: is the start point.*
- $t_k = e$ *: is the end point.*
- $t_{j+1} = t_j + i; j = 1 \ldots k-2$*: are the sample points, where* i*, is the increment in time and* $k = \lfloor \frac{e-s}{i} \rfloor \in \mathbb{N}_{\geq 0}$*, is the number of sample points.*

In this formal representation of a schedule, *sch*, i is considered to represent the interval in the form of the lowest sub-element type. For example, if the interval is 2 hours and 30 minutes, then $i = 150$ minutes.

Given the example in Listing 4.2, *sch* named *availabilityschedule* is a set $\{t_1, \ldots, t_k\}$ where:

$t_1 = s = 0$

$t_k = e = 44640$

$i = 1$ minute

$k = 44640$

The interval here is 31 days, then $e = 44640$ minutes (the number of minutes in this interval).

4.4.5 Functions

In WSLA, a `MeasurementDirective` is used as the basis for performing other WSLA computations to produce the topmost metric that represents the required `SLAParameter`. These computations are carried out through `Function`(s). WSLA defines a set of 17 function types in its standard specification. Each one corresponds to either series constructors (`TSConstructor`, `QConstructor`), arithmetic functions (`Plus`, `Minus`, `Divide`, `Multiply`), statistical functions (`Mean`, `Median`, `Mode`, `Sum`, `Max`, `Size`), or other functions (`TSSelect`, `Span`, `PercentageGreaterThanThreshold`,

[1]The time in WSLA is specified using either a `DateTime` format of the standard xsd schema or a IETF RFC 3060 [17]

`PercentageLessThanThreshold`, `NumberGreaterThanThreshold`, `NumberLessThanThreshold`, `ValueOccures`, `RateOfChange`, `Round`) [17].

All the aforementioned functions should have an operand which is either a `MeasurementDirective` or another `Function` output. As well as this operand, some functions require additional ones. For example, the time series constructor function needs a schedule, *sch*, to construct the series according to its time specification. Also, other functions may need a constant value for the reason of comparison like `ValueOccures` which uses a constant operand to compare the series value according to it and then returns the number of times of its occurrence. However, WSLA statistical functions do not need any extra operands.

Formally, the set $\mathcal{F}$ can be defined as a set of all functions in the WSLA specification. Hence, the set F defined for a particular *slap* is:

$$slap = (M, F, Sch), \text{if } M = \{m, \ldots, m_j\} : F = \{F_m, \ldots, F_{m_j}\}, F_m, \ldots, F_{m_j} \subset \mathcal{F},$$

where F_m is a set of functions applied on m. Each function, $F_{m,i} \in F_m$, represents the function that is of order i to be applied on m and is defined as:

Definition 10 *A WSLA* `Function` *is defined as a tuple* $F_{m,i} = (F_{name}, O)$*, where:*

- $i \in \mathcal{N}_{>0}$*: is the order in which this function is applied on a measurement* m*,* $i \in \{1 \ldots |F_m|\}$*.*
- F_{name}*: is the label, or name, of this function type.*
- O*: is a set of operands that this function needs in order to perform its functionality. An operand* $o \in O = \{sch, w, v_f, d_f, e_f, F_{m,i-1}\}$*, where sch is a schedule, the* $w, v_f, d_f, e_f \in \mathbb{R}$ *are constant values situated in* `Window`*,* `Value`*,* `Digit`*, and* `Element` *elements respectively;* $F_{m,i-1}$ *is the previous function output.*

Given the example in Listing 4.1, the function, $F_{m,1}$, named `TSConstructor`, can be written as:

$$F_{m,1} = (TSConstructor, m, \{0 \ldots 44640\}, 1440),$$

where m is defined previously; *TSConstructor* is defined in Section 4.5.2. The function $F_{m,2}$, named `Span`, can be written as:

$$F_{m,2} = (Span, F_{m,1}, 0),$$

where *Span* is defined in Section 4.5.2.

4.4.6 Formal Representation of the Common Order of WSLA Elements to Define an SLAParameter

The following assumption has been made to clarify and simplify the application of measurements, schedule and functions constituting an *slap* definition.

Assumption: WSLA rarely depends on a single measurement $m \in M$ to represent an `SLAParameter`, *slap*. Rather, it applies a set of functions on m, $F_m = \{F_{m,1}, \ldots, F_{m,|F_m|}\}$, where the function $F_{m,i}, i = 1, \ldots, |F_m|$ is the i-th function to be applied on m. The functions in F_m can be any of the types specified in Section 4.4.5. However, from an observation of existing WSLA contracts [16, 17, 54], a common order in which function types are applied has been obtained. Depending on the order in which WSLA functions are applied, it is easier to show the ultimate *slap* value. This order is adopted when mapping to SDES in Chapter 5 in order to achieve a clearer insight into the steps involved. Using this order does not eliminate the applicability of the proposed methodology, both theoretically and in the practical implementation, to any order through which the functions may applied. This common order for applying WSLA functions is described in the following steps:

1. Firstly, time series functions are considered to be used only for collecting measurement values (not values that come from other WSLA functions). This means that WSLA applies a time series function, *TSConstructor*, to create a time series that collects values of m using a schedule, *sch*. Hence, $F_{m,1}$ is always the *TSConstructor* function and its output is a series of measurements:

$$\{m(t_1), \ldots, m(t_k)\}, \{t_1, \ldots, t_k\} \in sch,$$

   where $m(t_j)$ is the measurement m at time $t_j, j = 1, \ldots, k$.

2. Secondly, a function, $F_{m,2}$, is applied in this series so that a single output is produced. All WSLA functions return a single value, except for series constructor functions and `RateOfChange`; these return a series instead of a single value.

3. Finally, additional functions $\{F_{m,3}, \ldots, F_{m,|F_m|}\}$ can be applied so that the exact *slap* is obtained.

It should be noted that, in steps 2 and 3 (described earlier), WSLA functions applied on a set of measurements or a function output may specify an additional operand, $o \in O - \{sch\}$ (as specified in Section 4.4.5). These WSLA functions can

be written in a monitoring case as: $F_{m,2} = (F_{name}, \{m(t_1), \ldots, m(t_k)\}, o), o \in \mathbb{R}$ for step 2, and as $F_{m,i} = (F_{name}, F_{m,i-1}, o), o \in \mathbb{R}, i = 3 \ldots |f_m|$ for step 3.

4.5 Defining the Monitoring Semantics of WSLA Elements

This section adds semantics to WSLA `MeasurementDirective` and `Function` elements depending on the formal representation given in the previous section. This is to avoid any misconceptions in understanding their meaning, as well as to help in mapping them correctly later. These semantics are given, depending on the monitoring nature of WSLA, to distinguish them from the stochastic model semantics presented in Chapter 5 when mapping them to SDES.

4.5.1 The Semantics of Measurement Directives

In the WSLA specification, all measurement directives are used to measure the specific QoS attributes of an object of a service, such as availability, although WSLA might not refer to them explicitly. For this reason, a measurement, $m \in \mathcal{M}$, should be assigned a precise semantic. Furthermore, it should be specified when to measure (instants, intervals), and where to measure (provider side, customer side or network point) [125].

In general, measured QoS metrics (i.e. WSLA measurements) have to be defined in such a way that gives the same perspective for both a service provider and a customer. In a service domain, a QoS metric may be referred to in an SLA using different syntaxes. Efforts have been made by researchers to address the need to unify metric semantics. An example of this is the work in [44] where the researcher attempted to enrich his WSDi semantic framework with different syntactical QoS attributes which had the same semantics. He also set the requirements for formalising QoS descriptions. However, it is has been proven that it is usually hard to decide on the semantic of a QoS metric without considering the domain that it is related to [44]. Since the primary domain for WSLA is a web service, the decision is made by the WslaCP methodology for its measurements to be assigned semantics depending on this domain.

Table 4.2 provides a brief summary of this section. It states the measurement directives available in WSLA and then shows three pieces of information for each measurement directive. In the first, the **Semantic** assigned to each measurement directive is specified. This is done by matching it with a common, well-defined QoS

Table 4.2: Summary of semantics added to the measurement directives

	WSLA Measurement Directives					
	Status/ Status-Request	***Invocation Count***	***Gauge***	***Counter***	***Response Time***	***Down Time***
Semantic	Status of Availability	Throughput	Queue size	Throughput	Processing Time	Down Time
When to take a Measurement	End of interval	During of interval	End an interval	During an interval	End an interval	During an interval
Where to take a Measurement	server side	server side	server side	server side	server side	server side

attribute of service-based systems. Other WSLA measurement directives, that have well-defined QoS attributes, are assigned one semantic from the number they can take. The second information that is given is **When to take the Measurement** and the third one is **Where to take the Measurement**. The former specifies whether the measurement will be taken at an instant or interval of time, while the latter specifies if the measurement will be taken at the server or customer side. The content of this table is described when presenting each measurement in what follows.

1- *Status and StatusRequest*: According to WSLA, `StatusRequest` gives 1 if the system is up and 0 otherwise, while `Status` gives true if the system is up and false if it is down [17]. This difference is ignored because it does not matter to the semantics. The measurements follow the syntax in Listing 4.4 with `resultType` of "*integer*" for `StatusRequest`. The URI is referred to using `<RequestURI>` tag.

- Semantics: This measurement can be related to the status of the well-known *availability* QoS attribute. *Availability* is the probability that the system is working:

$$A = u/T,$$

where, T is a time interval during which the system is observed, and u is the service uptime during this interval [126]. The status of availability means the system is either up or down (0 or 1). Thus, this means that $T = 1$ unit.

- When to take the measurement: since the status of a service operation should return either 0 or 1 (Yes, No), it has to be assessed at specific instants of time. This means that, when the WSLA schedule specifies intervals through a period to

collect the measurement, it considers the measurement to be taken at the end of each.

- Where to take the measurement: This measure is taken at the provider's side. The availability condition of a web service operation is complex and can be related to different aspects [127, 128]. The web service might be unavailable due to a software failure (such as when upgrading software or because of system overload), hardware failure (such as disk or server breakdown), security attack (such as denial of service attack), or human error (such as adjusting system parameters inaccurately) [128]. Availability of a service, from a service provider's perspective, might differ from that of a customer if the provider does not consider all these aspects of failures. For this reason, and in order to achieve the same perceived availability by both the service provider and the customer, the methodology assumes that the service will be unavailable due to hardware failure only. This means that if the service slows down due to overload, network clogging, or denial attack, this is not considered as a failure. This assumption simplifies the job of building the service model necessary for SLA compliance prediction because the provider can set the parameters related to this kind of failure better than he/she can do for the rest. This is because parameterising the probabilities of all kinds of failure in the service model may be too complex or unrealistic, making it difficult to predict the actual availability.

2- *InvocationCount*: WSLA defines this as "the number of usages of an operation per unit time" [17]. In other words, it corresponds to the *throughput* of a service operation. Its syntax follows Listing 4.4 with `resultType`="*integer*" and a `<CounterURI>` tag for specifying the URI.

- Semantics: This measurement can be related to the well-known *throughput* QoS attribute.

$$Th = CR/T,$$

where, CR is the number of completed requests during a time interval of length, T.

- When to take the measurement: Since throughput is a counting mechanism, it will be checked during an interval and retrieved at the end of it. This means that, when WSLA specifies intervals through a period to collect the measurement, it considers the measurement to be taken during those intervals, not at the end of them [126].

- Where to take the measurement: The throughput is always checked at the service provider side in the case of monitoring because it is related only to service hardware. No other constraints have to be considered that may affect it.

3- *Gauge*: This is defined in WSLA as "a non-negative integer that may increase or decrease; it is used to measure the current value of some entity" [17]. `Gauge` has a `resultType` of "*double*" and the URI is referred to using `<MeasurementURI>` tag.

- Semantics: In essence, there is no common QoS metric that relates directly to this measurement. However, it might be considered as a metric that returns a current *queue size*.

- When to take the measurement: The gauge will be checked at specific instants of time to give the current value of a system component. This means that, when WSLA specifies intervals through a period to collect the measurement, it considers the measurement to be taken at the end of each.

- Where to take the measurement: This will be on the server side because it is usually used to measure a service object that is situated on the provider side.

4- *Counter*: According to WSLA this "describes the relevant information to retrieve a counter from the instrumentation of a service or managed resource" [17]. It is used to count specific events of a service. `Counter` has a `resultType` of "*integer*" and the URI is referred to using the `<MeasurementURI>` tag.

`Counter` and `Gauge` are added in the latest version of the WSLA specification. This is because, at first, WSLA had measurement directives that were web service specific. However, the authors found that they needed a generic counter and gauge to specify any metric required.

- Semantics: This measurement also corresponds to the *throughput* of an operation as in *InvocationCount*.

- When to take the measure: The counter is checked during the interval and retrieved at the end of it.

- Where to take the measure: This will be taken at the server side because the objects whose throughput is measured are situated on the provider side.

5- *ResponseTime*: This is a well-known QoS metric. One of its different definitions will be chosen. The syntax of `ResponseTime` has a `resultType`= "*double*" and the URI is specified inside a `<MeasurementURI>` tag.

- Semantics: Response time, RT, can be looked at from different perspectives. For example, it may be considered as processing time, PT:

$$RT = PT$$

The work in [129] considered response time, RT, as:

$$RT = NL + PT,$$

where NL is network latency. The authors in [130] considered a new latency, called client latency, which was added to the previous one. Accordingly, the response time is defined as:

$$RT = CL + NL + SL,$$

where, CL is client latency, and SL is server latency (i.e. processing time).

In this methodology, the decision is made to use processing time (service latency) only as response time; hence, no network and client latencies are considered here. That is because most often the provider has no control over the rest. This choice of monitoring semantic for response time makes it easier to parameterise the model when predicting.

- When to take the measurement: The response time is checked at instant of time.

- Where to take the measurement: Since response time is chosen to be the processing time, it is checked at the server side.

6- *DownTime*: WSLA defines this as it gives a direct reading of the total time throughout which the system is considered to be at down status [17]. This measurement has a `resultType` of "*double*". `Downtime` does not specify any URI.

- Semantics: This measurement is the well-known *down Time* QoS attribute. It can be defined as [126]:

$$DT = TT - UT,$$

where, DT is the down time, TT is the total observed time, and UT is uptime during the total time. Using availability probability,A, down time can be written as:

$$DT = \frac{UT}{A} - UT$$

- When to take the measurement: This down time is checked during the interval and retrieved at the end of it.

- Where to take the measurement: As in *StatusRequest*, the down time due to the network being down is not considered. For this reason, down time will be checked at the server side.

4.5.2 Mathematical Definition of WSLA Function Semantics

WSLA functions are represented formally by assigning a mathematical definition for each of them. This is described in what follows.

1- Time Series Constructor: Most of the time, this function is the first function to be applied on `MeasurementDirective`. Furthermore, its output forms the basis for WSLA's statistical functions as they are used to store measurement values according to a specific `Schedule` so that additional computations can be performed on them easily.

A WSLA `TSConstructor` function creates a time series of a specific size. Each element in this series is a single measurement or function, evaluated at time specified in the schedule that it depends on. Formally, this can be defined as:

Definition 11 *A WSLA* `TSConstructor` *function, denoted as TSConstructor, creates a series of a specific size,* $w \in \mathbb{N}_{\geq 0}$*. Each element in this series is a single measurement,* $m \in M$*, possibly a result of a function,* $F_{m,i} \in F_m$*, evaluated at time,* $t_j \in sch, j = 1 \ldots k, k \in \mathbb{N}_{\geq 0}$*.*

$$TSConstructor(h, w, sch) = \{h(t_j), \ldots, h(t_{j+w})\}, j + w \leq k, h \in M \cup F$$

2- Queue Constructor: A WSLA `QConstructor` function creates a collection of values of a specific size, $w \in \mathbb{N}_{\geq 0}$, without depending on a schedule. Alternatively, the values are pushed by events that take these values from a uri and put them in the queue.

Definition 12 *A WSLA* `QConstructor` *function, denoted as QConstructor, creates an array of specific size,* $w \in \mathbb{N}$*. Each element in this array is a single measurement or function evaluated using some triggering events.*

$$QConstructor(h, w) = \{h(i), \ldots, h(i + w)\}, i, w \in \mathbb{N}_{\geq 0}, h \in M \cup F$$

3- Time Series Select: A WSLA `TSSelect` function is applied on a time series, created by the `TSConstructor` function, to select an element of a specific index, i.

Definition 13 *A WSLA* `TSSelect` *function, denoted as TSSelect, is defined as:*

$$TSSelect(\{h(t_j), \ldots, h(t_{j+w})\}, t_i) = h(t_i),$$

where $t_i \in sch, j \leq i \leq j + w, j = 1 \ldots k, h \in M \cup F$

4- Number Greater Than Threshold: A WSLA `NumberGreaterThanThreshold` is applied on a time series output to return the total number of elements greater than a specific value, e.

Definition 14 *A WSLA* `NumberGreaterThanThreshold` *function, denoted as NGTT, is defined as:*

$$NGTT(\{h(t_1), \ldots, h(t_k)\}, e) = |\{h(t_j)|h(t_j) > e, j = 1 \ldots k\}|,$$

where $e \in \mathbb{N}_{\geq 0}, t_j \in sch, h \in M \cup F$

5- Number Less Than Threshold: A WSLA `NumberLessThanThreshold` is applied on a time series output to return the total number of elements less than a specific value, e.

Definition 15 *A WSLA* `NumberLessThanThreshold` *function, denoted as NLTT, is defined as:*

$$NLTT(\{h(t_1), \ldots, h(t_k)\}, e) = |\{h(t_j)|h(t_j) < e, j = 1 \ldots k\}|,$$

where $e \in \mathbb{N}_{\geq 0}, t_j \in sch, h \in M \cup F$

6- Percentage Greater Than Threshold: A WSLA `PercentageGreaterThanThreshold` is applied on a time series output to return the percentage of elements greater than a specific value, e.

Definition 16 *A WSLA* `PercentageGreaterThanThreshold` *function, denoted as PGTT, is defined as:*

$$PGTT(\{h(t_1), \ldots, h(t_k)\}, e) = \frac{NGTT(\{h(t_1), \ldots, h(t_k)\}, e)}{k} * 100$$

7- Percentage Less Than Threshold: A WSLA `PercentageLessThanThreshold` is applied on a time series output to return the percentage of elements less than a specific value, e.

Definition 17 *A WSLA* `PercentageLessThanThreshold` *function, denoted as PLTT, is defined as:*

$$PLTT(\{h(t_1), \ldots, h(t_k)\}, e) = \frac{NLTT(\{h(t_1), \ldots, h(t_k)\}, e)}{k} * 100$$

8- Span: A WSLA `Span` is applied on a time series to return for a specific position in the time series, the maximum length of an uninterrupted sequence of a value, e, ending at that position.

Definition 18 *A WSLA* `Span` *function, denoted as* Span*, is defined as:*

$$Span(\{h(t_1), \ldots, h(t_k)\}, e) = Max(s_1, \ldots, s_k),$$

where Max is the maximum function and $s_j, j = 1, \ldots, k$ *is defined as follows:*

$$s_j = \begin{cases} u & \text{if } h(t_j) = e \wedge \ldots \wedge h(t_{j-u+1}) = e \wedge h(t_{j-u}) \neq e, \\ & \text{with } e < u < j \\ 0 & \text{if } h(t_j) \neq e \\ j & \text{otherwise (that is, } h(t_1) = e, \ldots, h(t_j) = e) \end{cases}$$

9- Mean: A WSLA `Mean` is the well-known arithmetic mean applied on a time series to return its mean.

Definition 19 *A WSLA* `Mean` *function, denoted as* Mean *is defined as:*

$$Mean(\{h(t_1), \ldots, h(t_k)\}) = \frac{\sum_{i=1}^{k} h(t_i)}{k}$$

10- Median: A WSLA `Median` is the well-known arithmetic Median for ungrouped data applied on a time series to return its median.

Definition 20 *A WSLA* `Median` *function, denoted as Median, of the ordered series* $h(t_{z1}), \ldots, h(t_{zk}), h(t_{z1}) \leq h(t_{zj}) \leq h(t_{zk})$ *of size k is defined as:*

$$Median(\{h(t_{z1}), \ldots, h(t_{zk})\}) = \begin{cases} h(t_{\frac{k+1}{2}}) & \text{if } k \text{ is odd,} \\ \\ \frac{h(t_{\frac{k}{2}}) + h(t_{\frac{k}{2}+1})}{2} & \text{if } k \text{ is even.} \end{cases}$$

where $0 \leq z1, zj, zk \leq k$.

11- Size: A WSLA `Size` is applied on a time series to return its size.

Definition 21 *A WSLA* `Size` *function, denoted as Size, is defined as:*

$$Size(\{h(t_1), \ldots, h(t_k)\}) = |\{h(t_1), \ldots, h(t_k)\}| = k$$

12- Sum: A WSLA `Sum` is the well-known arithmetic Sum applied on a time series to add its numeric elements.

Definition 22 *A WSLA* `Sum` *function, denoted as Sum, is defined as:*

$$Sum(\{h(t_1), \ldots, h(t_k)\}) = \sum_{j=1}^{k} h(t_j)$$

13- Arithmetic Functions: WSLA arithmetic functions are used to divide, add, multiply or subtract two operands. These operands can be a constant, a measurement, or a function output.

Definition 23 *A WSLA* `Minus/Plus/Divide/Multiply` *function, denoted as Minus/Plus/Divide/Multiply, are the well-known arithmetic functions:*

$$Minus(o_1, o_2) = o_1 - o_2,$$

$$Plus(o_1, o_2) = o_1 + o_2,$$

$$Divide(o_1, o_2) = o_1/o_2,$$

$$Multiply(o_1, o_2) = o_1 \times o_2,$$

where $o_1, o_2 \in M \cup F \cup V_f$, *where* $v_f \in V_f$ *is an integer value* $\in \mathcal{N}_{\geq 0}$.

14- Maximum: A WSLA `Max` is the well-known maximum function that returns the maximum value in a series.

Definition 24 *A WSLA* `Max` *function, denoted as Max, is defined as:*

$$Max(\{h(t_1), \ldots, h(t_k)\}) = h(t_a), \text{ for some } t_a \in \{t_1 \ldots t_k\},$$

where $h(t_a) \geq h(t_i), \forall t_i \in \{t_1, \ldots, t_k\}$

15- Mode: A WSLA `Mode` is the well-known mode function applied on the time series to return the most frequently occurring value within it.

Definition 25 *A WSLA* `Mode` *function, denoted as Mode, is defined as:*

$$Mode(\{h(t_1), \ldots, h(t_k)\}) = h(t_a), \text{ for some } t_a \in \{t_1 \ldots t_k\},$$

where $freq(h(t_a)) \geq freq(h(t_i)), \forall t_i \in \{t_1, \ldots, t_k\}$ *and* $freq(h(t_i))$ *is the frequency with which the item* $h(t_i)$ *exists in the series.*

16- Rate of Change: A WSLA `RateOfChange` is applied on a time series and returns a new time series containing the rate at which their values have changed.

Definition 26 *A WSLA* `RateOfChange` *function, denoted as RoC, is defined as:*

$$RoC(\{h(t_1),\ldots,h(t_k)\},e) = \{\frac{h(t_2)-h(t_1)}{t_2-t_1},\ldots,\frac{h(t_k)-h(t_{k-1})}{t_k-t_{k-1}}\}, e = t_k$$

17- Round: A WSLA `Round` is the well-known round function that returns the decimal number rounded to a specific decimal place, d.

Definition 27 *A WSLA* `Round` *function, denoted as Round*[1]*, is defined as:*

$$Round(n,d) = \begin{cases} \frac{[n\times 10^d]}{10^d} & \text{if } (10\times((n\times 10^d)-([n\times 10^d]))) < 5 \\ \\ \frac{[n\times 10^d]+1}{10^d} & \text{if } (10\times((n\times 10^d)-([n\times 10^d]))) \geq 5 \end{cases}$$

All WSLA functions return a single value, $r \in \mathbb{R}$, except for time series constructors *TSConstructor* and *QConstructor*, and *RoC* functions. However, the results of these functions are always manipulated by another WSLA function to produce a single value that represents an *slap* value. According to this, applying the set of all the WSLA functions available for one *slap* will produce a single value $r \in \mathbb{R}$.

4.6 Related Work

The first requirement for addressing the semantic ambiguity of WSLA contract, outlined in Section 4.2, is addressed by defining the formal representation of the structure of WSLA elements through mathematical notations using tuples. The second requirement is carried out by defining monitoring-related semantics to WSLA elements. This is achieved by using mathematical formulae representing WSLA functions and by assigning semantics for WSLA measurement directives.

Semantic ambiguity exists in XML-based SLA specifications other than WSLA such as the Web Service Offerings Language (WSOL) [131]. Some of these SLA languages have made steps towards resolving this issue in different ways. For example, SLAng achieves precise semantics by modelling its structure using UML[2] and OCL constraints[3]. Another attempt to add rigid semantics to an SLA specification in the literature is by incorporating ontologies that add formal semantics

[1] $[m]$ is used to refer to the integer part of the number m.
[2] http://www.uml.org/
[3] http://www.omg.org/spec/OCL/2.0/

to a Web Service (WS) description model [44]. In addition, the authors in [132] expressed WS-agreement schema using the Web Ontology Language (OWL) [133] and SLO constraints using the Semantic Web Rule Language (SWRL)[1] as a rule language. Since OWL cannot express a relationship between properties, the work in [41] proposed a new semantic-enabled SLA model for SLA monitoring using OWL-S ontology for a web service. In this model, the SLO expressions are written using SWRL to allow both a service provider and a customer to have a common understanding when building the contract and to allow the model to be read by a machine automatically. OWL-S was used in this model to describe terms such as SLA parameters, measurements, functions, and service operations. However, the predicate is defined using SWRL rules to identify violating conditions and the correction actions. Although OWL-S was designed to describe the functional requirements for a web service, it has limited ability in terms of describing QoS metrics. For this reason, OWL-Q was created to complement OWL-S in describing QoS metrics, where QoS guarantees are represented using SWRL [134].

One of the main reasons for adding semantics to the aforementioned SLA contracts and QoS definitions was to use them in QoS matching where several QoS offers are compared for equality. Another reason for adding semantics to describe a web service was to use them in web service discovery, replacing the UDDI's syntactic discovery [135]. Also, modelling SLO constraints using SWRL rules was used in achieving automatic SLA monitoring, as in the work in [41].

4.7 Conclusion

This chapter addresses the problem of the semantic ambiguity of WSLA elements. To eliminate this ambiguity, this chapter provides first a formal representation of the structure of the main elements. Second, it adds semantics to WSLA measurement directives whose meaning might be vague for the service provider and consumer; finally, it adds mathematical definitions to WSLA functions that suit the monitoring case. The contribution of this chapter is to allow for a better understanding of the structure and semantics of WSLA which is fundamental for the mapping process. In the next chapter, this formal definition of WSLA's structure and semantics is utilised as the basis for the mapping process to SDES.

[1] http://www.w3.org/Submission/SWRL/

Chapter 5

Formal Mapping of WSLA Contracts on SDES Models

This chapter covers the second fold of the WslaCP methodology, outlined in Section 3.3.1, to implement theoretically the last six phases of the SlaCP methodology that were described in Section 3.2.1. WslaCP implementation of the first phase of the SlaCP (i.e. the SLA Interpretation phase) was described in Chapter 4. This was achieved by formalising the structure of WSLA elements and adding mathematical semantics to them. The remaining six phases (i.e. the SLA-Model Mapping, Model Completion, Model Specialisation, Model Solving, Metric Composition, and Decision) are implemented and described from a theoretical point of view in this chapter. This description is presented from the perspective of the SLA-Model Mapping phase because this phase contains most of the theoretical contributions. Another reason behind embedding the implementation of the phases in one mapping process is to provide better integrity and continuity in terms of the flow of the information, and also because the rest of the subtle and fine-grained details of all phases relate mostly to the practical implementation of the methodology and are hence described in detail in the information on the tool implementation in Chapter 6. The mapping process is considered also as a central point because the WslaCP methodology does not offer any newly invented solving algorithm; rather, it depends on an existing one and hence no theoretical contribution has been made regarding this particular issue.

The main contribution of this chapter is the detailed formal mapping of WSLA contracts on SDES models which is the core of the WslaCP methodology. This includes the mapping of WSLA operations, measurement directives and time constraints on SDES primitives, the mapping of WSLA functions on the model output (i.e. on the random variables representing the output of the SDES model), and finally the mapping of the SLO expression as an evaluation function that produces

the SLA compliance probability.

The chapter is organised as follows. In Section 5.1, an outline of the steps required in the mapping process is presented, each mapping step is described in Section 5.2 in detail, a discussion is provided in Section 5.3 and finally, the chapter concludes with Section 5.4.

5.1 Outline of the Mapping Process

In the WslaCP methodology, WSLA elements need to be mapped firstly on an abstract stochastic model description before being translated into a user-chosen modelling formalism. For this reason, the WslaCP methodology uses an SDES formalism as a canonical form for the stochastic model of the service. The SDES formalism generalises the notation of the primitives that are used usually in a stochastic model, such as the set of reward variables as RV, the set of state variables as SV, and the set of actions as A (this was presented in Section 2.5.1).

The mapping process from WSLA to SDES is depicted in Figure 5.1. This figure consists of two gray rectangles. The former represents the **Formal Semantics** of the elements in the **WSLA Document** (this is the output of the SLA Interpretation phase), while the latter represents the components used in the **Formal Mapping**, which are the **SDES Model**, the **SDES Reward Variable** and the **SDES Model**

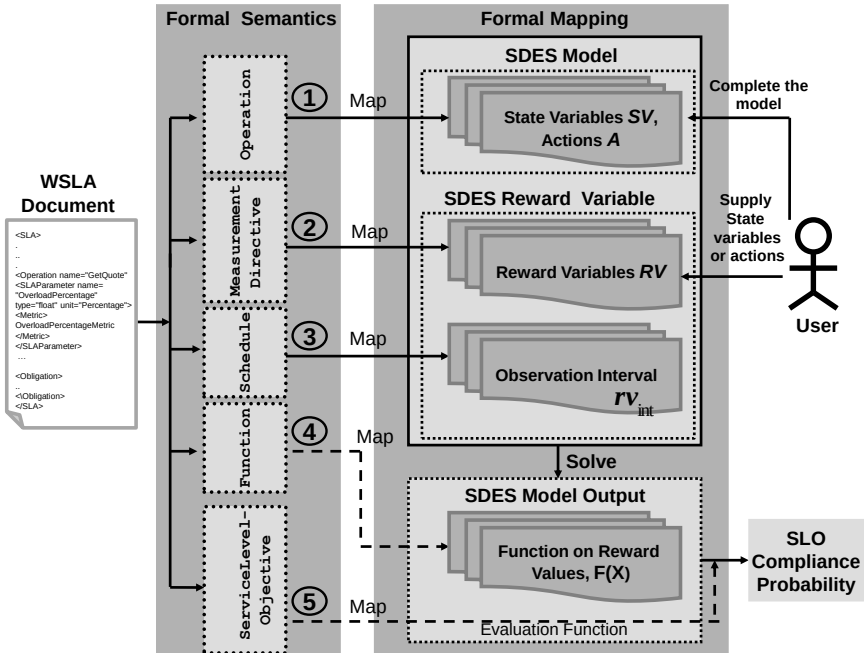

Figure 5.1: The mapping process from WSLA to SDES

Output. The first two components have been placed inside a solid-line rectangle because they are related to the actual service model while the latter is related to the output of solving this model.

The mapping process consists of five steps (these are numbered in the figure) that define a set of mapping rules specifying how different WSLA elements are mapped to SDES. These are outlined in what follows and then they are described in detail in the next section.

Step 1: `Operation` mapping. It provides a systematic translation of WSDL operations into the SDES model primitives $sv \in SV$ and $a \in A$.

Step 2: `MeasurementDirective`(s) Mapping. It provides a systematic translation of all measured metrics into the SDES reward variables RV.

Step 3: `Schedule` Mapping. It provides a systematic translation to obtain the set of observation intervals, rv_{int}, of the reward variable. This can be a set of observation intervals that are either instants or intervals of time.

Step 4: `Function`(s) Mapping. Each WSLA function is associated to a mathematical semantic suitable to the stochastic nature of the model output, $F(X)$, in order to specify further the reward variables in SDES.

Step 5: `ServiceLevelObjective` Mapping. The outcome of this mapping is an evaluation function which allows SLO compliance probability to be produced, i.e., it determines the probability with which the service level agreed to has been met.

Steps 1, 2 and 3 of this mapping process map WSLA elements to the actual SDES primitives (solid arrows in Figure 5.1), while steps 4 and 5 map WSLA elements to the model output represented as random variable X (the dashed arrow in Figure 5.1).

All the aforementioned steps are intended to be automated as much as is possible. However, some manual steps are still required. As can be seen in Figure 5.1, the mapping must be aided in its first and second steps by the **User** to complete the SDES model creation and to supply the right state variables/actions that are necessary to define the reward variable. This user interference is pointed out in Section 5.2.2.

Although this outline promises several mapping steps, it does not include information regarding the SlaCP SLA-Model Mapping phase only. However, all the theoretical concepts related to the remaining six phases are described implicitly. When describing each mapping step in Section 5.2, an indication is made regarding the elements required from the other phases related to this step.

5.2 The Detailed Mapping: Adding Stochastic Semantics to WSLA

This section describes the mapping steps outlined in Section 5.1. To illustrate them, the example of a stock quote service, presented in Listings 4.1 and 4.2, is utilized. This aids the understanding of the different steps and also allows the reader to narrow down the theoretical concepts.

Mapping WSLA to SDES means that WSLA elements have to be represented according to the stochastic nature of the service model rather than the monitoring nature of the SLA. Before starting the mapping description, it should be recalled that the SDES model is represented by the following tuple: $SDES = (SV, A, S, RV)$, where $rv \in RV$ is given as a tuple: $(rv_{rate}, rv_{imp}, rv_{int}, rv_{avg})$, and that the formal description of WSLA's SLO elements is given by a tuple $slo = (slap, c, v, vs, ve)$, where $slap = (M, F, sch)$. Given these tuples, the mapping of the formal WSLA elements into the SDES primitive is carried out in the following subsections.

5.2.1 Service Operation Mapping

WSLA defines an *slap* for a specific service object (find line 9 in Listing 4.2). Since the WSLA service object is mainly an operation [17], then mapping this operation, *op* (defined in Definition 8), to SDES will be carried out according to the SLA-Model Mapping phase, presented in Section 3.2.1. This is as follows:

op is mapped formally as an action $a_{op} \in A$, and a state variable $sv_{op} \in SV$, connected to the input of this action.

Recalling the example in Listing 4.2, the mapping of the service operation, named `GetQuote`, is a state variable, $sv_{GetQuote} \in SV$, and an action, $a_{GetQuote} \in A$.

If all service objects available in a WSLA document are mapped as pairs of state variables/actions, a part of the SDES model, consisting of a set of these pairs, can be produced automatically for the user.

5.2.2 MeasurementDirective(s) Mapping

The formal set that contains WSLA's seven measurement types was described in Section 4.4.3 as follows:

$$\mathcal{M} = \{Status, StatusRequest, Counter, Gauge, ResponseTime, DownTime, InvocationCount\}$$

A measurement, $m \in \mathcal{M}$, is the core unit in computing an `SLAParameter` *slap*

value. Hence, its value should be predicted first. Since no runtime measurement information is available before a service's deployment, the measurement directive should be added to the model of this service to produce its expected value. Having said this, the methodology maps WSLA measurement directives into SDES reward variables. This includes the determination of three points:

1. The type of reward variable, impulse or rate.
2. The content of its reward function.
3. The time to solve this reward variable, instant or interval.

For determining *the type of reward variable* that represents a measurement directive, typically, reward variables of a stochastic model are used for predicting some performance attributes, as described in Section 4.5.1. Hence, each measurement directive is expressed as a rate/impulse reward variable in a way suits its semantics. This mapping is referred to formally using the function $MtoRV$ as follows:

$$MtoRV : \mathcal{M} \rightarrow RV$$

$$\forall m \in \mathcal{M} : MtoRV(m) = rv \in RV, rv = (rv_{rate}, rv_{imp}, rv_{int}, rv_{avg}),$$

where rv_{rate} defines the rate reward function if the reward variable rv is rate-based, while rv_{imp} defines the impulse reward function otherwise. These reward functions specify respectively the rewards earned if the model spends time in a specific state or when an action has fired. The type of reward variable of each measurement directive will be defined when presenting its mapping.

Regarding the determination of *the content of the reward function*, after deciding the type of reward variable, its reward function has to be specified. Because a rate/impulse reward function depends on the model state variables/actions respectively, a correlation to the right primitives in the service model has to be made in order to build a correct function. Since *slap* is defined in the WSLA per operation, its measurement is related to this operation. Accordingly, the reward function will relate either to the state variable a_{op} or the action sv_{op} that results from this operation mapping. Hence, even though a complete model does not exist yet, the state variables and actions generated may be used to build the reward function. For example, *InvocationCount* is related to the a_{op} action, as discussed later in Section 5.2.2.2, and its reward function returns 1 whenever this action fires. Although it is reasonable to relate a reward function to a produced model primitive in the way described earlier, other measurements may not depend on these primitives directly.

These can be related to primitives that influence a_{op} or sv_{op}. For example, *Status-Request* is related to the ability of the action a_{op} to fire; hence, its reward function may be linked to state variables that inhabit the firing of this action. In this case, the user has to specify the primitives needed. Otherwise, a URI of a measurement directive may be used to indicate the action or the state variable required for this function. All the previous aspects are discussed when each measurement mapping is presented.

Concerning the determination of *the time to solve the reward variables,* after deciding what type of reward variable can represent a measurement, the time at which this variable has to be evaluated (rv_{int}) should be specified. Usually, reward variables are collected, as defined in [29, 97], at an instant of time, an interval of time, or a time-averaged interval of time. The boundary of these intervals can stretch to infinity, producing a steady state measure. Given these types of time evaluation, which of these best suits a measurement prediction should be specified. WSLA, being monitoring-centred, makes statements about values observed at regular time instants that are asynchronous in respect to the system, rather than about states/transitions of the underlying system or the stochastic model of that system. Consequently, the model has to evaluate a reward variable for every instant when an observation is

Table 5.1: Summary of Mapping MeasurementDirective(s) to SDES reward variables

WSLA to SDES Mapping		**WSLA Measurement Directives**					
		Status/ Status-Request	**Invocation Count**	**Gauge**	**Counter**	**Response Time**	**Down Time**
SDES Primitives	**rv type**	rv_{rate}	rv_{imp}	rv_{rate}	rv_{imp}	rv_{rate}	rv_{rate}
	rv_{int} [lo, hi]	$lo = hi$	$lo < hi$	$lo = hi$	$lo < hi$	$lo = hi$	$lo < hi$
	rv_{avg}	*False*	*False*	*False*	*False*	*False*	*False*
	sv/a Hint for reward function	$sv_{up} = 1$ or user defined	a_{op}	sv_{op} or user defined	a_{op} or a_{uri} or user defined	sv_{end} or user defined	$sv_{up} = 0$ or user defined
	sv/a Hint for SDES model	sv_{up}, sv_{down} a_{fail}, a_{repair} or user defined	a_{op} already defined	sv_{op} already defined	a_{op} already defined or a_{uri}	sv_{end} or closed model	sv_{up}, sv_{down} a_{fail}, a_{repair} or user defined
	User Input	availability condition	automated	automated	automated or a_{uri}	automated or sv_{end}	availability condition

made. According to the type of measurement, the evaluation can happen exactly at a specified instant, or be accumulated between instants. Hence, the reward variable could be an instant $[lo, hi], lo <= i$ or an interval $[lo, hi], lo < hi$ of times with $rv_{avg} = false$. This is described when each measurement mapping is presented.

Table 5.1 summarises all the information needed regarding the mapping of all the WSLA measurement directive types into SDES reward variables. This information, given the aforementioned discussion of the three points regarding reward variable specification, includes: the reward variable type; the type of evaluation interval rv_{int} and whether it is averaged or not rv_{avg}; the hint provided for the automatic construction of the reward function; and the hint for the SDES model's automatic creation. The input of the user for completing each reward function definition is also pointed out. This information is presented in detail in the following subsections where an unambiguous mapping from each measurement, m, to a reward variable, rv, in SDES is provided.

5.2.2.1 StatusRequest and Status

StatusRequest gives 1 if the system is up and 0 otherwise, while `Status` gives a true/false value [17]. This difference is not important when modelling; hence, in the methodology, they are treated identically.

- Reward variable type and function: In Section 4.5.1, *StatusRequest* is defined as the status of an *Availability* of the service operation. For this reason, this measurement is mapped as a rate reward variable that returns 1 while the service is in an up state and 0 otherwise. If $\Sigma^* \in \Sigma$ is the set of system states under which the SDES model is considered to be up and working, then the reward function template is as follows:

$$rv = \begin{cases} rv_{imp}(a) = 0 \quad \forall a \in A \\ rv_{rate}(\sigma) = \begin{cases} 1 & \text{if } \sigma \in \Sigma^* \\ 0 & \text{otherwise} \end{cases} \end{cases}$$

- Evaluation Interval: WSLA retrieves this measurement at specific time instances. Hence, it is represented as instant of time reward variable: i.e., $rv_{int} = [lo, hi]$, with $lo = hi$ & $rv_{avg} = False$.

- Hint for reward function: The assumption is that the reward function refers to the status of the availability of a WSDL operation represented as a_{op} (or the service status as a whole). The states under which this operation is working cannot be derived automatically from WSLA; hence, this operation is user-defined and no hint can be given. However, if the indication that will be specified in the

hint for the SDES model is used, then $sv_{up} = 1$ can be used in the reward function.

- Hint for SDES model: This indicates a need to include service up/down states in the model. If a simple failing/repairing mechanism is desired, this can be automated by including two state variables, $sv_{up} = 1, sv_{down} = 0 \in SV$, to indicate the up/down states, and two actions, $a_{fail}, a_{repair} \in A$, to reflect the fail/repair procedures. Then, sv_{up} should be connected to a_{op} to prevent its firing when this place is empty. If a more complicated up/down mechanism is desired, the user has to specify it manually.

- User input: The user has to specify the system states Σ^* that correspond to an available service operation.

An example of mapping *StatusRequest* that exists in Listing 4.2 is as follows:

$$rv = \begin{cases} rv_{imp}(a) = 0 \quad \forall a \in A \\ rv_{rate}(\sigma) = \begin{cases} 1 & \text{if } \sigma \in \Sigma^* \\ 0 & \text{otherwise} \end{cases} \\ rv_{int} = [lo, hi], lo = hi \\ rv_{avg} = false. \end{cases}$$

The template of the above reward function is produced automatically and the user has to define Σ^*, the service's up states. If the service is working when the number of CPUs is greater than three, then the user has to specify: the state variable reflecting this number; the relation " $>$ "; and the value 3. In the SDES model, this equates to the condition $sv_{noOfCPU}(\sigma) > 3, \forall \sigma \in \Sigma$. If the the hint for the SDES model is correct, then the condition $sv_{up}(\sigma) = 1$ will be specified automatically.

5.2.2.2 InvocationCount

This is mapped to SDES as follows:

- Reward variable type and function: In Section 4.5.1, *InvocationCount* was defined as the throughput of a service operation. Since an impulse reward is used to reflect the counting mechanism for a specific action, using impulse reward is more natural for describing this measurement than a rate reward. Accordingly, the natural manner in which to define *InvocationCount* in SDES is to associate an impulse reward of value 1 each time the action, $a_{op} \in A$, that represents the WSDL operation is fired. Accordingly, the reward function template will be defined as

follows:

$$rv = \begin{cases} rv_{rate}(\sigma) = 0 \quad \forall \sigma \in \Sigma \\ rv_{imp}(a) = \begin{cases} 1 & \text{if } a = a_{op} \\ 0 & otherwise \end{cases} \end{cases}$$

- Evaluation Interval: The evaluation of the reward variable at instant of time is not realistic for impulse reward because the action may complete a short time before or after the instant [97]. The interpretation of WSLA's usage of this measurement is the increment in the number of the service invocations from one reading to the next [17]. For this reason, there is a need to count all the firings of an action from the last observed instant until the currently observed one. Hence, the reward variable should be evaluated as an interval of time reward variable between specific instants to keep track of the increment in the invocation counting, i.e., $rv_{int} = [lo, hi]$, with $lo < hi$, & $rv_{avg} = False$.

- Hint for reward function: This includes the action that represents the WSDL operation, a_{op}, which has already been specified in Section 5.2.1. This is because the invocation of the service is reflected by this action.

- Hint for SDES model: The inclusion of the action, a_{op}, is already specified in Section 5.2.1. For this reason, no additional hint can be provided.

- User input: Since this measurement refers to the WSDL operation, the methodology assumes that the reward function refers to the action, a_{op}. However, if the user has built the model from scratch and has assigned a different name to the action representing this operation, then he/she has to specify this action as a_{op}. Hence, unlike *StatusRequest* where the availability condition should be modelled and specified, the throughput can be retrieved automatically when modelling the service operation since there are no other constraints that have to be modelled that may affect it.

5.2.2.3 Gauge

This is mapped to SDES as follows:

- Reward variable type and function: In Section 4.5.1, *Gauge* was defined as returning the current value or queue size of a service entity. Hence, it can be mapped as a reward variable that returns the current value of an SDES primitive. In essence, *Gauge* corresponds to the current value of a state variable and, in SDES terms, rate as well as impulse rewards can add to it. The reward definition is then unrestricted, and the user can assign any rewards to the gauge. However, the methodology provides a special gauge, corresponding to a single state variable representing the gauge value (which is usually the case). Depending on the model

at hand, this simplifies the job of the user. If it is assumed that the state variable that holds the number of a particular task in the service is $sv_{op} \in SV$, as defined in Section 5.2.1, with $sv_{op}(\sigma)$ being the value of sv_{op} in a state σ, then the reward function template is defined as:

$$rv = \begin{cases} rv_{imp}(a) = 0 \quad \forall a \in A \\ rv_{rate}(\sigma) = \begin{cases} sv_{op}(\sigma) & \forall \sigma \in \Sigma \\ 0 & otherwise \end{cases} \end{cases}$$

- Evaluation Interval: The reward variable is an instant of time variable to give the current value of a service entity represented in a state variable, i.e., $rv_{int} = [lo, hi]$, with $lo = hi$ & $rv_{avg} = False$.

- Hint for reward function: This includes the state variable sv_{op}, which represents the requests queuing for the WSDL operation; this has already been specified in Section 5.2.1.

- Hint for SDES model: The state variable sv_{op} is already specified. For this reason, no additional hint can be provided.

- User input: Since this measurement refers to the WSDL operation, the methodology assumes that the reward function refers automatically to the state variable sv_{op} retrieved in Section 5.2.1. However, if this state variable is not the relevant one or if the user built the model and assigned a different name to the state variable representing the incoming requests to the operation, then the user should choose or introduce a state variable sv_{op}.

5.2.2.4 Counter

This is mapped to SDES as follows:

- Reward variable type and function: In Section 4.5.1, *Counter* was defined as being related to the throughput of a service operation. For this reason, it can be mapped as an impulse reward variable of an action $a_i \in A$ in the model as in *InvocationCount*. The only difference is that it can refer to any action in the model.

$$rv = \begin{cases} rv_{rate}(\sigma) = 0 \quad \forall \sigma \in \Sigma \\ rv_{imp}(a) = \begin{cases} 1 & \text{if } a = a_i \\ 0 & otherwise \end{cases} \end{cases}$$

- Evaluation Interval: This reward variable is an interval of time reward variable, i.e., $rv_{int} = [lo, hi]$, with $lo < hi$ & $rv_{avg} = False$.

- Hint for reward function: This includes an action that represents the WSDL operation a_{op} which was already specified in Section 5.2.1. If this action is not the

intended one, the measurement URI can be used as a hint to the required action that performs a special function. In the example below, $ipPacketsIn$ hints to choose an action that indicates an IP packet arrival.

```
<MeasurementDirective xsi:type="wsla:Counter" resultType="double">
 <MeasurementURI>http://support1.com/ipPacketsIn</MeasurementURI>
<MeasurementDirective>
```

- Hint for SDES model: No additional hint can be provided since the action a_{op} is already specified. However, if the action a_{op} is not the one aimed to measure its counter, a URI value can hint to add an action a_{uri} that represents it.

- User input: By default, the reward function refers automatically to the action representing the WSDL operation, in this case, $a_i = a_{op}$ so no input is required. However, if this is not the case, the user has to specify the action a_i.

5.2.2.5 ResponseTime

This is mapped to SDES as follows:

- Reward variable type and function: To express the response time in terms of rewards, different methods can be used. For example, an additional state variable $sv_{end} \in SV$ can be added to the SDES model to signal the receipt of the response. In this case, sv_{end} is initially set to 0 and can jump to 1 once only, indicating the response has been received. Then, $RT(t)$, the probability that the response time is less than t, is equal to the probability that the state variable is 1 at time t. Hence, the response time of an operation is determined by checking, at each time instance, if the state variable equals 1. That is:

$$RT(t) = P(response\,time \leq t) = P(sv_{end}(\sigma) = 1\ at\ time\ t)$$

This is represented by using a rate-based reward function such as:

$$rv = \begin{cases} rv_{imp}(a) = 0 \quad \forall a \in A \\ rv_{rate}(\sigma) = \begin{cases} 1 & \text{if } \sigma \in \Sigma^* \\ 0 & \text{otherwise} \end{cases} \end{cases}$$

where Σ^* represents all system states σ and where $sv_{end}(\sigma) = 1$. The response time distribution is then computed by determining the expected reward at time t. This leaves the user with one complicating factor, one that is well-known when computing response times: response times computed in above manner depend on the initial state. Often it is appropriate to take the steady-state distribution as the initial state, but this depends on the circumstances. Hence, it can be left to the

user to set an appropriate initial state. This is not completely satisfactory, since it requires the user to possess good modelling judgment.

Another way of computing response time is when the model is closed. In this case, the state variable sv_{op} can be used as an indicator of whether a response has been returned. However, this may be useful if the model has only one user or under an assumption that the first request leaving sv_{op} is the first one to return to it.

- Evaluation Interval: This reward variable is an instant of time reward variable to check the service response at this instant, i.e., $rv_{int} = [lo, hi]$, with $lo = hi$ & $rv_{avg} = False$.

- Hint for reward function: This refers to the response time of the WSDL operation a_{op}. Hence, if the first type of computing response time is used, the hint that can be offered to the user is to use the additional state variable sv_{end} that reflects the receipt of the response. However, if the closed model response time is used, the state variable sv_{op}, specified in Section 5.2.1, can be used in the reward function to indicate the receipt of the response.

- Hint for SDES model: As specified in the reward function's hint, this measurement can indicate that the user should add a state variable that reflects the response receipt or should consider a closed model of the service.

- User input: By default, the reward function can also refer to the sv_{op} if the closed model is used. However, if this is not the case, the user has to specify the state variable $sv_{end} = 0$ in the model and set to 1 when the response is received; the user also has to determine an appropriate initial state for the model.

5.2.2.6 DownTime

This is mapped to SDES as follows:

- Reward variable type and function: Since *DownTime* gives the total down time of an operation, the mapping is similar to that of *Status*, but is measured as an interval of time rather than an instant of time.

$$rv = \begin{cases} rv_{imp}(a) = 0 \quad \forall a \in A \\ rv_{rate}(\sigma) = \begin{cases} 0 & \text{if } \sigma \in \Sigma^* \\ 1 & \text{otherwise} \end{cases} \end{cases}$$

where $\Sigma^* \in \Sigma$ is the set of system state under which this SDES model is considered to be up. Hence, this reward function returns 0 if the model resides in one of these up states.

- Evaluation Interval: This reward variable is an interval of time reward variable to check the system's down period. This means: $rv_{int} = [lo, hi]$, with $lo < hi$

and $rv_{avg} = False$.

- Hint for reward function: This is the same as the one defined in Section 5.2.2.1. However, $sv_{up} = 0$ should be used in the reward function instead.

- Hint for SDES model: This is the same as the one defined in Section 5.2.2.1.

- User input: The user has to specify the states Σ^* that correspond to an available service, as with the *StatusRequest* user input defined in Section 5.2.2.1.

The outcome of the mapping of the service operation and the measurement directives' hints (and after the user has completed the model creation) represents a complete service model that is the outcome of the SlaCP Model Completion phase.

5.2.3 Schedule Mapping

After mapping each measurement $m \in \mathcal{M}$ to a specific reward variable $rv \in RV$ and determining if it is an instant or interval of time variable, what these instants/intervals are needs to be determined. For this reason, it is of particular interest to map the WSLA monitoring times defined in the schedule, *sch*, to time in SDES. WSLA's schedule, *sch*, is defined in Definition 9 as a set of time points as follows:

$$sch = \{t_1, \ldots, t_k\}; t_1 = s, t_{j+1} = t_j + i, j = 1, \ldots, k-2, t_k = e$$

This is depicted in the lower part of Figure 5.2. For simplicity, the methodology considers the starting time as the zero instant $t_1 = 0$ and the end time t_k is the subtraction of the start date and the end date taken from the WSLA. This subtraction value is represented according to the smallest measures for the increment i.

These time points of WSLA are mapped onto an SDES observation interval.

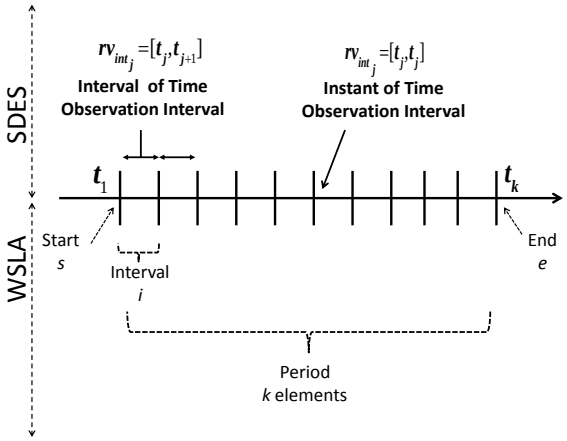

Figure 5.2: Mapping the WSLA schedule into the SDES observation interval

Since SDES has a primitive rv_{int} (defined in Definition 2) that defines a single observation interval for a reward variable $rv \in RV$, there is a need to define a set $\{rv_{int}\}$ that contains multiple observation intervals for each reward variable:

$$\{rv_{int}\} = \{rv_{int_j} | rv_{int_j} = [lo_j, hi_j]\}, \quad lo_j, hi_j \in \mathbb{R}_{\geq 0}, j = 1, \ldots, k$$

Because the reward variable $rv \in RV$ could be either an instant or interval, the boundaries lo_j, hi_j of each rv_{int_j} will vary accordingly.

In cases where $m \in \mathcal{M}$ is mapped as an instant of time reward variable, then $lo_j = hi_j$ in each observation interval rv_{int_j}. Hence, *sch* is mapped as a set of instants of time observation intervals as depicted in the right upper part of Figure 5.2. This set is written as:

$$\{rv_{int}\} = \{[t_j, t_j]\}, j = 1, \ldots, k.$$

However, if $m \in \mathcal{M}$ is mapped as an interval of time reward variable, then $lo_j < hi_j$ in each observation interval rv_{int_j}. In this case, *sch* is mapped as a set of intervals of time observation intervals where each interval is between two sequential time points in *sch*. This is as depicted in the left upper part of Figure 5.2. This set is written formally as:

$$\{rv_{int}\} = \{[t_j, t_{j+1}]\}, j = 1, \ldots, k-1.$$

An example of mapping the schedule in line number 2 of Listing 4.2 to the SDES model is as follows. Since *StatusRequest* is an instant of time reward variable, the model needs to provide instant of time results at the following points in time:

$$\{rv_{int}\} = \{[0; 0]; [1; 1]; \ldots; [44640; 44640]\}$$

Here, the unit of the increment time is in minutes (line 7); hence the start time point is 0 and the end time of 44640 is obtained by expressing one month (31 days) in minutes.

The outcome of mapping measurement directives and schedule (after translating them to a concrete stochastic model) represents the SlaCP Model Specialisation phase. This contains a complete reward model ready to be solved. For the next step, this model is assumed to be solved and its outcomes are obtained.

5.2.4 Function(s) Mapping

The formal set that contains WSLA's function types was described in Section 4.4.5 as follows:

$$\mathcal{F} = \{TSConstructor, QConstructor, TSSelect, Size, Mean, Median, Mode, Round, Sum, Max, ValueOccurs, Span, RateOFChange, PercentageGreaterThanTh-reshold, PercentageLessThanThreshold, NumberGreaterThanThreshold, NumberLessThanThreshold, ArithmeticFunction\}$$

These functions are not mapped into the actual SDES primitives, but on the results of solving the reward variables. In the previous three sections, all the elements required for preparing the reward model were discussed and hence the Model Solving phase can be conducted. During this phase, the solver has to solve the model to produce the expected values of the reward variables. These values are the input of WSLA functions which raises another difficulty when applying these functions to the values produced. This is because WSLA, being monitoring-dependent, has an input of $\in \mathbb{N}_{\geq 0}$ to most of its functions, while the output of the model is expected values or distribution. Consequently, WSLA functions must be mapped to suitable derivations from the result of the SDES model. In order to clarify what these derivations are, the mapping is provided for each of the steps, as presented in Section 4.4.6. This describes the common order in which WSLA elements are applied:

1. Since a measurement is mapped as an rv and a schedule is mapped as $\{rv_{int}\}$, then the time series constructor function in SDES evaluates the reward variable $rv \in RV$ for each evaluation interval in $\{rv_{int}\}$. This is expressed as a set:

   $$\{rv(t_1), \ldots, rv(t_k)\},$$

   where $rv(t_j)$ is the reward variable with the evaluation interval $rv_{int_j} = [t_j, t_j]$ in the case of an instant of time reward variable, and $rv_{int_j} = [t_j, t_{j+1}]$ in the case of an interval of time reward variable. In SDES, each $rv(t_j)$ can be thought of as a random variable, $X_{t_j} : \Sigma \to \mathbb{R}$. Accordingly, the previous set can be written as a set of random variables as follows:

   $$\{X_{t_1}, \ldots, X_{t_k}\}$$

2. The function $F_{m,2}$ is applied on the above set of random variables. Any function over a set of random variables results in a new random variable whose

probability distribution is determined by the probability distribution of each random variable [136].

$$X_{F_{m,2}} = F_{m,2}(X_{t_1}, \ldots, X_{t_k})$$

3. The rest of the functions $F_{m,3}, \ldots, F_{m,|F_m|}$ in the set F_m will be applied in sequence. This also results in a new random variable each time a new function is applied.

$$X_{F_{m,i}} = F_{m,i}(X_{F_{m,i-1}}), i = 3, \ldots, |F_m|$$

The random variable $X_{F_{m,|F_m|}}$ that results from applying the last function $F_{m,|F_m|} \in F_m$, represents the value of *slap*.

If WSLA functions specify an additional operand $o \in O - \{sch\}$ (as specified in Section 4.4.6), these functions after mapping to SDES can be written as: $F_{m,2}(X_{t_1}, \ldots, X_{t_k}, o), o \in \mathbb{R}$, for step 2 and as $F_{m,i}(X_{F_{m,i-1}}, o), o \in \mathbb{R}$, for step 3.

In what follows, an exact mapping of each WSLA function is described. The functions are mapped according to the mathematical definition given in Section 4.5.2 but with stochastic model semantics which is represented as a set of random variables.

1- Time Series Constructor: The time series constructor function that represents the reward variable values taken according to specified intervals is already defined in the first step presented earlier, as a set of random variables:

$$TSConstructor(rv, \{rv_{int}\}, w) = \{X_{t_j}, \ldots, X_{t_{j+w}}\},$$

where $j + w \leq k$, and X_{t_j} represents the random variable evaluated during $rv_{int_j} \in \{rv_{int}\}$. The values $TSConstructor(m, sch, w) = m(t_j), \ldots, m(t_{j+w})$ when monitoring (Definition 11) represents only one realisation $x_{i_{t_j}}, \ldots, x_{i_{t_{j+w}}}$ of $X_{t_j}, \ldots, X_{t_{j+w}}$ after mapping. If $j = 1, w = k$, then

$$TSConstructor(rv, \{rv_{int}\}, 1) = \{X_{t_k}, \ldots, X_{t_k}\},$$

The output random variables can be continuous or discrete depending on the measurement type. For example, response time and down time can take any number $\in \mathbb{R}_{\geq 0}$ while the other measurements take a discrete value. Status can take either 0 or 1, while counter, gauge, and invocation count can take any number $\in \mathbb{N}_{\geq 0}$.

2- Queue Constructor: This function is monitoring-dependent because it depends on events, that are implemented in deployment stage, to push the values inside the queue. For this reason, it is treated like the $TSConstructor$ function in the context

of WSLA compliance prediction. Hence, the events are replaced by a schedule and the scheduling choice is left to the user.

3- Time Series Select: The time series select function after mapping to SDES output represents one random variable, X_{t_j}, at an observation interval, rv_{int_j}, of the set of random variables representing the output of the $TSConstructor$.

$$TSSelect(\{X_{t_1}, \ldots, X_{t_k}\}, rv_{int_j}) = X_{t_j},$$

where $j \in \{1 \ldots k\}$. The value $TSSelect(m(t_1), ...m(t_k), t_j) = m(t_j)$ when monitoring (Definition 13) represents one realisation $x_{i_{t_j}}$ of X_{t_j} after mapping, where $i \in \{1, \ldots, n\}$ and n is the number of realisation.

4- Number Greater than Threshold: The $NGTT$ function is applied on a set of random variables to return a new random variable, X_{NGTT}, in which each element, $x_{i_{NGTT}}$, represents the number of elements greater than a threshold in the i-th realisation $\{x_{i_{t_1}}, \ldots, x_{i_{t_k}}\}$ of $\{X_{t_1}, \ldots, X_{t_k}\}$.

$$NGTT(\{X_{t_1}, \ldots, X_{t_k}\}, e) = X_{NGTT},$$

where $x_{i_{NGTT}} \in X_{NGTT}$ is given as:

$$x_{i_{NGTT}} = |\{x_{i_{t_j}} | x_{i_{t_j}} > e, j \ \in \{1, \ldots, k\}\}|, i \in \{1, \ldots, n\},$$

where n is the number of realisations.

5- Number Less than Threshold: The $NLTT$ function is applied on a set of random variables to return a new random variable, X_{NLTT}, in which each element, $x_{i_{NLTT}}$, represents the number of elements less than a threshold in the i-th realisation $\{x_{i_{t_1}}, \ldots, x_{i_{t_k}}\}$ of $\{X_{t_1}, \ldots, X_{t_k}\}$.

$$NLTT(\{X_{t_1}, \ldots, X_{t_k}\}, e) = X_{NLTT},$$

where $x_{i_{NLTT}} \in X_{NLTT}$ is given as:

$$x_{i_{NLTT}} = |\{x_{i_{t_j}} | x_{i_{t_j}} < e, j \ \in \{1, \ldots, k\}\}|, i \in \{1, \ldots, n\},$$

where n is the number of realisations.

6- Percentage Greater than Threshold: The $PGTT$ function is applied on a set of random variables to return a new random variable, X_{PGTT}, in which each element, $x_{i_{PGTT}}$, represents the percentage of elements greater than a threshold in the i-th realisation $\{x_{i_{t_1}}, \ldots, x_{i_{t_k}}\}$ of $\{X_{t_1}, \ldots, X_{t_k}\}$.

$$PGTT(\{X_{t_1}, \ldots, X_{t_k}\}, e) = X_{PGTT},$$

where $x_{i_{PGTT}} \in X_{PGTT}$ is given as:

$$x_{i_{PGTT}} = \frac{|\{x_{i_{t_j}} | x_{i_{t_j}} > e, j \ \in \{1, \ldots, k\}\}|}{k}, i \in \{1, \ldots, n\},$$

where n is the number of realisations.

7- Percentage Less than Threshold: The $PLTT$ function is applied on a set of random variables to return a new random variable, X_{PLTT}, in which each element, $x_{i_{PLTT}}$, represents the percentage of elements less than a threshold in the i-th realisation $\{x_{i_{t_1}}, \ldots, x_{i_{t_k}}\}$ of $\{X_{t_1}, \ldots, X_{t_k}\}$.

$$PLTT(\{X_{t_1}, \ldots, X_{t_k}\}, e) = X_{PLTT},$$

where $x_{i_{PLTT}} \in X_{PLTT}$ is given as:

$$x_{i_{PLTT}} = \frac{|\{x_{i_{t_j}} | x_{i_{t_j}} < e, j \ \in \{1, \ldots, k\}\}|}{k}, i \in \{1, \ldots, n\},$$

where n is the number of realisations.

8- Span:

The *Span* function is applied on a set of random variables to return a new random variable, X_{Span}, in which each element, $x_{i_{Span}}$, represents the maximum number of consecutive occurrences of a value, v, in the i-th realisation $\{x_{i_{t_1}}, \ldots, x_{i_{t_k}}\}$ of $\{X_{t_1}, \ldots, X_{t_k}\}$.

$$Span(\{X_{t_1}, \ldots, X_{t_k}\}, e) = X_{Span},$$

where $x_{i_{Span}} \in X_{Span}, i \in \{1 \ldots n\}$ is given as:

$$x_{i_{Span}} = Max(\{v_1, \ldots, v_k\}),$$

where Max is the function that return the maximum value in a set, and $v_j, j \in \{1, \ldots, k\}$ is given as:

$$v_j = \begin{cases} u & \text{if } x_{i_{t_j}} = e \wedge \ldots \wedge x_{i_{t_{j-u+1}}} = e \wedge x_{i_{t_{j-u}}} \neq e, \\ & \text{with } e < u < j \\ 0 & \text{if } x_{i_{t_j}} \neq e \\ i & \text{otherwise (that is, } x_{i_{t_1}} = e, \ldots, x_{i_{t_j}} = e) \end{cases}$$

where n is the number of realisations.

9- Mean: The *Mean* function is applied on a set of random variables to return a new random variable, X_{Mean}, in which each element, $x_{i_{Mean}}$, represents the mean of the elements in the i-th realisation $\{x_{i_{t_1}}, \ldots, x_{i_{t_k}}\}$ of $\{X_{t_1}, \ldots, X_{t_k}\}$.

$$Mean(\{X_{t_1}, \ldots, X_{t_k}\}) = X_{Mean},$$

where $x_{i_{Mean}} \in X_{Mean}$ is given as:

$$x_{i_{Mean}} = \frac{\sum_{j=1}^{k}(x_{i_{t_j}})}{k}, i \in \{1, \ldots, n\}$$

where n is the number of realisations.

10- Median: The *Median* function is applied on a set of random variables to return a new random variable, X_{Median}, in which each element, $x_{i_{Median}}$, represents the median of the elements in the i-th realisation $\{x_{i_{t_1}}, \ldots, x_{i_{t_k}}\}$ of $\{X_{t_1}, \ldots, X_{t_k}\}$.

$$Median(\{X_{t_1}, \ldots, X_{t_k}\}) = X_{Median},$$

Suppose that $x_{i_{t_{z1}}}, \ldots, x_{i_{t_{zk}}}, x_{i_{t_{z1}}} \leq x_{i_{t_{zj}}} \leq x_{i_{t_{zk}}}$, $0 \leq z1, zj, zk \leq k$, is the ordered i-th realisation, then $x_{i_{Median}} \in X_{Median}, i \in \{1, \ldots, n\}$ is given as:

$$x_{i_{Median}} = \begin{cases} x_{i_{t_{\frac{k+1}{2}}}}, & \text{if k is odd,} \\ \frac{x_{i_{t_{\frac{k}{2}}}} + x_{i_{t_{\frac{k}{2}+1}}}}{2}, & \text{if k is even.} \end{cases}$$

where n is the number of realisations.

11- Size: *Size* is applied on a set of random variables to return a new random variable, X_{Size}, in which each element, $x_{i_{Size}}$, represents the number of elements in the i-th realisation $\{x_{i_{t_1}}, \ldots, x_{i_{t_k}}\}$ of $\{X_{t_1}, \ldots, X_{t_k}\}$.

$$Size(\{X_{t_1}, \ldots, X_{t_k}\}) = X_{Size},$$

where $x_{i_{Size}} \in X_{Size}$ is given as:

$$x_{i_{Size}} = |\{x_{i_{t_1}}, \ldots, x_{i_{t_k}}, j \in \{1, \ldots, k\}\}|, i \in \{1, \ldots, n\}$$

where n is the number of realisations.

12- Sum: *Sum* is applied on a set of random variables to return a new random variable, X_{Sum}, in which each element, $x_{i_{Sum}}$, represents the summation of elements in the i-th realisation $\{x_{i_{t_1}}, \ldots, x_{i_{t_k}}\}$ of $\{X_{t_1}, \ldots, X_{t_k}\}$.

$$Sum(\{X_{t_1}, \ldots, X_{t_k}\}) = X_{Sum},$$

where $x_{i_{Sum}} \in X_{Sum}$ is given as:

$$x_{i_{Sum}} = \sum_{j=1}^{k} (x_{i_{t_j}}), i \in \{1, \ldots, n\},$$

where n is the number of realisations.

13- Arithmetic Functions: Arithmetic functions are used when predicting to divide, add, multiply or subtract two operands. These operands can be a random variable X_{t_j}, a constant, or a function output. If arithmetic functions are applied on two random variables, they return a new random variable, $X_{ArithmeticFunction}$, in which each element, $x_{i_{ArithmeticFunction}}$, represents the result from applying the arithmetic function on the i-th realisation of each random variable i.e. $\{x_{i_{t_j}}, x_{i_{t_w}}\}$ of $\{X_{t_j}, X_{t_w}\}$. For example, if the arithmetic function is *Add*, it is applied on two random variables as follows:

$$Add(X_{t_j} + X_{t_w}) = X_{t_j} + X_{t_w} = X_{Add},$$

where $x_{i_{Add}} \in X_{Add}$ is given as:

$$x_{i_{Add}} = (x_{i_{t_j}} + x_{i_{t_w}}); j, w \in \{1, \ldots, k\}; i \in \{1, \ldots, n\},$$

where n is the number of realisations.

14- Maximum: The Max function is applied on a set of random variables to return a new random variable, X_{Max}, in which each element, $x_{i_{Max}}$, represents the maximum of the elements in the i-th realisation $\{x_{i_{t_1}}, \ldots, x_{i_{t_k}}\}$ of $\{X_{t_1}, \ldots, X_{t_k}\}$.

$$Max(\{X_{t_1}, \ldots, X_{t_k}\}) = X_{Max},$$

where $x_{i_{Max}} \in X_{Max}, i \in \{1 \ldots n\}$ is given as:

$$x_{i_{Max}} = Max(\{x_{i_{t_1}}, \ldots, x_{i_{t_k}}\}) = x_{i_{t_a}}, \text{ for some a } \in \{1, \ldots, k\},$$

where, $x_{i_{t_a}} \geq x_{i_{t_j}}, \forall j \in \{1, \ldots, k\}$, and n is the number of realisations.

15- Mode: $Mode$ is applied on a set of random variables to return a new random variable, X_{Mode}, in which each element, $x_{i_{Mode}}$, represents the mode of the elements in the i-th realisation $\{x_{i_{t_1}}, \ldots, x_{i_{t_k}}\}$ of $\{X_{t_1}, \ldots, X_{t_k}\}$.

$$Mode(\{X_{t_1}, \ldots, X_{t_k}\}) = X_{Mode},$$

where $x_{i_{Mode}} \in X_{Mode}, i \in \{1, \ldots, n\}$ is given as:

$$x_{i_{Mode}} = Mode(\{x_{i_{t_1}}, \ldots, x_{i_{t_k}}\}) = x_{i_{t_a}}, \text{ for some a} \in \{1, \ldots, k\},$$

where, $freq(x_{i_{t_a}}) \geq freq(x_{i_{t_j}}), \forall j \in \{1, \ldots, k\}$, where $freq(x_{i_{t_a}})$ is the frequency at which the item $x_{i_{t_a}}$ exists in the series, and n is the number of realisations.

16- RateOfChange: The RoC function is applied on a set of random variables to return a new random variable, X_{RoC}, in which each element, $x_{i_{RoC}}$, represents the rate of change of the elements in the i-th realisation $\{x_{i_{t_1}}, \ldots, x_{i_{t_k}}\}$ of $\{X_{t_1}, \ldots, X_{t_k}\}$.

$$RoC(\{X_{t_1}, \ldots, X_{t_k}\}, e) = X_{RoC},$$

where $e = rv_{int_k}$, and $x_{i_{RoC}} \in X_{RoC}, i \in \{1, \ldots, n\}$ is given as:

$$x_{i_{RoC}} = \{\frac{x_{i_{t_2}} - x_{i_{t_1}}}{t_2 - t_1}, \ldots, \frac{x_{i_{t_k}} - x_{i_{t_{k-1}}}}{t_k - t_{k-1}}\}, j \in \{1, \ldots, k\}$$

where n is the number of realisations.

17- Round: If *Round* is applied on a random variable X_{t_j}, it returns a new random variable, X_{Round}, in which each element, $x_{i_{Round}}$, represents the decimal number rounded to a specific decimal place, d, for the i-th realisation $x_{i_{t_j}}$ of X_{t_j}.

$$Round(X_{t_j}, d) = X_{Round},$$

where $x_{i_{Round}} \in X_{Round}, i \in \{1, \ldots, n\}$ is given as:

$$x_{i_{Round}} = \begin{cases} \frac{[x_{i_{t_j}} \times 10^d]}{10^d} & \text{if } (10 \times ((x_{i_{t_j}} \times 10^d) - ([x_{i_{t_j}} \times 10^d]))) < 5 \\ \frac{[x_{i_{t_j}} \times 10^d] + 1}{10^d} & \text{if } (10 \times ((x_{i_{t_j}} \times 10^d) - ([x_{i_{t_j}} \times 10^d]))) \geq 5 \end{cases}$$

where $j \in \{1, \ldots, k\}$, and n is the number of realisations.

An example of mapping the functions in Listing 4.2 to the SDES model is as follows. In this example, two functions are identified. The first one is the time series function $F_{m,1}$ named *TSConstructor* which is mapped to a set of random variables $\{X_{t_1}, \ldots, X_{t_k}\}$. This represents the reward variable at each time instant. The random variables' state space is $\{0, 1\}$, since the system can be either up (1) or down (0). Thus, for $j = 1, \ldots, k$:

$$X_{t_j} = \begin{cases} 1 & \text{if system is up at time } t_j \\ 0 & \text{if system is down at time } t_j \end{cases}$$

The $F_{m,2}$ named *Span* is the second function. It counts the number of consecutive random variables with an identical value, which is 0 in this example. For $j = 1, \ldots, k$ in each of the n realisations, the j-th element of the $x_{i_{Span}} \in X_{Span}$, as given in the formal definition of *Span* in this section, then is:

$$v_j = \begin{cases} u & \text{if } x_{it_j} = 0 \wedge \ldots \wedge x_{it_{j-u+1}} = 0 \wedge xi_{t_{j-u}} = 1, \\ & \text{with } 0 < u < j \\ 0 & \text{if } x_{it_j} = 1 \\ j & \text{otherwise (that is, } x_{it_1} = 0, \ldots, x_{it_j} = 0) \end{cases}$$

The outcome of the step of mapping the functions represents the SlaCP Metric Composition phase. In this phase, all the functions available in WSLA are applied on the model outcome to return a single random variable for each *slap*. In the next section, the Decision phase is conducted by using the evaluation function which results from mapping the SLO to produce the WSLA compliance probability.

5.2.5 SLO Threshold Mapping

In the previous section, the mapping of all the functions in the set F_m results in a single random variable $X_{F_{m,|F_m|}}$. This represents `SLAParameter`, *slap*. Hence, the random variable X_{slap} can be defined as:

$$X_{slap} \triangleq X_{F_m,|F_m|}$$

Depending on this random variable, an *slo* will be evaluated. This is the last step in the mapping process and in the methodology as a whole. The evaluation function, *Eval*, maps the random variable into a single value representing the compliance probability:

$$Eval(X_{slap}) = pr \in [0, 1]$$

This evaluation is accomplished by performing a comparison of a type c of an *slap* value against a value v (as defined in Section 4.4.1.1). For example, if $c = '\leq'$ then the probability that an *slo* is met is as follows:

$$pr = P(slo) = P(X_{slap} \leq v)$$

Note that *vs* and *ve* of the validity period specified for an *slo* have not been considered in the evaluation function of *slo*. This is because they have often the same value as the start *s* and the end *e* period of the schedule *sch* defined within the *slap*. Thus, *vs* and *ve* are implicit in the definition of *sch* and hence all values of *slap* are already between *vs* and *ve*.

An example of mapping the SLO in Listing 4.1 to SDES is as follows. In this example, the SLO is satisfied if all *Span* values for $j = 1, \ldots, k$ are smaller than the agreed value of $v = 10$. So, the *slo* is evaluated using the random variable X_{Span}, that is:

$$P(slo) = P(X_{slap} < 10) = P(X_{Span} < 10)$$

5.3 Discussion

In this section, a number of questions are addressed regarding some aspects of the mapping process. These questions are as follows:

1. Why are WSLA functions not mapped on the model primitives?
2. Does the mapping process work for a composite service?
3. Can the methodology work for different function orders?
4. Why was the choice made to use transient reward variables instead of steady-state reward variables?
5. How can the initial state distribution that is necessary for solving the model be retrieved?
6. Is there a different way of computing the response time value?
7. How does the methodology help in providing a service model?

The answer to the first question *"Why are WSLA functions not mapped on the model primitives?"* is as follows: In this methodology, the measurement directives are mapped as reward variables on the SDES model automatically. Then, this model is solved to produce a prediction of these values. Later, the remaining WSLA functions are mapped on the model results (not directly on the model primitives) to produce the SLA parameter; then, an evaluation of the SLO compliance is carried out accordingly. Hence, in this methodology, the model is used to predict measurement directive values only. Another way of predicting SLOs could be to map the whole SLO expression, and not only a measurement, on the model: i.e. including WSLA

functions as part of the reward variable. Although this might be straightforward for some functions, such as `Mean`, it is prohibited by the nature of the tool solvers for others. This is because WSLA functions depend on the value of a reward variable at many instants of time not only a single one. However, the tool solver cannot use the results of a reward variable evaluated at specific instant as an input to the same reward variable in the same model [34]. In other words, measurement directives cannot be predicted and used in the same model in the same run. For this reason, the model is used to predict measurement directive only. In the early stage of this research work, some WSLA functions were mapped as a part of the reward function. This was implemented for the SRN model and the SPNP tool solver [137]. Nevertheless, after realising the aforementioned problem, the direction was changed so that WSLA functions are now mapped instead on the solver output, in particular on the random variable realisations.

For the second question *"Does the mapping process work for a composite service?"*, in a composite service, each service has an SLA with each of the other services. Hence, the methodology of predicting WSLA compliance will not differ whether it is for a composite or a single service because each SLA between two services is mapped independently. SLAParameter (and measurement directive in turn) is usually defined for a specific operation inside the SLA, not for all the services' operation in a single service. As an example, if WSLA specifies a response time measurement, it is for a specific operation in the composite service, not the overall response time for all operations of all services. Hence, the mapping process can be accomplished as normal. The only difficulty in the case of a composite service might be in producing an adequate model that is able to represent the communication among the services correctly.

Regarding to the third question *"Can the methodology work for different function orders?"*, the choice of the order of the function applications selected in Section 5.2.4 will not affect the applicability of the WslaCP methodology as the functions can be applied on a reward variable in any other order. The only reason for the selected order is because this is the most common; it also allows the description of the mapping to be clearer. In a case where the time constructor function is not the first to be applied on the reward variable, then it might be that each instant of a reward variable is computed separately by different calls of the solver (rather than producing a reward variable directly at multiple instances).

The answer to the fourth question *"Why was the choice made to use transient reward variables instead of steady-state reward variables?"* is as follows: The methodology tries to emulate the monitoring case specified in WSLA when the mapping to

SDES is carried out. For this reason, the reward variable is assumed to be solved at an instant or interval of time rather than in a steady state since it is more natural for the dynamic changes of the service. Another way of mapping the time for solving the reward variable is by using steady state metrics and investigating how these can correspond to transient reward metrics. However, the theoretically challenging question of how time-dependent metrics (specified via monitoring times) can be expressed approximately as steady-state measures (for reasons of efficiency) is challenging. Also, because the problem of applying WSLA functions on suitable derivations of these steady-state metrics will raise its head again.

For the fifth question *"How can the initial state distribution that is necessary for solving the model be retrieved?"*, the problem of providing the correct initial state distribution (which is important, not only for response time evaluation, but for any transient measure) is a delicate task that is not (and probably cannot be) supported by the automatic mapping process proposed in this chapter. Hence, it is assumed that parameterising the model is achieved by a user.

Regarding the sixth question *"Is there a different way of computing the response time value?"*, there is a cleaner way of analysing the probability distribution of a response time in the models than that proposed in the mapping. This might be done by using passage time computation methods. However, since the methodology aims to represent measurements as reward variables, the choice has been to adhere to the method proposed in this chapter.

The answer to the seventh question *"How does the methodology help in providing a service model?"* is as follows: Using the operations mapping presented in Section 5.2.1, the user can have a set of pairs, each containing a state variable and an action. Another piece of information that can help in the automatic creation of a model that suits WSLA, is to consider the type of measurement directives that exists in a WSLA document. Knowing this, the user will obtain some idea of and an insight into what WSLA expects the model to pertain to and produce. For example, if a measurement *StatusRequest* exists in a WSLA document, the user is informed that he/she should consider in the model how that the service might go down. This can be supported either manually or automatically. The hints indicated by the existence of measurement were discussed in Section 5.2.2. For example, information for building the service model resulting from mapping WSLA elements to SDES, as shown in Listing 4.2, appears in Table 5.2.

Although this partial model is not complete and lacks some essential information, such as including other state variables/actions, parameterising the delay of its

Table 5.2: The part of the service model as a result of mapping WSLA elements in Listing 4.2 to SDES

	WSLA	SDES	
		State Variables SV	**Actions** A
op	GetQuote	$sv_{GetQuote} \in SV$	$a_{GetQuote} \in A$
m	StatusRequest	$sv_{up}, sv_{down} \in SV$	$a_{fail}, a_{repair} \in A$

actions, and determining the initial state, it helps a user of the methodology by offering a basis for constructing the complete service model when this job is delegated to him/her. Another technique to make sure that the model reflects all the needs of WSLA, is to consider all the hints from all types of measurement directive and then to include them in the model so that the model can accommodate any future changes in the WSLA contract.

5.4 Conclusion

This chapter describes the second fold of the WslaCP methodology that theoretically implements the final six phases of the SlaCP methodology. The contribution of this chapter is to describe the mapping of WSLA contracts on SDES models. This was achieved by conducting a mapping process of five steps: mapping operations as model primitives, mapping measurement directives as reward variables, mapping a schedule as observation intervals, mapping functions to suitable derivations from the model's results, and finally, mapping an SLO into an evaluation function that performs the subsequent check of the adequacy of the modelled service with respect to the service level objective threshold. The mapping process aimed to be automated; however, the role of the user remains a requisite, even if the rewards function is standardised in a template. The user's role is to complete the model definition first and then to identify the incarnation, in the model, of the specified SDES primitives in the reward variable.

Chapter 6

A Software Tool Architecture for SLA Compliance Prediction

This chapter addresses the need to construct a software tool that supports the SLA compliance prediction methodology. The first part of this chapter employs the theoretical foundation of the SlaCP methodology, described in Section 3.2.1, in designing a general tool architecture that aids its users to predict the probability of SLA compliance. This was indicated in the tool designer perspective presented in Section 3.2.3. The contribution made by this part lies in providing a set of architectural components for building the SlaCP tool and describing the design of each component. The design of these components focuses on increasing the tool's modularity in order to accommodate different SLA languages and several stochastic models. It also makes the most of the available solutions to automate, as much as possible, the mapping of SLA elements to a stochastic model which leads to the prediction of the required results. The second part of this chapter implements the proposed design of the tool for WSLA and Stochastic Petri Net (SPN) models by employing the WslaCP methodology presented in Chapters 4 and 5. The contribution of this part involves, firstly, constructing the `SDESsch` schema to provide a structured machine-readable language that represents the model-related primitives produced from mapping SLA to SDES. Secondly, it describes the implementation of each component proposed in the tool architectural design and shows how they work together.

This chapter is organised as follows: Section 6.1 recaps essentials concerning the SlaCP methodology and introduces the common stages used by researchers in the process of constructing a software tool. These stages are described in detail in Section 6.2 by describing the tool's requirements and then the tool architecture with its design. This section provides the detailed description of each architectural component and its design regardless of the type of SLA document or stochastic

model. The implementation of these design components is presented in Section 6.3. Section 6.4 then provides a related discussion concerning the tool's design and implementation. Finally, Section 6.5 concludes the chapter.

6.1 Introduction

The design of the SlaCP methodology proposed in Section 3.2.1 made it possible, through seven phases, to predict the probability of a service conforming to a predefined SLA. This was accomplished by performing the mapping of SLA metrics into stochastic model primitives, reward variables, and a set of functions defined on this model's output. Solving the model to generate the required results allows SlaCP to produce the likelihood of conforming to the SLO thresholds.

Since the SlaCP methodology incorporates seven interdependent phases, it is important to support the user with a software tool that exploits it. Having a tool that utilises the methodology's phases helps in automating their functionality as much as possible. Furthermore, it assists the user in choosing the correct inputs required for completing the prediction process when full automation is not possible. To construct such a tool, the SlaCP tool architecture has to be designed and implemented according to the theoretical basis of the design of the SlaCP methodology (addressed in Section 3.2.1) and in a way that reflects a user's perspective (addressed in Section 3.2.2). Attaining the latter perspective through the architectural design of the tool allows for minimum user interaction to produce the required result.

In software engineering, the design stage of a software system differs from the implementation stage because the latter needs notations and methods [138]. Hence, and to be able to contribute to the design of a generalised tool which is independent of the way a software designer might want to implement it, a distinction is made in this chapter between designing the SlaCP tool architecture and implementing it. For the former, the designer has to pass through certain stages, from setting up the tool's requirements, defining the architecture components, and describing the design of these components [138]. For the latter, the implementation requirements, along with the mode of implementing each design component, have to be specified. In the following section, the stages for designing the SlaCP tool are described in detail.

6.2 Tool Architecture and Design

The tool architecture sets the appropriate elements and components required to build a tool that is able to accomplish a set of predefined requirements, while the

tool design sets the algorithm and any details that reinforce the architecture with the required behaviour [138].

In this section, the tool's architectural requirements are defined in Section 6.2.1 while the architectural assumptions are described in Section 6.2.2. The core and detailed architectural components, together with their design, are described in Section 6.2.3. Finally, a different design of the SlaCP tool is outlined in Section 6.2.4.

6.2.1 Tool Architecture Requirements

In general, the requirements of the tool architecture define the essential information and handling, together with their properties, that the tool designer has to consider when constructing the tool [138]. These requirements are classified as functional if they are related to actual software functionality, or as non-functional if they are related to software quality aspects [139, 140].

For the SlaCP tool, both the functional and non-functional architectural requirements that should be considered are defined in the following two subsections.

6.2.1.1 Functional requirements

The functional requirements of the SlaCP tool are characterised as follows:

1. Tool automation. The tool has to employ the SlaCP phases of Section 3.2.1 in software engines that perform the phases' functionality automatically as much as possible. The automation can be achieved by incorporating different parsers which implement the algorithms that each phase requires. To achieve a higher level of automation, the output and input of these phases have to be available in a machine-readable format. Having such interchangeable inputs and outputs allows the parsers to read, write or update their contents automatically. It also allows for the passing on of results to the 'downstream' software.

2. Tool modularity. The SlaCP tool has to be modular across formalisms and languages which include: multiple stochastic modelling formalisms, multiple SLA languages, and different implementation programming languages. To achieve this modularity, the following requirements have to be considered:

   (a) SlaCP has to be modular across different stochastic models and solvers. To achieve this, any primary outputs of SlaCP should be written independently of the type of the stochastic modelling formalism used to model the service and the type of solving techniques used to solve this model.

Given these independent outputs, SlaCP has to provide embedded translators to perform a translation to and from them. These translators, with simple alterations, can translate SlaCP outputs to be compatible with new stochastic modelling formalisms and solution techniques.

(b) SlaCP has to support multiple SLA languages. To achieve this, the parser has to match different measured metrics that have the same semantics to one category and then map them accordingly. The same has to be considered for different composite metrics that have the same functionality. In other words, the SLA parser has to be abstract enough to convey the meaning of different measured and composite metrics.

(c) The implementation of a tool architecture usually incorporates one of a number of different programming languages (e.g. Java [141]) for building its skeleton and connecting components. Hence, to satisfy the latter type of tool modularity, the architecture has to set any file the tool has to deal with internally (not across modelling and solving techniques) as a set of interfaces with abstract functionality; this is in order to be able to accommodate different implementations using different desired languages. For example, the functions representing SLA composite metrics are specified firstly as interfaces only to allow for the preferred implementation later.

6.2.1.2 Non-functional requirements

The non-functional requirements of the SlaCP tool are characterised as follows:

1. The tool has to be extensible without modifying its core. The tool has to be able easily to adapt the following components: a new SLA language by utilising/amending an additional parser to allocate/match the relevant measured and composite metrics; a new stochastic model by utilising an additional translator which will translate to it from the abstract model; new measured and computed metrics by incorporating a suitable mapper which will map from them to the stochastic model; and any additional parsing or modelling modules for special purposes that will allow the tool to achieve extra automation power. An example of the latter is a module for parsing a service description document in order to generate a complete service model automatically. The tool should also be able to conform to changes in the model's parameter values or in SLO threshold values by incorporating additional methods to set and obtain these values.

2. Tool accessibility. The tool has to be easy to use with clear GUIs which describe the required inputs, together with means that help the user to enter them.

3. Usability and re-usability of the tool. The different components of the SlaCP tool have to be re-usable in such a way that allows the designer to exploit them in new tools with different orientations. For example, an SLA parser component can be used in a tool that syntactically validates SLA documents.

6.2.2 Architectural Assumptions

The tool specifies a set of assumptions regarding phases of the SlaCP methodology, the model of the service, and the SLA document. These assumptions are as follows:

Assumptions regarding phases of the SlaCP methodology: As stated in the first requirement in Section 6.2.1.1, the tool architecture will implement the SlaCP phases in 'engines' that accomplish the functionality of these phases. It is assumed that the tool architecture will identify both a modelling and a solving engine that the rest of the SlaCP engines, defined in Section 6.2.3, can interact with. It is not necessary to design and implement these engines because several existing commercial and researcher modelling and solving tools can be exploited to deliver the functionality necessary for this part of the tool architecture. For this reason, and to avoid re-implementing an existing functionality, these engines are assumed to be present as part of a well-known modelling tool; this offers modules for creating and solving stochastic models. Accordingly, a plug-in tool will fulfil the role of the modelling and solving engines (automating the Model Completion and Model Solving phases); this has to be augmented with the SlaCP tool. The SlaCP tool, in turn, has to set up communication with this tool in a way that will minimise or eliminate user interaction.

Assumptions regarding SLA documents: As stated in the assumptions concerning the SlaCP methodology in Section 3.1.4, the SLA document is pre-defined and valid and is therefore used as an input to the tool. To help in achieving the first functional requirement of the tool's architecture (i.e., tool automation), the SLA document is assumed to be available in a machine-readable format in order for the tool to parse it automatically. This assumption is intuitive since most SLA specification languages are XML-based languages and are electronically available. Since the SLA is assumed to be syntactically correct, there is no need for the parser to perform any validity checks.

Assumptions regarding the model: These include two points:

1. The description of the stochastic model (i.e. service model) can be (a) predefined or (b) not yet available. For the former, the model is constructed previously, either manually, or using any stochastic modelling tool such as Möbius [32]. In this case, the model is used as an input of the tool. For the latter, the model creation is aided by the engine representing the SLA-Model Mapping phase to produce some model primitives, or by a complementary engine able to produce the model automatically from a specific service description document. However, since no optimal model can be generated automatically, extra primitives may need to be added and model validation may have to be performed manually. In all cases, the parameters of the model primitives (such as firing rates), along with the model's initial state, are assumed to be predefined.

2. The description of the stochastic model has to be available in a machine-readable format (i.e. in an exchange format). Most commercial or researcher modelling tools include a graphical representation of the stochastic model in addition to a textual one that can be produced from it automatically. Having the model in a text-based file allows the tool to parse it automatically (the first functional requirement) in order to extract some or add other information. For example, the tool can produce automatically a list of all the state variables and actions of a model. This list can be displayed in the tool's GUI so that it aids the user in choosing the correct primitives that are necessary for completing the reward variable templates when implementing the SLA-Model Mapping phase. Also, the tool can insert the produced reward variables into the appropriate place in the model file and prepare this file with suitable commands to call the solver automatically. The use of the model's textual file is described in detail when the tool's implementation is presented in Section 6.3.

6.2.3 The Tool's Architectural Components and their Design

In the design of the SlaCP tool, the functionality of each component is considered as a black-box which takes an input and produces an output. This is so that the way these boxes are implemented does not affect the concepts of the SlaCP methodology.

The architectural components of the SlaCP tool, whose design is presented in this section, are depicted in Figure 6.1. This contains two parts illustrated using two rectangles. The left-hand gray rectangle is the key part representing the created *'SlaCP Entity'* while the right-hand one represents an existing *'Plugged-in Modelling*

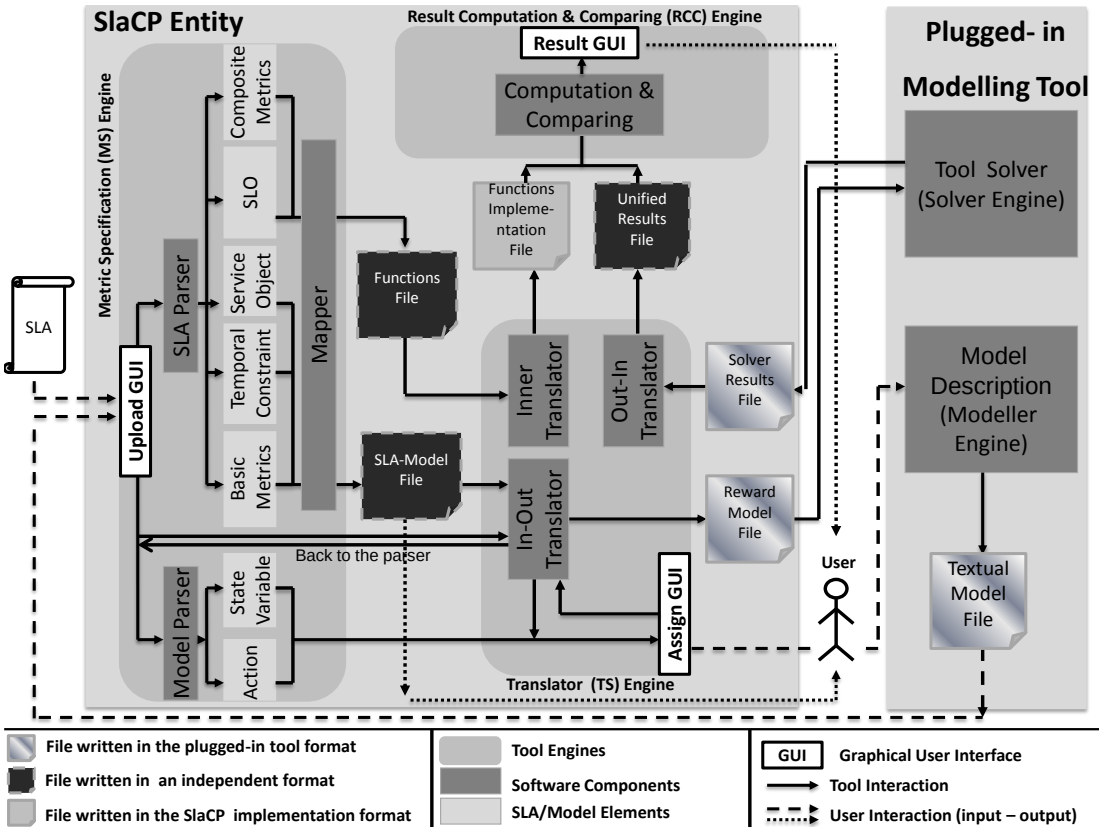

Figure 6.1: Tool architecture of SLA Compliance Prediction (SlaCP)

tool' with which SlaCP has to be augmented so as to create and solve the model. The difference between these parts is that the former is an original design while the latter depends on well-defined techniques. The functionality of the components in the two parts is outlined in what follows; this is described in forthcoming subsections.

The architecture of the *SlaCP Entity* consists of three engines: namely, the *Metric Specification Engine* (MS Engine), the *Translator Engine* (TS Engine) and the *Result Computation and Comparison Engine* (RCC Engine). These engines are depicted as rounded gray rectangles inside the *SlaCP Entity* rectangle in Figure 6.1 to distinguish them from their design components. The three engines have to perform, in total, the functionality of five phases of the methodology design; this was presented in Section 3.2.1. These phases are: the SLA Interpretation, SLA-Model Mapping, Model Specialisation, Metric Composition, and Decision phases.

The first engine, MS, has to parse the file containing the SLA in order to map it into the *SLA-Model File* and *Functions File* which are specified in an independent format. These files are represented as black folded-corner papers in the figure. It also has to parse the model description file, *Textual Model File*, to extract informa-

tion that aids user interaction later. The second engine, TS, has to translate and implement the files generated from the MS Engine into the plugged-in tool and the SlaCP implementation formats respectively. These files are the *Reward Model File* and the *Functions Implementation File*; they are represented in the figure using dark gray and gradient gray folded-corner papers respectively. After this, this engine has to handle the *Solver Results File*, written in the plugged-in tool's solver format, into an independent format, written into the *Unified Results File*, so it can be used by the third engine. Finally, the last engine, RCC, has to apply the functions to the handled results and then compare the final output against the specified threshold in order to produce the SLA compliance probability.

The architecture of the second part of the tool, i.e. the *'Plugged-in Modelling Tool'*, consists of two engines: namely, the *'Modeller Engine'* and *'Solver Engine'*. The two engines are depicted as dark-gray rectangles inside the *Plugged-in Modelling Tool* rectangle in Figure 6.1. These two engines perform the functionality of the Model Completion and Model Solving phases respectively.

The SlaCP engines have to interact with each other and with the plugged-in modelling tool, represented as solid lines in Figure 6.1, in order to accomplish the ultimate goal of the tool. Each engine also has to communicate with a *User* through dedicated GUIs, represented as white boxes in the figure, when an input is necessary from the user (represented as dashed lines) or when an output is presented to the user (represented as dotted lines). The *User* figure plays different roles in this tool; these could include the role of SLA engineers, service providers/engineers or modellers.

In what follows, each engine's design is described along with its inputs, software components, outputs, its ability to be fully automated, and its interactions with the other engines. Please note that all design components depicted in Figure 6.1 are referred to using an *emphasised* font.

6.2.3.1 Metric Specification Engine (MS Engine)

The *Metric Specification Engine* (MS Engine) has to emulate the function of the first two phases of the SlaCP theoretical methodology: namely, the SLA Interpretation, and the SLA-Model Mapping phases. It also has to contain another feature that is not included in the theoretical phases which is parsing the file that contains the service model. This feature is added to the tool design in order to increase the tool's level of automation and to increase the help offered to the user; this is described later in this section.

The MS Engine is depicted in Figure 6.2. This is the same as Figure 6.1 but the related design components, together with their input and output, are the only ones

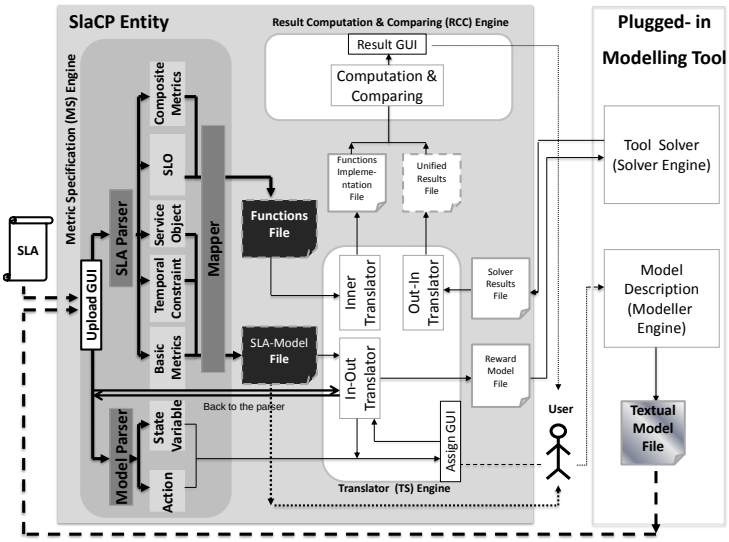

Figure 6.2: The *Metric Specification Engine* (MS Engine) in the SlaCP tool

that are highlighted. This engine is the first engine to be executed in the *SlaCP Entity*. It is responsible for parsing the files containing the SLA and the model description, and for mapping the parser output to the relevant stochastic model related metrics. Its design is as follows:

Engine Input: The MS Engine has two inputs: the *SLA* contract and the *Textual Model File* which is generated either manually by the user or semi-automatically with the aid of the tool. This input is achieved through a dedicated GUI, *Upload GUI*, from which the user can upload these files.

Software Components and their Output: This engine has three software components: namely, the *SLA Parser*, the *Model Parser*, and the *Mapper*. The design of each is as follows:

1. ***SLA Parser***: This emulates the SLA Interpretation phase by implementing an algorithm to parse the SLA automatically in order to extract the information that is relevant to the SLA prediction. This output information, as shown in the light-gray rectangles in Figure 6.2, is the same as specified in the SLA Interpretation phase. The outputs are: the *Service Object*, *Basic Metric*, *Temporal Constraint*, *Composite Metric* and the *SLO*.

2. ***Mapper***: This emulates the SLA-Model Mapping phase in order to produce two machine-readable files, the *SLA-Model File* and the *Functions File* (See Figure 6.2). The output of this engine is placed in two files because they

are used by different engines of the *SlaCP Entity* in order to be applied at different stages of the prediction process. The first file contains the model state variables/actions, the reward variables, and the time for solving them. This file is used to complete the model creation which is described in Paragraph a.2 of Section 6.2.3.2. The second file contains the interfaces of the mathematical functions representing the computation performed by composite metrics. Also, it contains a function representing the comparison of the SLO threshold with the predicted value of the ultimate composite metrics. This file is applied on the solver output which is described in Section 6.2.3.3.

A specific modelling-tool language is not sufficient to express the *SLA-Model File* since the user can choose from a wide range of modelling formalisms and tools, each of which has its own input language (such as CSPL for the SPNP tool, or C for Möbius). For this reason, as mentioned in Section 6.2.1.1, this file has to be written in an intermediate technical computing language that is able to express the outcome independently of the choice of the modelling tool or its solver. This intermediate language also has to be machine-readable. The *SLA-Model File* is considered to be an intermediate file that has to be translated into one that fits the chosen model. The same is true regarding the *Functions File* where the function interfaces have to be available in a machine-readable format to allow the SlaCP tool to read them, implement them using the preferred programming language, and apply them on the solver results automatically. Although this file is used internally by the SlaCP tool (i.e., it does not interact with the plugged-in modelling tool), the design choice here is also to write it in a unified mathematical language to be independent of the choice of the programming language (such as Java, for example) the designer wants to implement in the SlaCP.

3. ***Model Parser***: This does not relate directly to the theoretical foundation of the SlaCP methodology. Nevertheless, its usage in the tool is to offer extra help to the user in completing the prediction process correctly and easily (i.e., it adds an extra level of automation to the tool). The rationale behind using the *Model Parser* is to extract all the state variables and actions from the service model after the user has completed its creation or uploaded it to the tool (as a *Textual Model File*). The usage of this information is described in Paragraph a.2 of Section 6.2.3.2).

Ability for Automation of the MS Engine: This engine is fully automated since it implements different algorithms. The first one is for parsing the SLA file to extract the elements required for SLA prediction. The second one is for mapping SLA elements to produce the content of the *SLA-Model File* and *Functions File* in an independent format. The last algorithm is for parsing the model file to extract the available state variables and actions. The only interaction the user has to perform within this engine is to upload the files containing the SLA and the model description. Hence, the design of this engine achieves the first step of the methodology from a user's perspective, as presented in Section 3.2.2.

6.2.3.2 Translator Engine (TS Engine)

The *Translator Engine* (TS Engine), which is depicted in the coloured part of Figure 6.3, has to emulate the purpose of the Model Specialisation phase. It is used by the *SlaCP Entity* to communicate with the *Plugged-in Modelling Tool* and the *User*. In addition to this interaction, the TS Engine interacts within the *SlaCP Entity* with the other engines, namely, the MS and RCC engines. In what follows, each of these interactions is described with the required inputs, the software components and its outputs. A summary of these software components and the automation possibilities of this engine as a whole are discussed at the end of this section.

a)- TS Engine Interaction with the Plugged-in Tool and the User: As depicted in Figure 6.3, the *SlaCP Entity* has to deal with the *Plugged-in Modelling*

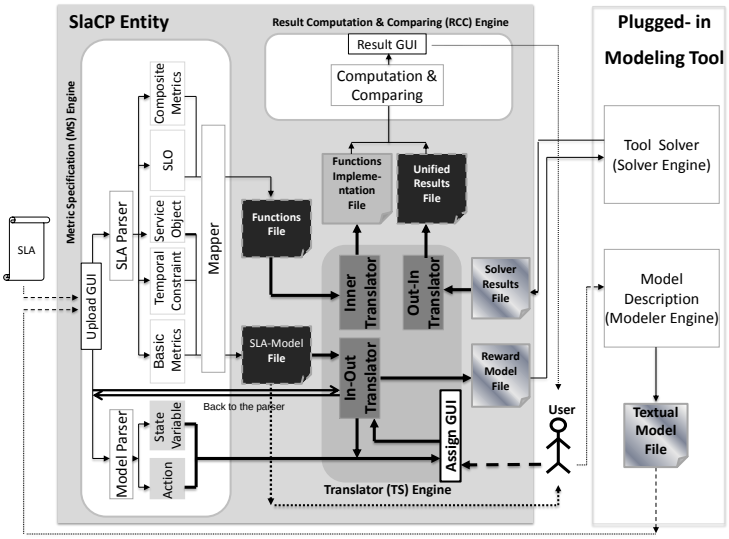

Figure 6.3: The *Translator Engine* (TS Engine) in the SlaCP tool

tool's solver in order to give information to it and to receive other information from it. One of the design options for providing such a communication, as presented earlier in Section 6.2.3.1, is to express any output from the SlaCP tool in a format independent of the one of the plugged-in tool. This is achieved by using two intermediate languages; each is suitable for expressing one of the MS Engine outputs. Accordingly, it is the TS Engine that is in charge of translating any input/output of the plugged-in tool to/from its specific language. The tool's modularity, extensibility and generality thus increases due to the ability of the TS Engine to provide translation to and from the input and output languages of new plugged-in tools by only making small changes to the algorithm that translates to and from the intermediate languages. However, integrating the translator engine inside the tool involves an immense amount of translation. This introduces extra time overheads and hence affects its speed.

To perform the aforementioned translations to and from the solver, a set of translating components is utilised inside the TS Engine. The number and type of these components depends on the way the plugged-in tool that represents the modeller/solver engines is integrated inside the SlaCP tool. The choice that will be made in Section 6.2.3.4 is to use a single tool that is capable of both modelling and solving a stochastic model. Given this choice, the engine's input, its components responsible for interacting with the plugged-in solver, along with the outputs, are as follows.

a.1)- Interaction Input: The inputs are firstly, the *SLA-Model File* that contains the generated model primitives, the reward variable templates, and the specified time to solve them; secondly, the *Textual Model File* which contains the model description in case the user completes its creation or builds it from scratch; and finally, the output of the *Model Parser* which includes all the state variables and actions available in the model.

a.2)- Software Components and their Outputs: The design choice for interacting with the plugged-in tool solver would be to embed two translating components (*In-Out Translator* and *Out-In Translator*) within the TS Engine, as depicted in Figure 6.3. The justification for using two translating components depends on the input and output of the plugged-in tool as described in what follows.

1- ***In-Out Translator***: This performs two kinds of translation. The first is to produce the *Textual Model File* while the second is to generate the *Reward Model File*. This is described in what follows.

The first translation depends on whether or not the *Textual Model File* is predefined. If it is predefined, this translation is ignored because the *Textual Model File* will be already written according to the plugged-in tool's language. However, if this file is not already defined, the *In-Out Translator* takes the *SLA-Model File* from the MS Engine and translates its state variables/actions into fragments compatible with the language used by the plugged-in tool. After doing this, it asks the user to complete the creation of the model description textually. When the user has completed this interaction, the translator saves the model into the *Textual Model File* and sends it back to the *Model Parser* of the MS Engine (the *back to parser* arrow in Figure 6.3) in order to obtain the required parser outputs.

The second translation also obtains the *SLA-Model File* and translates its abstract reward variables into fragments compatible with the language used by the plugged-in tool. Performing this translation allows the solver of the plugged-in tool to understand the input, therefore producing the desired results.

The latter translation cannot be fully completed automatically since the reward variables stored in the *SLA-Model File* are templates only. Each of these templates requires either the user's confirmation of its correctness or the user's interaction to complete it with the relevant model primitives. The former case occurs when the intended state variable or action representing the service object is used in the template. However, the latter occurs when the primitives used within these templates are unknown or do not reflect the intended service object in the model. Hence, the TS Engine has to inquire the user while performing this type of translation.

The GUI related to this engine, *Assign Engine*, displays the set of state variables and actions existing in the model by using the *Model Parser* output in a drop-down list tailored to the type of the reward variable. This means that, if the reward variable is rate-based, it displays the state variables; otherwise, it displays the actions. The user, with the tool's assistance, has to choose the one that completes the definition of each reward variable in the *SLA-Model File*. For example, if the reward variable concerns an object throughput, the tool will ask the user to choose the name of the action representing this object from a list of all the actions in the model. Using the drop-down list helps the user to choose the right primitives because it eliminates the need to input the model primitives manually. It also prevents any typo the user might make when writing down these primitives.

After completing the reward variable definitions and translating them, the translator takes the *Textual Model File* and automatically inserts these into the suitable place in the file. It also inserts any execution commands necessary for solving the model. In addition, it automatically embeds the time needed to solve the reward

functions (stored in the same file) in a format suitable to the solver. The output of this engine is a *Reward Model File* that contains the model description, the reward variables, the time needed for solving them, and the execution commands. This file is used as a solver input that is called implicitly by the *SlaCP Entity*.

2-***Out-In Translator***: This translates the *Solver Results File* into a unified format and writes it into the *Unified Results File* to be used later as input to the RCC Engine. This is because the file that stores the solver outputs has a unique extension, syntax and semantic for each modelling tool. Hence, this file has to be translated into a new file, written in a unified format, to allow it to be sufficiently general to be used by the SlaCP Entity and across other tools if needed.

b)- TS Engine Interaction with the RCC Engine: In addition to the previous interaction with the user and the plugged-in tool, the TS Engine, as depicted in Figure 6.3, is responsible for translating and preparing two of the tool outputs to be used as input to the RCC Engine. First, as described earlier in the *Out-In Translator* component, it translates the *Solver Results File*, which contains the output of the solver, into the *Unified Results File*, which represents the results in an independent format so that they are portable. These results are presented in a form that the SlaCP can understand so that further computations can be applied. Second, it translates and implements the *Functions File*, which is the second output of the MS Engine. The second translation compels the TS Engine to have extra input and an additional software component than the ones specified in the paragraph above.

b.1)- Interaction Input: The *Functions File* that contains the interfaces of WSLA functions.

b.2)- Software Component and its Outputs: The component required for this translation is the *Inner Translator*. It translates and implements the functions' interfaces into a language that the SlaCP tool can understand (i.e. the language that SlaCP APIs are built in, such as C or Java). It then writes these into the *Functions Implementation File*.

Given what is described earlier for this type of interaction, the TS Engine has to prepare any output files generated from the plugged-in tool or the MS Engine in a format that the RCC Engine can understand and then write them into the *Unified Results File* and the *Functions Implementation File* respectively.

c)- Interaction of the TS Engine with the MS Engine: The interaction that TS Engine performs with the MS Engine lies in obtaining the independent-format generated files (the *SLA-Model File* and the *Functions File*) and translating them to be understood by the plugged-in tool solver and the RCC Engine. These interactions were presented earlier. To recall them, the TS Engine translates the *SLA-Model File* into a format specific to the plugged-in tool in order for it to be used as a solver input. It also translates and implements the functions' interfaces written in the *Functions File* into a language the SlaCP tool is implemented with so that it can be used as input to the RCC Engine.

d)- The Software Components of the TS Engine for all the Interactions: According to the three types of TS interaction previously described, the TS Engine in this design has to perform three types of translation in total. The translating components used inside the TS Engine are therefore: the *In-Out Translator*, the *Out-In Translator*, and the *Inner Translator*. Accordingly, the TS Engine can be considered as the core engine in the SlaCP Entity because all the other engines interact through it. It is also the one that helps the tool to be portable across a range of modelling tools and implementation languages.

3)- Ability for Automation of the Whole TS Engine: The TS Engine mechanism can be considered as semi-automated since the user's interaction is needed to complete the model's creation and to provide the connection between reward variable templates and stochastic model primitives. However, all the other required translations and implementations has to be performed in a fully automatic way. In addition, the process of assigning the reward is aided by a dedicated GUI in which the user can choose the most suitable stochastic model primitives that correspond to the state variables or actions in the reward variable templates. In this way, the rewards will be assigned correctly, depending on the specific model. Hence, the design of this engine is compatible with the third step of the theoretical methodology from the user's perspective; this was presented in Section 3.2.2.

6.2.3.3 Result Computation and Comparison Engine (RCC Engine)

The *Result Computation and Comparison Engine* (RCC Engine), depicted in Figure 6.4, has to emulate the functionality of the last two phases of the theoretical SlaCP design: namely, the Model Composition and Decision phases. This engine is the last one in the *SlaCP Entity* and it accomplishes the ultimate goal of the SlaCP methodology. Its design is as follows:

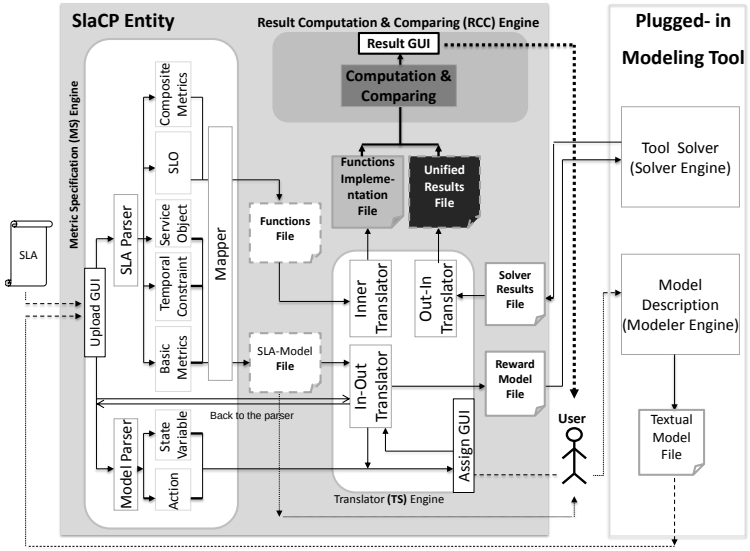

Figure 6.4: The *Result Computation and Comparison Engine* (RCC Engine) in the SlaCP tool

Engine Input: The RCC Engine has two inputs. The first one is the *Functions Implementation File* that contains both the implementation of the mathematical functions to be applied on the solver's output, and the evaluation function which computes the ultimate probability of the SLA compliance. The second input is *Unified Results File* that contains the solver results.

Software Component and its Outputs: RCC Engine has a *Computation & Comparing* component which automatically applies all the functions specified in the *Functions Implementation File* on the solver output specified in the *Unified Results File* in order to derive the desired SLA computed metric. After obtaining this value, the engine uses the evaluation function, stored in the same file, to compare the value to the specified SLO threshold. The outcome of this function is the SLA compliance probability that is depicted to the user through the *Result GUI*.

Ability for Automation of the RCC Engine: This engine is fully automated because it implements an algorithm for parsing the input files, producing the final result, and then comparing it. Hence, the design of this engine is compatible with the fourth step of the methodology from a user's perspective which was presented in Section 3.2.2.

6.2.3.4 The Modeller and Solver Engines

In this section, the design choice of the Solver and Modeller engines (i.e. plugging in an existing modelling tool into the SlaCP tool) is presented. Then, the requirements which need to be available in this plugged-in tool are identified.

The Design Choice of Plugging-in an Existing Modelling Tool: As described in Section 6.2.2, one of the design options of the SlaCP tool is to use existing model creation tools and solving techniques to represent the Model Completion and Model Solving phases. The plan here is to assemble them in the proposed SlaCP tool to aid in building the stochastic model and providing a better prediction accuracy when solving the generated reward model. Usually, the solving techniques can belong either to an integrated tool that both builds and solves the model, or they can be stand-alone solvers. Deciding whether to choose integrated or independent environments for building and solving the model affects the way they are plugged into the SlaCP. In addition, this affects the design choice of the other engines in the *SlaCP Entity*. The design choice made in this thesis is to use a single integrated plugged-in tool to fulfil both the Model Completion and Model Solving phases (i.e. the Modeller and Solver engines). This choice is made because using a single tool is more intuitive and powerful in obtaining an accurate result. The integration is accomplished in the way described in what follows.

The *Plugged-in Modelling Tool* is depicted in the coloured part of Figure 6.5. The *Model Description* represents the Model Completion phase, while the *Tool Solver* integrated with this particular tool represents the Model Solving phase.

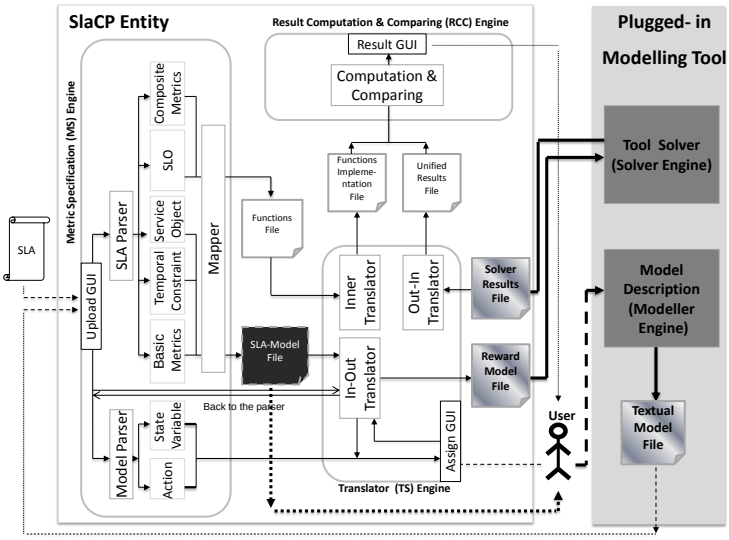

Figure 6.5: The *Plugged-in Modelling Tool* in the SlaCP tool

For the former, the user can produce the model from scratch in a graphical format; the plugged-in tool then translates it automatically into a textual one (a tool requirement). Alternatively, the textual format could be written manually by an advanced user if he/she is sophisticated in terms of producing such a model. Hence, the output required from the plugged-in tool is the *Textual Model File*, as depicted in Figure 6.5). This is usually written in a format specific to the plugged-in tool. If the user does not want to build the model from scratch, the *SLA-Model File* can aid model creation by giving a set of service model primitives mapped from an SLA; this then allows the user to complete the creation of the model.

For the latter part of the *Plugged-in Modelling Tool*, the solver is called automatically from the command line by the *SlaCP Entity* to solve the model after augmenting it with the relevant reward variables, time intervals, and the commands that solve it. The output of the solver that runs in the background is a file called *Solver Results File*. This contains the results in a format specific to the chosen tool.

In this chapter, concentration focuses on this design choice as it is the one used in the implementation. Another design choice, that of plugging in the modelling and solving tools, is presented in Section 6.2.4.

The Requirements of the Plugged-in Modelling Tool: There are several well-known modelling tools that can be plugged into the SlaCP tool. However, the choice of this tool depends on the existence of a set of features. These are as follows:

1. The preferred type of modelling formalism. Although the model-related primitives generated in the *SLA-Model File* are written in an abstract stochastic model, these need to be transferred into a concrete modelling formalism to allow the solver to solve them. The user has to be comfortable with the choice in a way that allows him/her to build or complete the service model correctly. For example, the Möbius tool can be used to build and solve models written in stochastic extensions of Petri nets (Stochastic Activity Network (SAN)), Markov Chains and extensions, and Stochastic Process Algebras (SPA). Another example is the SPNP tool that can be used to build and solve models written in the Stochastic Reward Net (SRN).

2. The ability to define impulse/rate reward variables. In other words, the chosen stochastic model has to be defined with an underlying Markov Reward Model (MRM) [142] to allow the definition of the reward variables representing the SLA measured metrics. The tool also has to allow this definition on the model

level rather than the state level. This is because assigning rewards on the state level tends to be too complicated and time-consuming for large models.

3. The ability to represent the model in a textual format to allow it to be parsed automatically; also, to allow the *SlaCP Entity* to call the solver and pass the model file to it automatically. For example, the SPNP tool has a simple input language called CSPL; this is based on C programming language. This language represents the model textually with its state variables, actions, reward variables and solving commands in a single file.

4. The simplicity of the textual description of the model. It is not enough for the model to be available textually; it also has to be simple, allowing the parser to extract the required information intuitively. For example, it is hard for multiple files representing a model to be used as input to the SlaCP tool; they are also difficult to parse. This complex textual description can be found in tools that employ hierarchical modelling such as Möbius. This has a textual description consisting of a set of classes, written in C language, specifying the atomic, composed, reward, study, and solver models. Each of these classes is written to a separate file in a separate folder [32].

These requirements are extended when the tool's implementation is described in Section 6.3.1.

6.2.4 Discussion: Alternative Design of the SlaCP Tool

The design choice of the SlaCP tool presented in Section 6.2.3.4 depends on using a single plugged-in modelling tool for building and solving the model. Although this design is modular, it involves a heavy translation workload. Given this, an alternative design choice can be considered. This design attempts to minimise the effort of translation by making a change to the way the plugged-in tool is augmented and, as a consequence, in the design choice of the MS and TS engines. This alternative design depends on choosing a stand-alone solver to solve any model independent of the chosen modelling tool used to describe the model. This means that the model can be developed using any tool associated with a stochastic model of choice while the solver has to solve this model after translating it into the solver input format.

To place this into the context of the SlaCP methodology, Figure 6.6 depicts this alternative design of the SlaCP tool. The main difference between this design and the one presented in Figure 6.1 is that the right-hand side rectangle is now divided into two parts: the *Plugged-in Solver* and the *Plugged-in Modelling Tool*. This means

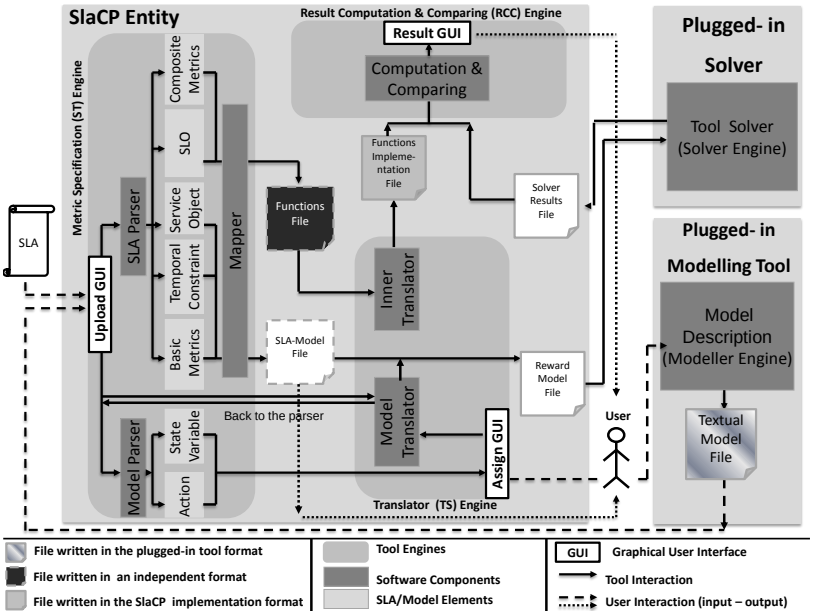

Figure 6.6: Alternative design choice of the SlaCP tool

that the model description of the *Plugged-in Modelling Tool* implements the Model Completion phase while a single stand-alone *Plugged-in Solver* (independent of the tool) implements the Model Solving phase. For the former, the description of its output is the same as described in Section 6.2.3.4. However, for the latter, the design consideration regarding the translating components responsible for communicating with the plugged-in solver and their output differs from that outlined in Section 6.2.3.4. In addition, the format type of the *SLA-Model File* that represents the output of the MS Engine is different. This is described in what follows.

Inside the TS Engine, the choice is to embed a single translating component, the *Model Translator* (as depicted in Figure 6.6)[1]. The *Model Translator* takes the *Textual Model File* and translates it into fragments which are compatible with the input format of the plugged-in solver. It then takes the *SLA-Model File* and directly inserts these fragments into the relevant place in the file. This is because, in this design, the *SLA-Model File* is written, from the beginning, according to the solver input format since this solver is always being used; i.e., it is not in a generalised format and so there is no need to translate it. The *Model Translator* then updates the file with any execution commands necessary for solving the model. The output of the TS engine is the *Reward Model File*; this is used as input to the solver, called implicitly from the *SlaCP Entity* to produce the *Solver Results File*. This contains

[1] An example of a similar design is the PDETool where a model, developed using one of different formalisms, is translated into an SDES file which is the input of the SimGine simulation [143].

the solver output which does not have to be translated because it always has a fixed syntax since this solver is always being used.

According to what has been previously described, in this design, the MS engine produces the *SLA-Model File* giving the solver-input format instead of an independent one. In addition, the TS engine performs only two types of translation in total. The translating components used inside the TS engine are therefore the *Model Translator* and the *Inner Translator* which is the same as defined in Section 6.2.3.2.

The difference between the two choices of design for the SlaCP tool, presented in Figures 6.1 and 6.6, is that the second minimises the overheads of communication and the translation load. It is also easier to implement, and has a faster execution time. However, it is less modular. The first design, instead, is believed to be more modular and powerful in obtaining accurate results. Other differences regarding the format of the SlaCP files are depicted in Table 6.1.

In this table, **Design 1** uses a single plugged-in tool for modelling and solving the model. In this design, the *SLA-Model File* is written in a unified format; thus, it is translated into a format compatible with the *Textual Model File*. Hence, the core of the actual textual model file is left intact and the rewards are inserted into it after they have been translated. On the other hand, **Design 2** uses a single stand-alone solver. In this design, the *SLA-Model File* is written in a solver specific format and hence, no translation is required. Instead, the translation is carried out on the textual model file into the solver-specific format. In this case, the core of the actual textual model file is modified and the content of the *SLA-Model File* is inserted directly into it without being translated. In both designs, the *Textual model File* can be written in one of multiple modelling formalisms. The *Reward Model File* is written in the format of the chosen modelling tool in the first design, while it is written in the solver format in the second design. Finally, in the second design, the *Unified Results File* does not exist since the solver always has the same format; this is the opposite in the first design which is written in a unified format.

Table 6.1: SlaCP files formats: differences between the two SlaCP designs

	SlaCP file format				
	SLA-Model File	**Textual Model File**	**Reward Model File**	**Solver Results File**	**Unified Results File**
Design 1	Unified format	Multiple formalisms	Modelling tool format	Solver's specific format	Unified format
Design 2	Solver format	Multiple formalisms	Solver format	Solver's specific format	N/A

6.3 Implementation

The implementation of a tool design defines the way in which the design components are represented in all their details. It specifies the algorithms and data types for which the architectural design, together with its requirements, is fulfilled [138]. The design components specified in Section 6.2.3 for the SlaCP tool are the MS engine, TS engine, and RCC engine, and the plugged-in modelling tool.

To implement the SlaCP design components, a WSLA contract is used as a concrete representation of the SLA document and the Stochastic Petri Net (SPN) is used as a representation of the stochastic modelling formalism. Hence, to be specific to the WSLA's context when implementing the *SLA Parser* and *Mapper* of the MS engine (in terms of service objects, time constraints, basic and composite metric definitions, and the SLOs), the implementation of these components exploits the WslaCP methodology represented in Chapters 4 and 5. Accordingly, the tool that implements the SlaCP design components using the WslaCP methodology is called from now on the WslaCP tool.

The WslaCP tool skeleton, algorithms, software components and all communication among these components (or what are called the Application Programming Interface (API)[1]) are built and implemented using Java [141] through Eclipse IDE [144]. Java is used because it can create Object-Oriented applications that are portable across platforms since it runs on the Java Virtual Machine. The implementation also exploits Java Swing [145] to build a set of Graphical User Interfaces (GUIs) that assist the user through the different steps of the tool. Java Swing is used instead of the Abstract Window Toolkit (AWT)[2] because it is a lightweight component that can therefore be used across platforms.

In the following subsections, the implementation requirements of the WslaCP tool are specified in Section 6.3.1. Later, the implementation of the design of each of the architectural components, that was specified in Section 6.2.3, is described from Section 6.3.2 to Section 6.3.5.

6.3.1 Tool Implementation Requirements

To implement the tool design, the following requirements have to be considered:

1. In order to express the *SLA-Model File*, which represents the model-related *Mapper* outputs, the implementation has to provide an intermediate machine-readable language in which to write this file. This has to be abstract and it

[1] http://en.wikipedia.org/wiki/Application_programming_interface
[2] http://docs.oracle.com/javase/1.5.0/docs/guide/awt/index.html

is desirable that it depends on the SDES formalism in order to exploit the WslaCP methodology which maps the WSLA to this SDES.

2. To be able to express the *Functions File* containing the functions' interfaces which are applied on the solver output, the implementation has to provide a mathematical machine-readable language in which to write this file.

6.3.2 The Implementation of the Plugged-in Tool

The implementation of the plugged-in modelling tool is described first as it forms the basis for describing the implementation of the SlaCP engines. The choice of the modelling tool, as described in Section 6.2.3.4, is related to the preferred type of stochastic model. In this thesis, the Stochastic Petri Net was chosen to represent the service's concrete model. Hence, the chosen plugged-in tool has to be able to build and solve SPN. In what follows, additional implementation requirements for choosing the plugged-in tool are stated. Then, the choice of this tool is determined.

6.3.2.1 Implementation Requirements of the Plugged-in Tool

The four requirements outlined in Section 6.2.3.4 have to be considered when choosing the plugged-in tool, in addition to the following requirements: the ability of the tool to solve the model using a transient simulation, and the ability to retrieve the simulation observations (replicas) from this tool. This is described in what follows.

The Ability to Solve the Model using a Transient Simulation: The choice to use simulation rather than an analytic solver is made for a number of reasons:

1. The monitoring nature of the SLA composite metrics requires, for their input, the raw data of the measured QoS metrics. Hence, in SLA prediction, using the results of the reward variable (i.e. the normal solver output such as the expected values) for applying a function, such as counting the occurrence of a specific value, is not valid. Using simulation instead of an analytic solver permits the extraction of the raw simulation replicas underlying the expected results of the solver. This provides the data in the form required by the composite metrics. Applying the functions representing the composite metrics to each replica, and then computing for each the ability to meet the *slo*, allows the tool to find the compliance probability from among all the replicas.

2. If a reward variable distribution (during an interval of time/at an instant of time) is produced as an output of a solver, there is no possibility of producing

the probability that this reward variable at time, t, had n past consecutive occurrences of a specific value, v (on which some WSLA functions depend). Hence, there is no possibility of computing WSLA functions given this distribution. This is because each reward variable was considered in Section 5.2.4 as a random variable, and WSLA functions depend on counting or comparing mechanisms of these random variables. Hence, the probability that X equals a and Y equals b, $pr(X = a \wedge Y = b)$, is required by WSLA functions to determine a probability occurrence of a specific value. However, this probability cannot be equal to the multiplication of individual probabilities: i.e. $pr(X = a \wedge Y = b)$ is equal to $pr(A = a|Y = b).pr(Y = b)$ rather than $pr(X = a).pr(Y = b)$ because the random variables are dependent. This cannot be obtained using the available tools, as much as is currently known. These tools normally produce the distribution only at each instant or produce the accumulated rewards up until a specific instant. Accordingly, using the replicas generated from the simulation output can resolve this issue because each one represents one possible behaviour of the running service; hence, any function can be applied, as in the monitoring case.

3. Using a terminating (transient) simulation is more appropriate than a steady-state one since the reward variables are evaluated, either at a finite instant of time, or during a finite interval of time [114], as defined in Section 5.2.3.

The Ability to Retrieve the Simulation Observations (Replicas): This is an essential requirement since not all tools write the simulation replicas into a file that the user can obtain. The rationale behind this is that the replicas cannot be produced off-line from the distribution/expected value of a reward variable because the result at a specific instant is dependent on past instants.

6.3.2.2 The Chosen Modelling Tool

The aforementioned requirements for the chosen modelling tool, in addition to the ones presented in Section 6.2.3.4, make it difficult to decide which tool can be implemented as the plugged-in tool. This is because, from among the ones that have been studied in this thesis, there is no single tool that accomplishes all these requirements. A summary of the comparison of the tools, which were briefly described in Section 2.6.4, is presented in Table 6.2. The tools, as stated earlier are required to possess the following: SPN type, reward definition, textual input, simple input, transient simulation and simulation trace. This is described in detail in what follows.

1. Möbius: This tool can model and solve multiple modelling formalisms of which the Stochastic Activity Network (SAN) is one. The reward variables are defined on the net level. Möbius can define hierarchal models and so, for that reason, multiple C-based textual files are generated for each model. However, the syntax of these files is complicated, making it difficult to parse and update its content automatically and to call the solver implicitly. The Möbius model can be solved using transient simulation; its trace can be generated and obtained.

2. SPNP: This tool can build and solve SPN Reward Models, especially the Stochastic Reward Net (SRN) [146]. It permits the definition of reward variables at the net level. SPNP has textual representation using the CSPL language. This textual model is available in a single simple file that can be extracted easily. Finally, SPNP can be used to obtain transient measures using discrete event simulation. However, there is no way of extracting the simulation trace. Also, simulation runs for multiple time instants do not work properly.

3. SHARPE: This tool can build and solve models of different formalisms including the Generalized Stochastic Petri Net (GSPN). SHARPE does not allow reward definition on the net level; instead, reward rates are inserted at the state level by enumerating each state transition and the reward given for each of them. SHARPE has a textual representation using MRM enumeration; this

Table 6.2: Comparison of different modelling tools with WslaCP requirements

	WslaCP Tool Requirements					
SPN Modelling Tool	**SPN Type**	**Reward Definition**	**Textual Input**	**Input Simplicity**	**Transient Simulation**	**Simulation Trace**
Möbius v. 2.3.1	SAN	Net level	Yes C-Based	Hierarchical, multiple files, complicated	Yes	Yes
SPNP v. 6.0	SRN	Net level	Yes CSPL file	Simple, single, and compact	Yes, not working properly	No
SHARPE v. 1.01	GSPN	State level	Yes, MRM file	Simple, not well structured	No	N/A
GreatSPN v. 2.0.2	GSPN	Net Level, limited expression power	Yes, BNF file	Two files, cannot be tested	Yes	Yes
PIPE v. 2.5	GSPN	No	Yes, XML/ PNML file	Simple, but not enough	No transient simulation/ analysis	N/A

can be extracted easily but it is not well-structured. SHARPE can solve the model using an analytic-numeric solver only.

4. GreatSPN: This tool is used for building and solving models built using a Generalized Stochastic Petri Net (GSPN) and its coloured extension. GreatSPN can define rate and impulse rewards as user-defined performance results (or performance indices) that have a limited expression power. GreatSPN 2.0.2 stores the graphical representation of the model in two ASCII files with the extensions *.net* and *.dat*. This textual description of the model is written according to the Backus-Naur Form (BNF)[1] format. It can be solved for discrete event simulation and it stores its trace file. These files cannot be tested or investigated because they are UNIX based.

5. PIPE: This tool is an open-source tool for modelling and solving models built using the Generalized Stochastic Petri Nets (GSPNs) [147]. PIPE does not allow the definition of reward variables; instead, non-reward based performance statistics can be derived, such as passage time analysis (using the DNAmaca interface), or state space analysis, such as the expected number of tokens in a place. The latter can be accomplished using GSPN analysis or a simulation [147]. PIPE is XML/PNML[2] based so the model is converted into this format before it is solved. Although this file is simple and place/transitions can be derived easily from it, it is, however, inadequate because it does not support the definition of analysis module control commands. PIPE does not support transient analysis either analytically or by using simulation.

Although there is no single tool that satisfies all the requirements for the plugged-in tool, the decision is made to augment WslaCP with two tools: SPNP and Möbius. This is because they fulfil most of these requirements. In what follows, an overview of the main strengths and drawbacks of the chosen tools are reviewed.

SPNP Tool: The factors for choosing SPNP lie in its ability to represent Stochastic Petri Nets (SRN in particular), as well as, its flexibility. It allows models to be built graphically, using the SRN editor, or textually, using the CSPL language (a C-based language specific to SPNP). The CSPL file contains a description of the model and has many built-in SPNP functions that are necessary to solve it later. This CSPL file corresponds exactly to the graphical representation of the model.

[1] en.wikipedia.org/wiki/BackusNaur_Form
[2] http://www.pnml.org/

According to this, the *Textual Model File* needed by the WslaCP tool can be generated in two ways. The first one is to write the model (or complete it) manually using SPNP-dedicated CSPL statements, while the second is to use the SPNP editor to build an SRN model graphically. This will automatically produce the equivalent CSPL file. The previous characteristic is the most important factor to support the use of SPNP because the WslaCP tool would then be able to extract/insert the code of interest automatically from/to this CSPL file and then call this file to be solved from the command line. For example, the WslaCP can retrieve the information regarding state variables and actions and add new constructs representing a model's behaviour, reward functions, time to solve them, variable definitions, and other SPNP statements to aid in solving the model and producing the desired results.

A negative aspect of using SPNP lies in its transient simulation. This can only produce a single reward variable value at a time although it allows the definition of a set of instants to solve the reward variable.This has already been discovered and was reported to the SPNP team while this research was being conducted. Unfortunately, this is yet to be fixed. Also, another missing requirement in using SPNP is that there is no possibility of accessing the log file that contains the simulation replicas. The reasons for continuing to use SPNP despite its inability to serve the purpose are stated in Section 6.4.

Möbius Tool: The factors for choosing Möbius lie in its capability of representing Stochastic Petri Nets (SAN in particular), as well as, its powerful transient simulation and the ability to extract the simulation observations (replicas) necessary for completing the prediction process. These replicas are stored in a text file that can be extracted directly by the WslaCP to perform the required computations.

The disadvantage of using Möbius is its inability to offer a single compact textual file that represents the model as a whole. This prevents the WslaCP tool reading from or inserting into it any code automatically. This is because, since Möbius is so powerful in terms of building a hierarchical model, a model can be composed of a set of atomic and composite models that are written as C classes. Hence, many files are produced for a single model with each being devoted to a specific functionality (e.g. atomic, composite, study, reward, solver files). This is very complicated to upload to, as well as in terms of being analysed by the WslaCP tool. In addition, it is hard to call the Möbius solver from the command line and pass onto it the required files.

Recalling Figure 6.1, the service model has to be available in a textual format. Using the SPNP tool means that the *Textual Model File* is written in CSPL, while it is written in C language when using Möbius.

6.3.3 MS Engine Implementation

In this section, the implementation of the *Metric Specification Engine* is described. The software components of the MS Engine (i.e. the *Model Parser*, *SLA Parser* and the *Mapper*), designed in Section 6.2.3.1, are implemented using three Java executables. The *Model Parser* executable utilises a text parser which uses a Java Scanner class to parse the textual model file using regular expressions. On the other hand, the *SLA Parser* executable utilises a DOM parser to parse the XML-based WSLA file. The DOM parser is used instead of the SAX because the SLA document is usually not large and the parser needs to visit different locations at the same time. This parser implements an algorithm to extract the prediction-related elements automatically from the WSLA and write them into a new XML file; this is a short version of the WSLA file. The reason for producing this file is to obtain a separate WSLA document that includes only the information related to prediction. Starting from this new file, the *Mapper* executable uses another DOM parser that implements an algorithm to perform the mapping from WSLA elements in order to produce the two files required: the *SLA-Model File* and the *Functions File*.

According to the mapping of service operation, measurement, and schedule described in Sections 5.2.1, 5.2.2, and 5.2.3 respectively, the *SLA-Model File* contains some model primitives, together with the reward variables and the time to solve them. This means it contains those elements from the WSLA that the modelling tool needs to produce their values. The *Functions File*, as noted in Section 5.2.4, contains the computational functions applied on the solver result in order to produce the desired SLA compliance probability. As described in the tool requirements, these files are written in a format that does not depend on the kind of augmented modelling tool. Hence, the implementation choice for the *Functions File* is to write it using Matlab. However, for the *SLA-Model File*, an `SDESsch` schema is developed whose instants are used as the intermediate language to write this file. The two implementation languages, along with a sample file representing the *SLA-Model File* and the *Functions File*, are described in the following sub-sections.

6.3.3.1 `SDESsch` Schema for Expressing the *SLA-Model File*

The `SDESsch` schema is developed to represent the *SLA-Model File*, i.e. the WSLA elements mapped to the SDES model, in a machine-readable format following the normal SDES formalism context. The value of this schema is that it enables the WslaCP tool to read and manipulate the *SLA-Model File* automatically. In addition, it allows this file to be used by different modelling formalisms/tools by translating

it into the chosen formalism/tool-specific language.

An XML schema for representing an SDES model was developed in [148] for its instances to be used as inputs to the SimGine simulation. However, the developed schema in this work differs in that it does not represent the whole SDES model. Rather, it only considers the SDES elements related to the results of mapping the SLA elements. Also, the rate reward functions of the reward variable are represented here in a more formal way, rather than the one in [148] where a subset of C++ fragments was used to represent the reward function. In addition, the `SDESsch` considers new elements that are not related directly to the SDES model description and reward variable definitions. Instead, they are used to offer extra help to the user in terms of choosing the appropriate primitives to complete the model or the reward function.

The entity declaration DTD of the `SDESSch` is defined in Listing 6.1. The rationale for using DTD is to give the reader a complete overview of the elements of the schema. For a full description of `SDESSch`, please refer to Appendix A.

Listing 6.1: The DTD of the `SDESSch` Schema

```
<?xml version="1.0" encoding="UTF-8"?><!ENTITY %sdes_prefix "sdes">
<!ENTITY %sdes_prefix.. "%sdes_prefix;:"><!ENTITY % documentElementAttributes"
xmlns:%sdes_prefix;CDATA 'D:/writingup2012/sdes'">
<!--element name mappings-->
....Omitted
<!--element and attribute declarations-->
<!ELEMENT %sdes..SV; (%sdes..Name;, (%sdes..Value;)?)>
<!ELEMENT %sdes..A; (%sdes..Name;, (%sdes..Rate;)?, (%sdes..InputS;)?,
                    (%sdes..OutputS;)?)>
<!ELEMENT %sdes..RV; ((%sdes..rvRate;)?, (%sdes..rvImp;)?, %sdes..rvInt;,
                    (%sdes..hint;)?)>
<!ATTLIST %sdes..RV;
        type (Gauge | Counter | InvocationCount | ResponseTime | DownTime | Status)
                    #IMPLIED
        name CDATA #IMPLIED>
<!ELEMENT %sdes..SDES; ((%sdes..SV;)+, (%sdes..A;)+, (%sdes..RV;)+)>
<!ATTLIST %sdes..SDES;
        name CDATA #IMPLIED
        %documentElementAttributes;>
<!ELEMENT %sdes..Name; (#PCDATA)>
<!ELEMENT %sdes..Value; (#PCDATA)>
<!ELEMENT %sdes..Rate; (#PCDATA)>
<!ELEMENT %sdes..InputS; (#PCDATA)>
<!ELEMENT %sdes..OutputS; (#PCDATA)>
<!ELEMENT %sdes..rvRate; ((%sdes..IF;, (%sdes..ElseIf;)?, (%sdes..ElseReturn;)?) |
    (%sdes..Return;))>
<!ATTLIST %sdes..rvRate;
        %documentElementAttributes;>
<!ELEMENT %sdes..IF; (%sdes..condition;, %sdes..return;)>
<!ELEMENT %sdes..ElseIf; (%sdes..condition;, %sdes..return;)>
<!ELEMENT %sdes..ElseReturn; (#PCDATA)>
```

```
<!ELEMENT %sdes..Return; (#PCDATA)>
<!ELEMENT %sdes..stv; (#PCDATA)>
<!ELEMENT %sdes..R; (#PCDATA)>
<!ELEMENT %sdes..V; (#PCDATA)>
<!ELEMENT %sdes..And; ((%sdes..condition;, %sdes..condition;))>
<!ELEMENT %sdes..Or; ((%sdes..condition;, %sdes..condition;))>
<!ELEMENT %sdes..Not; (%sdes..condition;)>
<!ELEMENT %sdes..condition; (((%sdes..stv;, %sdes..R;, %sdes..V;) | %sdes..And; | %
    sdes..Or; | %sdes..Not;))>
<!ELEMENT %sdes..rvImp; (%sdes..ac;, %sdes..return;)>
<!ATTLIST %sdes..rvImp;
        %documentElementAttributes;>
<!ELEMENT %sdes..ac; (#PCDATA)>
<!ELEMENT %sdes..return; (#PCDATA)>
<!ELEMENT %sdes..hint; (%sdes..svH; | %sdes..aH;)>
<!ATTLIST %sdes..hint;
        %documentElementAttributes;>
<!ELEMENT %sdes..svH; (#PCDATA)>
<!ELEMENT %sdes..aH; (#PCDATA)>
<!ELEMENT %sdes..rvInt; (%sdes..S;, %sdes..E;, %sdes..I;)>
<!ATTLIST %sdes..rvInt;
        type (Instant | Interval) #IMPLIED
        %documentElementAttributes;>
<!ELEMENT %sdes..S; (#PCDATA)>
<!ELEMENT %sdes..E; (#PCDATA)>
<!ELEMENT %sdes..I; (#PCDATA)>
```

Depending on the `SDESSch` schema, and on mapping WSLA measurements to SDES represented in Section 5.2.2, the *SLA-Model File* can be in one of two cases, depending on the reward function template status: i.e., whether it is complete or incomplete. There is a need to distinguish between these two cases because the user has to react differently to them in a later step. These cases are as follows:

1- The Reward Function Template is Complete: This is the case when the measurement directive used in the WSLA is one of the following: *Gauge*, *Counter*, or *InvocationCount* when the reward function is generated completely with a specific state variable/action that represents the service operation. In this case, the user has to confirm only, in a later step, that the automatically chosen state variable/action is correct; otherwise, he/she has to select the correct one from the set of other state variables/actions in the model. An example of this type of reward function template inside a reward variable generated by the MS Engine, is presented in Listing 6.2.

Listing 6.2: An *SLA-Model File* with a complete template

```
<sdes:SDES xmlns:sdes="D:/writingup2012/sdes"   xmlns:xsi="http://www.w3.org/2001/
    XMLSchema-instance" xsi:schemaLocation="D:/writingup2012/sdes file:///D:/
    writingup2012/sdesXML/SDESschema.xsd" name= "DemoService">
<!-- Model Primitives -->
 1:   <sdes:SV>
 2:          <sdes:name>GetQote_s</sdes:name>
```

```
 3:    </sdes:SV>
 4:    <sdes:A>
 5:          <sdes:name>GetQuote_a</sdes:name>
 6:          <sdes:InputS>GetQuote_s</sdes:InputS>
 7:    </sdes:A>
<!-- Reward Variable: Function + Time -->
 8:    <sdes:RV type="Gauge" name="Gauge_GetQuote">
 9:        <sdes:rvRate>
10:           <sdes:Return>GetQuote_s</sdes:Return>
11:        </sdes:rvRate>
12:        <sdes:rvInt type="Instant">
13:            <sdes:S>0</sdes:S>
14:            <sdes:E>100</sdes:E>
15:            <sdes:I>10</sdes:I>
16:        </sdes:rvInt>
17:    </sdes:RV>
</sdes:SDES>
```

In this listing, the mapping of the service object (i.e. the `GetQuote` operation) is defined by a single state variable `GetQuote_s` inside the `sdes:SV` element (line 2) and a single action `GetQuote_a` inside the `sdes:A` element (line 5) where the state variable `GetQuote_s` is the input of this action (line 6). The reward variable that represents the *Gauge* measurement is represented as a rate reward `rvRate` inside the `sdes:RV` element (line 9) that its reward function returns the value of the state variable `GetQuote_s` (line 10). The name of this reward variable is created automatically by attaching the name of the measurement to the name of the operation `Gauge_GetQuote` (line 8). It is then solved as an instant of time (line 12), starting at 0 and ending at 100 with a step size of 10 defined inside the `S, E, I` elements respectively (lines 13, 14, and 15). The contents in this listing are produced automatically. The user has only to confirm that the state variable used for the reward variable is `GetQuote_s`[1]. If it is not, he/she has to choose the correct state variable from the set of state variables available in the model[2].

2- The Reward Function Template is Incomplete: This is the case when the measurement is one of the following: *ResponseTime*, *Status*, or *DownTime* when the reward function does not refer only to the model's primitives representing the service operation. This case is more complicated than the former because it includes defining new state variables, a condition, and a return statement. Although in Section 5.2.2 primitives were assigned automatically to the mapping of these measurements, they

[1]If some primitives of the service model are generated automatically from SLA, the name of the state variables and actions come from the name of the service operation. Hence, the chosen state variable in the reward variable is typically the one specified in the templates.

[2]This case may occur when the model is built manually by a user. In this case, he/she might represent the model primitives or assign their name differently.

are, however, often more complicated and hence the reward functions are left blank. Thus, the user, in a later step, has to choose the state variable with its condition and the return value which completes the reward function definition. An example of this case, and depending on Listing 4.2, is the XML instance in Listing 6.3.

Listing 6.3: An *SLA-Model File* with incomplete reward function template employing Listing 4.2

```
 1:   <sdes:SV>
 2:          <sdes:name>GetQuote_s</sdes:name>
 3:   </sdes:SV>
 4:   <sdes:A>
 5:         <sdes:name>GetQuote_a</sdes:name>
 6:         <sdes:InputS>GetQuote_s</sdes:InputS>
 7:   </sdes:A>
   <!-- Reward Variable: Function + Time -->
 8:<sdes:RV type="Status" name="Status_GetQuote">
 9:  <sdes:rvRate>
10:    <sdes:IF>
11:       <sdes:condition/>
12:       <sdes:return/>
13:    </sdes:IF>
14:     <sdes:ElseReturn> </sdes:ElseReturn>
15:  </sdes:rvRate>
16:  <sdes:rvInt type="Instant">
17:      <sdes:S>0</sdes:S>
18:      <sdes:E>44640</sdes:E>
19:      <sdes:I>1</sdes:I>
20:  </sdes:rvInt>
21: </sdes:RV>
```

In this listing, the reward variable that represents the *Status* measurement is defined as a rate reward (line 9). The rate reward function is composed of two elements, `condition` (line 11) and `return` (line 12). The user has to choose in a later step what condition is required in order for the service to be up and what reward is earned while this condition is true. The user might also specify via `ElseReturn` (line 14) what the reward will be earned while the condition is false. In `SDESsch` there is a flexibility in defining multiple conditions to express the actual condition of the service status. An example of completing the reward function template of Listing 6.3 with multiple conditions is displayed in the content of Listing 6.4.

Listing 6.4: Completing the reward function template of Listing 6.3

```
 2:  <sdes:rvRate>
 3:    <sdes:IF>
 4:      <sdes:condition>
 5:        <sdes:And>
 6:          <sdes:condition>
 7:            <sdes:sv>DF</sdes:sv>
 8:            <sdes:R>Less</sdes:R>
 9:            <sdes:V>10</sdes:V>
10:          </sdes:condition>
```

```
11:          <sdes:condition>
12:             <sdes:sv>TR</sdes:sv>
13:             <sdes:R>Less</sdes:R>
14:             <sdes:V>5</sdes:V>
15:          </sdes:condition>
16:       </sdes:And>
17:    </sdes:condition>
18:    <sdes:return>1</sdes:return>
19:    </sdes:IF>
20:      <sdes:ElseReturn>0</sdes:ElseReturn>
21:  </sdes:rvRate>
```

In this listing, the condition is composed of two nested conditions (lines 6 and 11) joined by the `And` logical operator (line 5). The first one specifies a state variable `DF` to be `Less` than 10 (lines 7, 8 and 9), while the second specifies a state variable `TR` to be `Less` than 5 (lines 12, 13 and 14). The reward function will return 1 (line 18) if the condition is valid and 0 (line 20) otherwise.

The *SLA-Model File* produced from the MS Engine executable is assigned a name which is equal to the service name (*DemoService* for this particular example in Listing 4.2); it has an extension of **.m** (to denote a set of model primitives).

6.3.3.2 Matlab for Expressing the *Functions File*

Matlab (MATrix LABoratory) [149] is incorporated in this implementation to represent the *Functions File*. Using Matlab to express the content of the *Functions File* means a computer can read WSLA undisputed mathematical meaning clearly. Also, it allows different implementation languages to make use of and implement this file. Matlab is chosen since it is powerful enough in expressing maths and because well-known programming languages, such as C or Java, have libraries that can automatically map code written in Matlab. Other mathematical languages such as OpenMath [150] can be chosen; however, Matlab's coding is more concise.

A Matlab equivalent functions' header is produced automatically by the Mapper executable of the MS Engine for each function in the WSLA document and for the SLO. These functions are applied either on the result of the model solver, the *Solver Results File*, or on the result of another Matlab function. Although Matlab is used to express the header of the functions in the *Functions File*, the functionality (function implementation) of the more complex WSLA functions may also be specified in Matlab in order to help implement them correctly in the desired language. Accordingly, the notation used in the *Functions File* regarding data types, variable definitions, and function definitions will follow Matlab-style language.

The functions in the *Functions File* can be in one of several cases. There is a need to distinguish between these cases in order to know which functionality needs

to be implemented using Matalb and which do not. These cases are as follows:

1. If a WSLA function is one of the time series functions, it will be represented as an array equals to the solver output. This is presented in Table 6.3.

Table 6.3: WSLA functions equal to solver output

WSLA Function	Math in Matlab
TSConstructor/QConstructor	$function[TSC[]] = solver\_output$

2. If a WSLA function has the same functionality as an existing Matlab function, it will be referred to it directly. This means that, if the WSLA function is a well-known Matlab function, only the header of the function will be produced since its functionality is intuitive. Well-defined WSLA functions, together with their corresponding Matlab functions, are presented in Table 6.4.

Table 6.4: Matlab functions equivalent to WSLA functions

WSLA Function	Math in Matlab
Size	$function[size] = length(TSC[])$
Mean	$function[w_{mean}] = mean(TSC[])$
Median	$function[w_{median}] = Median(TSC[])$
Mode	$function[w_{mode}] = mode(TSC[])$
Round/Truncate	$function[w_{round}] = round(v)$
sum	$function[w_{sum}] = sum(TSC[])$
Max	$function[w_{max}] = max(TSC[])$
Plus/Minus/Multiply/Divide	+/-/*//

3. If a WSLA function is not equivalent to any well-defined Matlab function, the header, along with the function body (functionality), will be defined in Matlab to prevent any misunderstanding when it is implemented. The complex WSLA

Table 6.5: Matlab function' headers of the complex WSLA functions

WSLA Function	Math in MATLAB
TSSelect	$function[v] = TSS(TSC[], x)$
ValueOccurs	$function[v[]] = VO(TSC[], x)$
PercentageGreaterThanThreshold	$function[v[]] = PGTT(TSC[], x)$
PercentageLessThanThreshold	$function[v[]] = PLTT(TSC[], x)$
NumberGreaterThanThreshold	$function[v[]] = NGTT(TSC[], x)$
NumberLessThanThreshold	$function[v[]] = NLTT(TSC[], x)$
Span	$function[v[]] = Span(TSC[], x)$
RateOfChange	$function[v[]] = RoC(TSC[], sch[], x)$

function headers are presented in Table 6.5. It appears from this table that the header does not provide any particular meaning. Hence, the functionality definition is important, as is specified in the following example.

Table 6.6: Sample of the *Functions File* that contains the functions of Listing 4.2

```
//Functions Headers
  function[TSC[]]=solver_output
  function[xCount[]]= Span(TSC[],0)
  function[slo]= Less(xCount[],10)

//Complex Functions Functionality
function[xCount[]]= Span(TSC[],0)
 xCount=0;
 i=0;
 for i=0:1:length(TSC[])
   if TSC(i)= 0
    xCount=xCount+1;
    xCount[i]=xCount;
   else
    xCount=0;
    xCount[i]=xCount;
   end
 end
return(xCount[])
end
```

Table 6.6 contains an example of a sample output of the *Functions File* that contains the functions of Listing 4.2. The output file contains the method headers and then the functionality descriptions of the complex ones. In this table, the functionality of the *Span* function is defined; this is not stated explicitly in WSLA and does not have any equivalent in Matlab. Hence, the pseudo code of the *Span* functionality is stated because it gives a better degree of confidence when implementing the function than writing *Span* only. Others WSLA function implementations in Matlab are specified in Appendix B.

The *Functions File* produced from the MS Engine executable is assigned a name which is equal to the service name (*DemoService* for this example); it has an extension of **.f** (to denote a set of **f**unctions).

6.3.4 TS Engine Implementation

In this section, the implementation of the *Translator Engine* is described. The three software components of the TS Engine (i.e. *In-Out Translator*, *Out-In Translator* and *Inner Translator*), which were designed in Section 6.2.3.2, are implemented as Java executables. These executables utilise a set of algorithms to translate the

intermediate files (i.e. *SLA-Model File* and *Functions File*) into the plugged-in tool's input language and the WslaCP implementation language respectively. In what follows, the implementation of these three software components is presented.

6.3.4.1 The In-Out Translator Implementation

The *In-Out translator* translates the *SLA-Model File*, which is written according to `SDESSch`, into the input language of the tool that solves the type of SDES model being used; this generates the *Reward Model File*. Before generating this file, the translator creates part of the *Textual Model File* (if it is not predefined) by translating the model-related parts of the *SLA-Model File*; this is completed by the user. In the current status of the implementation, the WslaCP tool is capable of making the transformation to CSPL and, with a few alterations, it can be transformed into another language. For this reason, SPNP is used to describe the implementation of the In-Out translator. Given that the plugged-in tools is SPNP, the *Textual Model File* and the *Reward Model File* are expressed in CSPL. Hence, these files have the service name but with the extension "*.spnp*" and "*.c*" respectively (the latter represents the extension of the SPNP solver input file).

The translation of the *SLA-Model File* through the *In-Out translator* passes through the following steps:

1. Parsing the file to translate the model primitives being used.
2. Generating the GUI for completing the model creation by the user.
3. Parsing the file to recognise type of the reward variables being used.
4. Generating the GUI that suits the specific type of reward variable.
5. Accepting the user input that completes the reward function definition.
6. Creating the *Reward Model File* with the reward functions.
7. Adding the instant/interval of time required to solve the reward functions.

These steps are described in detail in what follows.

1- Parsing the File to Translate the Model Primitives being used: The translator parses the first part of the *SLA-Model File* to translate the state variables/actions into the corresponding places/transitions of the SPNP model. It then stores them in a text file that has the service name with the extension *.spnp* (*demoService.spnp* in this example). This file represents part of the *Textual Model*

Table 6.7: The translation of the model primitives of the *SLA-Model File* of Listing 6.3 from SDESSch into CSPL

```
/* ========= DEFINITION OF THE NET ========== */
void net(){
/* ====== PLACE ====== */
place("GetQuote_s");

/* ====== TRANSITION ====== */
/* Timed Transitions */
rateval("GetQuote_a",1);

/* ====== ARC ====== */
/* Input Arcs */
iarc("GetQuote_s","GetQuote_a");}
```

File. Recalling the information relating to model primitives in Listing 6.2, the automatic translation from SDESSch to CSPL can be as it appears in Table 6.7.

The second part of the *SLA-Model File* that contains the reward variable definitions is postponed to step 3 in order to separate the model completion step from the reward function completion step.

2- Generating the GUI for Completing the Model Creation by the User: After translating the model primitive into the plugged-in tool input (CSPL here), a

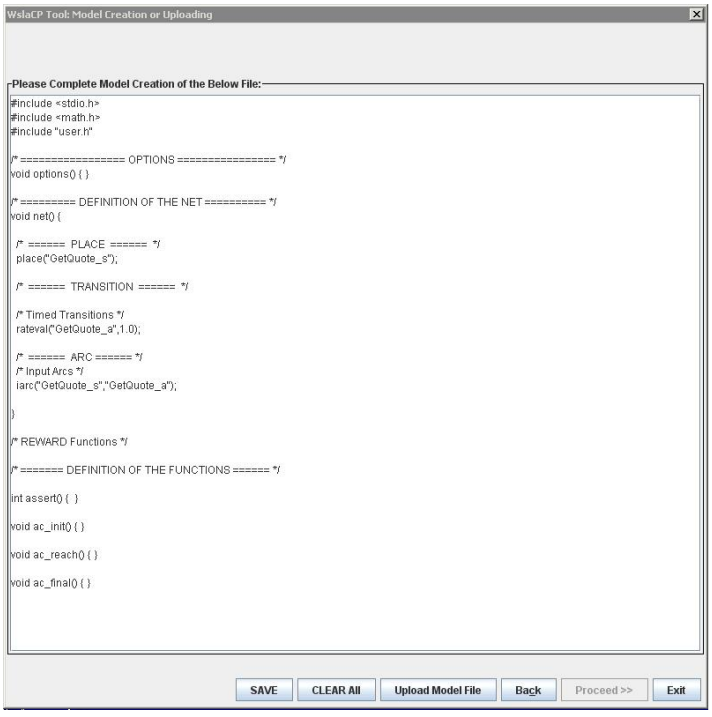

Figure 6.7: A WslaCP GUI for completing the model creation

GUI is presented to the user to allow the model creation to be completed manually (see Figure 6.7). After completing the model, the *Textual Model File* (which is *demoService.spnp* here) is updated. If the user does not want to complete the model creation manually, he/she can upload the model file that contains a predefined textual description of the model (see the 'Upload Model File' button in Figure 6.7). In this case, this file replaces the *Textual Model File*.

3- Parsing the File to Recognise the Type of Reward Variables being Used: This is necessary for deciding the type of GUI the tool will present in the next step. If the reward variable represents *Gauge*, *InvocationCount* or *Counter*, the reward variable template regarding the reward function is complete since it refers to the state variable or action representing the service operation (as specified in Section 6.3.3.1). This can be seen in what follows.

Gauge → `<sdes:rvRate>`
`<sdes:Return>`*state_variable_name*`</sdes:Return>`
`</sdes:rvRate>`

InvocationCount → `<sdes:rvImp>`
`<sdes:a>`*action_name*`</sdes:a>`
`<sdes:Return/>1</sdes:Return>`
`</sdes:rvImp>`

Counter → `<sdes:rvImp>`
`<sdes:a>`*action_name*`</sdes:a>`
`<sdes:Return/>1</sdes:Return>`
`</sdes:rvImp>`

Alternatively, if the reward variable represents *StatusRequest*, *Status*, *ResponseTime*, or *DownTime*, the reward function template is incomplete since it does not refer directly to the service operation primitives and hence it is hard for this to be generated automatically (as specified in Section 6.3.3.1). The part of the reward variable with regard to the incomplete reward function is presented in what follows.

StatusRequest, *ResponseTime*, *DownTime* →

```
<sdes:rvRate>
    <sdes:IF>
        </sdes:condition/>
        <sdes:return/>
    </sdes:IF>
    <sdes:ElseReturn/>
</sdes:rvRate>
```

The condition in this case is a place, an arithmetic relational operator, and a value, in addition to a logical operator, when multiple conditions are used.

4- Generating the GUI that Suits the Specific Type of Reward Variable: If the reward function templates are complete, the user has to specify whether or not the usage of the model primitives in this template is correct. If it is not, then he/she will be asked to choose the correct state variable or action from the model. However, if the reward function templates are incomplete, the user will be asked to specify the condition needed to complete the reward function. In either of these cases, the GUI will be tailored according to the type of reward variable. This means that the dedicated GUI will present only a suitable set of primitives (i.e. places and transitions available in the model) for this kind of reward variable. For example, for the reward variable representing *Status*, the user will be asked to choose the relation that specifies when the system is up. This is achieved through a user interface (Figure 6.8) that displays, in a drop-down list, all the places available in the model from which to choose a set of these with the corresponding number of tokens and an arithmetic relation.

Another example is when the reward variable represents *Gauge*. In this case, the

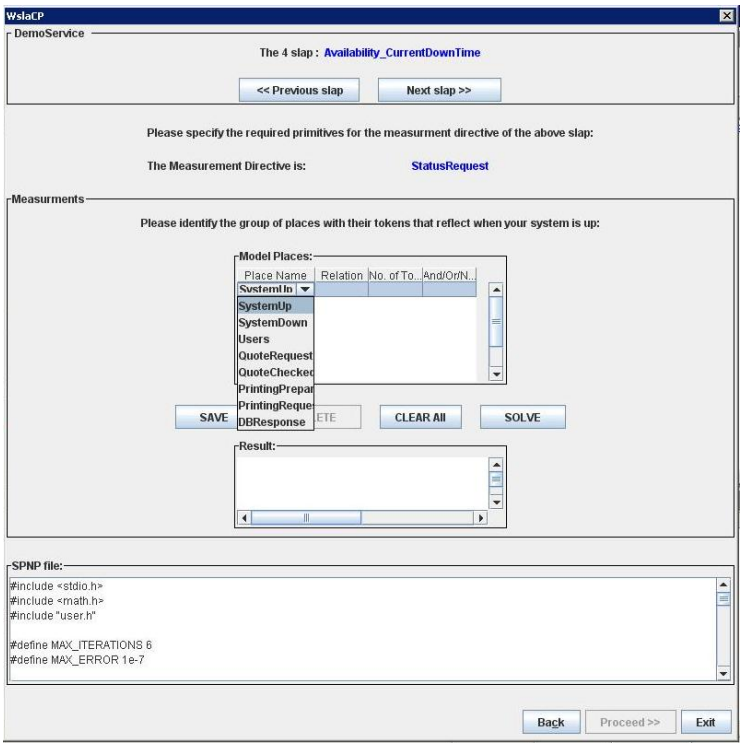

Figure 6.8: A WslaCP GUI for completing the reward definition of *Status*

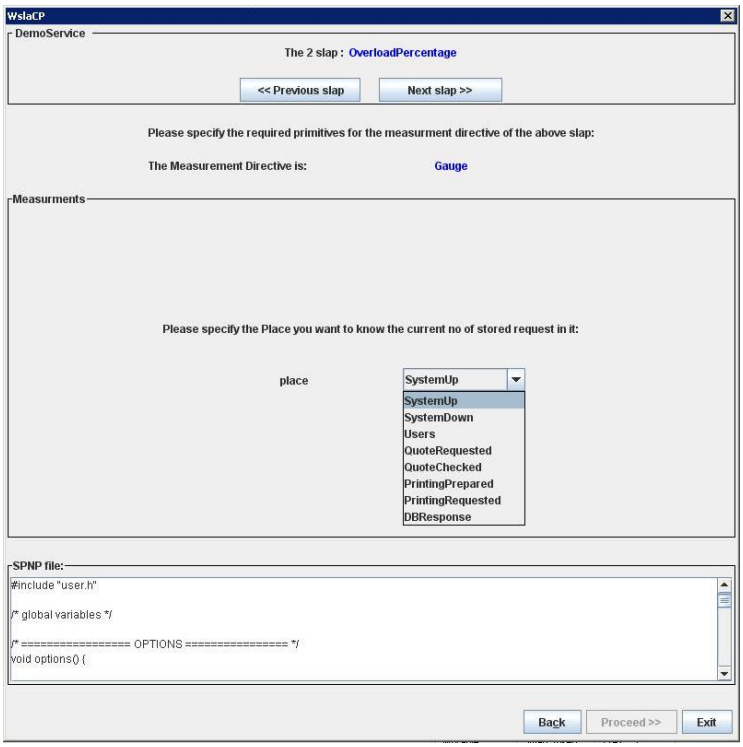

Figure 6.9: A WslaCP GUI for completing the reward definition of *Gauge*

user will be asked, through a GUI (Figure 6.9), to choose the state variable he/she wants for measuring gauge if this is not the state variable representing the service operation. This is achieved using a drop-down list that is populated automatically

Table 6.8: The updated reward function in the *SLA-Model File* of Listing 6.3

```
<sdes:RV type="Status" name="Status_GetQuote">
  <sdes:rvRate>
     <sdes:IF>
      <sdes:condition>
            <sdes:sv>UP</sdes:sv>
            <sdes:R>Equal</sdes:R>
            <sdes:V>1</sdes:V>
      </sdes:condition>
      <sdes:return>1</sdes:return>
     </sdes:IF>
     <sdes:ElseReturn>0</sdes:ElseReturn>
  </sdes:rvRate>
  <sdes:rvInt type="Instant">
     <sdes:S>0</sdes:S>
     <sdes:E>44640</sdes:E>
     <sdes:I>1</sdes:I>
  </sdes:rvInt>
</sdes:RV>
```

with all the places available in the model from which to choose an option.

5- Accepting the User Input that Completes the Reward Function Definition: Once the GUIs related to completing the reward function appear, the user has to choose the relevant model primitives. Once this is done, the tool will automatically update the *SLA-Model File* with the entered values. For example, if the value that is entered for completing the *Status* reward function is: the state variable *UP* is equal to a value of 1, then the updated *SLA-Model File* will appear as in Table 6.8.

Table 6.9: The equivalent CSPL code of the reward function presented in Table 6.8

```
double Status_GetQuote() {
   if (mark("UP")> 0)
    return (1.0);
   else
    return (0.0); }
```

The tool will then automatically generate the CSPL code of this completed reward function. For example, the code in Table 6.9 represents the CSPL equivalent of the reward template from Table 6.8. This generated CSPL code will be saved in the tool's memory to be used in the next step.

6- Creating the Reward Model File with the Reward Functions: In this step, the tool creates the *Reward Model File* (which represents the solver input) and it gives it the name of the service with the extension *.c*. The tool then copies the content of the model description that is taken from the *Textual Model File* (*demoService.spnp* in the running example) into this new file *demoService.c*. After this, it automatically inserts the generated CSPL reward functions into the appropriate place inside the *demoService.c* file; this is after the net definition and before the $ac\_final()$ function. This file content will appear in the lower part of Figure 6.8 each time a new reward variable is completed and translated.

7- Adding the Instant/Interval of Time Required to Solve the Reward Functions: After creating and inserting the reward functions into the *Reward Model File* (*demoService.c* here), the last part of the reward variable template in the *SLA-Model File* is used. This part concerns the time required to solve the reward variable and is translated by the tool automatically into the equivalent CSPL code for solving the reward functions.

Since the *Reward Model File* is used as the input to the solver of the plugged-in tool, the instructions for solving the model have to be inserted in the $ac\_final()$ function. This SPNP function is the one that tells the tool which outputs need to be computed. All the variable definitions necessary for the loop, which are defined in ac_final, are also generated and inserted automatically.

Table 6.10: A sample of the time generated from Table 6.8 for insertion into the *Reward Model File*

```
void ac_final(){
 int loop=0;
 /* Compute the reward function for the interval
 and time period specified in the schedule.  */
 for (loop=0; loop < 44640; loop+=1)
 {
 solve ((double) loop);
 expected(Status_GetQuote);
 }
```

Given the example in Table 6.8, the status of the system is checked every minute during a 44640 minute period. This is reflected in Table 6.10 through the use of a 'for' loop, where *solve*() is the function used to solve the SPNP model.

To inform the SPNP solver to use the simulation rather than the numerical solver, the command *iopt(IOP_SIMULATION,VAL_YES)* needs to be placed in an SPNP function called *option*(); this is also done automatically. Any other preferences, such as the simulation confidence interval, the simulation seeds, and the maximum and minimum replicas, also go here.

Given all the aforementioned steps performed by the *In-Out Translator*, the generated *Reward Model File* will contain a complete version of the model description, in addition to the reward variables and the time to solve them. Hence, this file is ready to be solved via the solver of the plugged-in modelling tool in order to produce the result of each reward variable at each time instant/interval.

Calling the solver, which is simulation-based, is triggered by clicking the solve button, as shown in Figure 6.8. The result of solving the model is the *Solver Results File* that contains n realisations for each reward variable at each instant. The number of realisations (replicas), n, is determined according to the user's preference regarding the confidence interval. These results are stored in a text file, the *Solver Results File*, in which each line represents one realisation of the reward variable at all the required time instants/intervals.

Given the implementation of the plugged-in tool in Section 6.3.2.2, SPNP cannot solve the output of the *In-Out Translator* using simulation. Hence, Möbius is used

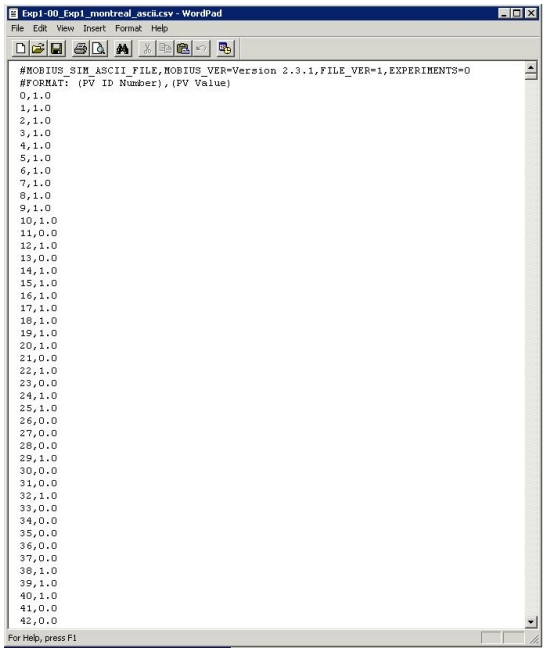

Figure 6.10: An example of Möbius trace file

instead. However, Möbius is not called implicitly from the WslaCP tool; rather, it is run manually by the user because of the limitation described in Section 6.3.2.2. The simulation trace file generated by Möbius (written in an ASCII format into *.txt* or *.cvs* files as shown in Figure 6.10) is taken then as input to the WslaCP tool to complete its work. This trace contains, in each line, the ID number for the obtained reward variable with the value of this variable. These are separated by a comma.

6.3.4.2 The Out-In Translator Implementation

The *Out-In Translator* translates the simulation trace stored in the *Solver Results File*[1] (*DemoService.cvs* here for Möbius) into a unified format. This translator implements an algorithm to read each line and then stores it in a matrix of size equal to the $number\ of\ realizations \times total\ number\ of\ instants/intervals$. This algorithm ignores any text that is specific to the solver format. For example, if the simulation trace has 100 realizations and the reward variable is obtained for 44640 time instants (as specified in Table 6.10), then the new *Unified Results File* will look like the one illustrated in Table 6.11.

Inside this file, the simulation replicas related to a specific reward variable are preceded by its name. This is a combination of the $MeasurementDirective$ name and the *Operation* name separated by an underscore sign (*Status_GetQuote* for the

[1]SPNP's output file extension is *.out*

Table 6.11: A sample output of the *Unified Results File*

```
Status_GetQuote results:
   1:   1   1   0   0   0   0   0  ...  0   0   0
   2:   1   1   1   0   0   0   0  ...  0   0   1
...
 100:   1   0   0   0   0   0   1  ...  0   0   0
```

running example). The whole generated file (i.e. the *Unified Results File*) will be stored under a name which is a combination of the service name and the word *result* separated by an underscore sign (*demoService_result.txt* for the running example).

6.3.4.3 The Inner Translator Implementation

This translator implements the functions specified in the *Functions File* in the language in which the WslaCP is implemented; in this case, this is Java. This implementation is stored in a separate *Functions Implementation File* that has the service name with the extension $.fi$ (to denote the **f**unction's **i**mplementation). Because of its simplicity, a sample file is not presented here.

6.3.5 Implementation of the RCC Engine

The *Computation & Comparing* software component in this engine is implemented as a Java executable that uses a specific algorithm in order to predict the probability of SLA compliance. The implemented algorithm is presented in Table 6.12. Its inputs are the simulation replicas, R, that are stored in the *Unified Results File* and the functions, $Func$(s), that are stored in the *Functions Implementation File*. For each replica $R_i, i \leq n$ (line 8), where n is the total number of replicas, the algorithm takes the value of the reward variable v_j for each instant $j \leq w$ (line 9), where w is the number of time instants/intervals for which the reward variable are solved, and applies all the functions $Func$(s) in sequence (line 10). The result of applying the final function for each replica will be w different values of the *slap* (this is represented in line 11 using $slap_{v_j}$). These results are compared by the algorithm against the threshold specified in the evaluation function of the *slo*. The result will be w answers of True/False for each replica (this is represented in line 12 using slo_{v_j}). The algorithm then counts the number of $True$ answers, Y, (which means the *slo* is met) for each value of each replica (lines 14 and 15) and then divides the sum by the number of instants w to produce the probability that *slo* is met in this replica (this is represented using pr_{R_i} in line 16). It then computes the summation of the probability compliance for each replica and divides it by n to give the probability of

Table 6.12: An algorithm for computing the *slo* compliance probability from simulation replicas

1: /*Input : R: Simulation replicas of the *Unified Results File**/
2: /* $Func$(s): Functions of the *Functions Implementation File**/
3: /*Output : Finding SLA compliance probability*/
4: /*Variables: n: Number of replicas*/
5: /* w: Number of time instants/intervals in each replica*/
6: /* v_j: Value of the reward variable at index $j \leq w$*/
7: //After Solving the Model:
8: For each replica $R_i, i \leq n$
9: For each value $v_{j \leq w}$
10: 1.1 Apply all the $Func$(s) in sequence;
11: 1.2 Assign the final value to SLA parameter, $slap_{v_j}$;
12: 1.3 Evaluate SLO to True/False, slo_{v_j};
13: End;
14: Y=0 (number of instances satisfying *slo* threshold at each slo_{v_j});
15: For all $slo_{v_j}, j \in w$
16: If (slo_{v_j}=*True*) then Y=Y+1;
17: $pr_{R_i}=\frac{Y}{w}$;
18: End;
19: End;
20:Return ($pr_{slo} = \frac{\sum pr_{R_i}}{n}$);

SLA compliance pr_{slo} for all the n replicas (line 20).

To simplify this, the algorithm presented in Table 6.12 is applied for the functions in the example of Listing 6.6. This means that the simulation output of Listing 6.11 will be parsed first to apply the *Span* function on it, as appear in Listing 6.5.

Listing 6.5: Applying the *Span* function on the simulation output of Table 6.11

```
  1: 0 0 1 2 3 4 5 ...8 9 10
  2: 0 0 0 1 2 3 4 ...5 6 0
...
100: 0 1 2 3 4 5 0 ...1 2 3
```

Then, the result will be compared against SLO threshold as specified in Listing 6.6. This means for each instant in the replica, if the result is less than 10 then true will be returned (`T`) else it will be false (`F`). This is appeared in Listing 6.6.

Listing 6.6: Evaluating the result of Listing 6.5 to True/False

```
  1: T T T T T T T...T T F
  2: T T T T T T T...T T T
...
100: T T T T T T T...T T T
```

The probability of SLO compliance for each replica is computed, as appeared in

Listing 6.7, by summing the true value and dividing them by the number of instants.

Listing 6.7: Computing the probability of *slo* compliance for each replica

```
  1: 13000/44640
  2: 14530/44640
......
100: 23400/44640
```

Finally, the ultimate SLO compliance probability will be the summation of the compliance probability of all the replicas divided by the number of replicas. This is displayed to the user through the engine specified GUI.

After describing the implementation of all the tool engines, the information related to the WslaCP output files is summarised in Table 6.13. This table contains the design name of the output file, its implementation name and its extension.

Table 6.13: A summary of the implementation of the WslaCP's output files

Output File Design Name	**Output File Implementation Name**	**Extension**
SLA-Model File	service_name	.m
Functions File	service_name	.f
Textual Model File	service_name	.spnp
Reward Model File	service_name	.c (SPNP)
Solver Results File	service_name	.txt/.cvs (Möbius) .out (SPNP)
Unified Results File	service_name_result	.txt
Functions Implementation File	service_name	.fi

6.4 Discussion

To explain the implementation choices that were made for the WslaCP tool, it is necessary to discuss briefly the challenges faced when carrying out the research for this PhD.

The reason for using SPNP in spite of its inability to satisfy the WslaCP requirements presented in Section 6.3.2.2 regarding the ability to run a transient simulation and retrieve its replicas, is that the WslaCP tool was originally built to be used with the transient analytic solver of SPNP rather than its simulation[1]. This tool was capable of implementing all the phases of the methodology for only a specific set of WSLA functions. This is because deriving a numerical solution of the WSLA metrics is prohibitively difficult, and sometimes impossible, for various WSLA functions.

[1]This was described at an early stage of this thesis in [137].

When it was realised that using an analytic solution was not possible for all WSLA functions, the direction of the work was modified to use simulation replicas as a way of accommodating all the WSLA functions. For this reason, the decision was taken to change the tool's design with regard to solving the model. Unfortunately, the SPNP transient simulation was not working properly and the simulation trace could not be tracked. In the light of this problem, the direction of the work had to change once again to use Möbius in the WslaCP tool instead of SPNP. Although using Möbius was possible and gave the intended results for the step of producing the reward variable results onwards, it was difficult to implement the tool steps in terms of solving the model backwards (i.e. the steps relating to parsing the model file, manipulating it, and then calling the solver implicitly).

Given these problems, other tools had to be investigated in order to check their applicability to the new requirements of the WslaCP tool. However, this was achieved for only a limited number of tools because of time limitations. Hence, it was not possible to search among all the stochastic modelling tools to find a more suitable one that would satisfy all the needs of the WslaCP tool. Because no tool from among the ones studied was ideal for implementing the proposed tool design, a step back was made to use both SPNP and Möbius in the following way. SPNP is used to implement the tool design from the point where the SLA document is uploaded until the reward model file creation is completed and the solver is called, while Möbius implements the tool design from the point at which the model is solved until the *slo* compliance prediction is produced.

The aforementioned implementation may seem unsatisfactory because the WslaCP tool does not provide one continuous flow from the SLA input to its compliance prediction. However, this implementation, using SPNP and Möbius, can be completed in the future when the SPNP simulation bug is solved, or when an easy way to produce a textual input of the Möbius model can be worked out. Hence, there is no reason why the architectural design cannot be fully implemented when the aforementioned solution are satisfied or when a plugged-in tool that satisfies all the tool requirements is utilised.

The implemented components of the WslaCP tool can be exploited for other purposes: for example, for learning purposes regarding SLA content and stochastic modelling, or as parts of other tools with different orientations, such as a tool which may validate SLA content. This implementation has also led to a different design choice that might be applied in future work, as specified in Section 6.2.4.

It should be noted that the change in the research's direction towards using simulation replicas and to consider each reward variable as a random variable (as

Table 6.14: Features implemented in the WslaCP tool

✓	supported	N/A	Not applicable (WslaCP tool specific)
✗	Not supported	∼	supported for a single tool

Design Engines		SPNP	Möbius	WslaCP
RS Engine	**SLA Parser**	N/A	N/A	✓
	Model Parser	✓	✗	∼
	Mapper	N/A	N/A	✓
TS Engine	**In-Out Translator**	✓	✗	∼
	Out-In Translator	✗	✓	∼
	Inner Translator	N/A	N/A	✓
RCC Engine	**Computation & Comparing**	✗	✓	∼

presented in [15]), might be achievable without obtaining these replicas if the current software modelling tools were capable of computing functions over random variables. Since this was hard to tackle in practice, simulation replicas was the issue to explore.

A summary of what the current implementation of the WslaCP tool is capable of doing is presented in Table 6.14. For the *RS Engine* in this table, the *SLA Parser* is not related to the plugged-in tool; hence, it is supported by WslaCP. The same is true for the *Mapper*. However, for the *Model Parser*, the WslaCP can parse the model represented in SPNP simple textual format, but this is not supported for Möbius. For the *TS Engine*, the *In-Out Translator* is implemented for SPNP but not for Möbius. The opposite is true for the *Out-In Translator* where only Möbius can solve the model using a transient simulation and allow the replicas to be extracted. The *Inner Translator*, on the other hand, is not related to the plugged-in tool; hence, it is implemented fully by the WslaCP. Finally, although the RCC Engine is not related to the plugged-in tool directly, it takes its output as an input to the *Computation & Comparing* components. Hence, this engine can be considered as supporting Möbius only.

6.5 Conclusion

This chapter addresses the architecture design and implementation of a software tool that automates the process of predicting SLA compliance. The chapter has two main contributions: firstly, the architectural design of the SlaCP tool that automates the SlaCP methodology; secondly, the implementation of this tool design based on WslaCP methodology. Also in this chapter, arguments concerning the inability to provide a complete working tool implementation are provided. In the next chapter, an evaluation of the WslaCP software tool is conducted via a case study employing a stock quote service.

Chapter 7

A Case-Study Based Methodology Evaluation

This chapter presents a case study to evaluate the applicability of the WslaCP methodology presented in Chapter 5, and to evaluate the usability of the tool presented in Chapter 6. The value of this evaluation is to investigate the degree to which both the methodology and the tool can achieve their objectives. Furthermore, it aims to explore the level of help and automation both the methodology and the tool offer to their users. It also discusses the areas that need to be enhanced. In addition, it introduces the use of an add-on feature to the WslaCP methodology, namely the use of a WSDL document in the automatic creation of the model, thus increasing the level of the automation. The chosen case study is of a stock quote service, used to predict the service's ability to satisfy performance thresholds specified within a WSLA contract. These thresholds are defined using three types of SLO: an SLO with a simple expression, an SLO with nested expressions, and an SLO with hard to evaluate measurement types.

The remainder of this chapter is organised as follows: Section 7.1 introduces the steps normally included in an evaluation process; in Section 7.2, the case study used in this work is outlined; the methodology evaluation is discussed in Section 7.3, while the tool evaluation is examined in Section 7.4. Finally, Section 7.5 concludes the chapter.

7.1 Introduction

The evaluation of a piece of work (such as a program, a methodology or a framework) means that it will be discussed, explored and assessed against the degree to which it

achieves the goals and objectives it was designed for. It helps the evaluator to detect and explore areas of weakness and strength [151]. The fundamentals for carrying out an evaluation in general, according to Bond in [151], are:

1. Framing the evaluation: To determine the context of the evaluation (i.e. describing its environment and the factors that influence it) and the specification of the targeted users for the specified piece of work.

2. Defining evaluation goal and objectives. The goal is the broad target of what the proposed work wants to achieve, while the objectives are the detailed steps towards accomplishing it. After setting the goals and objectives, the evaluation uses formative or summative questions. The former are objective-related and they inspect the degree at which the work achieves its objectives, while the latter are goal-related; they inspect the impact of these objectives on the main goal. In this step, the kind of quantitative and qualitative data that have to be collected should be determined. The former are numerical and usually measurement-based; they can be expressed using averages, percentages and so on. The latter are word-based and can be obtained mostly by observation. The quantitative and qualitative data are then used as indicators to judge the degree to which the goal and objectives are met.

3. Looking for the evidence; also called data collection. It defines what resources or methods supply the evaluator with the desired data. This may include experiments, questionnaires, interviews, focus groups, etc.

4. Interpreting the evidence. To examine the obtained data and results, evaluate them against the indicators, and discuss how they reflect on the work. The evaluator may assess how to use these results in enhancing the work.

The aforementioned evaluation steps are used in the context of the evaluation of the WslaCP methodology in Section 7.3 and the tool in Section 7.4.

7.2 The Case Study

The example is an on-line stock quote service built to suit the WSLA contract described in the WSLA manual [17]. The aim of this case study is to predict the probability that this service will meet three kinds of SLOs defined in a WSLA document. The first SLO of this case study (Section 7.2.2.1) has been used in our paper [15] as a simple example to demonstrate the applicability of the theoretical

methodology when considering a single SLO with simple expression. This SLA is extended in this thesis to cover another aspect of WSLA contracts, namely nested SLO expressions, and hard to evaluate measurement directives such as response time.

This section is structured as follows: the stock quote service is described in Section 7.2.1 while the WSLA contract with its SLOs is illustrated in Section 7.2.2. The WSDL contract of this service is described in Section 7.2.3 and is used later in building the service model.

7.2.1 Service Description

In the stock quote service, the service provider offers to deliver two kinds of service for a particular number of users. Therefore, the web service employs two main operations allowing the user either to request a stock quote (GetQuote) or to print its history (PrintQuote). The request operation is triggered when the customer requests a quote, causing the service to store this request in a queue. The service then checks the quote value and creates a response which waits in another queue before being sent back to the user. The sending and receiving mechanism in this service is assumed to be durable. However, checking the stock value may fail, in which case the quote requests wait in the queue until the service is up again. The printing operation, added in this thesis work, is triggered when the customer submits a request to print a file of a quote history. The service checks the database to retrieve all the related information and then prepares them in a file for printing. The database is considered reliable too.

The provider of this stock quote service offers an SLA with three different SLOs:

1. The GetQuote operation is offered with high availability. The provider specifies that the service down time will not reach a continuous ten minute threshold during the last month of the year.

2. The GetQuote operation promises a transaction rate of more than 1000 transactions/hour if the load, when greater than 80%, does not exceed 30% of the time in the last month of the year.

3. The PrintQuote operation is provided with a reasonable response time. The SLO guarantees that a history file will be available for printing by the end user after, at most, 15 seconds in the last month of the year.

7.2.2 WSLA Contract of the Stock Quote Service

The WSLA contract of the stock quote service is presented in Listing C.1 in Appendix C. For practical reasons, this contract is presented in this section in three listings; each contains one SLO with all the information related to it. The first two SLO are taken from the WSLA manual [17] and are defined for the `GetQuote` operation. The third SLO is a modified version from [17] and is defined for the `PrintQuote` operation. The three SLOs are described separately in what follows.

7.2.2.1 An SLO with Simple Expression: GetQuote Availability

The first SLO offered in WSLA refers to the down time of the GetQuote operation which will be less than ten minutes during the last month of the year. Its complete WSLA syntax is presented in Listing 7.1.

Listing 7.1: The "ContinuousDownTimeSLO" service level objective

```
 1:<ServiceDefinition  name="DemoService">
 2:  <Schedule name="availabilityschedule">
 3:    <Period>
 4:      <Start>2001-11-30T14:00:00.000-05:00</Start>
 5:      <End>2001-12-31T14:00:00.000-05:00</End>
 6:    </Period>
 7:    <Interval> <Minutes>1</Minutes> </Interval>
 8:  </Schedule>

 9:<Operation name="GetQuote" xsi:type="WSDLSOAPOperationDescriptionType">
10: <SLAParameter name="Availability_CurrentDownTime" type="long"  unit="minutes">
11:    <Metric>CurrentDownTime</Metric>
12: </SLAParameter>

13: <Metric name="CurrentDownTime" type="long" unit="minutes">
14:    <Function xsi:type="Span" resultType="double">
15:     <Metric>StatusTimeSeries</Metric>
16:     <Value> <LongScalar>0</LongScalar> </Value>
17:   </Function>
18: </Metric>

19: <Metric name="StatusTimeSeries" type="TS" unit="">
20:  <Function xsi:type="TSConstructor" resultType="TS">
21:    <Schedule>availabilityschedule</Schedule>
22:    <Metric>MeasuredStatus</Metric>
23:    <Window>1440</Window>
24:  </Function>
25: </Metric>

26: <Metric name="MeasuredStatus" type="integer" unit="">
27:  <MeasurementDirective xsi:type="StatusRequest" resultType="integer">
28:    <RequestURI>http://ymeasurement.com/StatusRequest/GetQuote</RequestURI>
29:  </MeasurementDirective>
30: </Metric>
```

```
31:  <WSDLFile>DemoService.wsdl</WSDLFile>
32:  <SOAPBindingName>SOAPNotificationBinding</SOAPBindingName>
33:  <SOAPOperationName>getQuote</SOAPOperationName>
34: </Operation>
35:</ServiceDefinition>

36:<Obligations>
37: <ServiceLevelObjective name="ContinuousDowntimeSLO">
38:  <Validity>
39:   <Start>2001-11-30T14:00:00.000-05:00</Start>
40:   <End>2001-12-31T14:00:00.000-05:00</End>
41:  </Validity>
42:  <Expression>
43:   <Predicate xsi:type="Less">
44:      <SLAParameter>Availability_CurrentDownTime</SLAParameter>
45:      <Value>10</Value>
46:   </Predicate>
47:  </Expression>
48: </ServiceLevelObjective>
49:</Obligations>
```

This listing consists of a `ServiceDefinition` section (line 1) to define the SLA parameter with its metrics, and an `Obligation` section (line 36) to define the SLO with its threshold and validity period. These sections are described in detail in Section 4.3.1.1 (please refer to Listings 4.1 and 4.2 with their descriptions). For this reason, their description are not repeated here. The only part of this listing that was not included in Section 4.3.1.1 is described in what follows.

Each operation in WSLA refers to a file that describes it. Any kind of service description document can be used [17]. However, given that WSLA is used mostly for web services, the WSDL [152] document is used to define this service operation. Inside WSLA, the name of this WSDL file is defined in the `WSDLFile` element (line 31). It is `DemoService.wsdl` in this example. Another piece of information may also be specified, such as the name of the binding, along with the service SOAP operation name. These are defined in the `SOAPBindingName` and `SOAPOperationName` elements respectively (lines 32 and 33).

7.2.2.2 An SLO with Nested Expressions: GetQuote Transaction Rate

The second SLO offered in WSLA is that the service is able to deal with more than 1000 transactions per hour if, and only if, the service experiences a heavy load of 80% for less than 30% of the time and this during the last month. Its complete WSLA syntax is presented in Listing 7.2.

Listing 7.2: The "ConditionalSLOForTransactionRate" service level objective

```
1:<ServiceDefinition  name="DemoService">
2: <Schedule name="businessdayschedule">
```

```
 3:    <Period>
 4:       <Start>2001-11-30T14:00:00.000-05:00</Start>
 5:       <End>2001-12-31T14:00:00.000-05:00</End>
 6:    </Period>
 7:    <Interval> <Minutes>1440</Minutes> </Interval>
 8: </Schedule>

 9: <Schedule name="5minuteschedule">
10:    <Period>
11:      <Start>2001-11-30T14:00:00.000-05:00</Start>
12:      <End>2001-12-31T14:00:00.000-05:00</End>
13:    </Period>
14:    <Interval> <Minutes>5</Minutes> </Interval>
15: </Schedule>

16: <Schedule name="hourlyschedule">
17:  <Period>
18:    <Start>2001-11-30T14:00:00.000-05:00</Start>
19:    <End>2001-12-31T14:00:00.000-05:00</End>
20:  </Period>
21:  <Interval> <Minutes>60</Minutes> </Interval>
22: </Schedule>

23: <Operation name="GetQuote" xsi:type="WSDLSOAPOperationDescriptionType">
24:  <SLAParameter name="OverloadPercentage"  type="float" unit="Percentage">
25:   <Metric>OverloadPercentageMetric</Metric>
26:  </SLAParameter>

27:  <SLAParameter name="TransactionRate"  type="float" unit="transactions/hour">
28:   <Metric>Transactions</Metric>
29:  </SLAParameter>

30:  <Metric name="OverloadPercentageMetric" type="float" unit="Percentage">
31:   <Function xsi:type="PercentageGreaterThanThreshold" resultType="float">
32:    <Schedule>businessdayschedule</Schedule>
33:    <Metric>UtilizationTimeSeries</Metric>
34:    <Value> <LongScalar>0.8</LongScalar> </Value>
35:   </Function>
36:  </Metric>

37:  <Metric name="UtilizationTimeSeries" type="TS" unit="">
38:   <Function xsi:type="TSConstructor" resultType="float">
39:    <Schedule>5minuteschedule</Schedule>
40:    <Metric>ProbedUtilization</Metric>
41:    <Window>12</Window>
42:   </Function>
43:  </Metric>

44:  <Metric name="ProbedUtilization" type="float" unit="">
45:   <MeasurementDirective xsi:type="Gauge" resultType="float">
46:     <RequestURL>http://acme.com/SystemUtil</RequestURL>
47:   </MeasurementDirective>
48:  </Metric>

49:  <Metric name="Transactions" type="long" unit="transactions">
50:   <Function xsi:type="Minus" resultType="double">
```

```
51:     <Operand>
52:      <Function xsi:type="TSSelect" resultType="long">
53:       <Operand> <Metric>SumTransactionTimeSeries</Metric> </Operand>
54:       <Element>0</Element>
55:      </Function>
56:     </Operand>
57:     <Operand>
58:      <Function xsi:type="TSSelect" resultType="long">
59:       <Operand> <Metric>SumTransactionTimeSeries</Metric></Operand>
60:       <Element>-1</Element>
61:      </Function>
62:     </Operand>
63:    </Function>
64:   </Metric>

65:   <Metric name="SumTransactionTimeSeries" type="TS" unit="transactions">
66:    <Function xsi:type="TSConstructor" resultType="TS">
67:     <Schedule>hourlyschedule</Schedule>
68:     <Metric>SumTransactions</Metric>
69:     <Window>2</Window>
70:    </Function>
71:   </Metric>

72:   <Metric name="SumTransactions" type="long" unit="tansactions">
73:    <MeasurementDirective xsi:type="InvocationCount" resultType="long"/>
74:   </Metric>
75:   <WSDLFile>DemoService.wsdl</WSDLFile>
76:   <SOAPBindingName>SOAPNotificationBinding</SOAPBindingName>
77:   <SOAPOperationName>getQuote</SOAPOperationName>
78:  </Operation>
79:</ServiceDefinition>

80:<Obligations>
81:  <ServiceLevelObjective name="ConditionalSLOForTransactionRate">
82:   <Validity>
83:    <Start>2001-11-30T14:00:00.000-05:00</Start>
84:    <End>2001-12-31T14:00:00.000-05:00</End>
85:   </Validity>
86:   <Expression>
87:    <Implies>
88:     <Expression>
89:      <Predicate xsi:type="Less">
90:       <SLAParameter>OverloadPercentage</SLAParameter>
91:       <Value>0.3</Value>
92:      </Predicate>
93:     </Expression>
94:     <Expression>
95:      <Predicate xsi:type="Greater">
96:       <SLAParameter>TransactionRate</SLAParameter>
97:       <Value>1000</Value>
98:      </Predicate>
99:     </Expression>
100:   </Implies>
101:  </Expression>
102:</ServiceLevelObjective>
103:</Obligations>
```

In the WSLA contract presented in this listing, this SLO is described in two sections as follows. In the `Obligations` section (line 80), an SLO called `ConditionalSLOForTransitionRate` is defined (line 81) using nested expressions joined by the `Implies` logical operator (line 87). The first expression (line 88) specifies a predicate of type `Less` (line 89) to compare the SLA parameter `OverloadPercentage` (line 90) with a value of 0.3 (line 91). The second expression specifies a predicate of type `Greater` (line 95) to compare the SLA parameter `TransactionRate` (line 96) with a value of 1000 (line 97). This SLO is valid through the period of December (lines 83 and 84).

In the `ServiceDefinition` section (line 1), it is clear that the intended SLA parameters `OverloadPercentage` (line 24) and `TransactionRate` (line 27) are defined for the `GetQuote` WSDL operation (line 23). This section also specifies how these parameters are computed.

For the former SLA parameter, the `Gauge` measurement (line 45) in the `ProbedUtilization` metric checks the current value of resource utilisation. This is used as input to the `UtilizationTimeSeries` metric (line 37) which uses a `TSConstructor` function (line 38) to define a series of these values; these are obtained every five minutes (according to a `5minutesschedule` in line 39). In turn, each day (according to a `businessdaysschedule` in line 32), the `OverloadPercentageMetric` metric applies a `PercentageGreaterThenThreshold` function (line 31) on that series that gives the percentage of elements whose value is greater than 80%. Finally, this metric value is used as the `SLAParameter` value (line 25). The `5minutesschedule` is defined to collect values every 5 minutes (line 14) for one month (lines 11 and 12). Also, the `businessdaysschedule` is defined to collect values every 1440 minutes (line 7) for one month (lines 4 and 5).

For the latter SLA parameter, the `InvocationCount` measurement (line 73) in the `SumTransaction` metric returns the number of invocations of the GetQuote operation. This is used as input to the `SumTransactionTimeSeries` metric (line 65) that uses a `TSConstructor` function (line 66) to define a series of these values which is obtained each hour (according to a `hourlySchedule` in line 67). In turn, the `Transactions` metric (line 49) applies a `Minus` function (line 50) on that series to give the difference between the current (line 54) and the previous (line 60) value of two entries from this series. The current and previous values are selected using the `TSSelect` function (lines 52 and 58). Finally, this metric value is used as the `SLAParameter` value (line 28). The `hourlySchedule` is defined to collect values every 60 minutes (line 21) for one month (lines 18 and 19).

7.2.2.3 An SLO with Hard-to-Evaluate Measurement: PrintQuote Response Time

The third SLO offered in WSLA is that the printing operation is able to prepare the history file and send it back to the user in less than 15 seconds and this is during the last month of the year. Its complete WSLA syntax is presented in Listing 7.3.

Listing 7.3: The "PrintingResponseTime" service level objective

```
 1:<ServiceDefinition  name="DemoService">
 2: <Schedule name="ResponseSchedule">
 3:  <Period>
 4:    <Start>2001-11-30T14:00:00.000-05:00</Start>
 5:    <End>2001-12-31T14:00:00.000-05:00</End>
 6:  </Period>
 7:  <Interval><Seconds>15</Seconds></Interval>
 8: </Schedule>

 9: <Operation name="PrintQuote" xsi:type="WSDLSOAPOperationDescriptionType">
10: <SLAParameter name="MaxResponseTime" type="double" unit="seconds">
11:  <Metric>MaximumResponseTime</Metric>
12: </SLAParameter>

13: <Metric name="MaximumResponseTime " type="long" unit="seconds">
14:  <Function xsi:type="Max" resultType="double">
15:   <Metric>ResponseTimeSeries</Metric>
16:  </Function>
17: </Metric>

18: <Metric name="ResponseTimeSeries" type="TS" unit="seconds">
19:  <Function xsi:type="TSConstructor" resultType="TS">
20:   <Schedule>ResponseSchedule</Schedule>
21:   <Metric>ResponseTimeMetric</Metric>
22:   <Window>4</Window>
23:  </Function>
24: </Metric>

25:  <Metric name="ResponseTimeMetric" type="double" unit="seconds">
26:   <MeasurementDirective xsi:type="ResponseTime" resultType="double">
27:    <RequestURI>http://ymeasurement.com/ResponseTime/PrintQuote</RequestURI>
28:   </MeasurementDirective>
29:  </Metric>

30:  <WSDLFile>DemoService.wsdl</WSDLFile>
31:  <SOAPBindingName>SOAPNotificationBinding</SOAPBindingName>
32:  <SOAPOperationName>PrintQuote</SOAPOperationName>
33: </Operation>
34:</ServiceDefinition>

35:<Obligations>
36: <ServiceLevelObjective name="PrintingResponseTime">
37:  <Validity>
38:   <Start>2001-11-30T14:00:00.000-05:00</Start>
39:   <End>2001-12-31T14:00:00.000-05:00</End>
40:  </Validity>
```

```
41:  <Expression>
42:   <Predicate xsi:type="Less">
43:    <SLAParameter>MaxResponseTime</SLAParameter>
44:    <Value>15</Value>
45:   </Predicate>
46:  </Expression>
47: </ServiceLevelObjective>
48:</Obligations>
```

This SLO is described in two sections of WSLA as follows. In the `Obligations` section (line 35), an SLO called `PrintingResponseTime` (line 36) is defined using a simple expression that specifies a predicate of type `Less` (line 42); this compares the SLA parameter `MaxResponseTime` (line 43) with a value of 15 (line 44). This SLO is valid through the validity period of December (lines 38 and 39).

In the `ServiceDefinition` section, the required SLA parameter `MaxResponseTime` (line 10) is defined for the `PrintQuote` WSDL operation (line 9). This operation is defined in the `DemoService.wsdl` file (line 30). To define how this parameter is computed, the `ResponseTime` measurement (line 26) in the `ResponseTimeMetric` metric returns the response time of a printing request. This is used as input to the `ResponseTimeSeries` metric (line 18) that uses a `TSConstructor` function (line 19) to define a series of these values entered every 15 seconds (according to a `ResponseSchedule` in line 20). In turn, the `MaximumResponseTime` metric applies a `Max` function (line 14) on that series to give the maximum response time. Finally, this metric value is used as the `SLAParameter` value (line 11). In the same `ServiceDefinition` section, the `ResponseSchedule` is defined to collect values every 15 seconds (line 7) for one month.

7.2.3 The WSDL File of the Stock Quote Service

WSLA can contain a reference to a WSDL document that describes service operations and its management actions. WSDL is described in this section because it is used in Section 7.3.1.3 to help in creating the service model.

WSDL can be seen as complementary to WSLA. This is because WSLA is used to specify the quantitative attributes that both the service provider and customer agree on, and the way of measuring and computing their values. On the other hand, WSDL is used to describe the actual web service and how it communicates with its application. To clarify further, WSDL is used as an input when creating the actual service or implementing it in the real world. Later, WSLA is used as an input to a management agent that is responsible for testing the service against the agreed contract.

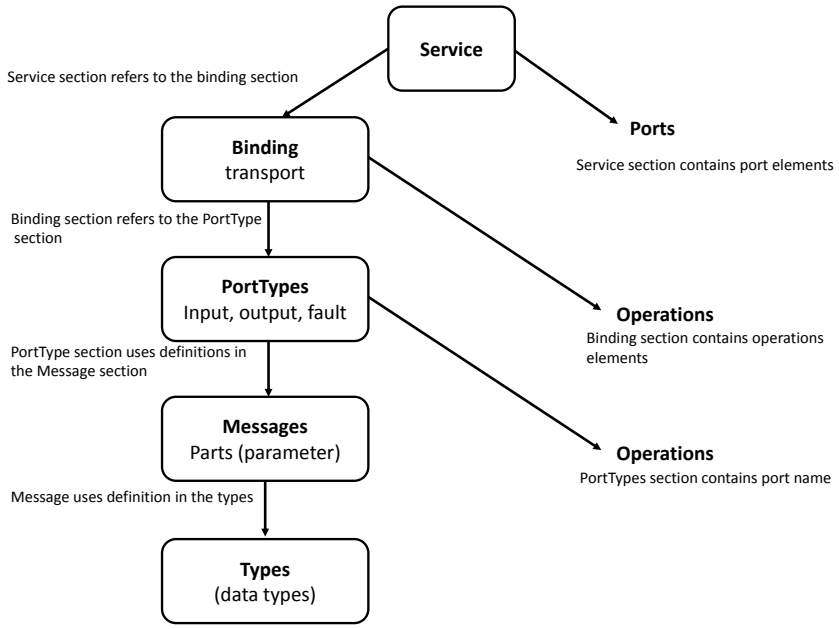

Figure 7.1: WSDL Abstract Definition

WSDL is an XML-based document that contains elements which are used to describe the web service, its location, how to access it, the communication messages and their format, the service operations and their input and output information [153]. To simplify the relations of the previous elements, Figure 7.1 shows the abstract definition of WSDL that is taken from the work in [153]. This is important to simplify the idea of representing WSDL elements as Stochastic Petri Net (SPN) primitives when this is discussed later in Section 7.3.1.3. The WSDL elements are [153]:

- **Type**: defines the types employed by a message element.
- **Message**: defines the transferred data.
- **PortType**: defines a set of operations where each **operation** has an **input**, **output**, and **fault** messages.
- **Binding**: defines the type of communication protocol which could be SOAP, HTTPGET, HTTPPOST, and MIME.
- **Service**: defines a set of ports; each **port** relates a location with a binding. This location contains the address of the file that contains the service method which the client wants to invoke. A service refers to one portType with several operations in this portType. A WSDL file can contain multiple services.

The complete WSDL contract of the stock quote service is presented in Listing 7.4. Part of this file is adopted from [154].

Listing 7.4: The WSDL file of the stock quote service.

```
 1:<?xml version="1.0" encoding="UTF-8"?>
 2: <definitions name="DemoService" targetNamespace="http://example.com.wsdl/
     DemoService/" xmlns="http://schemas.xmlsoap.org/wsdl/"  xmlns:tns="http://
     example.com.wsdl/DemoService/" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 3:  <message name="getQuoteRequest">
 4:   <part name="ticker" type="xsd:float"/>
 5:  </message>
 5:  <message name="getQuoteResponse">
 6:   <part name="result" type="xsd:float"/>
 7:  </message>
 8:  <message name="printQuoteRequest">
 9:   <part name="ticker" type="xsd:string"/>
10:  </message>
11:  <message name="printQuoteResponse">
12:   <part name="result" type="xsd:string"/>
13:  </message>

14:  <portType name="StockQuote_get">
15:   <operation name="getQuote">
16:    <input message="tns:getQuoteRequest" name="getQuoteRequest"/>
17:    <output message="tns:getQuoteResponse" name="getQuoteResponse"/>
18:   </operation>
19:  </portType>
20:  <portType name="StockQuote_print">
21:   <operation name="printQuote">
22:    <input message="tns:printQuoteRequest" name="printQuoteRequest"/>
23:    <output message="tns:printQuoteResponse" name="printQuoteResponse"/>
24:   </operation>
25:  </portType>

26:  <binding name="StockQuoteBinding_get" type="tns:StockQuote_get">
27:   <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/
    http"/>
28:   <operation name="getQuote">
29:     <soap:operation soapAction="http://localhost/getQuote"/>
30:     <input>
31:       <soap:body use="literal"/>
32:     </input>
33:     <output>
34:       <soap:body use="literal"/>
35:     </output>
36:   </operation>
37:  </binding>
38:  <binding name="StockQuoteBinding_print" type="tns:StockQuote_print">
39:   <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/
    http"/>
40:   <operation name="printQuote">
41:    <soap:operation soapAction="http://localhost/printQuote"/>
42:    <input>
43:     <soap:body use="literal"/>
44:    </input>
```

```
45:     <output>
46:      <soap:body use="literal"/>
47:     </output>
48:    </operation>
49:   </binding>

50: <service name="StockQuoteService">
51:   <port name="StockQuotePort_get" binding="tns:StockQuoteBinding_get">
52:    <soap:address location="'http://localhost/StockQuoteService.asmx"/>
53:   </port>
54:   <port name="StockQuotePort_print" binding="tns:StockQuoteBinding_print">
55:    <soap:address location="'http://localhost/StockQuoteService.asmx"/>
56:   </port>
67: </service>
58:</definitions>
```

In this listing, the `StockQuoteService` is defined (line 50). It defines two ports: `StockQuotePort_get` and `StockQuotePort_print` (lines 51 and 54). These ports refer to the bindings which are `StockQuoteBinding_get` and `StockQuoteBinding_print` which are defined in lines 26 and 38 respectively. These bindings in turn refer in their `type` attributes to the `PortType`(s) which are `StockQuote_get` and `StockQuote_print`; these are defined in lines 14 and 20 respectively. These port types define two `Operation`(s) which are `getQuote` and `printQuote` (lines 15 and 21); these operations are the ones used in the WSLA example. They also refer to the `message`(s) defined from lines 3 to 13.

7.3 Evaluation of the WslaCP Methodology

In this section, an evaluation of the theoretical aspects of the WslaCP methodology based on the described case study is provided. Recalling the evaluation fundamentals presented in Section 7.1, four steps should be conducted to complete the evaluation process. These steps can be written in the context of evaluating the WslaCP methodology as follows:

Framing the evaluation: The targeted users of the WslaCP methodology, as defined in Section 3.1.1, are service providers/engineers, SLA engineers, or modellers. Hence, the evaluation should be addressed from their perspectives.

Defining evaluation goals and objectives and evaluation methods: The evaluation of the WslaCP methodology is assessed by a case study. This evaluation is based on the degree to which the methodology can achieve the aim and objectives it was designed for. The research aim and objectives were stated in Section 1.4. A number of questions are formulated to evaluate them; these are aligned with some of the research questions addressed in Section 1.3. These questions are regarding:

1. *Automatic Model Creation*: The questions related to this are as follows:
   - Q1: Can a service's stochastic model be generated automatically?
   - Q2: Is there any difference if the automatic model creation is carried out before or after deploying the service in the real world?
   - Q3: Can a WSDL file aid this automatic creation of a service model?
2. *Methodology's Applicability*: The questions related to this are as follows:
   - Q1: Is the methodology applicable in a realistic scenario (i.e. can all its proposed steps be achieved)?
   - Q2: Is it scalable for more complicated scenarios?
   - Q3: Does the generated model reflect the actual service?
3. *Methodology's Generality*: The questions related to this are as follows:
   - Q1- Is the methodology general enough to be applicable for different SLAs?
   - Q2- Is the methodology general enough to be applicable for different stochastic models?
4. *User Support*: The questions related to this are as follows:
   - Q1: What degree of automation and help does this methodology offer to its users?
   - Q2: Does it accomplish the aim of minimal user interaction?

For the last two steps of the fundamentals of evaluation, which are **Looking for the evidence** and **Interpreting the evidence**, the described case study has been utilised to answer the aforementioned evaluation questions. This is then used as feedback to enhance the methodology in a way that better addresses the questions.

In the following sub-sections, each of the aforementioned evaluation questions, along with looking for evidence and interpreting it, are discussed in detail.

7.3.1 Automatic Model Creation

To answer the questions regarding the ability to generate a service model automatically, the case study is utilised in Section 7.3.1.1. The answers are provided in Section 7.3.1.2 and finally, a complementary feature of using a WSDL file to enhance the automatic model creation is described in Section 7.3.1.3.

7.3.1.1 Looking for the Evidence

The stock quote service is modelled in a specific SDES model, namely the Stochastic Petri Net (SPN). Recalling the three SLOs in Listings 7.1, 7.2 and 7.3, and the service operation mapping and measurements mapping presented in Sections 5.2.1 and 5.2.2, the following SPN primitives can be produced automatically from WSLA:

1. The service operation of Listing 7.1, `GetQuote`, is mapped as depicted in the upper part of Figure 7.2. This includes a place *GetQuote_s* with a connected transition *GetQuote_a*.

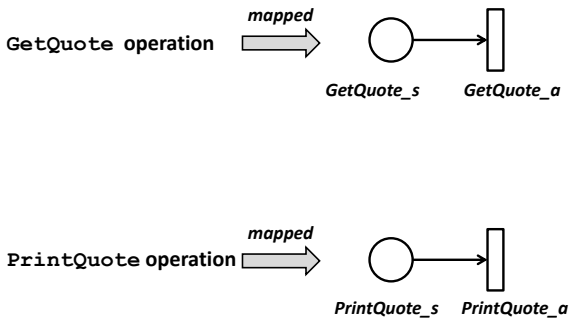

Figure 7.2: Mapping service operations in Listings 7.1 and 7.3 to SPN

2. The service operation of Listing 7.3, `PrintQuote`, is mapped as depicted in the bottom part of Figure 7.2. This also includes a place *PrintQuote_s* with a connected transition *PrintQuote_a*.

3. The *StatusRequest* measurement directive of the `GetQuote` operation in Listing 7.1 is mapped as depicted in the upper part of Figure 7.3. This includes

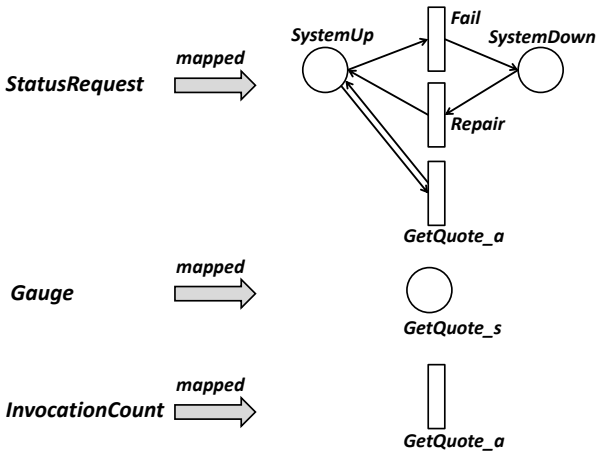

Figure 7.3: Mapping measurement directives in Listings 7.1 and 7.2 to SPN

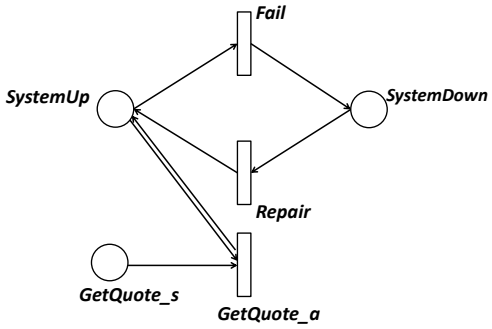

Figure 7.4: Merging SPNs in Figure 7.3 and the upper part of Figure 7.2

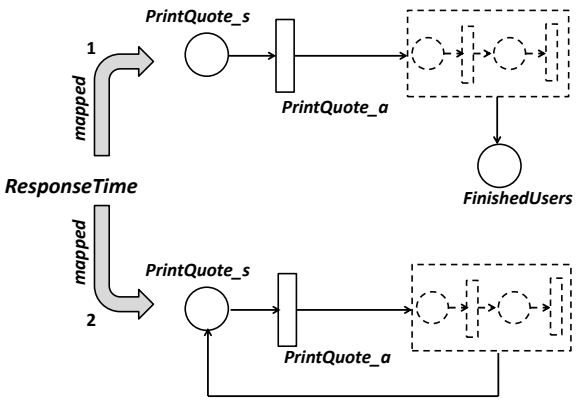

Figure 7.5: Mapping the measurement directive of Listing 7.3 to SPN

the simplest fail-repair mechanism allowing the *GetQuote_a* transition to be enabled/disabled.

4. The *Gauge* and the *InvocationCount* measurement directives of the `GetQuote` operation in Listing 7.2 are mapped as depicted in the middle and bottom parts of Figure 7.3. This includes a place *GetQuote_s* and a transition *GetQuote_a* respectively. The former represents the queue that stores the incoming requests while the latter represents the operation execution.

5. The set of primitives in Figure 7.3 and the upper part of Figure 7.2 can be merged into the model depicted in Figure 7.4.

6. The *ResponseTime* measurement directive of the `PrintQuote` operation in Listing 7.3 is mapped as depicted in Figure 7.5. This is done (1) using a *FinishedUsers* place to prompt the completion of the printing request in case of an open model, or (2) using an arc that links back to the *PrintQuote_s* place to prompt the completion of the printing request in the case of a closed model. In Figure 7.3, the dashed parts represent unspecified sets of places/transitions.

7. The model parts in Figures 7.4 and 7.5 are merged into the model in Figure 7.6 where the first way of mapping the response time, presented in Figure 7.5, is utilised. In this figure, the model is extended by including the dotted parts which represent an extra place (*Users*) and a transition (t). The latter represents the choice of either requesting or printing a quote, while the former represents a fixed number of users to prevent a state space explosion if the action is used alone (the action will fire continuously). These two primitives are added only when the WSLA document contains more than one operation.

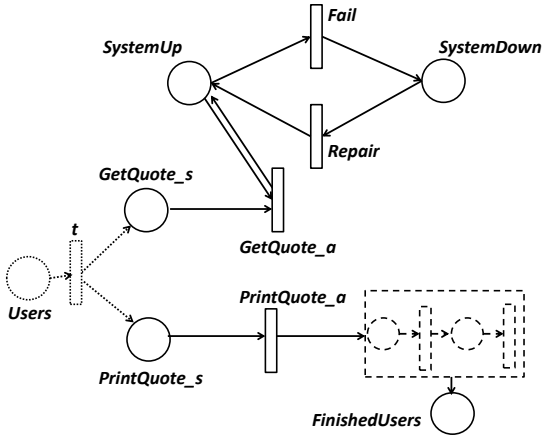

Figure 7.6: Merging SPN parts of Figures 7.4 and 7.5

The model in Figure 7.6, generated from mapping service operations and measurement directives, is incomplete because the *GetQuote_a* is not connected to an output place. This can be enhanced automatically by connecting the *GetQuote_a* back to the *Users* place as appears in the dotted parts of Figure 7.7.

All the transitions are assigned a firing rate of value 1 by default except for repair-fail transitions that are assigned 0.1 and 0.9 firing rates respectively. The user can then change these values manually if necessary.

The reward variables (i.e. reward functions with evaluation intervals), that are generated automatically for the measurement directives and schedules of Listings 7.1, 7.2 and 7.3, are depicted in Table 7.1. These are not described here because of their simplicity.

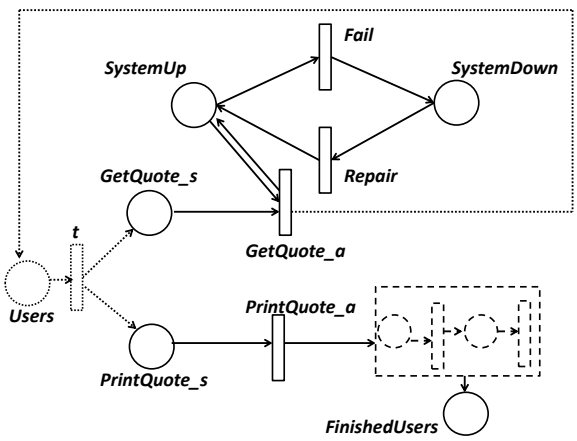

Figure 7.7: Completing the model of Figure 7.6

Table 7.1: The reward variables generated from mapping measurement directives and the schedules of Listings 7.1, 7.2 and 7.3

Measurement Directives & Schedules	**Reward Function** (rv_{rate}/rv_{int})	**Evaluation Intervals** ($\{rv_{int}\}$)
`StatusRequest` & `Businessdayschedule`	$rv_{rate}(\sigma) = \begin{cases} 1 & \text{if } SystemUp = 1 \\ 0 & \text{otherwise} \end{cases}$	At instants:{[0,0],[1440, 1440],. . . ,[44640,44640]}
`Gauge` & `5minuteschedule`	$rv_{rate}(\sigma) = \begin{cases} GetQuote_s(\sigma) & \forall\sigma \in \Sigma \\ 0 & otherwise \end{cases}$	At instants:{[0,0],[5, 5],. . . ,[44635,44640]}
`InvocationCount` & `hourlyschedule`	$rv_{imp}(a) = \begin{cases} 1 & \text{if } a = GetQuote_a \\ 0 & otherwise \end{cases}$	During intervals:{[0,60], [60,120],. . . ,[44580,44640]}
`ResponseTime` & `ResponseSchedule`	$rv_{rate}(\sigma) = \begin{cases} 1 & \text{if } FinishedUsers{=}1 \\ 0 & \text{otherwise} \end{cases}$	At instants:{[0,0], [15,15],. . . ,[44625,44640]}

7.3.1.2 Interpreting the Evidence

The model created in Figure 7.7 with the reward variables of Table 7.1 are the outcome of the automatic model creation. This was produced by mapping the available operations and measurement directives in WSLA to a specific state variable or action, or a combination of them. Furthermore, this was derived by considering the hints of including some fail-repair mechanism and a response time measuring mechanism in the form of a set of transitions/places/arcs primitives. Comparing this model with the one built manually by a user (depicted in Figure 7.11) implies that only a small amount of information regarding the stochastic model of the service can be known automatically from the SLA alone; this information is limited and sometimes scattered. Given this, and as appears in Figure 7.7, a number of shortcomings in the automatic production of a stochastic service model are recognised:

1. The produced model is abstract. Since there is no way to know the exact implementation of the GetQuote and PrintQuote operations from WSLA alone (i.e. how these operations perform their work), they cannot be reflected truly in the model. For example, the manually created model in Figure 7.11 specifies, in its bottom part, that the *RequestPrinting* operation (that represents the *printQuote_a* operation in Figure 7.7) checks the database for the quote history before preparing the file for printing and sending it back to the user (this is accomplished using *RequestFromDB*, *CreatingPrintingHistory* and *SendPrinting*). This was not included in the model that is created automatically.

2. The model is not complete. There is no way either to assign a true firing

rate automatically to the transitions, or to know the true initial state of the model. All these rates are assumed and hence may not reflect the real service parameters.

3. Some mechanisms that are used in the model may not be desired. For example, *StatusRequest* gives a hint to specify the up and down status of the service. The automatically chosen mechanism (depicted in Figure 7.7) might be trivial and not wanted by the user who possibly wants to replace it with a more complicated one. The same problem is true for *ResponseTime* where the user might be interested in a different mechanism to measure the response, especially since the one used (using *FinishedUsers* in Figure 7.7) can reflect the response time for one-customer only or promote the assumption that the first token to leave the *Users* place is the first one to arrive at the *FinishedUsers* place. If more than one-customer is used in the *Users* place, the response time might not be truly reflected since the tokens in SPN are not marked; hence the returned token to the *FinishedUsers* place might not be the one which left the *Users* place first. For example, in Figure 7.11, the user specifies the response time for printing a quote history without considering the *FinishedUsers* place chosen in Figure 7.7. Instead, the user links the *SendPrinting* transition back to the *Users* place; hence the response time is measured by tracking the token from the time it left the *Users* place until it returned to it.

In the light of on these shortcomings, the evaluation questions related to the automatic model creation addressed in Section 7.3 can be answered as follows:

Q1*: Can a service's stochastic model be generated automatically?*

Not fully. Only an abstract and incomplete model can be generated automatically from the WSLA specification alone. Given the abstraction and incompleteness of the generated model, the user of the methodology has to refine the model, assign concrete firing rates, and assign the model's initial state. However, the reward variables, which depend on the generated model, can be automatically derived without user interaction, as appears in Table 7.1. Nevertheless, when the model is completed by the user, this might imply some changes in the reward function content and hence user interference is required.

Q2*: Is there any difference if the automatic model creation is carried out before or after deploying it in the real world?*

No. The methodology itself is related to how WSLA can be used to create a service model automatically given the assumption that it is the only document the user has access to. This will not differ whether or not the service is deployed.

However, if the methodology is extended to use the service's definition documents, in addition to the SLA, then there will be a difference. Hence, if this methodology is used after deploying the service (or after its implementation), a number of files that contain either the business process (such as the Business Process Execution Language for Web Services (BPEL4WS)[155]) or the service work-flow (such as the Web Services Flow Language (WSFL)[156]) can be utilised to refine the model automatically given that these documents are accessible to the users. Furthermore, if the service is deployed, parameterising the model (transition firing rate and initial state) can be derived from data taken from the running service; these better reflect the model parameters.

Q3*: Can a WSDL file aid this automatic creation of a service model?*

Yes, if it is accessible. A user of the methodology might have access to the WSDL file that describes the service operations (this might be in the design, implementation or deployment stage). A WSDL file can then be utilised to add more primitives to the model. A service methods file, WSFL, or BPEL4WS can be used after the service's implementation to refine the model further, if the user is permitted access to them. Given that WSDL can add extra primitives to the model, the next section describes the way a WSDL file is used to serve this purpose.

7.3.1.3 Using a WSDL File in Building the Service Model

WSLA refers to a WSDL file to describe the service operations. Depending on this, a complementary feature that exploits WSDL mapping is added to the automatic creation of the service model. Hence, in this section, an investigation with regard to whether a Stochastic Petri Net model can be derived automatically from WSLA as well as WSDL is addressed. This is done by introducing the rules of mapping a WSDL to a timed Petri Net model (designed by [153]), then using these rules to map WSDL to an SDES model and SPN accordingly.

The Mapping Rules from WSDL to a Timed Petri Net Model: In the literature, some studies were undertaken with regard to accomplishing automatic mapping from WSDL to timed PNs[1]. The most detailed work was carried out by Javed in his thesis [153] where a set of rules for mapping WSDL elements to a timed PN model was defined as follows:

- A Place represents a PortType that contains the operation with the input and output messages.

[1]WSDL file is used for automated mapping from WSDL to a timed Petri Net model according to [157], and to GSPN according to [158].

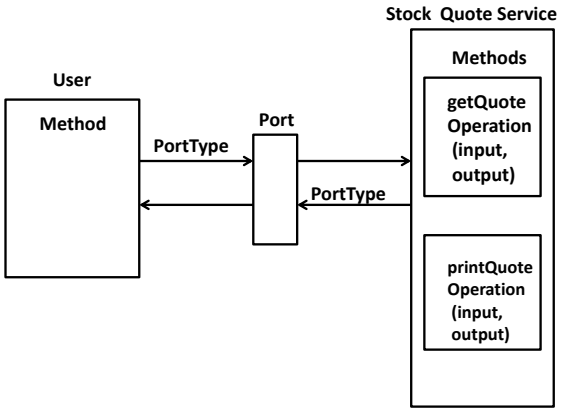

Figure 7.8: WSDL model for user interaction

- A Transition represents a Service-Port that contains Name, Binding, and Location.
- An Arc represents a Binding containing the PortType and Protocol.

A WSDL file describes what the service can do, i.e. service operations (methods), in an abstract way without expressing how to implement them. However, the parameters and their data type, required to invoke these methods, are placed inside the WSDL file published by this service [153]. The methods represent the business process computations which may contain conditional statements, loops, and other service method calls. These computations are stored in a separate file which is referred to by the WSDL's Port element. If this file is reachable (i.e. if it exists), then the rules for mapping it to timed Petri Net according to [153] are as follows:

- Places represent the data storage.
- Transitions represent the computational primitives.

The mapping by Javed [153] does not depend on a WSDL file only, but also on the WSDL flow model implied between the web service and its client, or what is called a WSDL model for user interaction. For example, Figure 7.8 depicts the WSDL flow model of the example in this case study. To map this to a timed PN, Method and Ports are represented as transitions, while PortType and Service are represented as places.

The Mapping Rules from WSDL to an SDES Model: According to the predefined rules for mapping a WSDL file and its flow model[1], these rules can be

[1]The method file that is referred to from a WSDL file is ignored in this mapping.

generalised for mapping WSDL and its flow model into the stochastic model of the specified service $SDES = (SV, A, S, RV)$. This is accomplished as follows:

1. Each PortType in the WSDL file and each service in the WSDL model for user interaction represents a state variable $sv \in SV$.

2. Each Port in the WSDL file and each method in the WSDL model for user interaction represents an action $a \in A$.

3. Each WSDL binding represents an arc.

According to the predefined rules of mapping a WSDL file and its flow model, the SPN model in Figure 7.9 can be produced automatically from the WSDL file presented in Listing 7.4 and the flow model presented in Figure 7.8.

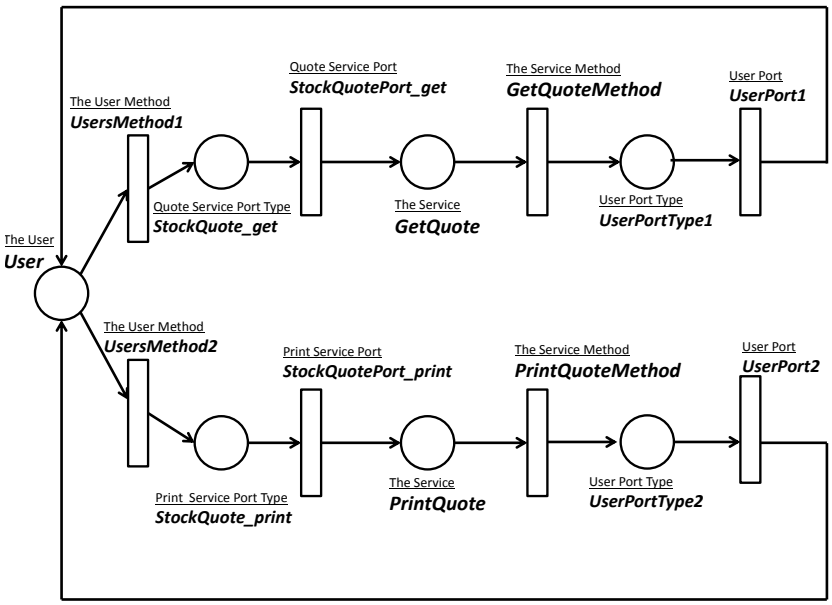

Figure 7.9: Mapping the WSDL file of Listing 7.4, and the WSDL user interaction model of Figure 7.8 to SPN

The underlined labels in Figure 7.9 represent the type of element being modelled (such as User, Service, Port, PortType, and Method), while the **bold labels** represent the name assigned to this element after mapping it. Some of these element names are taken from the WSDL file of Listing 7.4, such as the *StockQuote_get*, *StockQuotePort_get* and *GetQuote*, while the others are generated automatically, such as the *GetQuoteMethod*, *UserPortType1*, and *UserPort1*. The latter elements are generated from the WSDL model for user interaction; for that they are assigned

randomly generated names. However, the former elements are generated from a WSDL file; for that they are assigned the name of the elements in this file.

The model generated from the WSDL mapping in Figure 7.9 is also abstract in that the actual service method implementations are not defined. However, comparing the model produced from WSLA alone (in Figure 7.7) and the model built manually by the user (in Figure 7.11), the WSDL model is much more complete. Accordingly, using WSDL can partially extend the automatic model creation so that an additional set of information can be provided to a user of the methodology.

The WSDL generated model does not reflect the up-down mechanism specified in WSLA for the *StatusRequest*; hence, the model can be enhanced automatically by attaching the primitives describing this mechanism to the *GetQuoteMethod* transition. To reflect the *ResponseTime*, there is no need to add an extra place since the generated model is closed and the response time can be computed from the token returning to the *User* place. Accordingly, the generated service model using both WSDL and WSLA, as shown in Figure 7.10 where the added part is depicted using dashed lines, is a better option for automatic model creation.

If the generated model of Figure 7.10 is used for the automatic model creation, the state variable and action used within the reward function of the reward variable should be changed to refer to the names of the PortType and Port respectively. This is because they were referred to previously (in the WSLA generated model of Figure

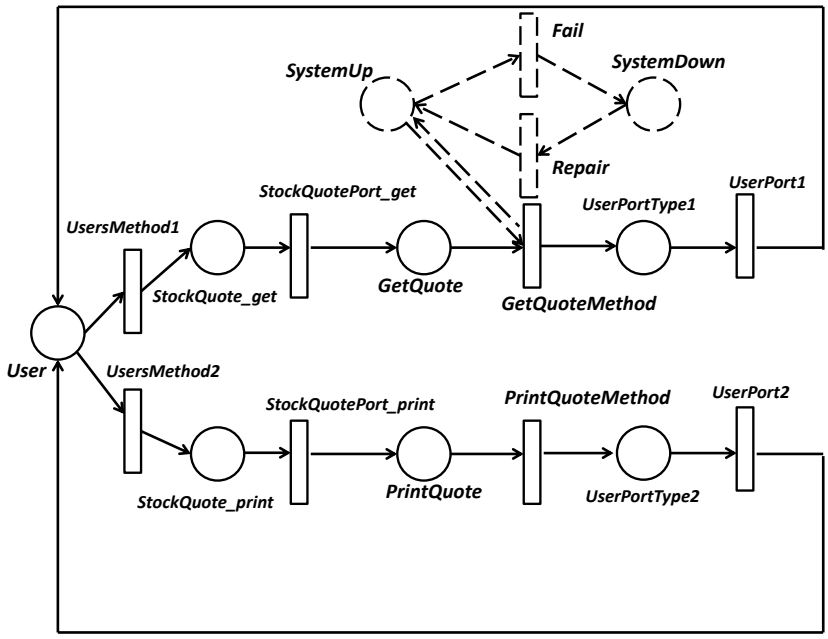

Figure 7.10: Mapping the WSDL and WSLA of the stock quote service to SPN

7.7) using the operation name with the extensions _s and _a respectively.

7.3.2 Methodology's Applicability

This section investigates the ability to apply the WslaCP methodology for a real WSLA and a service model. This is discussed in Section 7.3.2.1, while the answers to the evaluation questions are provided in Section 7.3.2.2.

7.3.2.1 Looking for the Evidence

The theoretical basis of the WslaCP methodology, described in Chapters 4 and 5, includes defining WSLA semantics and then carrying out a five-step mapping process. This process consists of the mapping of the: service operation, measurement directives, functions, schedules and the SLO. The SLO in Listing 4.1 and the SLA-Parameter in Listing 4.2 (which are the same as the first SLO in Listing 7.1) were utilised as a running example for each mapping step in order to show the methodology applicability. This was described from Section 5.2.1 to Section 5.2.5. For this reason, there is no need to study the remaining SLOs in this case study as no new outcome can be obtained.

7.3.2.2 Interpreting the Evidence

The evaluation questions regarding the methodology's applicability are answered below as follows:

Q1- *Is the methodology applicable in a realistic scenarios?*

Yes. All steps are applicable. The only difficulty is in mapping the WSLA functions (as described in Section 5.2.4). This is because, during prediction, these functions depend on the expected value of the solver outputs as their inputs rather than on the monitoring values of the running service. WSLA functions depend on counting, comparing, or performing other statistics on the monitored values of the measurement directives. This is hard to do using their expected values. For this reason, and to solve this issue, the output of the reward variables (representing these measurements) is considered as a random variable. Hence, the functions were applied on realisations of this random variable.

Q2- *Is it scalable for a more complicated scenario?*

Theoretically, yes. The methodology can be scalable for different composite services with different SLA and WSDL documents being defined among these services. It is also scalable for more than one SLO, for longer evaluation periods, and for more

fine-grained interval units. However, this could be time consuming with regard to the tool support especially when obtaining the simulation trace.

Q3- *Does the generated model reflect the actual service?*

In some ways, yes. The final model, generated in Figure 7.10 from the automatic mapping of WSLA and WSDL, is abstract: i.e. the actual implementation of the method is not there. Solving this model in order to produce the value of the desired metrics might differ from the model that reflects the actual implementation of the service (in Figure 7.11). Although the generated model is abstract, it gives some idea about the expected performance of the service. Besides, it is all the user may have at hand. This is because, before the service's implementation, the user may have only the WSLA and WSDL documents at hand with which to start predictions. In addition, even after deployment, if the user of the methodology is a customer who wishes to predict SLA compliance before agreeing on it, he/she may only have access to these documents because the provider will not publish more. In a later focus, the model could be refined and might be expanded to cover more automatic aspects of the model by using a methods' file or work-flow documents.

7.3.3 Methodology's Generality

To answer the questions about the methodology's ability to accommodate several SLA languages and modelling formalisms, the case study and examples from a new SLA specification are utilised in Section 7.3.3.1. The answers are provided in Section 7.3.3.2.

7.3.3.1 Looking for the Evidence

In this section, an SLA written in a WS-Agreement specification is presented in order to investigate its mapping possibilities. Listing 7.5 presents a snapshot of this SLA [1].

Listing 7.5: A snapshot of an SLA "with measured metric" written in the WS-Agreement specification

```
1:<wsag:ServiceProperties wsag:Name="AvailabilityProperties" wsag:ServiceName="
    GPS0001">
2: <wsag:Variables>
3:  <wsag:Variable  wsag:Name="ResponseTime"  wsag:Metric="metric:Duration">
4:   <wsag:Location>qos:ResponseTime</wsag:Location>
5:  </wsag:Variable>
6: </wsag:Variables>
7:</wsag:ServiceProperties>
```

[1] This example is taken from http://serviceqos.wikispaces.com/WSAgExample

```
<!-- statements to offered service level(s) -->
8:<wsag:GuaranteeTerm Name="FastReaction" Obligated="ServiceProvider">
 9: <wsag:ServiceScope ServiceName="GPS0001">
10:  http://www.gps.com/coordsservice/getcoords
11: </wsag:ServiceScope>
12: <wsag:QualifyingCondition>
13:  applied when current time in week working hours
14: </wsag:QualifyingCondition>
15: <wsag:ServiceLevelObjective>
16:  <wsag:KPITarget>
17:   <wsag:KPIName>FastResponseTime</wsag:KPIName>
18:   <wsag:Target>
19:    //Variable/@Name="ResponseTime" LOWERTHAN 1 second
20:   </wsag:Target>
21:  </wsag:KPITarget>
22: </wsag:ServiceLevelObjective>
23:</wsag:GuaranteeTerm>
```

In this example, the service has an SLO called `FastResponseTime` (line 15) which states that the `ResponseTime` will be less than 1 second (line 19). This SLO is defined in particular for the `getcoords` operation (defined inside the `ServiceScope` in line 10) for the period that covers a week's working hours (defined inside the `QualifyingCondition` in line 13). The `ResponseTime` variable is defined in the `ServiceProperties` element as a `Duration` metric (line 3).

To map such an SLA to SDES, the methodology has to distinguish the five elements in SLA in order to map them later. These elements are the service object, measured metric, temporal constraint, composite metric and the SLO. From the SLA in Listing 7.5, the value of these elements are as follows:

- The service object is the `getcoords` operation.
- The measured metric is the `ResponseTime`.
- The temporal constraint regarding the period is the `week working hours`. However, the interval is not specified.
- The composite metric is null.
- The SLO threshold is `LOWERTHAN 1 second`.

From the elements retrieved earlier, the service operation and measured metric can be mapped normally according to WslaCP (or the general SlaCP methodology). However, since the interval of time at which to measure the response time inside the week period is not specified, then there is no way to figure out the evaluation interval during which the reward variable representing the response time has to be evaluated. In addition, this validity period is written in English which prevents the

automatic reading and interpretation of it. The composite metric in this example does not exist since the response time is defined without applying any function on it. Finally, the SLO threshold and comparison types are stated explicitly; hence they can be mapped as in WslaCP.

Another example of an SLA with a composite metric that is written according to the WS-Agreement is presented in Listing 7.6.

Listing 7.6: A snapshot of an SLA "with composite metric" written in the WS-Agreement specification

```
 1:<wsag:ServiceProperties wsag:Name="AvailabilityProperties" wsag:ServiceName="
     GPS0001">
 2: <wsag:Variables>
 3:  <wsag:Variable wsag:Name="AvgThroughput" wsag:Metric="
     metric:ThroughputofArrival">
 4:   <wsag:Location>qos:ThroughputofArrival</wsag:Location>
 5:  </wsag:Variable>
 6: </wsag:Variables>
 7:</wsag:ServiceProperties>

<!-- statements to offered service level(s) -->
8:<wsag:GuaranteeTerm Name="FastReaction" Obligated="ServiceProvider">
 9: <wsag:ServiceScope ServiceName="GPS0001">
10:   http://www.gps.com/coordsservice/getcoords
11: </wsag:ServiceScope>
12: <wsag:QualifyingCondition>
13:  applied when current time in week working hours
14: </wsag:QualifyingCondition>
15: <wsag:ServiceLevelObjective>
16:  <wsag:KPITarget>
17:   <wsag:KPIName>AvgThroughputLimit</wsag:KPIName>
18:   <wsag:Target>
19:     //Variable/@Name="AvgThroughput" LOWERTHAN 1000 transactions
20:   </wsag:Target>
21:  </wsag:KPITarget>
22: </wsag:ServiceLevelObjective>
23:</wsag:GuaranteeTerm>
```

Here, the service has an SLO called `AvgThroughputLimit` (line 17) which states that the `AvgThroughput` will be less than 1000 (line 19). This SLO is defined for the `getcoords` operation (line 10) for the period that covers the week's working hours (line 13). The `AvgThroughput` variable is defined as a `ThroughputofArrival` metric (line 3). From this SLA, the values of the required elements are as follows:

- The service object is the `getcoords` operation.
- The measured metric is the `Throughput`.
- The temporal constraint regarding the period is the `week working hours`. However, the interval is not specified.

- The composite metric is `Average`.
- The SLO threshold is `LOWERTHAN 1000 transactions`.

The problem in this example is that the variable `AvgThroughput`, which is defined inside the SLO, is not defined explicitly inside the `ServiceProperties` element (line 1). Here it is assumed that the measured metric is the `Throughput` and that the composite metric is derived using the `Average` function. However, this is not formal and it is hard to accomplish this automatically. In addition, the time intervals when the average of the throughput metric is taken are also unknown.

7.3.3.2 Interpreting the Evidence

Depending on mapping the SLA of Listings 7.5 and 7.6 in the previous section, and depending also on the discussion presented throughout this thesis, the evaluation questions regarding the methodology's generality can be answered as follows:

Q1- *Is the methodology general enough to be applicable to different SLAs?*

Possibly. Since the methodology is aimed at SLAs with QoS metrics built using a constructive ontology, its application might be limited to the type of SLA languages that accomplish this principle. For example, depending on the mapping in Listing 7.6, the definition of the `AvgThroughput` metric is not explicitly stated; hence, the automatic decomposition of its semantic in order to retrieve the measured metric with the functions applied on it, is not possible. Thus, using a constructive ontology does place a restriction on the type of SLA that is employed. However, some SLAs are moving towards implementing this ontology in order define their QoS metrics such as in the work of [58] who adopted the functions used by WSLA to define precisely its QoS metrics. Another restriction regarding the application of the methodology might be considered in the mapping of Listing 7.5. In this example, and even though the QoS metric used is measured (i.e. there is no need for constructive ontology in this case), the absence of the interval definition (i.e. the time needed to take the measure) also poses a problem in applying the methodology. However, this can be solved under the assumption that the measured metric is checked every second (depending on the fact that the response time in this SLO has a threshold of 1 second).

Q2- *Is the methodology general enough to be applicable for different stochastic models?*

Yes. Since the methodology is already defined for SDES, which is an abstract high-level stochastic modelling formalism, the methodology can be used for any stochastic model. This is because only a simple translation from the SDES model to

the preferred one is required. However, since the methodology depends on mapping QoS measured metrics (i.e. measurement directives in WSLA) as reward variables, the stochastic models must have an underlying Markov Reward Model (MRM) to be able to define the reward variable. Accordingly, using rewards as part of the methodology limits the type of stochastic model that can be used. However, since the models that depend on the underlying MRM are widely used and are supported with many software tools to build and solve them, no real restriction can be imposed.

7.3.4 User Support

To answer the questions about the methodology's ability to support its users, the case study is utilised in Section 7.3.4.1. The answers are provided in Section 7.3.4.2.

7.3.4.1 Looking for the Evidence

Depending on Figure 7.7 and Table 7.1, an incomplete model of the service with its reward variable can be generated automatically from a WSLA document. Hence, the user has to complete the model creation and then assign the right primitives to the reward functions.

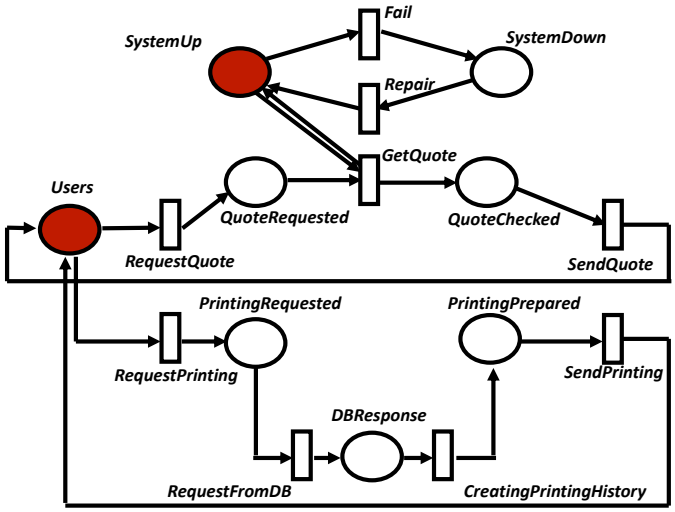

Figure 7.11: The SPN model of the stock quote service, completed by the user manually

Suppose that the user completes the model manually, as in Figure 7.11 where the red places indicate that these places are not empty. This SPN model is closed. It contains a *Users* place which represents a pool of all the customers who may want to check a quote value or print a quote history at any time. A user cannot send a

new request until it receives the result of a previous one. This allows the token in the places to be treated as a request, a response or a user.

The first operation of this service is represented in the upper part of Figure 7.11. When requesting a quote value, the transition *GetQuote* is fired and the request is queued in the place *QuoteRequested*. After this, the quote value is checked by firing the transition *GetQuote* which is suspended if the service is down. When the service is up again, the quote value is checked and the response is queued again in the place *QuoteChecked*; this is sent to the user when firing the transition *SendQuote*. The service up/down states are described in this model using a single token in *SystemUp* and *SystemDown* places respectively. The model alternates between the up/down states through the `Repair` and `Fail` transitions respectively.

The second operation of the stock quote service is represented in the lower part of Figure 7.11. When requesting the printing of a quote history, the transition *RequestPrinting* is fired and the request is queued in the place *PrintingRequested*. After this, the history of the quote values is retrieved from the database by firing the transition *RequestFromDB*, which is assumed to be reliable. After retrieving the quote history, the printing file is prepared through firing the transition *Creating-PrintingHistory* and the response is queued again in the place *PrintingPrepared* to be sent to the user by firing the transition *SendPrinting*. All activities in this model have exponential distributions.

Accordingly to the model in Figure 7.11, the reward functions that are automatically generated in Table 7.1 are no longer valid due to changes in the names of the primitives and in the model's structure. Hence, the reward functions have to be updated by the user with the new primitives to complete their definitions.

Table 7.2: The reward functions completed by the user

Measurement Directive	**Automatically Generated Reward Function**	**User Manual Input**	**Automatically Updated Reward Functions**
StatusRequest	$\begin{cases} 1 & \text{if } SystemUp = 1 \\ 0 & \text{otherwise} \end{cases}$	-	-
Gauge	$\begin{cases} GetQuote\_s(\sigma) & \forall\sigma \\ 0 & else \end{cases}$	*Quote-Requested*	$\begin{cases} QuoteRequested(\sigma) & \forall\sigma \\ 0 & else \end{cases}$
InvocationCount	$\begin{cases} 1 & \text{if } a = GetQuote\_a \\ 0 & otherwise \end{cases}$	*GetQuote*	$\begin{cases} 1 & \text{if } a = GetQuote \\ 0 & otherwise \end{cases}$
ResponseTime	$\begin{cases} 1 & \text{if } FinishedUsers{=}1 \\ 0 & \text{otherwise} \end{cases}$	*Users*	$\begin{cases} 1 & \text{if } Users{=}1 \\ 0 & \text{otherwise} \end{cases}$

Table 7.2 shows the automatically generated reward functions taken from Table 7.1, in addition to the user manual input (which depends on the model in Figure 7.11) and the updated reward functions. When the user does not want to make any changes, he/she only confirms this (this is represented using the dash (-) sign in the table). For example, in this table, the reward function of the *StatusRequest* measurement directive does not need any change since the same up/down mechanism is used in both models (in Figures 7.7 and 7.11).

After finishing the reward function creation, no interaction from the user is needed. The methodology can perform the remaining steps automatically in terms of producing the compliance probability.

7.3.4.2 Interpreting the Evidence

Depending on the previous section, the evaluation questions regarding the support this methodology offers to its users are answered as follows:

Q1- *What degree of automation and help does this methodology offer to its users?*

It can be considered to offer a reasonable amount of help in different areas as follows:

- Completing the model: Comparing the model completed by the user (Figure 7.11) and the one generated from WSLA automatically (Figure 7.7), it can be seen that there is some kind of similarity. This means that the automatically generated model has offered a basis and some sort of structure for completing the desired model; hence, it helps the user to envisage the final model of the service. For example, the user in Figure 7.11 kept the fail/repair mechanism while he/she did change the mechanism related to the response time (i.e. deleting the *FinishedUsers* place). The user also added more detail to the printing operation, which involved making requests of the database (*RequestFromDB*); this was expected using the dashed box in Model 7.7.

- Updating the reward function: Looking at the user's input in Table 7.2, the user would not be involved in taking care of building the reward function with the fine grained information or with the required evaluation intervals. All that is needed is a confirmation of the correction to the reward function or the supplying of the desired place/transition.

- Overall steps: All other steps of the methodology are automated (except uploading the SLA and the model files). Accordingly, reasonable help and automation are offered to users.

Q2- *Does it accomplish the aim of minimal user interaction?*

It is not ideal. According to the previous answer, user interaction is still necessary to complete the model and to help in assigning the model primitives correctly. However, even when user interaction is needed, help is offered by the methodology. This might not be a perfect solution but it is a step forward in helping the user to perform SLA compliance prediction for his/her predefined SLA. Hence, although the aim of the methodology is for it to be as automated as possible, this cannot be done (as ideally) unless the user of the methodology has access to other service supporting documents that help in reflecting a real implementation of this service.

7.4 Evaluation of the WslaCP Tool

In this section, an evaluation of the WslaCP tool based on the described case study is provided. Recalling the evaluation fundamentals presented in Section 7.1, four steps should be conducted to complete the evaluation process. These steps can be written in the context of evaluating the WslaCP tool as follows:

Framing the evaluation: For the WslaCP tool, the targeted users are the same as for the WslaCP methodology: i.e. service providers/engineers, SLA engineers, or modellers.

Defining evaluation goals and objectives and evaluation methods: The evaluation is conducted using the case study as in the WslaCP methodology evaluation. This evaluation is based on the degree to which the tool can achieve the aim and objectives it was designed for. A number of questions can be formulated to evaluate the aim and objectives of the tool. These are aligned with the research questions addressed in Section 1.4 and concern:

1. *Tool's Applicability*: The questions related to this are as follows:
   - Q1- Does the tool implement the WslaCP methodology?
   - Q2- Is the tool applicable for real example scenarios? Is it scalable for more complicated scenarios?
2. *User Support*: The questions related to this are as follows:
   - Q1- Is the tool GUI usable and user-friendly?
   - Q2- What degree of automation and help does this tool offer to its users?

For the last two steps, which are **Looking for the evidence** and **Interpreting the evidence**, the described case study is utilised to answer the aforementioned evaluation questions.

In the following sub-sections, the evaluation questions, along with looking for evidence and interpreting it, are discussed in detail.

7.4.1 Tool's Applicability

To answer the questions about the ability to use the WslaCP tool for a real WSLA and web service model, the case study is utilised in Section 7.4.1.1. The answers are provided in Section 7.4.1.2.

7.4.1.1 Looking for the Evidence

This section investigates the way the tool utilises the WslaCP methodology for a real WSLA (the described WSLA for the stock quote service) and a real SPN model. This is achieved by examining the GUIs that the tool uses to interact with its users. The main GUIs the user has to follow in the WslaCP tool are specified as follows:

1. Uploading the WSLA file.
2. Completing the model creation or uploading its file.
3. Completing the reward function definition.
4. Solving the model.
5. Displaying the results.

These steps are described in detail in what follows.

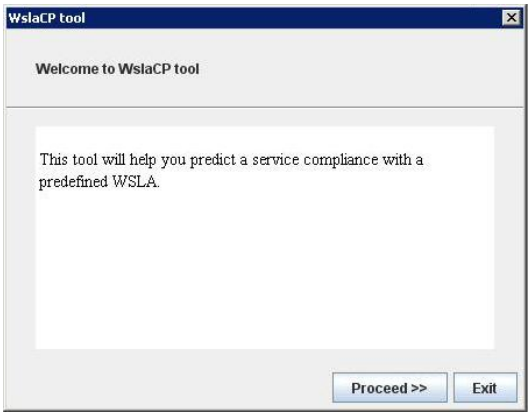

Figure 7.12: The WslaCP welcoming GUI

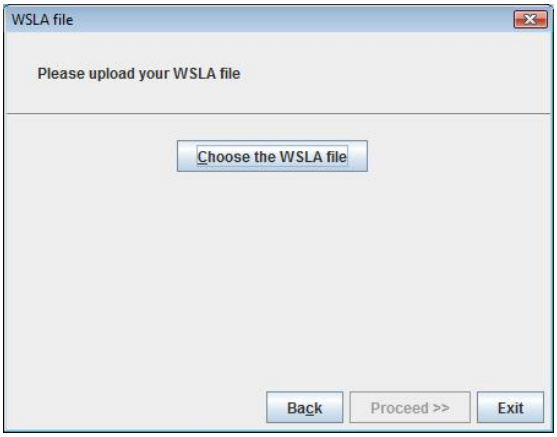

Figure 7.13: The WslaCP GUI for uploading a WSLA contract

1- Uploading the WSLA File: When executing the WslaCP tool, a welcoming GUI appears, as depicted in Figure 7.12. After clicking the proceed button, the next GUI, depicted in Figure 7.13, is presented. This GUI contains a button that opens a file chooser for uploading the WSLA contract of the stock quote service.

2- Completing the Model Creation or Uploading its File: Once the file is uploaded, the tool stores a copy of it in a specific directory (*WslaCP\WSLA_DOC*).

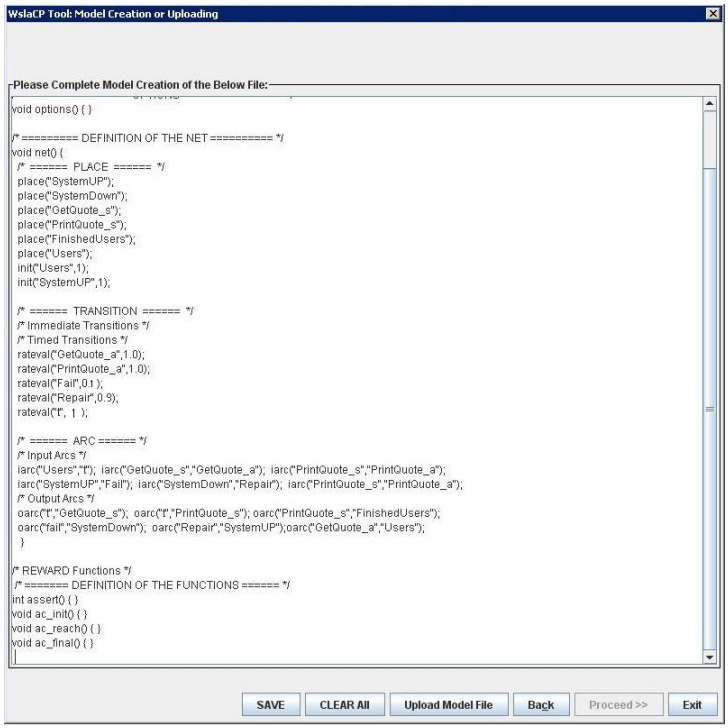

Figure 7.14: A WslaCP GUI for completing the model creation

The tool then examines this file automatically to produce a shorter version of it by omitting all the unnecessary information. This makes the mapping in the next step faster. This new file is stored in a new XML document called *CompactSLA.xml*. This file is used by the tool and is hidden from the user. However, the user can find it in the (*WslaCP\WSLA_DOC*) directory. After generating this file, the tool then parses it automatically to produce the *SLA-Model File* (recalling Figure 6.1) and stores it in the same directory. This file is used by the tool in a later step.

As described in Section 6.3.4.1, the part relating to the model primitives in the *SLA-Model File* is translated automatically to CSPL. After this, the user has to complete the model creation. However, in the current implementation of the tool, it is not possible to do this in a graphical format; instead it has to be done in a text-based format, as appears in Figure 7.14. The content of this figure is the textual CSPL representation of the automatically generated model depicted in Figure 7.7. The user has to modify this and then save the changes. However, if the user wants to upload an existing file that contains a complete model definition, he/she can use the 'Upload Model File' button. If the user completes the model manually, as specified in Figure 7.11, then the CSPL file representing it will be as presented in Listing 7.7. This file is stored in the folder (*WslaCP\SPNP_DOC*).

Listing 7.7: The CSPL file of the SPN model depicted in Figure 7.11

```
#include <stdio.h>
#include "user.h"

/* global variables */

/* ================= OPTIONS ================ */
void options() { }

/* ========= DEFINITION OF THE NET ========== */
void net() {
  /*  ======  PLACE  ======  */
  place("Users"); init("Users",10); place("QuoteChecked");
  place("QuoteRequested"); place("SystemUp");  init("SystemUp",1);
  place("SystemDown");place("PrintingRequested");
  place("PrintingPrepared"); place("DBResponse");

  /*  ======  TRANSITION  ======  */
  /* Timed Transitions */
  rateval("RequestQuote",1); rateval("GetQuote",1); rateval("SendQupte",1);
  rateval("Fail",0.1);  rateval("Repair",0.9);
  rateval("RequestPrinting",1); rateval("CreatingPrintingHistory",1);
  rateval("RequestFromDB",1); rateval("SendPrinting",1);

  /*  ======  ARC ====== */
  /* Input Arcs */
  iarc("Users","RequestQuote"); iarc("QuoteRequested","GetQuote");
  iarc("QuoteChecked","SendQuote"); iarc("SystemDown","Repair");
```

```
  iarc("SystemUp","Fail");
  iarc("Users","RequestPrinting"); iarc("PrintingRequested","RequestFromDB");
  iarc("DBResponse","CreatingPrintingHistory");
  iarc("PrintingPrepared","SendPrinting");

  /* Output Arcs */
  oarc("RequestQuote","QuoteRequested"); oarc("GetQuote","QuoteChecked");
  oarc("SendQuote","Users"); oarc("Fail","SystemDown");  oarc("Repair","SystemUp");
  oarc("RequestPrinting","PrintingRequested"); oarc("RequestFromDB","DBResponse");
  oarc("CreatingPrintingHistory","PrintingPrepared"); oarc("SendPrinting","Users");
}

/* REWARD Functions */

/* ======= DEFINITION OF THE FUNCTIONS ====== */
int assert() {}
void ac_init() { /* Information on the net structure */ }
void ac_reach() { /* Information on the reachability graph */ }
void ac_final() {}
```

After completing or uploading the model file, the tool parses it to extract all the available transitions and places in order to help the user choose the most suitable primitives to complete the reward function definition in the next step. These details are hidden from the user.

3- Completing the Reward Function Definition: After completing the model, the tool extracts all the reward function templates for the available measurement directives that are situated in the second part of the *SLA-Model File*. From these templates, the tool automatically specifies the parts that need a user's assistance, as well as those that can be translated directly to the solver input language without user interaction. Accordingly, the tool generates, for each of the seven measurements available in the WSLA specification, a tailored GUI that is dedicated to receive the required input from the user. These GUIs provide a brief description of the measurement used and what *slap* it is related to. They also present some description about the information that needs to be entered with a window that presents only the suitable set of primitives (i.e. places and transitions available in the model) for completing the reward function of this kind of measurement. This is aided by drop-down lists that are populated automatically by using the output of the model file.

In this case study, four reward functions were generated, as specified in Table 7.1. These are presented in the tool GUIs one after another for completing their definition. For example, Figure 7.15 presents the GUI related to the reward function of the *StatusRequest* measurement that is part of the first SLO. The tool recognises automatically that it should present a grid inside this GUI. The grid columns are

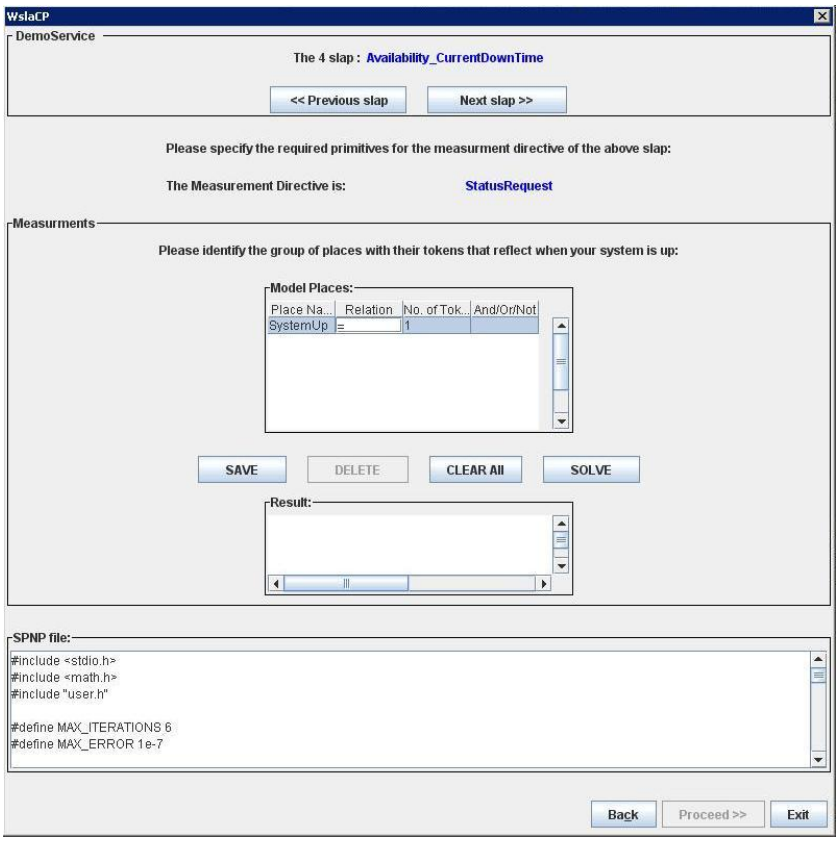

Figure 7.15: A WslaCP GUI for completing the *StatusRequest* reward function

populated automatically with the following information: a list of all the places available in the first column, a list of the common arithmetic relations in the second column, a text box for entering an integer number (token number) in the third column, and a list of the common boolean operators in the fourth column. When this GUI is presented, the tool automatically populates the reward function template of this measurement, as shown in Table 7.1, inside the grid of this GUI (in this case, it is : $SystemUp = 1$). The user then has to confirm the correctness of this availability condition or modify it appropriately and then click the save button. Using lists inside the grid makes it easier for the user to modify the availability condition because the user can choose from them rather than entering the values manually where it is possible to make a spelling error or a mistake.

Once the changes are saved, the tool automatically generates the CSPL reward function, in Listing 7.8, that meets with the existing or the modified condition.

Listing 7.8: The CSPL code equivalent to the StatusRequest measurement directive in Figure 7.15

```
double StatusRequest_GetQuote() {
if (mark("SysemUp")=1)
```

```
return (1.0);
else return (0.0); }
```

This reward function is inserted directly in the appropriate place in the SPNP file (in the 'REWARD Functions' area in Listing 7.7). The user can view this from the text area related to the SPNP file shown at the bottom of Figure 7.15.

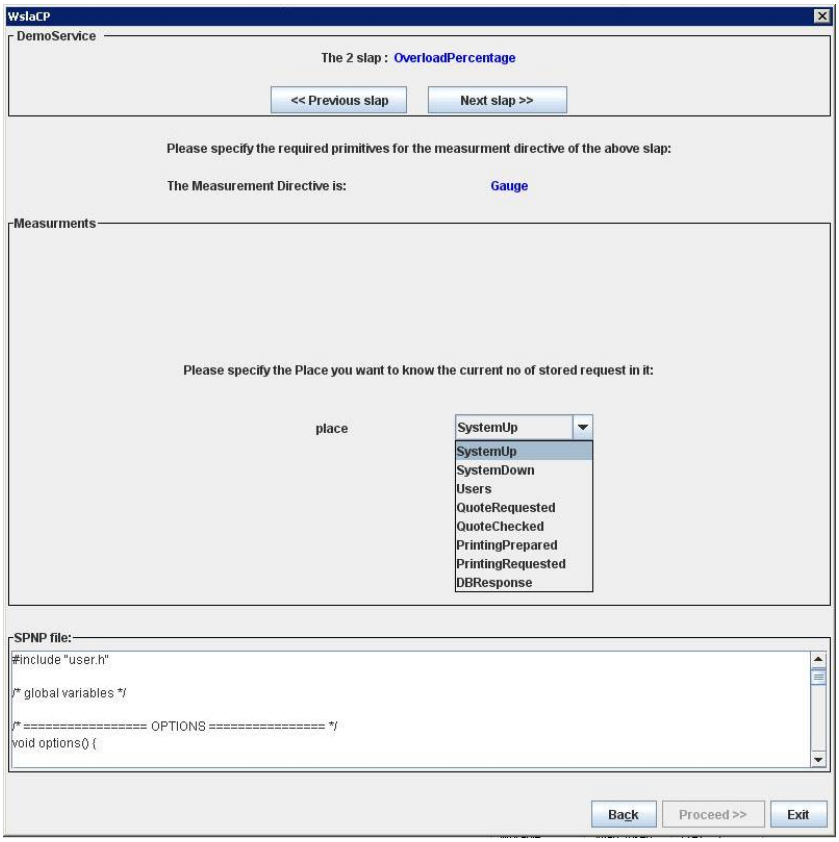

Figure 7.16: WslaCP GUI for completing the *Gauge* reward variable

Another example of the GUI that is presented for completing the reward function definition is depicted in Figure 7.16. This GUI concerns the *Gauge* measurement directive that is part of the `OverloadPercentage` SLA parameter used in the second SLO. Its reward function template, as illustrated in to Table 7.1, is presented using the place name (*GetQuote_s*). If it is not correct to use this place, which is the case here, the user has to choose the desired place from the drop-down list (which is *QuoteRequested*)[1].

Listing 7.9: The CSPL code equivalent to the Gauge measurements

```
double Gauge_GetQuote() {
```

[1]This reward function is a rate reward returning the number of tokens in his place.

```
return (mark("QuoteRequested"));}
```

The CSPL code (in Listing 7.9) is generated automatically from this GUI; this can be viewed by the user in the text area of Figure 7.16.

The GUIs related to *InvocationCount*, which is also part of the second SLO, and the *ResponseTime*, part of the third SLO, are not discussed here because they are almost similar.

4- Solving the Model: After completing the definition of all the reward functions, the user has to click the 'SOLVE' button. When this is done, the tool automatically updates the model file with the time required to solve the reward functions (these times are specified in the *SLA-Model File*) following CSPL syntax. The tool also adds the commands necessary to solve the model inside the *ac_final()* function of the model file and then calls the solver implicitly. All these are hidden from the user.

For example, in the first SLO, the reward function of *StatusRequest* is checked, as shown in Table 7.1, every 1440 minutes for one month (i.e. 31 days = 44640 minutes). This is automatically expressed in CSPL as in Listing 7.10.

Listing 7.10: The CSPL code equivalent to the time to solve the *StatusRequest* reward function

```
int loop=0;
/* Compute the reward function for the interval and time period specified in the
    schedule. */
for (loop=0; loop < 44640; loop+=1440)
{
solve ((double) loop);
expected(StatusRequest_GetQuote);
}
```

To clarify this further, the loop in Listing 7.10 begins at 0 until it reaches the upper bound of 44640 with increments of 1440. An SPNP function (*solve*) is used to solve the model according to these time instants; this computes the expected value (*expected*) of the reward function (*StatusRequest_GetQuote*) at each instant.

For the second SLO, the reward function of *Gauge* is checked, as in Table 7.1, every 5 minutes for one month. In addition, the reward function of *InvocationCount* is checked every 60 minutes for one month. This is automatically expressed in CSPL as in Listing 7.11.

Listing 7.11: The CSPL code equivalent to the time to solve the *Gauge* and *InvocationCount* reward functions

```
int loop=0;
/* Compute the reward function for the interval and time period specified in the
    schedule. */
```

```
for (loop=0; loop < 44640; loop+=5)
{
solve ((double) loop);
expected(Gauge_GetQuote);
}
for (loop=0; loop < 44640; loop+=60)
{
solve ((double) loop);
pr_cum_expected(InvocationCount_GetQuote);
}
```

where *pr_cum_expected()* is used to compute the expected accumulated value (interval of time) for the reward functions. For the third SLO, the reward function of *ResponseTime* is checked, as seen in Table 7.1, every 15 seconds for one month. This is automatically expressed in CSPL as in Listing 7.12.

Listing 7.12: The CSPL code equivalent to solve the *ResponseTime* reward function

```
int loop=0;
/* Compute the reward function for the interval and time period specified in the
    schedule. */
for (loop=0; loop < 44640; loop+=15)
{
solve ((double) loop);
expected(ResponseTime_PrintQuote);
}
```

5- Displaying the Results: After the tool has run the plugged-in solver in order to obtain the required results, the tool automatically uses these results to compute the probability of SLA compliance; this is done for each SLO individually. However, as described in Section 6.4, SPNP does not provide the required simulation results; for this, Möbius is used from the point of solving the model onwards. Unfortunately, this mans that all the previous steps have to be undertaken manually by the user using the Möbius tool.

After receiving the simulation results from Möbius, the user has to upload the file containing these simulation replicas into the WslaCP tool in order to derive the compliance prediction for each SLO automatically. The fine-grained details of preparing the result file, parsing each replica, applying WSLA functions to it, evaluating its compliance with the SLO threshold, and finally computing the probability of SLO compliance among all the replicas, are hidden from the user. All the user can see is the result, as depicted in Figure 7.17.

Figure 7.17: WslaCP GUI for presenting the result of the SLO compliance

7.4.1.2 Interpreting the Evidence

Depending on the previous section, the evaluation questions regarding the tool's applicability can be answered as follows:

Q1- *Does the tool implement WslaCP methodology?*

Yes. All the steps are applicable. However, the difficult issue with regard to implementing the WslaCP methodology in the tool is that the mapping functions cannot be applied when the solver being used is analytic. This is because this type of solver depends on mathematical formulae to compute the expected results of the reward variables. Hence, it is not possible to retrieve the raw data that generate these values. However, when a simulation is used, applying the functions becomes feasible because the simulation depends on running the model multiple times and then generating the result depending on these runs. Using a simulation trace permits the application of the functions because each of them represents one instance of monitoring the running service and, in turn, a realisation of the random variable assumed in the WslaCP methodology.

A disadvantage of the implemented WslaCP tool is that it is implemented using two plugged-in modelling tools, SPNP and Möbius, and each can only be used for specific steps. This is because a plugged-in tool that is able to provide both a simple textual input and an accessible simulation trace file at the same time could not be found. Hence, although the tool implements all aspects of the WslaCP methodology, it cannot do this in a continuous fashion, using a single plugged-in tool, from the time the SLA file is uploaded and until the compliance probability is received.

Q2- *Is the tool applicable for real example scenarios? Is it scalable for more complicated scenarios?*

Yes, it is applicable for a real example scenario but it is not readily scalable. This is because the reward variable has to be solved for every instant when an observation is made; this makes the model expensive to solve. As the tool depends on the plugged-in tool to solve the reward variable and to store the replicas to a file, this proves to be very time consuming. In the first SLO example, the Möbius tool took more than four hours to complete a simulation run of 100 replicas using a

single 2.19 GHz processor. This run solved the reward variable for 44640 instants as specified in a month's validity period. However, the running time would be longer if a longer period were considered or if multiple SLOs were used. This raises problems with scalability which can be attributed to the simulation speed of the plugged-in tool, or the tool design that requires the extraction of the simulation replicas.

7.4.2 User Support

To be able to answer the questions regarding the ability of the WslaCP tool to support its users, the same example regarding the tool's applicability, as presented in Section 7.4.1.1, is utilised. The evaluation answers, related to interpreting the evidence, are provided in what follows:

Q1- *Are the tool GUIs usable and user-friendly?*

Yes, for most of the GUIs. This is because WslaCP GUIs provide good descriptions of the required inputs and an easy way of entering these inputs using drop-down lists. The only exception is the GUI relating to completing the creation of the service model (as depicted in Figure 7.13); this might put a burden on the user who has to complete the model textually in a CSPL format instead of carrying this out graphically.

Q2- *What degree of automation and help does this tool offer to its users?*

It offers a reasonable level of help. The aim of the WslaCP tool is to allow the user to predict the WSLA fulfilment by completing the service model and choosing some primitives for this model. However, in the current implementation this is not possible because of the use of two plugged-in modelling tools. Hence, the user has to carry out more steps manually than was the aim before the model can be solved by Möbius. These manual steps that are specific to Möbius include putting the reward functions into the model, assigning the time to solve them, running the simulation, and then uploading the result file into the WslaCP tool.

7.5 Conclusion

This chapter evaluates the WslaCP methodology and its tool in terms of achieving the aim and objectives they were designed for. The evaluation was carried out based on a case study of a stock quote service considering a number of evaluation questions. The contribution of this chapter is to demonstrate the following: first, the viability of the WslaCP methodology in terms of its applicability, generality, and user support; second, the use of the WSDL file in adding an extra level of automation to the model

creation; third, to show the feasibility of the WslaCP tool in term of its applicability and user support. The case study proves that the WslaCP methodology is general enough to accommodate different stochastic models. However, its generality with regard to adopting another SLA language is less easily applicable since it depends on SLAs with a constructive ontology. The methodology is considered applicable for a real scenario. In addition, it has been proved to offer reasonable help to its users as many details are hidden. The WslaCP methodology evaluation regarding the automatic model creation shows that only a part of the service model can be produced from WSLA, which is abstract and incomplete. Hence, the methodology was extended to allow the use of WSDL file in producing this model. Although the newly generated model does miss some information, it helps the user in constructing a complete service model when such a job is delegated to him/her. Finally, the tool which exploits WslaCP methodology is considered to implement all its aspects. However, this is achieved by using two plugged-in tools which prevents the user from obtaining one continuous solution, thus exposing him/her to more manual work. In the next chapter, a summary of the research's contributions and some potential future work are presented.

Chapter 8

Conclusion

In this thesis, SlaCP, a new engineering methodology that predicts the probability of SLA compliance, is proposed including the architectural design of a software tool that automates many of its aspects. The SlaCP methodology achieves compliance prediction by mapping SLA elements into metrics of a service's stochastic model which is created semi-automatically. It then uses the outcome of the solved model to specify further the desired SLA metrics. Finally, it predicts SLA compliance by comparing the predicted values of the SLA metrics with the agreed SLO thresholds. In order to show its applicability, an implementation of the SlaCP methodology was achieved by using WSLA and SDES; this is called WslaCP. The tool's architectural design was also implemented using SPNP and Möbius to automate most aspects of WslaCP. Finally, a case study was employed to evaluate both the WslaCP methodology and the tool. This indicated their usability and also revealed some limitations in specific areas. Furthermore, it showed the methodology's ability to offer an increased level of automation in the creation of the service model when using a WSDL file that describes the service operation.

The remainder of this chapter is organised as follows: Section 8.1 outlines the contributions made by this thesis. Section 8.2 then provides reflections on the research conducted in this thesis in the light of answering the research questions. Finally, Section 8.3 offers a brief discussion regarding possible future extensions to the work proposed in this thesis.

8.1 Summary of Contributions

The contributions made by this thesis are as follows:

1- SlaCP, a new engineering methodology for the automated prediction of the probability of SLA compliance (addressed in Chapter 3). The

methodology exploits a model-based approach to predict SLA compliance, as automatically as possible, through seven phases. These include: interpreting existing SLA metrics; mapping them into a generalised stochastic model (this includes primitives, reward variables, time to solve them, functions over solver output, and SLO evaluation function); completing the model creation; refining the model into a concrete one to be solved; solving the model to produce the desired values; using these to compose the ultimate SLA metrics; and finally evaluating them against the SLO thresholds to decide the probability of compliance. The design of the SlaCP methodology is described and addressed from two viewpoints: that of the methodology's user and that of its tool designer. This was done to provide interested users with a basis from which to understand the methodology, irrespective of the concrete SLA language and stochastic modelling formalism. To the best of our knowledge, this is the first research study that performs SLA prediction in an all-in-one methodology starting from a predefined SLA and ending with a compliance probability. Although the methodology offers a reasonable level of automation, user interaction is still required to complete the model creation and the reward function definitions. SlaCP can be considered to apply to stochastic models in general since it uses a generalised model that can be translated automatically into a concrete one. However, it is less easily applicable to SLA specifications in general since it assumes that the SLAs used employ a constructive ontology to define their QoS metrics.

2- WslaCP, a theoretical implementation of the proposed methodology for WSLA and SDES. This includes:

1. A formal representation of a WSLA agreement through an unambiguous mathematical representation (addressed in Chapter 4). This provides a complete and formal view of the WSLA elements necessary for prediction (rather than for monitoring). In addition, it paves the way towards achieving the correct application of the phases of the methodology. Some literature has already attempted to formalise general SLA specifications but only at a high level and without utilising the hierarchical definition of QoS metrics. In addition, the proposed representation in this contribution is tightly coupled only with prediction-related elements. The relation between the components of some formal elements (such as the SLO nested expressions) was difficult to reflect in mathematical terms; hence, this was accomplished using algorithms that are implemented by the tool.

2. A mapping process from WSLA to SDES (addressed in Chapter 5). This includes five steps: representing service operations as model primitives; mapping

measured QoS metrics as reward variables; mapping the time at which measured QoS metrics are taken as observation intervals; mapping the functions constituting QoS composite metrics as functions of the model outputs; and finally, mapping the SLO threshold and its comparison operator as an evaluation function to predict the probability of SLA compliance. This demonstrates the applicability of the generic methodology for concrete SLA and stochastic model specifications. It also permits better concrete understanding of the proposed methodology. This mapping helps in building a service model, but user interaction is still needed to complete the model.

3- The architectural design and implementation of a software tool. The former automates the SlaCP methodology, while the latter implements this design using WSLA, SPNP and Möbius (addressed in Chapter 6). This includes:

1. The SlaCP tool's architectural design: A set of architectural components are proposed, together with their design. This design aims at making the tool modular for use with different SLA specifications, modelling formalisms and implementation languages. This was achieved by expressing the tool's primary outputs in a language independent of the modelling tool used to build and solve the model, and the language utilised to implement the tool's APIs. The tool design also aims to automate the SlaCP methodology as much as possible to help users to predict SLA compliance with minimum interaction.

2. The WslaCP tool: The architectural design of the SlaCP tool was implemented for WSLA and an SPN model. It exploits the WslaCP methodology regarding the mapping details; hence, it is referred to as WslaCP. The implementation was accomplished using different techniques, both existing and novel. Existing tools, namely SPNP and Möbius, were used as plugged-in tools; Java was used to implement the rest of the tool engines and to provide the communication among them; the `SDESSch` novel schema was created as a means of representing the reward model in a machine-readable format; and finally, Matlab was used to represent the functions that were applied on the solver outputs. Using this tool, the user can predict SLA compliance with a minimum level of interaction. Furthermore, such a tool allows the user to adopt a new SLO threshold or service design according to their impact on SLA compliance. This tool cannot provide a continuous flow from uploading the SLA until the SLA compliance is produced due to the lack of a plugged-in tool that satisfies all the tool requirements. This left no choice other than to use two plugged-in tools;

each is suitable for a set of the steps involved in the WslaCP tool. This unfortunately increases the manual interaction required from the user.

4- Using WSDL as a new extension to WslaCP methodology (described in Chapter 7). The WslaCP methodology was extended by employing WSDL documents. Using WSDL to produce the service model improved the level of automation and completeness in the model's creation, although user interaction is still required to refine it, parameterise its parameters and specify its initial state.

8.2 Reflections on Research Outcomes

This section provides reflections on the research conducted in this thesis in the light of the research questions addressed in Section 1.3. In this section, each research question is answered first, then, a reflection on the overall thesis work is provided.

8.2.1 The First Research Question

The first research question was as follows: **Can an existing SLA be mapped theoretically to metrics of a stochastic model in an automated fashion?"** This question implies several sub-questions, as indicated in Section 1.3.

Q1- Are all SLA elements useful for prediction-related mapping or are some of them monitoring-related only? What are these elements?

A review of some SLA examples showed that SLAs are written for monitoring purposes. SLA elements are designed to measure QoS metrics using URIs or measurement directives of the running service. Moreover, the functions of its composite metrics are also designed to work on values of these measurements. Many elements also exist to manage the runtime relationship between the service provider and customers, and within the different elements of the SLA. For compliance prediction, not all elements are considered, only those elements that are used to build up the QoS metrics (these are the basic metrics, the temporal constraints to retrieve these metrics, and the composite metrics) with all the SLO threshold information (the acceptable numeric value, the arithmetic relation, and the validity period), in addition to elements that help in building the model of the service (these are the service objects and URIs). All the previous information allows this question to be answered as follows :"*Not all of SLA elements are necessary for its prediction: only elements related to model creation, QoS metric definitions and SLO information are used.*"

Q2- Does an SLA provide any information that helps in automatically creating

a complete service model or a part of it? If yes, what are these elements? Can other supporting documents enhance this automatic model creation?

An exploration was carried out to find out what elements inside an SLA can be used to produce model primitives. Since all QoS metrics inside an SLA are defined for a specific service object such as an operation, this can be utilised as a state variable and an action in the service model. This is because the requests that this service object handles are queued normally before they can be served; this can be represented as a state variable. In addition, the service object typically takes some time to service the request; this can be reflected as a firing delay of an action. Furthermore, basic QoS metrics in an SLA can be used as reward variables. These metrics usually describe a service attribute, and reward models can be utilised to retrieve their values. In addition, the temporal constraint inside an SLA, through which the basic QoS metrics are measured, can be used as an evaluation interval of the reward variables. Finally, since the derivation of some basic metric implies the presence of specific primitives inside the model, this can provide hints for model creation. An example of this is the QoS that measures availability; this implies the use of primitives that reflect the up/down states in the model. As a conclusion, a part of the reward model (i.e. a service model with reward variables) can be produced automatically from an SLA. This model contains pairs of state variables/actions (service objects), other primitives (hints from basic QoS metrics), reward variables (basic QoS metrics), and the time to solve them (temporal constraints). This allows the question to be answered as follows: *"An SLA can provide some information that helps to create automatically an abstract reward model which includes part of the service model and the reward variables defined in it"*.

Using WSLA as the SLA language, another supporting document can be used to enhance the level of detail and completeness in the automatically produced service model. The model generated from WSLA and WSDL is more complete. However, more work needs to be done to increase the level of detail in this model. This potentially can be done using service supporting documents such as WSFL or BPEL4WS. However, it is still difficult to parameterizse the model automatically. This allows the question to be answered as follows: *"A WSDL document that describes the service operations inside a WSLA document can be used to create a more detailed service model automatically."*

Q3- Assuming that such a service model is available, how do the prediction-related SLA elements correspond to the service model? In other words, are they mapped on the model primitives or are they captured by a function over the results of solving this model?

After completing the service model, the basic QoS metrics are mapped to it as reward variables; this creates a reward model. In addition, the temporal constraints are used to define the times at which these reward variables are solved. Functions that are used inside the composite metrics, on the other hand, cannot be mapped as a part of these reward variables. They need to be mapped as functions over the results of solving the reward models. This is because, in an early part of the work of this thesis, functions were mapped as parts of the reward variables. This proved to be possible for some functions but not for all. This is because WSLA functions, while monitoring, are applied on the values of a basic metric that are taken at different time instants. However, this cannot be reflected as part of a single reward variable because a reward variable value at a specific instant cannot be predicted and used by the same reward variable in the same model (i.e. a reward variable value at a specific time cannot be used as an input to itself). The impact of considering functions being applied on the reward model outcome rather than as part of the reward variable was explored using Möbius; this proved to be more reasonable. This allows the question to be answered as follows: *"The basic QoS metrics are mapped on the service model as reward variables in order to generate a reward model. However, the functions of the composite metrics are mapped on the output of solving this reward model."*

Q4- Given that the mapping from SLA into a stochastic model is feasible, to what extent can the mapping process be automated?

The research in this thesis does not provide a complete automation of the process of SLA compliance prediction. User interaction is always necessary to complete the model creation and to assign the necessary primitives that are relevant to the reward functions. However, the methodology succeeds in simplifying this process for the user as much as possible. This allows the question to be answered as follows: *"An automated mapping process from an SLA to a stochastic model of the service is only partly possible as user interaction is vital to complete the model creation and the assignment of reward functions."*

In the light of the analysis of these four sub-questions, the following answer holds for the first main question: "**An existing SLA can be mapped semi-automatically to produce a reward model that helps in predicting the SLA compliance probability.**"

8.2.2 The Second Research Question

The second research question was as follows: "**Is the theoretical mapping process applicable in a real example scenario?**" This question implies the following sub-questions:

Q1- Is the methodology, which exploits the research hypothesis, applicable before or after deploying the service in the real world? Is it useful for service providers and customers?

The SlaCP methodology that explores the research hypothesis was developed. It can be applied at any stage of service creation or SLA establishment. However, after service deployment, the user may have more supporting documents that may help in the automatic model creation, or data that will help in parameterising the model.

The SlaCP methodology is primarily targeted at service providers, modellers, or SLA engineers, helping them in designing a better SLA document with compliance probability that is better understood. A service customer may also use this methodology if he/she is interested in the ability of the desired service to comply with an SLA. To use the methodology, the customer must have the SLA and should be able to parameterise the service model. This allows the question to be answered as follows: *"An SLA compliance prediction methodology can help its users to predict SLA compliance probability in as automated way as possible, either after or before deploying the service."*

Q2- Assuming that the methodology is applicable before deploying the service, how will the user be able to obtain the necessary information for parameterising the model (e.g. delay time)? How can the initial state of the model, which is necessary for solving it, be determined (e.g. does this depend on simulation results or historical data)?

A review of some literature studies showed that most of the parameters in a model-based evaluation approach are either assumed or taken from historical data. In addition, some model parameters can sometimes be extracted from the log of a system that is similar to the one under consideration, or from scenarios that are running theoretically [78]. In this methodology, it is assumed that parameterising the model will be completed by a user and not investigated further. This allows the question to be answered as follows: *"The parameters of the service model that is used in the SLA compliance prediction methodology are assumed."*

Q3- Does the type of model affect the usage of the methodology? In other words, is the type of stochastic model (i.e. closed or open, steady state or transient)

important for mapping validity?

The model of a service can be open; that is, requests arrive to the model randomly at a specific rate, are served, and then leave the model (the number of requests is not fixed). The model may also be closed, where the number of requests is fixed [159]. The type of the model (i.e. whether closed or open) is not crucial for the proposed methodology since basic metrics can be mapped to both of them (for response time mapping, either choice is possible). Given that the SLA is monitoring centric, it naturally leads to QoS metric evaluation at instants/intervals of time instead of a steady state. Consequently, the model evaluates the reward variables for every instant when an observation is made. This allows the question to be answered as follows: *"The service model that is used in the SLA compliance prediction methodology is transient to reflect the monitoring nature of the SLA."*

Q4- Is there any difference in prediction if a service is composite (i.e. not single)? Can a service model for a composite service still be generated and used by the methodology assuming the hypothesis?

In a composite service, each service has a specific SLA for each of the other services. This cannot affect the application of the methodology as each service, with its SLA and WSDL files, is treated independently. This means that, for any SLA between any two services, the mapping to a reward model can be accomplished as normal. The whole model representing the composite metrics results by merging all the individual models. This allows the question to be answered as follows: *"There is no difference in prediction if the service is composite or single, and a reward model can still be generated partially for this service."*

Q5- Can an all-in-one software tool automate all aspects of the methodology?

An architectural design of a tool that automates the proposed SlaCP methodology was presented and a number of techniques were adopted to increase the tool's modularity and automation. This was done by implementing a set of intermediate files to represent the tool's output in a way that can be understood by engines reasoning about them. However, a number of issues restricted the implementation of all aspects of the architectural design in the implemented WslaCP tool. Functions cannot be applied on the expected values of the solver; rather, this has to be achieved on individual runs (replicas) obtained from a simulation rather than on results from an analytic model.

Möbius proved to be very powerful in obtaining the required simulation replicas but its complicated textual representation prevents the WslaCP tool from utilising Möbius during the first steps of the tool. The opposite is true for the SPNP

tool. Hence, due to time limitations, and in the light of the lack of a tool that could accomplish the WslaCP requirements from among the ones studied, WslaCP was implemented using the two previously mentioned modelling tools, SPNP and Möbius. All the previous information allows the question to be answered as follows: *"A software tool that automates the methodology can be designed and implemented."*

In the light of the analysis of the three aforementioned sub-questions, the following answer holds for the second main question: "**A theoretical mapping of an SLA to a stochastic model is applicable in a real scenario, and it can be implemented in a software tool that automates it.**"

In the light of answering the main research questions, the research hypothesis that *"the process of model-based SLA compliance prediction can be automated using an existing SLA document as the only input"*, as presented in Section 1.3, is not entirely valid because user interaction is still vital.

8.2.3 Overall Reflection

This section gives an overall opinion about the viability of the proposed methodology.

The methodology proposed in this thesis starts from an existing SLA contract and maps it on reward models. Purposely, well-established specification languages for defining the QoS metrics in an SLA, such as Performance Trees or the Continuous Stochastic Logic, were not used. The reason for this is that it would require the rewriting of an existing SLA in a new specification. Instead, the proposed methodology assumes a predefined SLA that the user already has and makes use of it.

The methodology is aimed at users who are non-specialists in the area of model-based evaluation and tries to help them as much as possible. However, although the methodology aims to help non-specialists to carry out SLA compliance prediction, it needs user interaction to perform some manual steps which have been shown to be non-trivial; these indeed require some modelling skills (such as the ability to complete a model creation and reward definitions). As a conclusion, the methodology can be considered to provide advantages, but it does not avoid the need for some expert involvement.

Finally, the main advantage of using the proposed SlaCP methodology and its implementation is to allow the SLA engineers or service providers to gain an early insight in the ability of the service to conform to a predefined SLA in a semi-automated fashion. This methodology can help them also in later stages of SLA negotiation and establishment to perform SLA management. This can aid in predicting the probability of breaching an SLA while the service is running.

8.3 Future Work

The work presented in this thesis can be enhanced and extended in terms of all of its three aspects: the methodology, the tool, and the evaluation. The possible future work is as follows:

1. The methodology: The areas that need to be enhanced regarding the theoretical basis of the methodology are as follows:

   (a) Regarding the SlaCP methodology, it would be useful to implement the general methodology presented in this thesis for different SLA languages. The methodology is aimed at SLAs with a constructive ontology for defining QoS metrics. An idea would be to extend it in a way that other SLAs, which define its QoSs without considering such an ontology, could also be used. For example, this could be accomplished by using a decomposition mechanism to transform the composite QoS metrics into measured metrics and the functions over them.

   (b) Regarding the WslaCP methodology, in this thesis, the intention was to use WSLA in a prediction context. Therefore, semantics were associated with its elements in order to make sure that they were obvious when mapping them. It would be useful to integrate the semantics with a WSLA contract or to develop a mechanism for agreeing on them. For the sake of practicality, these semantics could be written in a machine-readable format so they are accessible for the SLA parties to read.

2. The tool: The areas that need to be enhanced regarding the tool are as follows:

   (a) To integrate a WSDL mapping module in the tool.

   (b) To extend the tool so that it not only gives the probability of SLO compliance, but can also offer suggestions about areas of weakness that cause low SLA satisfaction, as well as possible solutions to this.

   (c) To allow the user to complete the model's creation graphically instead of achieving this using textual input.

   (d) To improve the usability of the WslaCP tool by improving the graphical user interface. This would be useful, especially when the user wants to change action rate values or the initial model state.

   (e) To find new estimation methods to produce prediction results more quickly, other than by storing and analysing the simulation replicas. This would reduce the time complexity and would make the solution more scalable.

3. The evaluation: The areas that need to be enhanced regarding the evaluation are as follows:

   (a) The proposed methodology could be connected to a monitoring framework to support the validation of the stochastic model against the real system; the proposed method could ensure the coherence of both the measured values and the model-derived ones.

   (b) Applying the tool to a running service. This would provide a comparison between the predicted SLA compliance values and the monitored ones.

Appendix A

SDES Schema, SDESSch

The SDESSch schema represents the *SLA-Model File*, which is the output of the *Metric Specification Engine* (MS Engine) in a unified machine-readable format following the notation used in the SDES formalism.

The root element of the SDESSch schema, as appears in Listing A.1, is the SDES of the type SDESType (line 9). This type consists of three main parts: SV, A, and RV (lines 3, 4, and 5). These describe the state variables, the actions, and the reward variables that are derived from the WSLA contract respectively.

Listing A.1: The three main elements in the SDESSch schema

```
1:<xsd:complexType name="SDESType">
2: <xsd:sequence>
3:  <xsd:element name="SV" type="sdes:SVType"  maxOccurs="unbounded"/>
4:  <xsd:element name="A"  type="sdes:AType"   maxOccurs="unbounded"/>
5:  <xsd:element name="RV" type="sdes:RVType" maxOccurs="unbounded"/>
6: </xsd:sequence>
7: <xsd:attribute name="name" type="xsd:string"/>
8:</xsd:complexType>

9:<xsd:element name="SDES" type="sdes:SDESType"/>
```

The name attribute (line 7) represents the name of the service that the WSLA document is defined for. SDES elements and types are defined in the name-space "sdes". The type of each of the main elements of the SDESSch schema is described in the following sections.

A.1 State Variables

The set of state variables, SV, that is derived from the mapping of the WSLA's service operation is defined using the SVType complex type. This type is presented

in Listing A.2.

Listing A.2: The definition of the state variable in the SDESSch schema

```
1:<xsd:complexType name="SVType">
2: <xsd:sequence>
3:  <xsd:element name="Name" type="xsd:string" minOccurs="1"
       maxOccurs="1"/>
4:  <xsd:element name="Value" type="xsd:double" minOccurs="0"
       maxOccurs="1"/>
5: </xsd:sequence>
6:</xsd:complexType>
```

The Name element in this listing (line 3) refers to the name of the state variable; this is equal to the name of the WSLA's service operation with the attached string ($\_s$). The Value element (line 4) represents the initial value of this state variable; this is set to 1 by default.

A.2 Actions

The set of actions, A, that is derived from mapping the WSLA's service operation is defined using the AType complex type; this is presented in Listing A.3.

Listing A.3: The definition of the action in the SDESSch schema

```
1:<xsd:complexType name="AType">
2: <xsd:sequence>
3:  <xsd:element name="Name" type="xsd:string" minOccurs="1"
       maxOccurs="1"/>
4:  <xsd:element name="Rate" type="xsd:double" minOccurs="0"
       maxOccurs="1"/>
5:  <xsd:element name="InputS" type="xsd:string" minOccurs="0"
       maxOccurs="1"/>
6:  <xsd:element name="OutputS" type="xsd:string" minOccurs="0"
       maxOccurs="1"/>
7: </xsd:sequence>
8:</xsd:complexType>
```

The Name element in this listing (line 3) refers to the name of the action; this is equal to the name of the WSLA's service operation with the attached string ($\_a$). The Rate element (line 4) represents the firing rate of this action which is set to 1 by default. The InputS and OutputS elements refer to the input and output state variables connected to this action.

A.3 Reward Variables

The set of reward variables, `RV`, that is derived from mapping WSLA's measurement directives is defined using the `RVType` complex type; this appears in Listing A.4.

Listing A.4: The definition of the reward variable in the `SDESSch` schema

```
 1:<xsd:complexType name="RVType">
 2: <xsd:sequence>
 3:  <xsd:element ref="sdes:rvRate" minOccurs="0" maxOccurs="1"/>
 4:  <xsd:element ref="sdes:rvImp" minOccurs="0" maxOccurs="1"/>
 5:  <xsd:element ref="sdes:rvInt" minOccurs="1" maxOccurs="1"/>
 6:  <xsd:element ref="sdes:hint" minOccurs="0" maxOccurs="1"/>
 7: </xsd:sequence>
 8:<xsd:attribute name="type" type="sdes:MesurementTypes"/>
 9:<xsd:attribute name="name" type="xsd:string"/>
10:</xsd:complexType>
```

This definition is compatible with the required outputs of mapping the measurement directives as reward variables (which is described in Section 5.2.2). These outputs are the reward variable type (rate or impulse), the time to solve it, and the hint to the user (the elements `rvRate`, `rvImp`, `rvInt`, and `hint` in lines 3, 4, 5, and 6 respectively). According to this definition, an element `RV` can contain, at most, one rate-based reward function; at most, one impulse-based reward function; only one reward variable evaluation interval and, at most, one hint. These elements and their types are described in detail in the forthcoming subsections. However, before this, the attributes of the `RVType` are described.

The first attribute, `type` (line 8), represents the type of the measurement directive, which can be one of six types (equivalent to the six types of the measurement directive in WSLA, where `Status` and `StatusRequest` are treated identically); these are specified in Listing A.5. The `type` attribute is needed to allow the tool to recognise how to deal with the template of the reward variables and what type of GUI is related to each of them.

Listing A.5: The type of the measurement directive in the `SDESSch` schema

```
<xsd:simpleType name="MesurementTypes">
 <xsd:restriction base="xsd:string">
  <xsd:enumeration value="Gauge"/>
  <xsd:enumeration value="Counter"/>
  <xsd:enumeration value="InvocationCount"/>
  <xsd:enumeration value="ResponseTime"/>
  <xsd:enumeration value="DownTime"/>
  <xsd:enumeration value="Status"/>
 </xsd:restriction>
</xsd:simpleType>
```

The second attribute, `name` (line 9 in Listing A.4), represents the name that is given to the reward variable. This name is generated automatically by assigning the name of the measurement directive to the name of the operation on which this measurement is defined separated by an underscore sign (_).

A.3.1 The Rate Reward Function

After defining the type of the reward variable, the reward function has to be specified. According to the mapping in Section 5.2.2, the measurement directives that are mapped as rate reward functions are: *Gauge*, *ResponseTime*, *Status* and *DownTime*. These rate reward functions, according to Section 5.2.2, can be in one of two types: the first specifies a state variable whose value is returned (for *Gauge*), while the second specifies a condition that should be satisfied (with state variables, relations and thresholds) along with the value it returns (for *Status*, *ResponseTime* and *DownTime*). Accordingly, the `rvRate` element is defined using the `RateType` complex type as appears in Listing A.6.

Listing A.6: The definition of the rate reward function in the `SDESSch` schema

```
 1:<xsd:element name="rvRate" type="sdes:RateType"/>

 2:<xsd:complexType name="RateType">
 3: <xsd:choice>
 4:  <xsd:sequence>
 5:   <xsd:element name="Return" type="xsd:string"/>
 6:  </xsd:sequence>
 7:  <xsd:sequence>
 8:   <xsd:element name="IF" type="sdes:IFType"/>
 9:   <xsd:element name="ElseIf" type="sdes:IFType" minOccurs="0"/>
10:   <xsd:element name="ElseReturn" type="xsd:double" minOccurs="0"/>
11:  </xsd:sequence>
12: </xsd:choice>
13:</xsd:complexType>

14:<xsd:complexType name="IFType">
15: <xsd:sequence>
16:  <xsd:element name="condition" type="sdes:LogicExpressionType"/>
17:  <xsd:element name="return" type="xsd:double"/>
18: </xsd:sequence>
19:</xsd:complexType>
```

For the first type of the rate reward function, the element `Return` (line 5) is used to specify the name of the state variable whose value should be returned. For the second type of the rate reward function, the condition is specified using the `IF`, `ElseIf`, and `ElseReturn` elements (lines 8, 9 and 10). All these elements

are of the type `IFType`; this specifies `condition` (line 16) and `return` (line 17) elements. The condition can be either simple or nested. This is reflected using a binary operator (And, Or) or a unary operator (Not). The binary and unary operators are specified using the complex type `BinaryOperatorType` and `UnaryOperatorType` respectively. These complex types are defined using another complex type, `LogicExpressionType`, as appears in Listing A.7.

Listing A.7: The definition of the condition in the `SDESSch` schema

```
 1:<xsd:complexType name="LogicExpressionType">
 2: <xsd:sequence>
 3:  <xsd:choice>
 4:   <xsd:sequence>
 5:    <xsd:element name="stv" type="xsd:string"/>
 6:    <xsd:element name="R" type="sdes:relationType"/>
 7:    <xsd:element name="V" type="xsd:double"/>
 8:   </xsd:sequence>
 9:   <xsd:element name="And" type="sdes:BinaryOperatorType"/>
10:   <xsd:element name="Or" type="sdes:BinaryOperatorType"/>
11:   <xsd:element name="Not" type="sdes:UnaryOperatorType"/>
12:  </xsd:choice>
13: </xsd:sequence>
14:</xsd:complexType>

15:<xsd:complexType name="BinaryOperatorType">
16: <xsd:sequence>
17:  <xsd:element name="condition" type="sdes:LogicExpressionType"
        minOccurs="2" maxOccurs="2"/>
18: </xsd:sequence>
19:</xsd:complexType>

20:<xsd:complexType name="UnaryOperatorType">
21: <xsd:sequence>
22:  <xsd:element name="condition" type="sdes:LogicExpressionType"
         minOccurs="1" maxOccurs="1"/>
23: </xsd:sequence>
24:</xsd:complexType>
```

In the `BinaryOperatorType` (line 15), two `condition`(s) are exactly specified (line 17), while for the `UnaryOperatorType` (line 20) one `condition` is only specified (line 22). Each condition of the aforementioned types is defined by the `LogicExpressionType` type (line 1). The simple condition of the `LogicExpressionType` can be defined by specifying a state variable `stv`, an arithmetic relation `R`, and a value `V` (lines 5, 6, and 7). However, the complex condition can be defined by specifying multiple simple ones that are joined using one of the logical operators `And, Or,` or `Not` (lines 9, 10, and 11). The arithmetic relation, `R`, is defined using the complex type `relationType` as presented in Listing A.8.

Listing A.8: The definition of the arithmetic relation in the SDESSch schema

```
<xsd:simpleType name="relationType">
 <xsd:restriction base="xsd:string">
  <xsd:enumeration value="Greater"/>
  <xsd:enumeration value="Less"/>
  <xsd:enumeration value="GreaterEqual"/>
  <xsd:enumeration value="LessEqual"/>
  <xsd:enumeration value="Equal"/>
 </xsd:restriction>
</xsd:simpleType>
```

A.3.2 The Impulse Reward Function

According to Section 5.2.2, the measurement directives that are mapped as impulse reward functions are: *Counter* and *InvocationCount*. For defining the impulse reward function, the action and the returned value should be determined. Accordingly, the `rvImp` element is defined using the `ImpulseType` complex type as appears in Listing A.9.

Listing A.9: The definition of the impulse reward functions in the SDESSch schema

```
1:<xsd:element name="rvImp" type="sdes:ImpulseType"/>

2:<xsd:complexType name="ImpulseType">
3:  <xsd:sequence>
4:   <xsd:element name="ac" type="xsd:string"/>
5:   <xsd:element name="return" type="xsd:double"/>
6:  </xsd:sequence>
7:</xsd:complexType>
```

The `ac` element (line 4) is used to specify the name of the action whose firing is considered, while the `return` element (line 5) specifies the value that should be returned when this action fires.

A.3.3 The Evaluation Interval

According to Section 5.2.3, the schedule determines the time at/during which the reward variable should be solved. This time is recognised as an evaluation interval which could be an instant or interval of time. Accordingly, the `rvInt` element is defined using the `rvIntervalType` complex type as appears in Listing A.10.

Listing A.10: The definition of the reward interval in the SDESSch schema

```
1:<xsd:element name="rvInt" type="sdes:rvIntervalType"/>

2:<xsd:complexType name="IntervalType">
```

```
 3:  <xsd:sequence>
 4:   <xsd:element name="S" type="xsd:double"/>
 5:   <xsd:element name="E" type="xsd:double"/>
 6:   <xsd:element name="I" type="xsd:double"/>
 7:  </xsd:sequence>
 8:  <xsd:attribute name="type" type="sdes:TimeType"/>
 9:</xsd:complexType>

10:<xsd:simpleType name="TimeType">
11: <xsd:restriction base="xsd:string">
12:  <xsd:enumeration value="Instant"/>
13:  <xsd:enumeration value="Interval"/>
14: </xsd:restriction>
15:</xsd:simpleType>
```

The `S`, `E` and `I` elements (lines 4, 5, and 6) are used to specify the start time, the end time, and the increment respectively. The `type` attribute (line 8) is defined using the `TimeType` complex type. It specifies whether the evaluation is done at an `Instant` of time (line 12) or an `Interval` of time (line 13).

A.3.4 The Reward Hint

The last element that is defined inside the `RV` element. This element, `hint`, gives an information about the state variable or the action that can be included in the reward function or in the model. Accordingly, the `hint` is defined using the `HintType` complex type as appears in Listing A.11.

Listing A.11: The hint definition in the `SDESSch` schema

```
1:<xsd:element name="hint" type="sdes:HintType"/>

2:<xsd:complexType name="HintType">
3:  <xsd:choice>
4:   <xsd:element name="svH" type="xsd:string"/>
5:   <xsd:element name="aH" type="xsd:string"/>
6:  </xsd:choice>
7:</xsd:complexType>
```

The `svH` and `aH` elements (lines 4 and 5) refer to the name of the state variable and the action accordingly.

It should be noted that when mapping the measurement directives in Section 5.2.2, the reward variable values were either instant or interval of time reward variables; i.e. they are not averaged ($rvAvg$ is always false). For this reason, there is no need to include a reference to $rvAvg$ in the `SDESSch` schema.

A.4 The Complete SDESSch Schema

The complete SDESSch Schema is presented in Listing A.12.

Listing A.12: The complete SDESSch Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:sdes="D:/
    writingup2012/sdes"  targetNamespace="D:/writingup2012/sdes"
    elementFormDefault="qualified">

  <xsd:complexType name="SDESType">
   <xsd:sequence>
    <xsd:element name="SV" type="sdes:SVType" maxOccurs="unbounded"/>
    <xsd:element name="A" type="sdes:AType" maxOccurs="unbounded"/>
    <xsd:element name="RV" type="sdes:RVType" maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attribute name="name" type="xsd:string"/>
  </xsd:complexType>

  <xsd:element name="SDES" type="sdes:SDESType"/>

  <xsd:complexType name="SVType">
   <xsd:sequence>
    <xsd:element name="Name" type="xsd:string" minOccurs="1"
        maxOccurs="1"/>
    <xsd:element name="Value" type="xsd:double" minOccurs="0"
         maxOccurs="1"/>
   </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="AType">
   <xsd:sequence>
    <xsd:element name="Name" type="xsd:string" minOccurs="1"
        maxOccurs="1"/>
    <xsd:element name="Rate" type="xsd:double" minOccurs="0"
        maxOccurs="1"/>
    <xsd:element name="InputS" type="xsd:string" minOccurs="0"
        maxOccurs="1"/>
    <xsd:element name="OutputS" type="xsd:string" minOccurs="0"
        maxOccurs="1"/>
   </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="RVType">
    <xsd:sequence>
    <xsd:element ref="sdes:rvRate" minOccurs="0" maxOccurs="1"/>
    <xsd:element ref="sdes:rvImp" minOccurs="0" maxOccurs="1"/>
    <xsd:element ref="sdes:rvInt" minOccurs="1" maxOccurs="1"/>
    <xsd:element ref="sdes:hint" minOccurs="0" maxOccurs="1"/>
   </xsd:sequence>
```

```
 <xsd:attribute name="type" type="sdes:MesurementTypes"/>
 <xsd:attribute name="name" type="xsd:string"/>
</xsd:complexType>

<xsd:element name="rvRate" type="sdes:RateType"/>

<xsd:complexType name="RateType">
 <xsd:choice>
  <xsd:sequence>
   <xsd:element name="IF" type="sdes:IFType"/>
   <xsd:element name="ElseIf" type="sdes:IFType" minOccurs="0"/>
   <xsd:element name="ElseReturn" type="xsd:double" minOccurs="0"/>
  </xsd:sequence>
  <xsd:sequence>
   <xsd:element name="Return" type="xsd:string"/>
  </xsd:sequence>
 </xsd:choice>
</xsd:complexType>

<xsd:complexType name="LogicExpressionType">
 <xsd:sequence>
  <xsd:choice>
   <xsd:sequence>
    <xsd:element name="stv" type="xsd:string"/>
    <xsd:element name="R" type="sdes:relationType"/>
    <xsd:element name="V" type="xsd:double"/>
   </xsd:sequence>
   <xsd:element name="And" type="sdes:BinaryOperatorType"/>
   <xsd:element name="Or" type="sdes:BinaryOperatorType"/>
   <xsd:element name="Not" type="sdes:UnaryOperatorType"/>
  </xsd:choice>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="BinaryOperatorType">
 <xsd:sequence>
  <xsd:element name="condition" type="sdes:LogicExpressionType"
      minOccurs="2" maxOccurs="2"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="UnaryOperatorType">
 <xsd:sequence>
  <xsd:element name="condition" type="sdes:LogicExpressionType"
      minOccurs="1" maxOccurs="1"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:element name="rvImp" type="sdes:ImpulseType"/>

<xsd:complexType name="ImpulseType">
```

```
 <xsd:sequence>
  <xsd:element name="ac" type="xsd:string"/>
  <xsd:element name="return" type="xsd:double"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="IFType">
 <xsd:sequence>
  <xsd:element name="condition" type="sdes:LogicExpressionType"/>
  <xsd:element name="return" type="xsd:double"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:element name="hint" type="sdes:HintType"/>

<xsd:complexType name="HintType">
 <xsd:choice>
  <xsd:element name="svH" type="xsd:string"/>
  <xsd:element name="aH" type="xsd:string"/>
 </xsd:choice>
</xsd:complexType>

<xsd:element name="rvInt" type="sdes:IntervalType"/>

<xsd:complexType name="IntervalType">
 <xsd:sequence>
  <xsd:element name="S" type="xsd:double"/>
  <xsd:element name="E" type="xsd:double"/>
  <xsd:element name="I" type="xsd:double"/>
 </xsd:sequence>
 <xsd:attribute name="type" type="sdes:TimeType"/>
</xsd:complexType>

<xsd:simpleType name="TimeType">
 <xsd:restriction base="xsd:string">
  <xsd:enumeration value="Instant"/>
  <xsd:enumeration value="Interval"/>
 </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="MesurementTypes">
 <xsd:restriction base="xsd:string">
  <xsd:enumeration value="Gauge"/>
  <xsd:enumeration value="Counter"/>
  <xsd:enumeration value="InvocationCount"/>
  <xsd:enumeration value="ResponseTime"/>
  <xsd:enumeration value="DownTime"/>
  <xsd:enumeration value="Status"/>
 </xsd:restriction>
</xsd:simpleType>
```

```
  <xsd:simpleType name="relationType">
   <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Greater"/>
    <xsd:enumeration value="Less"/>
    <xsd:enumeration value="GreaterEqual"/>
    <xsd:enumeration value="LessEqual"/>
    <xsd:enumeration value="Equal"/>
   </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Appendix B

Implementation of the Complex WSLA Functions in Matlab

In this appendix, the implementation of the complex WSLA functions is provided. These functions are the *TSSelect*, *ValueOccurs*, *PercentageGreaterThanThreshold*, *PercentageLessThanThreshold*, *NumberGreaterThanThreshold*, *NumberLessThanThreshold*, *Span* and *ValueOccurs*. The implementation of these functions resides inside the *Functions File* of the *Metric Specification Engine*. The implementation of these functions are described in the what follows.

1- TSSelect: As appears in Table B.1, this function takes the time series function's output, represented as the array *TSC[]*, and returns an index x from it.

Table B.1: TSSelect function implementation in Matlab

```
function[xCount]= TSS(TSC[],x)
  xCount= TSC(x);
  return(xCount);
end
```

2- ValueOccurs: As appears in Table B.2, this function takes the time series function's output, represented as the array *TSC[]*, and returns, at each index i of this series, the number of times a value x occurs in this series.

3- PercentageGreaterThanThreshold: As appears in Table B.3, this function takes the time series function's output, *TSC[]*, and returns the percentage of elements whose value are greater than a value x, and this at each index i of this series.

Table B.2: ValueOccurs function implementation in Matlab

```
function[xCount[]]= VO(TSC[],x)
 xCount=0;
 i=1;
 for i=1:1:length(TSC[])
   if TSC(i)= x
    xCount=xCount+1;
    xCount(i)=xCount;
   else
    xCount(i)=xCount;
   end
 end
return(xCount[])
end
```

Table B.3: PercentageGreaterThanThreshold function implementation in Matlab

```
function[xCount[]]= PGTT(TSC[],x)
 xCount=0;
 i=1;
 for i=1:1:length(TSC[])
   if TSC(i)> x
    xCount=xCount+1;
    xCount(i)=xCount/i;
   else
    xCount(i)=xCount/i;
   end
 end
return(xCount[])
end
```

Table B.4: PercentageLessThanThreshold function implementation in Matlab

```
function[xCount[]]= PLTT(TSC[],x)
 xCount=0;
 i=1;
 for i=1:1:length(TSC[])
   if TSC(i)< x
    xCount=xCount+1;
    xCount(i)=xCount/i;
   else
    xCount(i)=xCount/i;
   end
 end
return(xCount[])
end
```

4- PercentageLessThanThreshold: As appears in Table B.4, this function takes the time series function's output, *TSC[]*, and returns the percentage of elements whose value are less than a value x, and this at each index i of this series.

5- NumberGreaterThanThreshold: As appears in Table B.5, this function takes the time series function's output, *TSC[]*, and returns the number of elements whose value are greater than a value x, and this at each index i of this series.

Table B.5: NumberGreaterThanThreshold function implementation in Matlab

```
function[xCount[]]= NGTT(TSC[],x)
 xCount=0;
 i=1;
 for i=1:1:length(TSC[])
   if TSC(i)> x
    xCount=xCount+1;
    xCount(i)=xCount;
   else
    xCount(i)=xCount;
   end
 end
return(xCount[])
end
```

6- NumberLessThanThreshold: As appears in Table B.6, this function takes the time series function's output, *TSC[]*, and returns the number of elements whose value are less than a value x, and this at each index i of this series.

Table B.6: NumberLessThanThreshold function implementation in Matlab

```
function[xCount[]]= NLTT(TSC[],x)
 xCount=0;
 i=1;
 for i=1:1:length(TSC[])
   if TSC(i)< x
    xCount=xCount+1;
    xCount(i)=xCount;
   else
    xCount(i)=xCount;
   end
 end
return(xCount[])
end
```

7- Span: As appears in Table B.7, this function takes the time series function's output, *TSC[]*, and returns the length of the consecutive occurrence of a value x inside this series, and this at each index i of this series.

Table B.7: Span function implementation in Matlab

```
function[xCount[]]= Span(TSC[],x)
 xCount=0;
 i=0;
 for i=length(TSC[]):-1:1
   if TSC(i)= x
    xCount=xCount+1;
    xCount(i)=xCount;
   else
    xCount=0;
    xCount(i)=xCount;
   end
 end
return(xCount[])
end
```

8- RateOfChange: As appears in Table B.8, this function takes the time series function's output, *TSC[]*, and returns the rate at which its items are changed (the difference between the series value at the current index and the index before, divided by the difference between the schedule value at the current index and the index before), and this at each index i of this series.

Table B.8: RateOfChange function implementation in Matlab

```
function[xCount[]]= RoC(TSC[],sch[],x)
 xCount=0;
 i=1;
 for i=length(TSC[]):-1:1
    xCount(i)=(TSC(i)-TSC(i-1))/(sch(i)-sch(i-1));
 end
return(xCount[])
end
```

Appendix C

WSLA Contract of a Stock Quote Service

The complete WSLA contract of the stock quote service used in Chapter 7 is presented in this appendix in Listing C.1.

Listing C.1: WSLA contract of stock quote service

```
<SLA xmlns="http://www.ibm.com/wsla"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.ibm.com/wsla WSLA.xsd"
    name="DemoSLA123" >

<ServiceDefinition  name="DemoService">

 <Schedule name="businessdayschedule">
  <Period>
    <Start>2001-11-30T14:00:00.000-05:00</Start>
    <End>2001-12-31T14:00:00.000-05:00</End>
  </Period>
  <Interval>
    <Minutes>1440</Minutes>
  </Interval>
 </Schedule>

 <Schedule name="businessdayschedule">
     <Period>
        <Start>2001-11-30T14:00:00.000-05:00</Start>
        <End>2001-12-31T14:00:00.000-05:00</End>
     </Period>
     <Interval>
        <Minutes>1440</Minutes>
     </Interval>
 </Schedule>

 <Schedule name="5minuteschedule">
     <Period>
        <Start>2001-11-30T14:00:00.000-05:00</Start>
        <End>2001-12-31T14:00:00.000-05:00</End>
```

```
    </Period>
    <Interval>
      <Minutes>5</Minutes>
    </Interval>
</Schedule>

<Schedule name="ResponseSchedule">
    <Period>
      <Start>2001-11-30T14:00:00.000-05:00</Start>
      <End>2001-12-31T14:00:00.000-05:00</End>
    </Period>
    <Interval>
      <Seconds>15</Seconds>
    </Interval>
</Schedule>

<Operation name="GetQuote" xsi:type="WSDLSOAPOperationDescriptionType">

 <SLAParameter name="Availability_CurrentDownTime" type="long"
              unit="minutes">
   <Metric>CurrentDownTime</Metric>
 </SLAParameter>

<SLAParameter name="OverloadPercentage"  type="float"
                   unit="Percentage">
       <Metric>OverloadPercentageMetric</Metric>
  </SLAParameter>

  <SLAParameter name="TransactionRate"  type="float"
                   unit="transactions/hour">
       <Metric>Transactions</Metric>
 </SLAParameter>


 <Metric name="CurrentDownTime" type="long" unit="minutes">
   <Function xsi:type="Span" resultType="double">
    <Metric>StatusTimeSeries</Metric>
    <Value>
       <LongScalar>0</LongScalar>
    </Value>
  </Function>
 </Metric>

<Metric name="StatusTimeSeries" type="TS" unit="">
 <Function xsi:type="TSConstructor" resultType="TS">
   <Schedule>availabilityschedule</Schedule>
   <Metric>MeasuredStatus</Metric>
   <Window>1440</Window>
 </Function>
</Metric>

<Metric name="MeasuredStatus" type="integer" unit="">
 <MeasurementDirective xsi:type="StatusRequest"
                                      resultType="integer">
   <RequestURI>http://ymeasurement.com/StatusRequest/GetQuote
  </RequestURI>
```

```
 </MeasurementDirective>
</Metric>

 <Metric name="OverloadPercentageMetric" type="float" unit="Percentage">
    <Function xsi:type="PercentageGreaterThanThreshold" resultType="float">
        <Schedule>businessdayschedule</Schedule>
        <Metric>UtilizationTimeSeries</Metric>
        <Value>
            <LongScalar>0.8</LongScalar>
        </Value>
    </Function>
 </Metric>

 <Metric name="UtilizationTimeSeries" type="TS" unit="">
   <Function xsi:type="TSConstructor" resultType="float">
        <Schedule>5minuteschedule</Schedule>
        <Metric>ProbedUtilization</Metric>
        <Window>12</Window>
    </Function>
 </Metric>

 <Metric name="ProbedUtilization" type="float" unit="">
    <MeasurementDirective xsi:type="Gauge" resultType="float">
        <RequestURL>http://acme.com/SystemUtil</RequestURL>
    </MeasurementDirective>
 </Metric>

 <Metric name="Transactions" type="long" unit="transactions">
    <Function xsi:type="Minus" resultType="double">
       <Operand>
           <Function xsi:type="TSSelect" resultType="long">
                <Operand>
                    <Metric>SumTransactionTimeSeries</Metric>
               </Operand>
              <Element>0</Element>
           </Function>
       </Operand>
       <Operand>
           <Function xsi:type="TSSelect" resultType="long">
               <Operand>
                <Metric>SumTransactionTimeSeries</Metric>
              </Operand>
                <Element>-1</Element>
            </Function>
         </Operand>
    </Function>
 </Metric>

 <Metric name="SumTransactionTimeSeries" type="TS" unit="transactions">
     <Function xsi:type="TSConstructor" resultType="TS">
        <Schedule>hourlyschedule</Schedule>
        <Metric>SumTransactions</Metric>
        <Window>2</Window>
    </Function>
 </Metric>
```

```
   <Metric name="SumTransactions" type="long" unit="tansactions">
     <MeasurementDirective xsi:type="InvocationCount" resultType="long"/>
   </Metric>

  <WSDLFile>DemoService.wsdl</WSDLFile>
  <SOAPBindingName>SOAPNotificationBinding</SOAPBindingName>
  <SOAPOperationName>getQuote</SOAPOperationName>
 </Operation>

 <Operation name="PrintQuote" xsi:type="WSDLSOAPOperationDescriptionType">

  <SLAParameter name="MaxResponseTime" type="double"
                unit="seconds">
    <Metric>MaximumResponseTime</Metric>
  </SLAParameter>

  <Metric name="MaximumResponseTime " type="long" unit="minutes">
    <Function xsi:type="Max" resultType="double">
     <Metric>ResponseTimeSeries</Metric>
    </Function>
  </Metric>

 <Metric name="ResponseTimeSeries" type="TS" unit="seconds">
  <Function xsi:type="TSConstructor" resultType="TS">
    <Schedule>ResponseSchedule</Schedule>
    <Metric>ResponseTimeMetric</Metric>
    <Window>4</Window>
  </Function>
 </Metric>

 <Metric name="ResponseTimeMetric" type="double" unit="seconds">
  <MeasurementDirective xsi:type="ResponseTime"
                          resultType="double">
    <RequestURI>http://ymeasurement.com/ResponseTime/PrintQuote
   </RequestURI>
  </MeasurementDirective>
 </Metric>

  <WSDLFile>PrintService.wsdl</WSDLFile>
  <SOAPBindingName>SOAPNotificationBinding</SOAPBindingName>
  <SOAPOperationName>PrintQuote</SOAPOperationName>
 </Operation>

</ServiceDefinition>

<Obligations>
<ServiceLevelObjective name="ContinuousDowntimeSLO">
 <Obliged>ACMEProvider</Obliged>
  <Validity>
   <Start>2001-11-30T14:00:00.000-05:00</Start>
   <End>2001-12-31T14:00:00.000-05:00</End>
  </Validity>
 <Expression>
   <Predicate xsi:type="Less">
      <SLAParameter>Availability_CurrentDownTime</SLAParameter>
      <Value>10</Value>
```

```
   </Predicate>
 </Expression>
 </ServiceLevelObjective>

 <ServiceLevelObjective name="ConditionalSLOForTransactionRate">
     <Validity>
      <Start>2001-11-30T14:00:00.000-05:00</Start>
      <End>2001-12-31T14:00:00.000-05:00</End>
     </Validity>
     <Expression>
       <Implies>
         <Expression>
           <Predicate xsi:type="Less">
             <SLAParameter>OverloadPercentage</SLAParameter>
             <Value>0.3</Value>
          </Predicate>
         </Expression>
          <Expression>
           <Predicate xsi:type="Greater">
             <SLAParameter>TransactionRate</SLAParameter>
              <Value>1000</Value>
            </Predicate>
          </Expression>
      </Implies>
    </Expression>
   </ServiceLevelObjective>

<ServiceLevelObjective name="PrintingResponseTime">
     <Validity>
   <Start>2001-11-30T14:00:00.000-05:00</Start>
   <End>2001-12-31T14:00:00.000-05:00</End>
     </Validity>
     <Expression>
         <Predicate xsi:type="Less">
         <SLAParameter>MaxResponseTime</SLAParameter>
         <Value>15</Value>
       </Predicate>
     </Expression>
 </ServiceLevelObjective>

  </Obligations>
</SLA>
```

References

[1] Alain Andrieux, Karl Czajkowski, Asit Dan, Kate Keahey, Heiko Ludwig, Toshiyuki Nakata, Jim Pruyne, John Rofrano, Steve Tuecke, and Ming Xu. *Web Services Agreement Specification (WS-Agreement).* Open Grid Forum, version 2005/09 edition. xi, 2, 21, 23

[2] Katerina Goseva Popstojanova and Kishor Trivedi. Stochastic modeling formalisms for dependability, performance and performability. In *Performance Evaluation - Origins and Directions, Lecture Notes in Computer Science*, pages 403–422. Springer Verlag, 2000. 1, 2, 3, 32, 33, 44

[3] Michael P. Papazoglou. Service-oriented computing: concepts, characteristics and directions. In *WISE 2003. Proceedings of the Fourth International Conference on Web Information Systems Engineering, 2003*, pages 3 – 12, December 2003. 1, 16

[4] Ian Foster and Carl Kesselman. Grid resource management. chapter The Grid in a nutshell, pages 3–13. Kluwer Academic Publishers, 2004. 1

[5] Michael P. Papazoglou. *Web Services: Principles and Technology.* Pearson Education Limited, 2008. 1, 18, 19

[6] Peter Mell and Timothy Grance. The NIST definition of cloud computing, September 2011. 1

[7] J. W. Ross and G. Westerman. Preparing for utility computing: The role of IT architecture and relationship management. *IBM Systems Journal*, 43(1):5 –19, 2004. 1

[8] Michael P. Papazoglou. Service-oriented computing: A research roadmap. *International Journal of Cooperative Information Systems*, 17(02):223+, 2008. 1

[9] Li jie Jin, Vijay Machiraju, and Akhil Sahai. Analysis on service level agreement of web services. Technical report, HP Laboratories, 2002. 1, 2, 17

[10] Heiko Ludwig. Web services QoS: external SLAs and internal policies or: how do we deliver what we promise? In *Proceedings of the Fourth International Conference on Web Information Systems Engineering Workshops (WISEW03)*, pages 115–120, 2003. 1, 21, 73

[11] Daniel A. Menascè. QoS issues in web services. *Internet Computing, IEEE*, 6(6):72 – 75, nov/dec 2002. 1

[12] Akhil Sahai, Vijay Machiraju, Mehmet Sayal, Aad van Moorsel, and Fabio Casati. Automated SLA monitoring for web services. In M. Feridum, P. Kropf, and G. Babin, editors, *Management Technologies for E-Commerce and E-Business Applications. 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, volume 2506 of *Lecture Notes in Computer Science*, pages 28–41. Springer Verlag, 2002. 1, 2, 28, 29

[13] Linlin Wu and Rajkumar Buyya. Service level agreement (SLA) in utility computing systems, 2010. 2, 26

[14] Carlos Molina-Jimenez, James Pruyne, and Aad van Moorsel. The role of agreements in IT management software. In R. de Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems III*, volume 3549 of *Lecture Notes in Computer Science*, pages 36–58. Springer Verlag, 2005. 2

[15] Rouaa Yassin Kassab and Aad van Moorsel. Formal mapping of WSLA contracts on stochastic models. In *8th European Performance Engineering Workshop - EPEW 2011*, volume 6977 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2011. 2, 14, 82, 174, 176

[16] Heiko Ludwig, Alexander Keller, Asit Dan, and Richard King. A service level agreement language for dynamic electronic services. *Proceedings Fourth IEEE International Workshop on Advanced Issues of ECommerce and WebBased Information Systems WECWIS 2002*, 59(Wecwis):25–32, 2003. 2, 89

[17] Heiko Ludwig, Alexander Keller, Asit Dan, Richard P. King, and Richard Franck. Web service level agreement (WSLA) language specification, 2003. IBM Corporation. 2, 9, 20, 21, 22, 23, 68, 73, 74, 77, 81, 85, 87, 88, 89, 91, 92, 93, 94, 104, 107, 109, 176, 178, 179

[18] D. Davide Lamanna, James Skene, and Wolfgang Emmerich. SLAng: A language for defining service level agreements. In *The Ninth IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 100–106, 2003. 2, 21, 25

[19] D. Snelling, M. Fisher, and A. Basermann. An introduction to the NextGRID vision and achievements v1.0, 2008. Fujitsu Labs Europe, The University of Edinburgh and Members of the NextGRID Consortium. 2

[20] Philipp Masche, Paul Mckee, and Bryce Mitchell. The increasing role of service level agreements in B2B systems. In *WEBIST (2), Proceedings of the Second International Conference on Web Information Systems and Technologies*, pages 123–126, 2006. 2, 21

[21] Adrian Paschke and Elisabeth Schnappinger-Gerull. A categorization scheme for SLA metrics. In *Service Oriented Electronic Commerce*, pages 25–40, 2006. 2, 18, 19, 20, 74

[22] Claus Rautenstrauch and André Scholz. Performance engineering on the basis of performance service levels. In *Performance Engineering, State of the Art and Current Trends*, pages 68–77. Springer-Verlag, 2001. 2, 27

[23] Philipp Leitner, Branimir Wetzstein, Dimka Karastoyanova, Waldemar Hummer, Schahram Dustdar, and Frank Leymann. Preventing SLA violations in service compositions using aspect-based fragment substitution. In *ICSOC'10*, pages 365–380, 2010. 2

[24] Philipp Leitner, Branimir Wetzstein, Florian Rosenberg, Anton Michlmayr, Schahram Dustdar, and Frank Leymann. Runtime prediction of service level agreement violations for composite services. In *Proceedings of the 2009 international conference on Service-oriented computing*, ICSOC/ServiceWave'09, pages 176–186, Berlin, Heidelberg, 2009. Springer-Verlag. 3, 27, 29, 30, 33

[25] Nicholas J. Dingle, William J. Knottenbelt, and Lei Wang. Service level agreement specification, compliance prediction and monitoring with performance trees. In *22nd Annual European Simulation and Modelling Conference (ESM'08)*, pages 137–144, September 2008. 3, 5, 30, 56

[26] Marcelo Teixeira, Ricardo Massa, Cesar Oliveira, and Paulo Maciel. Planning service agreements in SOA-based systems through stochastic models. In *Pro-*

ceedings of the 2011 ACM Symposium on Applied Computing, SAC '11, pages 1576–1581, New York, NY, USA, 2011. ACM. 3, 33, 34, 65

[27] L.J.N. Franken and B.R. Haverkort. The performability manager. *Network, IEEE*, 8(1):24 –32, jan/feb 1994. 4

[28] Tamas Suto, Jeremy T.Bradley, and William J.Knottenbelt. Performance trees: A new approach to quantitative performance specification. In *in Proc. 14th IEEE/ACM Intl. Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS 2006*, pages 303–313, 2006. 4, 38, 45, 46

[29] Armin Zimmermann. *Stochastic Discrete Event Systems: Modeling, Evaluation, Applications*. Springer-Verlag New York, Inc., 2007. 9, 38, 39, 68, 106

[30] M. Ajmone Marsan. Stochastic Petri Nets: An elementary introduction. In *In Advances in Petri Nets*, pages 1–29. Springer, 1989. 9, 36, 37, 39, 40, 70

[31] Gianfranco Ciardo, Jogesh K. Muppala, and Kishor S. Trivedi. SPNP: Stochastic Petri Net Package. In *PNPM89. Proceedings of the Third International Workshop On Petri Nets and Performance Models, 1989, Kyoto, Japan*, pages 142–151. IEEE Computer Society Press, 1990. 9, 42, 43, 52

[32] William H. Sanders. *Möbius User Manual, Version 2.3.1*. University of Illinois, May 2010. 9, 47, 48, 49, 132, 145

[33] Tobias Unger, Frank Leymann, Stephanie Mauchart, and Thorsten Scheibler. Aggregation of service level agreements in the context of business processes. In *2008 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 43–52. IEEE, September 2008. 10, 84

[34] Rouaa Yassin Kassab and Aad van Moorsel. Mapping WSLA on reward constructs in Möbius. In *24th UK Performance Engineering Workshop*, pages 137–147, 2008. 13, 22, 23, 124

[35] Simon Edward Parkin, Rouaa Yassin Kassab, and Aad P. A. van Moorsel. The impact of unavailability on the effectiveness of enterprise information security technologies. In *ISAS'08*, pages 43–58, 2008. 14

[36] M.N. Huhns and M.P. Singh. Service-oriented computing: key concepts and principles. *Internet Computing, IEEE*, 9(1):75 – 81, jan-feb 2005. 16

[37] Heather Kreger. Web Services Conceptual Architecture (WSCA 1.0), 2001. 16, 17

[38] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the web services web: An introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6:86–93, March 2002. 17

[39] Zhen Liu, Mark S. Squillante, and Joel L. Wolf. On maximizing service-level-agreement profits. In *EC '01: Proceedings of the 3rd ACM conference on Electronic Commerce*, pages 213–223, New York, NY, USA, 2001. ACM Press. 17

[40] Emmanuel Marilly, Olivier Martinot, Stéphane Betgé-Brezetz, and Gérard Delégue. Requirements for service level agreement management, 2002. IP Operations and Management, 2002 IEEE Workshop. 17

[41] K. Fakhfakh, T. Chaari, S. Tazi, K. Drira, and M. Jmaiel. Semantic enabled framework for SLA monitoring. *International Journal on Advances in Software*, 2(1):36–34, 2009. 17, 100

[42] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. An approach for QoS-aware service composition based on genetic algorithms. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, GECCO '05, pages 1069–1075, 2005. 18

[43] Justin O'Sullivan, David Edmond, and Arthur H. M. ter Hofstede. Formal description of non-functional service properties, 2005. 18

[44] Kyriakos Kritikos and Dimitris Plexousakis. Requirements for QoS-based web service description and discovery. *IEEE Transactions on Services Computing*, 2:320–337, 2009. 18, 77, 90, 100

[45] Andrew N. Hiles. Service level agreements: Panacea or pain? *The TQM Magazine*, 6(2):14–16, 1994. 18, 19, 74

[46] M. Sathya, M. Swarnamugi, P. Dhavachelvan, and G. Sureshkumar. Evaluation of QoS based web- service selection techniques for service composition. *International Journal of Software Engineering (IJSE)*, 1:7390, 2010. 18

[47] H. J. Lee, M. S. Kim, J. W. Hong, and G. H. Lee. QoS parameters to network performance metrics mapping for SLA monitoring. 2002. 19, 34

[48] Katinka Wolter and Aad van Moorsel. The relationship between quality of service and business metrics: Monitoring, notification and optimization. Technical Report HPL-2001-96, HP Laboratories Palo Alto, April 2001. 19

[49] Christian N. Madu and Assumpta A. Madu. Dimensions of e-quality. *International Journal of Quality & Reliability Management*, 19(3):246–258, 2002. 19

[50] Anbazhagan Mani and Arun Nagarajan. Understanding quality of service for web services, January 2002. IBM DeveloperWorks. 19

[51] Kuyoro Shade O., Awodele O., Akinde Ronke O., and Okolie Samuel O. quality of service (Qos) issues in web service. *IJCSNS International Journal of Computer Science and Network Security*, 12(1):94–97, Janyary 2012. 19

[52] Eyhab Al-Masri and Qusay H. Mahmoud. Discovering the best web service: A neural network-based solution. In *Systems, Man and Cybernetics, 2009. SMC 2009. IEEE International Conference on*, pages 4250 –4255, oct. 2009. 19

[53] Aad van Moorsel. Metrics for the internet age: Quality of experience and quality of business. Technical report, 5th Performability Workshop, 2001. 20

[54] Alexander Keller and Heiko Ludwig. The WSLA framework: Specifying and monitoring service level agreements for web services. *Network and Systems Management*, 11(1):57–81, March 2003. 21, 22, 29, 89

[55] James Skene, D. Davide Lamanna, and Wolfgang Emmerich. Precise service level agreements. In *In: Proc. of 26th Intl. Conference on Software Engineering (ICSE)*, pages 179–188. IEEE Press, 2004. 23, 25

[56] Philip Bianco, Grace A. Lewis, and Paulo Merson. Service level agreements in service-oriented architecture environments. Technical Report CMU/SEI-2008-TN-021, Software Engineering Institute, September 2008. 24

[57] Paul Käranke and Stefan Kirn. Service level agreements: An evaluation from a business application perspective, 1988. 24

[58] 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. ACM, 2006. 24, 202

[59] Akhil Sahai, Anna Durante, and Vijay Machiraju. Towards automated SLA management for web services. Technical Report HPL-2001-310 (R.1), HP Laboratories Palo Alt, 2002. 26

[60] M. J. Buco, R. N. Chang, L. Z. Luan, C. Ward, J. L. Wolf, and P. S. Yu. Utility computing SLA management based upon business objectives. *IBM Syst. J.*, 43:159–178, January 2004. 27

[61] Christoph Rathfelder, Benjamin Klatt, Franz Brosch, and Samuel Kounev. Performance modeling for quality of service prediction in service-oriented systems. In Stephan Reiff-Marganiec and Marcel Tilly, editors, *Handbook of Research on Service-Oriented Systems and Non-Functional Properties: Future Directions*, pages 258–79. Hershey: IGI Global, 2012. 27

[62] Claudio Bartolini, Abdel Boulmakou, Athena Christodoulou, Andrew Farrell, Mathias Sall, and David Trastour. Management by contract: It management driven by business objectives. In *in Proceedings of the 11th Workshop of the HP OpenView University Association (HPOVUA 2004*, 2004. 27

[63] C. Bartolini, M. Salle, and D. Trastour. It service management driven by business objectives an application to incident management. In *Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP*, pages 45 –55, april 2006. 27

[64] Jacques Sauv, Filipe Marques, Anto Moura, Marcus Sampaio, Joo Jornada, and Eduardo Radziuk. SLA design from a business perspective. In *In Proceedings of DSOM 2005*. Springer, 2005. 28

[65] R. Nou and J. Torres. Heterogeneous QoS resource manager with prediction. In *Autonomic and Autonomous Systems, 2009. ICAS '09. Fifth International Conference on*, pages 69 –74, april 2009. 28

[66] Katja Gilly, Nigel Thomas, Carlos Juiz, and Ramon Puigjaner. Scalable QoS content-aware load balancing algorithm for a web switch based on classical policies. *Advanced Information Networking and Applications, International Conference on*, pages 934–941, 2008. 28

[67] Davide Lorenzoli and George Spanoudakis. Runtime prediction. In Philipp Wieder, Joe M. Butler, Wolfgang Theilmann, and Ramin Yahyapour, editors, *Service Level Agreements for Cloud Computing*, pages 139–152. Springer New York, 2011. 28

[68] Davide Lorenzoli and George Spanoudakis. EVEREST+: run-time SLA violations prediction. In *Proceedings of the 5th International Workshop on Middleware for Service Oriented Computing*, MW4SOC '10, pages 13–18, New York, NY, USA, 2010. ACM. 28

[69] Ernst Oberortner, Stefan Sobernig, Uwe Zdun, and Schahram Dustdar. Monitoring of performance-related QoS properties in service-oriented systems: A pattern-based architectural decision model. In *Proceedings of the 16th European Conference on Pattern Languages of Programs (EuroPLoP)*, Irsee, Germany, July 2011. 28, 29

[70] Florian Rosenberg, Christian Platzer, and Schahram Dustdar. Bootstrapping performance and dependability attributes ofweb services. In *Proceedings of the IEEE International Conference on Web Services*, ICWS '06, pages 205–212, Washington, DC, USA, 2006. IEEE Computer Society. 28

[71] 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. 28, 29

[72] 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. 29

[73] Chris Smith. *Uncertainty In Service Provisioning Relationships*. PhD thesis, Newcastle University. 30

[74] P. Leitner, A. Michlmayr, F. Rosenberg, and S. Dustdar. Monitoring, prediction and prevention of SLA violations in composite services. In *Web Services (ICWS), 2010 IEEE International Conference on*, pages 369 –376, july 2010. 30

[75] Ashok Argent-katwala and Jeremy T. Bradley. Functional performance specification with stochastic probes. In *Formal Methods and Stochastic Models for Performance Evaluation: Third European Performance Engineering Workshop (EPEW 2006). Number 4054 in LNCS, Springer-Verlag*, pages 31–46, 2006. 30, 31

[76] Richard A. Hayden, Jeremy T. Bradley, and Allan Clark. Performance specification and evaluation with unified stochastic probes and fluid analysis. *IEEE Transactions on Software Engineering*, 39(1):97–118, 2013. 30

[77] Samuel Kounev, Fabian Brosig, Nikolaus Huber, and Ralf Reussner. Towards self-aware performance and resource management in modern service-oriented systems. In *Services Computing (SCC), 2010 IEEE International Conference on*, pages 621 –624, july 2010. 32

[78] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: a survey. *Software Engineering, IEEE Transactions on*, 30(5):295 – 310, may 2004. 32, 33, 35, 224

[79] Daniel A. Menascé, Daniel Barbará, and Ronald Dodge. Preserving QoS of e-commerce sites through self-tuning: a performance model approach. In *Proceedings of the 3rd ACM conference on Electronic Commerce*, EC '01, pages 224–234, New York, NY, USA, 2001. ACM. 33

[80] Marcelo Teixeira and Pablo Sabadin Chaves. Planning databases service level agreements through stochastic Petri Nets. *JIDM*, 2(3):369–384, 2011. 34

[81] Giovanni Pacifici, Mike Spreitzer, Asser Tantawi, and Alaa Youssef. Performance management for cluster based web services. Technical report, IEEE Journal on Selected Areas in Communications, Volume 23, Issue, 2003. 34

[82] Markus Debusmann, Kurt Geihs, and Reinhold Kroeger. Unifying service level management using an MDA-based approach. In Raouf Boutaba and Seong B. Kim, editors, *Proceedings of the 9-th International IFIP/IEEE Network Operations and Management Symposium (NOMS 2004)*, pages 801–814. IEEE, 2004. 34

[83] M. Tian, A. Gramm, T. Naumowicz, H. Ritter, and J.S. Freie. A concept for QoS integration in web services. In *Web Information Systems Engineering Workshops, 2003. Proceedings. Fourth International Conference on*, pages 149 – 155, dec. 2003. 34

[84] Dorin B. Petriu and Murray Woodside. A metamodel for generating performance models from UML designs. In *Lecture Notes in Computer Science: The Unified Modelling Language: Modelling Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004 / Thomas*

Baar, Alfred Strohmeier, Ana Moreira, et al. (Eds.), volume 3273, pages 41–53. Springer-Verlag, 2004. 35

[85] Object Management Group. Uml profile for schedulability, performance, and time specification, 2002. OMG Adopted Specification ptc/02-03-02. 35

[86] Ihab Sbeity, Leonardo Brenner, and Mohamed Dbouk. Generating a performance stochastic model from UML specifications. *IJCSI International Journal of Computer Science Issues*, 8:13–21, 2011. 35

[87] Rob Pooley. Using UML to derive stochastic process algebra models. In *PROC. of XV UK Performance Engineering Workshop*, pages 23–33. 1999. 35

[88] Vincenzo Grassi, Raffaela Mirandola, and Antonino Sabetta. Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach. *J. Syst. Softw.*, 80:528–558, April 2007. 35

[89] Vincenzo Grassi, Raffaela Mirandola, Enrico Randazzo, and Antonino Sabetta. KLAPER: An intermediate language for model-driven predictive analysis of performance and reliability. In *The Common Component Modeling Example*, volume 5153 of *Lecture Notes in Computer Science*, pages 327–356. Springer Berlin / Heidelberg, 2008. 36

[90] Gordon P. Gu and Dorina C. Petriu. From UML to LQN by XML algebra-based model transformations. In *Proceedings of the 5th international workshop on Software and performance*, WOSP '05, pages 99–110, New York, NY, USA, 2005. ACM. 36

[91] Isi Mitrani. *Probabilistic Modelling*. Cambridge University Press, 1998. 36, 37

[92] U. Narayan Bhat and Gregory K. Miller. *Elements of Applied Stochastic Processes*. Wiley-Interscience, 3 edition, 2002. 37

[93] Boudewijn R. Haverkort. *Performance of Computer Communication Systems: A Model- Based Approach*. John Wiley & Sons, Inc., New York, NY, USA, 1998. 37

[94] Muhammad A. Qureshi and William H. Sanders. Reward model solution methods with impulse and rate rewards: An algorithm and numerical results. 38, 44, 46, 47

[95] Tamas Suto, Jeremy T. Bradley, and William J. Knottenbelt. Performance trees: Expressiveness and quantitative semantics. In *QEST'07, 4th International Conference on the Quantitative Evaluation of Systems*, pages 41–50. IEEE Computer Society, 2007. 38, 45, 51

[96] Håkan L. S. Younes and Reid G. Simmons. Statistical probabilistic model checking with a focus on time-bounded properties. *Inf. Comput.*, 204(9):1368–1409, September 2006. 38

[97] William H. Sanders and John F. Meyer. A unified approach for specifying measures of performance, dependability, and performability. *Dependable Computing and Fault-Tolerant Systems: Dependable Computing for Critical Applications*, 4:215–237, 1991. 39, 42, 47, 106, 109

[98] Marco Ajmone Marsan and Gianni Conte. A class of generalized stochastic Petri Nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems*, 2:93–122, 1984. 39, 40

[99] Muhammad A. Qureshi, William H. Sanders, Aad P. A. van Moorsel, and Reinhard German. Algorithms for the generation of state-level representations of stochastic activity networks with general reward structures. In *IEEE Transactions on Software Engineering*, pages 180–190. IEEE Comp. Soc. Press, 1995. 40

[100] Giovanni Chiola, Marco Ajmone Marsan, Gianfranco Balbo, and Gianni Conte. Generalized stochastic Petri Nets: A definition at the net level and its implications. *IEEE Transactions on Software Engineering*, 19:89–107, 1993. 41

[101] Willaim H. Sanders and John F. Meyer. Stochastic activity networks: formal definitions and concepts, 2001. Springer Lectures On Formal Methods And Performance Analysis. 41

[102] Daniel D. Deavours, Graham Clark, Tod Courtney, David Daly, Salem Derisavi, Jay M. Doyle, William H. Sanders, and Patrick G. Webster. The Möbius framework and its implementation. *IEEE Trans. Softw. Eng.*, 28(10):956–969, 2002. 41, 49, 50

[103] Mohammad Abdollahi Azgomi and Ali Movaghar. Application of stochastic activity networks on network modeling. In *The 10th International Conference on Software, Telecommunications and Computer Networks (SoftCOM'02).*

Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture, Split, CROATIE, 2002. 42

[104] Jogesh K. Muppala, Gianfranco Ciardo, and Kishor S. Trivedi. Stochastic reward nets for reliability prediction. In *Communications in Reliability, Maintainability and Serviceability*, pages 9–20, 1994. 42, 43

[105] Razib Hayat Khan and Poul E. Heegaard. Derivation of stochastic reward net (SRN) from UML specification considering cost efficient deployment management of collaborative service components. *International Journal on New Computer Architectures and Their Applications (IJNCAA)*, 2011. 43

[106] Robin A. Sahner, Kishor S. Trivedi, and Antonio Puliafito. *Performance and reliability analysis of computer systems: an example-based approach using the SHARPE software package*. Kluwer Academic Publishers, Norwell, MA, USA, 1996. 43, 52

[107] J. F. Meyer. On evaluating the performability of degradable computing systems. *IEEE Transactions on Computing*, 29(8):720–731, August 1980. 44

[108] Adnan Aziz, Kumud Sanwal, Vigyan Singhal, and Robert Brayton. Verifying continuous time markov chains. In *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*, pages 269–276. Springer-Verlag, 1996. 45

[109] Boudewijn R. Haverkort, Lucia Cloth, Holger Hermanns, and Joost-Pieter Katoen. Model checking performability properties. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 103–112. IEEE Computer Society, 2002. 45

[110] Nicholas J. Dingle, William J. Knottenbelt, and Tamas Suto. PIPE2: a tool for the performance evaluation of generalised stochastic Petri Nets. *SIGMETRICS Perform. Eval. Rev.*, 36(4):34–39, March 2009. 46, 51

[111] Tamas Suto. *Performance Trees: A Query Specification Formalism For Quantitative Performance Analysis*. PhD thesis, University of London, Imperial College London, Department of Computing. 46

[112] Kishor S. Trivedi, Gianfranco Ciardo, Manish Malhotra, and Robin A. Sahner. Dependability and performability analysis. In *Performance Evaluation of Computer and Communication Systems, Joint Tutorial Papers of Performance '93 and Sigmetrics '93*, pages 587–612. Springer-Verlag, 1993. 47

[113] B. Tuffin, P. K. Choudhary, C. Hirel, and K. S. Trivedi. Simulation versus analytic-numeric methods: illustrative examples. In *Proceedings of the 2nd international conference on Performance evaluation methodologies and tools*, ValueTools '07, pages 63:1–63:10. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007. 48, 52

[114] William H. Sanders. Simulation basics, 2002. http://www.ece.virginia.edu/~mv/edu/prob/stat/simulation-tips-from-uiuc.pdf. 48, 150

[115] David Daly, Daniel D. Deavours, Jay M. Doyle, Patrick G. Webster, and William H. Sanders. Möbius: An extensible tool for performance and dependability modeling. In *TOOLS '00: Proceedings of the 11th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools*, pages 332–336. Springer-Verlag, 2000. 49, 50

[116] Kishor S. Trivedi. *SPNP Users Manual, version 6.0 ed.* Duke University, 1999. 51

[117] Darren K. Brien, Nicholas J. Dingle, William J. Knottenbelt, Harini Kulatunga, and Tamas Suto. Performance trees: Implementation and distributed evaluation. In *PDMC'08, 7th International Workshop on Parallel and Distributed Methods in Verification*. Elsevier Sciencey, March 2008. 51

[118] Boudewijn R. Haverkort and Ignas G. Niemegeers. Performability modelling tools and techniques. *Perf. Ev*, 25:17–40, 1996. 52

[119] Performance Evaluation group. *GreatSPN Users Manual, version 2.0.2*. Department of Computer Science, University of Torino, 2001. 52

[120] Boudewijn Haverkort and Kishor Trivedi. Specification techniques for Markov reward models. *Discrete Event Dynamic Systems*, 3:219–247, 1993. 10.1007/BF01439850. 52

[121] Kishor Trivedi, Boudewijn Haverkort, Andy Rindos, and Varsha Mainkar. Techniques and tools for reliability and performance evaluation: Problems and perspectives. In Günter Haring and Gabriele Kotsis, editors, *Computer Performance Evaluation Modelling Techniques and Tools*, volume 794 of *Lecture Notes in Computer Science*, pages 1–24. Springer Berlin / Heidelberg, 1994. 52

[122] Soheib Baarir, Marco Beccuti, Davide Cerotti, Massimiliano De Pierro, Susanna Donatelli, and Giuliana Franceschinis. The GreatSPN tool: recent enhancements. *SIGMETRICS Perform. Eval. Rev.*, 36(4):4–9, March 2009. 52

[123] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernndezd, Michael Kay, Jonathan Robie, and Jerome Simon. XML path language (XPath) 2.0. Technical report, World Wide Web Consortium, January 2007. 79

[124] Altova XMLSpy, v2011r3 enterprise edition. `www.altova.com`, 2011. 81

[125] Daniel A. Menasce. Composing web services: a QoS view. *Internet Computing, IEEE*, 8(6):80 –90, nov.-dec. 2004. 90

[126] Shuping Ran. A model for web services discovery with QoS. *ACM SIGecom Exchanges*, 4:1–10, 2003. 91, 92, 94

[127] Mohamad Ibrahim Ladan. Web services metrics: A survey and a classification. In *2011 International Conference on Network and Electronics Engineering IPCSIT*, volume 11, 2011. 92

[128] Soila Pertet and Priya Narasimhan. Causes of failure in web applications. Technical Report CMU-PDL-05-109, Carnegie Mellon University, Dec 2005. 92

[129] I.V. Papaioannou, D.T. Tsesmetzis, I.G. Roussaki, and M.E. Anagnostou. A QoS ontology language for web-services. In *Advanced Information Networking and Applications, 2006. AINA 2006. 20th International Conference on*, volume 1, page 6 pp., april 2006. 93

[130] Web services quality factors version 1.0, July 2011. OASIS Committee Specification 01. 94

[131] Vladimir Tosic, Bernard Pagurek, and Kruti Patel. WSOL - a language for the formal specification of various constraints and classes of service for web services. Technical report, in the international conference on web services, ICWS03, 2002. 99

[132] Hao Wu and Hai Jin. Specifying web service agreement with OWL. In *Next Generation Web Services Practices, 2005. NWeSP 2005. International Conference on*, pages 109 – 114, aug. 2005. 100

[133] Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. From SHIQ and RDF to OWL: The making of a web ontology language. *Journal of Web Semantics*, 1:2003, 2003. 100

[134] Kyriakos Kritikos and Dimitris Plexousakis. Semantic QoS metric matching. In *Proceedings of the European Conference on Web Services*, ECOWS '06, pages 265–274, Washington, DC, USA, 2006. IEEE Computer Society. 100

[135] Kyriakos Kritikos and Dimitris Plexousakis. Service-oriented computing - icsoc 2007 workshops. chapter A Semantic QoS-Based Web Service Discovery Engine for Over-Constrained QoS Demands, pages 151–164. Springer-Verlag, Berlin, Heidelberg, 2009. 100

[136] Charles Miller Grinstead and James Laurie Snell. *Introduction to probability*. American Mathematical Society, 1997. 116

[137] Rouaa Yassin Kassab and Aad van Moorsel. Predicting compliance of WSLA contracts using automated model creation. School of Computing Science Technical Report Series 1204, School of Computing Science, Newcastle University, June 2010. 124, 172

[138] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, October 1992. 128, 129, 148

[139] Lihua Xu, Hadar Ziv, and Debra Richardson. Towards modeling nonfunctional requirements in software architecture. In *In Proceedings of Aspect-Oriented Software Design, Workshop on Aspect Oriented Requirements Engineering and Architecture Design*, 2005. 129

[140] Lawrence Chung and Julio do Prado Leite. On non-functional requirements in software engineering. In Alexander Borgida, Vinay Chaudhri, Paolo Giorgini, and Eric Yu, editors, *Conceptual Modeling: Foundations and Applications*, volume 5600 of *Lecture Notes in Computer Science*, pages 363–379. Springer Berlin / Heidelberg, 2009. 129

[141] Sun Microsystems. Java, 2012. `http://www.java.com`. 130, 148

[142] R.M. Smith, K.S. Trivedi, and A.V. Ramesh. Performability analysis: measures, an algorithm, and a case study. *Computers, IEEE Transactions on*, 37(4):406 –417, apr 1988. 144

[143] Ali Khalili, Amir Jalaly Bidgoly, and Mohammad Abdollahi Azgomi. PDETool: A multi-formalism modeling tool for discrete-event systems based on SDES description. In Giuliana Franceschinis and Karsten Wolf, editors, *Applications and Theory of Petri Nets*, volume 5606 of *Lecture Notes in Computer Science*, pages 343–352. Springer Berlin / Heidelberg, 2009. 146

[144] Eclipse Foundation. Eclipse classic 3.7.2, 2012. http://www.eclipse.org/. 148

[145] Sun Microsystems. A Swing architecture overview, 2012. http://java.sun.com/products/jfc/tsc/articles/architecture/. 148

[146] Gianfranco Ciardo, Jogesh K. Muppala, and Kishor S. Trivedi. Analyzing concurrent and fault-tolerant software using stochastic reward nets. *Journal of Parallel and Distributed Computing*, 15:255–269, 1992. 151

[147] Pere Bonet, Catalina M Llad, Ramon Puigjaner, and William J Knottenbelt. PIPE v2.5 : a Petri Net tool for performance modeling. *Proceedings of the 23rd Latin American Conference on Informatics*, 2007. 152

[148] Performance and Dependability Engineering Lab (PDE Lab) Computer Faculty, Iran University of Science and Technology. *SimGine version 1.0.*, 2009. 155

[149] MathWorks. MATLAB: The language of technical computing, 2012. http://www.mathworks.co.uk/products/matlab/. 159

[150] OpenMath Society. OpenMath, 2012. www.openmath.org/. 159

[151] Sally L. Bond, Sally E. Boyd, Kathleen A. Rapp, Jacqueline B. Raphael, and Beverly A. Sizemore. Taking stock: A practical guide to evaluating your own programs, 1997. 176

[152] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (WSDL) 1.1, 15 March 2001. IBM, Microsoft, Ariba. 179

[153] Muhammad Asif Javed. Petri Net modeling of web services. Master thesis, Faculty of the Graduate College of the Oklahoma state University. 185, 194, 195

[154] IBM. Stockquote.wsdl, 2012. http://pic.dhe.ibm.com/infocenter/ratdevz/v7r6/index.jsp?topic=/com.ibm.etools.est.doc/concepts/csfprj002.html. 186

[155] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business process execution language for web services version 1.1, May 2003. 194

[156] Frank Leymann. Web services flow language (WSFL 1.0), May 2001. 194

[157] Johnson P Thomas, Mathews Thomas, and George Ghinea. Modeling of web services flow. In *E-Commerce, 2003. CEC 2003. IEEE International Conference on*, pages 391 – 398, june 2003. 194

[158] Zhangxi Tan, Chuang Lin, Hao Yin, Ye Hong, and Guangxi Zhu. Approximate performance analysis of web services flow using stochastic Petri Net. In *In GCC*, 2004. 194

[159] Ward Whitt. Open and closed models for networks of queues. *AT&T Bell Laboratories Technical Journal*, 63(9):1911–1979, 1984. 225