Dipartimento di Informatica
Università di L'Aquila
Via Vetoio, I-67100 L'Aquila, Italy
http://www.di.univaq.it

Dissertation

Model-based Performance Analysis of Software Architectures

Antinisca Di Marco

2005

Advisor: Prof. ssa P. Inverardi

To Vittorio,
the man who gave
me back the smile.

ABSTRACT

Over the last decade, research has addressed the importance of integrating quantitative validation in the software development process, in order to meet performance requirements. Performance problems may be so severe that they can require considerable changes at any stage of software life cycle. Especially at the earlier phases inaccurate decisions may imply an expensive rework, possibly involving the overall software system structure.

The validation and verification of performance requirements is an important task that should be accomplished at any stage of the software life cycle. To this purpose, many approaches have been proposed. Although several of these approaches have been successfully applied, we are still far from considering performance analysis as an integrated activity into software development which effectively supports all phases of the software life cycle. We believe that model-based performance analysis techniques facilitate such an integration, since models represent tools that can be refined during the software life cycle while the development progresses, and also at run time they provide fast and suitable support in dynamic management of the system performance.

The aim of this thesis is to achieve model-based performance analysis of software architecture of component-based systems. We consider performance models that describe the software system at the architecture level. The thesis deals with the usage of such analysis at two different phases of the software life cycle: at the design level and at run time. The former, called predictive performance analysis, helps the designers during the software architecture definition step. The latter is used to dynamically manage the performance attributes when the system runs.

We devise a new predictive performance analysis methodology (approach, etc.) that can be applied at the software architecture level when information on the hardware platform is still missing.

To evaluate the applicability of existing approach to predictive performance analysis, we apply two existing software performance analysis approaches, one based on Stochastic Process Algebras and one based on the simulation models, to a real telecommunication software system. This study allows us to identify their suitability to real complex software system and to define the main suitable features an approach to predictive analysis should have to be accepted by software designers.

We also introduce a framework to cope with the integration of functional and non-functional analyses at the software architecture level. Such a framework allows to deal with inter-relationships between functional and non-functional aspects that would not necessarily emerge from separate analysis. The integration framework is still a working progress and we aim to implement it in the next future.

Finally, we define a framework to dynamically reconfigure a component-based application in order to manage its performance while it runs. The framework monitors the application performance and, when some problem occurs, it chooses the next configuration for the software system. Such decision is taken on the basis of feedback provided by the on-line evaluation of performance models of several pre-defined and feasible reconfiguration alternatives.

ACKNOWLEDGMENTS

It is hard to sum up in one page all the support received from people for my PH.D. studies.

Undoubtedly, I learned a lot under the supervision of Paola Inverardi. First of all the ability to overcome unpredictable problems and to be flexible to the many exigencies can occur in the real life. She has always treated me and my colleagues at an even level, giving us the opportunity to emerge and to meet important people in our research fields. I want to thank her for all of this.

I want to thank the reviewers of this thesis, Dorina Petriu and Jeff Maggee for their great job and valuable advices that allowed me to improve the quality of this manuscript. In particular, I find Dorina Petriu a special woman able to express great humanity.

I want to thank the researchers I collaborated with in the past three years: Simonetta Balsamo, Benedetto Intrigila, Marta Simeoni, Giuseppe Della Penna, Alfonso Pierantonio, Francesco Lo Presti, Connie Smith, Catalina Lladó, Lloyd Williams, Mauro Caporuscio, Moreno Marzolla, and Igor Melatti.

In particular, Marta Simeoni was very closed to me at the beginning of my Ph.D. program, when everything seems to be big and insurmountable. She gave me many useful and meaningful advices.

I want to say thank to the Software Engineering Group of the Computer Science Department of University College London that gave and still today gives me the great opportunity to work with. This group was closed to me in the last, and even more difficult, part of the Ph.D. program, when all the work of three years had to be compressed and organized in this manuscript. In particular, I want to thank Cecilia Mascolo and Wolfgang Emmerich that show me every day their deem and credit in my work and my skills.

Special acknowledgments go to Vincenzo Grassi and Raffaela Mirandola that are two special persons whom I had not chance to work with, even if they were and are always available to give me (technical and human) support.

The SEA Group with which I shared my experience and troubles. All of you are special. In particular, I want to thank Fabio Mancinelli because he shared with me many thinks and opinions, and Mauro Caporuscio hoping that he never changes his mind and behavior.

In my department in L'Aquila, I want to give special thanks to Michele Flammini who spent considerable time to encourage me.

To Dora that was my ray of hope and thoughtlessness in my dark days. To Fernanda which is the mirror where my soul reflects itself. To Valeria for her continuous presence and human support.

To Monica and Romano which always feel myself part of the family as if I was since long time. Even if we meet each other few times in this three years, they always dedicate a lot of time to reassure myself.

Last, but not least, to Vittorio to exist. He has the ability to feel myself close to him even when there are thousands of miles among us.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Over the last decade, research has addressed the importance of integrating quantitative validation in the software development process, in order to meet non-functional requirements. Among these, performance is one of the most influential factors to be considered since performance problems may be so severe that they can require considerable changes at any stage of software life cycle, in particular at the software architecture level or design phase and, in the worst cases, they can even impact the requirements level. Independently of the software process, the early design phases may heavily affect the software development and the quality of the final software product. Therefore inaccurate decisions at early phases may imply an expensive rework, possibly involving the overall software system. Traditional software development methods focus on software correctness, and deal with performance issues later in the development process. But this development style called “fix-it-later” approach, brought large-sized projects to fail[89].

In the research community there has been a growing interest in the (early) validation of performance requirements and many approaches have been proposed. Although several of these approaches have been successfully applied, we are still far from seeing performance analysis as an integrated activity into software development which effectively supports all phases of the software life cycle. We believe that model-based performance analysis techniques facilitate such an integration, since models represent tools that can be refined during the software life cycle while the development progresses.

In the software practice, it is generally acknowledged that the lack of performance requirement validation during the software development process is mostly due to the knowledge gap between software engineers/architects and quality assurance experts (as special skills required) rather than to performance foundational issues. Moreover, short time to market constraints make this situation even more critical. In this scenario software modeling notations and tools may play a crucial role to fill this gap as well as to shorten the performance validation time.

With the growing complexity and size of modern distributed software systems the need of tools to support design decisions and manage performance of a software system at run time are becoming a critical issue.

Software development teams often have to decide among different functionally equivalent design alternatives relying only on their own skills and experience. This choice is driven by non-functional factors such as performance, reliability, and topological/economical constraints. The criticality of these attributes is high even in software systems where non functional requirements are not explicitly expressed. Even in a component-based distributed software system the attributes such as performance, dependability, maintainability determine the quality of the product and the success of a software development. Component-based software systems are developed by assembling existing components. Software components are independent, compositional and deployable units which interact each others to provide services to the user. On one side, the problem of assessing the quality of single components is an active research area, on the other side even if it could be possible to assume ”good quality” of components the quality of the assembled software system would not always be guaranteed. Hence it is mandatory to pursue further investigation, especially in the early development phases, on how components interact each other and with the environment.

For software systems whose performance requirements are strict, in addition to performance validation

at the design time, performance attributes should be monitored during their execution, in order to react opportunely every time performance degradations are experienced. In this direction, recently, growing attention has been focused on run-time management of Quality of Service of complex software systems. In this context, self-adaptation of applications based on run-time monitoring and dynamic reconfiguration is considered a useful technique to manage QoS in complex systems. Many frameworks for dynamic reconfiguration have been recently proposed for this aim. These frameworks lay on monitoring, reconfiguration and on–line model-based analysis to manage/negotiate QoS level of software systems at run time. They share the idea of modifying the application configuration when the threshold of a critical QoS index is crossed. The choice of the new configuration for improving the QoS of the system is based on the current status of the managed software application.

We believe that the combined action of predictive analysis at the early phases of the software development process, and adaptation at run time, are the key points to assure the fulfilment of the performance constraints for the software systems.

MOTIVATIONS AND CONTRIBUTIONS

The aim of this thesis is to achieve a deep insight in model-based performance analysis of software architecture of component-based systems. We consider performance models that describe the software system at the architecture level even if the analysis can be carried on at any stage of the software life cycle. The thesis will deal with the usage of such analysis at two different phases of the software life cycle: design level and run time. The former, called predictive performance analysis, helps the designers during the software architecture definition step. Errors in such development phase could prevent the success of the whole project. The latter instead is used to dynamically manage the performance attributes when the system runs. The choice of the software architecture level is not casual: the performance analysis is strongly based on the dynamics of the software system and, at the design level, the software architecture is the first software artifact describing the dynamics of the software system. Indeed, at run time we have represented the managed software system at the software architecture level to remove unnecessary details that can make the target model not tractable. The on–line model evaluation in fact imposes strict requirements on the analysis that has to be performed as quickly as possible to guarantee timely reactions when some performance problems occur.

The contributions of this thesis can be summarized as follows:

Predictive Performance Analysis. On the basis of a study on the state of the art, we devise a new predictive performance analysis methodology that can be applied at the software architecture level, when information on the hardware platform on which the final system shall run is still missing. The step we here focus on is the definition of an automated transformation from software model into a performance model. To achieve this goal several decisions should be taken at the beginning of the realization: the software notation, that is the entry data for the analysis process, the target notation that is the performance model that has to be generated, the additional information needed to carry on the analysis (such as for example operational profile and workload) , and where and how such an information should be specified. The thesis will address all such points. Making automated such a transformation helps to fill the gap among the software and performance experts and to integrate the validation of the performance requirements in the software life cycle without delaying the development process.

Recently, many approaches to performance analysis of software systems at the software architecture level have been defined. However, their application to real and complex case studies is still limited even if it helps understanding the capabilities, complexity and limits of such approaches. The thesis will discuss the application of two existing software performance analysis approaches, one based on Stochastic Process Algebras and one based on the simulation models, to a real telecommunication software system. Thanks to this experiment we are able to point out, for each used methodology,

different figure of merits in terms of performance modeling, analysis and feedbacks at the design level that can derive from the performance results interpretation. Moreover the use of different techniques can provide the software designer of a more precise and comprehensive picture on the software architecture. Thus, the concurrent/complementary application of several analysis techniques will overcome the problems of the single techniques and conduct to more faithful analysis results.

We also introduce a framework to cope with the integration of functional and non-functional analysis at the software architecture level. Such a framework allows to deal with inter-relationships between functional and non-functional aspects that would not necessarily emerge from separate analysis. We also present how a fully automated software performance approach can be properly integrated into the framework.

Performance Management of the Software System at Run-time. We define a framework able to dynamically reconfigure an application in order to manage the performance of the software system at run-time. The framework monitors the performance of the application and, when some problem occurs, decides the new application configuration on the basis of feedback provided by the on–line evaluation of performance models of several, pre-defined feasible alternatives. The choice of the new system configuration might consider several factors, such as, for example resources needed to implement the new configuration.

OUTLINE OF THE THESIS

Chapter 2 reviews the main software and performance notations, and gives for each notation an example of modelling a simple case study. The chapter concludes with a study on the three main performance notations usually adopted in software performance engineering, i.e. Queueing Networks, Stochastic Process Algebras and Generalized Stochastic Petri Nets. This study originates from our interest to determine which notation may be more acceptable for a software designer to carry on a performance analysis, and under which assumptions on the designer skills and on the software development environment this is true. To this extent we consider the three major notations at work on a simple example.

After the above chapters containing background notions, the thesis is composed by three parts.

Part I- Predictive Performance Analysis: From Software Models to Performance Models. This part is composed by the Chapters 3, 4 and 5 and deals with the predictive performance analysis under different points of views. In the Chapter 3 the principal approaches to performance model generation from the software models are surveyed. The approaches start from a description of the software system at different level of abstraction and generate a ready-to-validate performance model. The reviewed methodologies are finally classified with respect to several dimensions. In Chapter 4 we present our approach to performance model generation. The approach defines rules to generate a Queueing Network model from a set of UML 2.0 diagrams describing the software architecture of a component-based software system. The application of such approach is shown on a simplified e-commerce system. Finally, in Chapter 5 two different existing predictive performance analyses are applied to a real telecommunication system. The aim of this chapter is twofold: to study the applicability of such approaches on real software systems in order to identify their limits. In particular, we are interested on how they can be improved, and to discuss the advantages and the disadvantages of the applied techniques and how to take advantage of the simultaneous usage of different methodologies to easily integrate the predictive performance analysis in real industrial contexts.

Part II- Integration of Predictive Functional and Non-Functional Analyses. This part is composed by the Chapters 6 and 7. In the first one a tool integration framework (namely TOOL•one)is presented. This framework copes with the integration of functional and non-functional analysis at the software architecture level. Chapter 7 presents an automated software performance engineering approach that can be integrated in the TOOL•one framework.

Part III- Model-based Performance Analysis in System Dynamic Reconfiguration. This part copes with the work done during the last period of the Ph.D. program. It is composed by Chapter 8 where we discuss the usage of the model-based performance analysis in dynamic reconfiguration of component-based software system. In this chapter a new reconfiguration process is devised. The next configuration of the software system is chosen among the possible ones on the basis of the performance indices. The process has been used up today to dynamically manage the performance of the *SIENA* event publish/subscribe middleware.

RELATED PUBLICATIONS

Part of this thesis comes from published papers, as follows. Chapter 2 and Chapter 3, on the Software and Performance models and on the state of the art of the Predictive Analysis of Software System, can be considered the evolution of two papers: [61] where we compare three different performance notations from a software designer perspective, and [27] where the most relevant early performance analysis approaches have been reviewed and classified according to several dimensions. Chapter 4 extends two papers, [72] and [73], which describe our proposed approach to predictive performance analysis at the software architecture level. This approach is defined upon to component-based software systems and it generates a Queueing Network model from UML 2.0 diagrams describing the software architecture of the software system. Chapter 5 extends the papers [58] and [31], which present the application of two different predictive performance analyses on a real telecommunication system. Chapter 6 and 7 extend the papers in [62] and [146], respectively. The first one introduces a framework allowing integrated functional and non-functional analysis on software systems during the design phase. The second one presents our effort to make a predictive performance analysis approach fully automated. Chapter 8 refines the work in [55] and [51]. In [55] it is defined a reconfiguration process to dynamic manage performance attributes of component-based software systems. This process is based on the evaluation of performance models representing the software system at the architectural detail. In [51], this process is used to dynamically reconfigure *SIENA* middleware when performance constraints are not satisfied.

During my Ph.D. program, I published two more papers [71] and [70], that, for sake of brevity, are not part of this thesis. Both of them deal with a technique called Xere (*XML E*ntity *R*elationship *E*xchange) for the XML–DBMS integration and describe a mapping algorithm that allows a *natural* translation of XML Schemas into Entity-Relationship diagrams. We also discuss the soundness and completeness of the Xere mapping and we show its implementation in XSLT and Java.

CHAPTER 2

SOFTWARE MODELS AND PERFORMANCE MODELS

In the software practice, it is generally acknowledged that the lack of performance requirement validation in the software life cycle is mostly due to the knowledge gap between software engineers/architects and quality assurance experts rather than to foundational issues. Software specialists use specific notations to describe software systems, since the first phases of the software life cycle, that are far from the ones used by performance experts. In fact, while the software notations emphasize the structure and the dynamics of the software system in an abstract and general way, the performance notations provide capabilities in modelling performance specific aspects of the software system needed to validate the software system with respect to the performance requirements. In this scenario software modeling notations and tools may play a crucial role to fill this gap as well as to shorten the performance validation time. The more close the software modeling notation to performance one is the more little the gap is.

This chapter briefly reviews the most used software and performance modelling notations. For each notation, both software and performance one, an example of modeling over the XML Translator system introduced in the following is shown.

Moreover we here report a study about the three main performance notations usually used in software performance engineering, i.e. Queueing Network, Stochastic Process Algebra and Stochastic Timed Petri Net. This study originates from our interest to investigate the impact that a performance model notation may have on the software development when this is integrated with a performance analysis approach. In particular, we would like to determine which notation may be more acceptable for a software designer and under which assumptions on the designer skills and on the software development environment this is true. To this extent we consider the three major notations at work over the XML Translator system.

The work this chapter discusses has been outlined in [61] and it is described here in details.

CASE STUDY: A XML TRANSLATOR - In this section it is presented the simple system used to show the software and performance modeling and the performance notation comparison. We decided to use a common and simple case study in order to show similarities and differences among the reviewed notations. Moreover, it helps in emphasizing the expressiveness of each notation, especially in the performance notation study that concludes the chapter.

The software system we consider is called XML Translator (XT). It automatically builds an XML document from a text document with respect to a given XML schema [156]. The text document has a fixed structure to allow the automatic identification of its specific parts that are then emphasized by using the XML tags defined in the given XML Schema.

The XT system reads a text document, and it creates a new XML file with the information content of the text document suitably formatted with respect to the considered XML syntax [154] and the XML Schema. The system builds the new file by iterative steps in which it identifies useful information and marks it up. Multiple users can concurrently connect to the system and request its services.

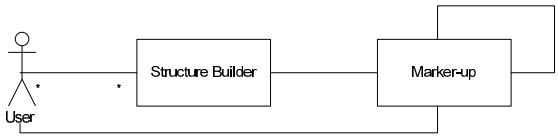


Figure 2.1: Static Description of XT system

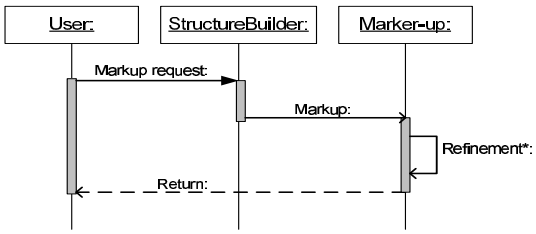


Figure 2.2: Behavioral Description of XT system

From this first description of the system we can identify two distinct software components:

- a *StructureBuilder*, that preprocesses the text file to create its XML related content (i.e. XML special characters) conform to the XML syntax rules. The output of this step is a new text file semantically equivalent to the former, but syntactically different. It also creates an XML file according with the established XML Schema, containing only XML tags that describe the structure of the document.
- a *Marker* that, by using a heuristic approach, localizes useful information in the text document, singles out it by significant tags from the XML Schema and inserts this chunk of information in the XML file. This component works iteratively on the XML version of the document for an unknown number of times until it is not acceptable (i.e., it does emphasize most of the useful information under certain heuristic conditions).

A static description of XT system is shown in Figure 2.1, whereas its behavior is defined by means of the UML Sequence Diagram [6] in Figure 2.2, which shows that all the interactions among XT components are asynchronous.

2.1 Software Specification Models

Software engineers describe static and dynamic aspects of a software system by using ad-hoc models. The static description consists of the identification of software modules or components and of their interconnections, e.g. see Figure 2.1. The dynamics of a software system concerns its behavior at run time. There exists many notations to describe the behavior of a software system. In the following we shortly review Automata [99], Process Algebra [120], Petri Nets [132], Message Sequence Charts (MSC) [137], Unified Modeling Language (UML) Diagrams [6] and Use Case Maps (UCM) [15]. We also show how the XT system could be modelled by means of such tools.

2.1.1 Automata

Automaton [99] is a simple mathematical and expressive formalism that allows to model cooperation and synchronization between subsystems, concurrent and not. It is a compositional formalism where a system

is modelled as a set of states and its behavior is described by transitions between them, triggered by some input symbol.

More formally an automaton is composed of a (possibly infinite) set of states Q, a set of input symbols Σ and a function $\delta : Q \times \Sigma \rightarrow Q$ that defines the transitions between states. In Q there is a special state $q_0 \in Q$, the initial state from which all computations start, and a set of final states $F \subset Q$ reached by the system at the end of correct finite computations [99]. It is always possible to associate a direct labelled graph to an automaton, called State Transition Graph (or State Transition Diagram), where nodes represent the states and labelled edges represent transitions of the automata triggered by the input symbols associated to the edges.

Automata can also be composed through composition operators, notably the parallel one that composes two automata A and B by allowing the interleaving combination of A and B transitions.

There exist many types of automata, among which we consider: *deterministic automata* with a deterministic transition function, that is the transition between states is fully determined by the current state and the input symbol; *non deterministic automata* with a transition function that allows more state transitions for the same input symbol from a given state; *stochastic automata* which are non deterministic automata where the next state of the system is determined by a probabilistic value associated to each possibility.

XML TRANSLATOR AUTOMATON - The state transition graph of the XML Translator automaton is shown in Figure 2.3 (c). States are pairs of elements, that model the Structure Builder component state and the Marker-up component state respectively. The initial state of the automaton is $< q_0, q'_0 >$ where both components are inactive. This state is also the final one where the system correctly terminates the computation. Since the XML Translator automaton is obtained by the parallel composition of the Structure Builder and of the Marker-up components automata (Figure 2.3 (a) and (b) respectively), we analyze their behavior separately.

The Structure Builder component transits from the state q_0 to the state q_1 when it receives a markup request and it starts the preprocessing phase. At the end of its elaboration it sends the event markup to the Marker-up component and returns to the initial state q_0 where it waits for new requests.

When the Marker-up component receives the markup event, it moves from its initial state q'_0 to state q'_1 where it remains until the refinement processing has been completed. Eventually the component moves back to its initial state.

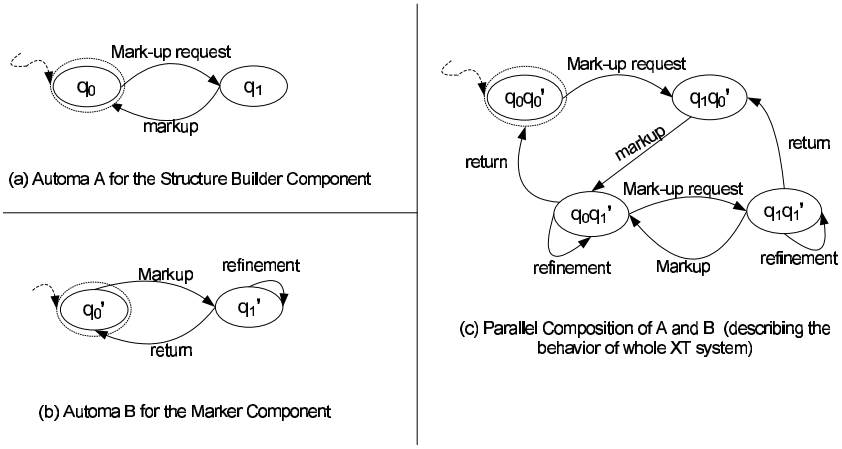


Figure 2.3: State Transition Graph for the XML Translator Automaton.

2.1.2 PROCESS ALGEBRAS

System behavior generally consists of processes which are the control mechanisms for the manipulation of data. System behavior tends to be composed of several processes that are executed concurrently, where these processes exchange data in order to influence each other's behavior.

For the purpose of mathematical reasoning, it is often convenient to represent the system behavior algebraically in the form of terms. The formalism used for this representation is the Process Algebra.

Process Algebras, such as CCS [120] and CSP [96], are a widely known modeling technique for the functional analysis of concurrent systems. These are described as collections of entities, or *processes*, executing atomic *actions*, which are used to describe concurrent behaviors which synchronize in order to communicate. Processes can be composed by means of a set of operators, which include different forms of parallel composition.

Process algebras provide a formal model of concurrent systems, which is abstract (the internal behavior of the system components can be disregarded) and compositional (systems can be modelled in terms of the interactions of their subsystems). The semantics of these calculi is usually defined in terms of Labelled Transition Systems (LTS) following the structural operating semantics approach. Moreover Process Algebra formalism is used to detect undesirable properties and to formally derive desirable properties of a system specification. Notably, process algebra can be used to verify that a system displays the desired external behavior, meaning that for each input the correct output is produced. T

Process Algebras can describe systems at different levels of abstraction. Many notions of equivalence or pre-order are defined to study the relationship between different descriptions of the same system. Behavioral equivalences allow one to prove that two different system specifications are equivalent when "uninteresting" details are ignored, while pre-orders are suitable for proving that a low level specification is a satisfactory implementation of a more abstract one.

A PROCESS ALGEBRA MODEL FOR THE XML TRANSLATOR SYSTEM - The process algebra model for the XML Translator system consists of two main processes, a StructureBuilder process and a Marker process, that perform actions to satisfy the requests. To model the asynchrony in the system we introduce two further processes representing two queues. The first queue, *Queue1*, buffers the requests from the users and generates text formatting requests for the StructureBuilder. The second queue, *Queue2*, models the asynchronous connection point from the StructureBuilder to the Marker component.

The specification we present also defines a *User* process that models the user behavior. A user does some work, then enqueues a service request in *Queue1* and waits for the results from the XML Translator.

The behavior of the whole system is specified by putting in parallel all the processes. Figure 2.4 shows the specification of the XML Translator system with three concurrent users and queues of capacity three, modelled by using the TIPP Process Algebra [93]. The behavior of the StructureBuilder process is recursively defined by a sequence of three actions: *deq1*, the process dequeues a request from its buffer (if any), *preprocessing*, the process does its work, and *enq3*, the process forwards a request to the Marker process. For what concerns the Marker process, it dequeues a request, if any, from its buffer (*deq2* action), it executes the *markup* action and then, in a non deterministic way, it can decide to make a refinement or to return the control to the User. Eventually it restarts its execution.

2.1.3 PETRI NETS

Petri Nets (PN) are a formal modeling technique to specify synchronization behavior of concurrent systems. A PN [132] is defined by a set of *places*, a set of *transitions*, an *input function* relating places to

```
specification System

behaviour

        (User|||User|||User)|[enq1,arrival]|(Queue1(0,3)|[deq1]|
        StructureBuilder|[enq3]|Queue2(0,3)|[deq2,enq2]|Marker)
where
        process User := work; enq1; arrival; User endproc

        process Queue1(n,k) := [n>0] -> (deq1; Queue1(n-1,k)) []
                               [n<k] -> (enq1;Queue1(n+1,k)) endproc

        process Queue2(n,k) := [n>0] -> (deq2;Queue2(n-1,k)) []
                                        [n<k] -> (enq3;Queue2(n+1,k)) []
                                        [n<k] -> (enq2;Queue2(n+1,k)) endproc

        process StructureBuilder := deq1; pre-processing; enq3; StructureBuilder
endproc

        process Marker := deq2; markup; ((refinement; enq2; Marker) []
                                            (backtousers;  arrival;  Marker))     endproc
endspec
```

Figure 2.4: Process Algebra Model for the XML Translator.

transitions, an *output function* relating transition to places, and a *marking* function, associating to each place a nonnegative integer number where the sets of places and transitions are disjoint sets.

PN have a graphical representation: places are represented by circles, transitions by bars, input function by arcs directed from places to transitions, output function by arcs directed from transitions to places, and marking by bullets, called *tokens*, depicted inside the corresponding places. Tokens distributed among places define the state of the net. The dynamic behavior of a PN is described by the sequence of transition *firings* that change the marking of places (hence the system state). Firing rules define whether a transition is *enabled* or not.

Petri nets could be considered an extension of Finite State Automata giving a new definition of state and transition: each state in Petri Nets is a set of partial and independent states of automata and, in general, a transition does not consider the global state of the system, but only a part. Moreover, two events that can happen independently are represented by two concurrent net transitions, instead in an automata a transition prevents from concurrently verifying other ones. Petri Nets may also model asynchronous systems, where events must take place under a defined frequency.

The main characteristics of PN are the following:(i) causal dependencies and interdependencies among events may be represented explicitly. A non-interleaving, partial order relation of concurrency is introduced for events which are independent of each other; (ii) systems may be represented at different levels of abstraction; (iii) PN support formal verification of functional properties of systems.

A PETRI NET FOR THE XML TRANSLATOR - Figure 2.5 shows the initial configuration of the PN model corresponding to the XML Translator system. Each user is represented by a sub-net consisting of two places and two transitions, shown as a shaded area at the top of the figure. When two tokens are present in P_{2i-1}, the User i is in its initial state and it is ready to produce a request to the XT system. Moreover, the $work_i$ transition is enabled. When $work_i$ fires, the two tokens in P_{2i-1} are consumed (they disappear from P_{2i-1}) and one token is transferred in P_{2i} and the other is enqueued in Q_1. The first indicates that the user i is waiting for the processing result and the second represents the service request forwarded to the StructureBuilder component. The t_i transition in the user sub-net will fire when the XT system returns the service response (one token is in Q_0) and the User i transits in its initial state.

Similarly, the two system components are modelled by the corresponding sub-nets identified by the shaded area at the bottom of the figure. The StructureBuilder is composed by two transitions (deq_1 modeling

the service request receiving, and the $Preproc$ for its pre-processing operation) and two places (SB_1 and SB_2). One token in SB_1 means that the StructureBuilder is waiting for a request whereas a token in the SB_2 means that the component is busy. Similar modeling is done for the Marker component that has one place (M_3) and two transitions ($refinement$ and $back$) more needed to model the decision to refine or to send back the users the result of the work. Places labelled Q_0, Q_1, Q_2 model the queues for asynchronous communication. Dynamically, user requests enter Q_1 to access the Structure Builder. Then its output is enqueued in Q_2 to access Marker. Eventually the processed request returns to the User.

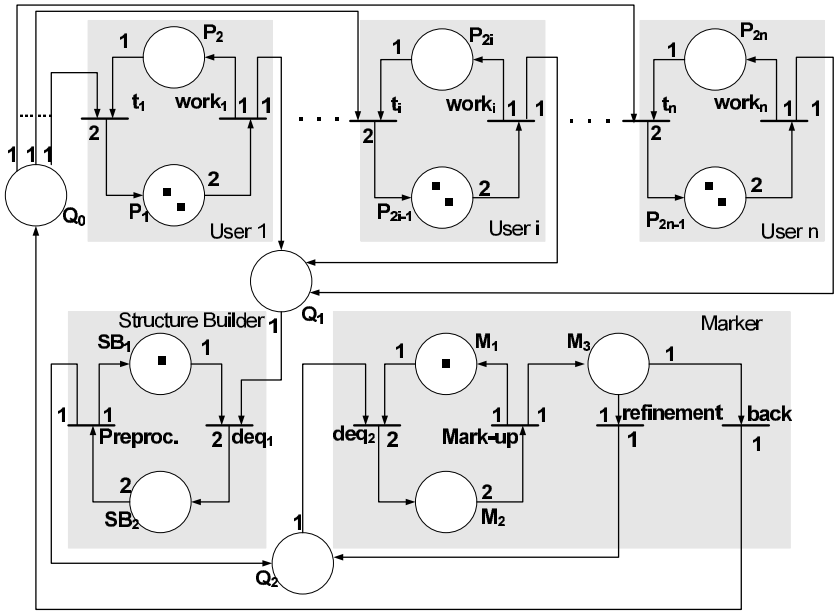


Figure 2.5: Petri Net Model of the XMLTranslator.

2.1.4 MESSAGE SEQUENCE CHARTS

Message Sequence Charts (MSC) is a language to describe the interaction among a number of independent message-passing instances (e.g. components, objects or processes) or between instances and the environment. This language is specified by the International Telecommunication Union (ITU) in [137]. MSC is a scenario language that describes the communication among instances, i.e., the messages sent, messages received, and the local events, together with the ordering between them. One MSC describes a partial behavior of a system.

Additionally, it allows for expressing restrictions on transmitted data values and on the timing of events. MSC is also a graphical language which specify two-dimensional diagrams, where each instance lifetime is represented as a vertical line, while a message is represented by a horizontal or slanted arrow from the sending process to the receiving one. MSC supports complete and incomplete specifications and it can be used at different levels of abstraction. It allows to develop structured design since simple scenarios described by basic MSC can be combined to form more complete specifications by means of high-level MSC.

In connection with other languages, MSC are used to support methodologies for systems specification, design, simulation, testing, and documentation.

MSC FOR THE XMLTRANSLATOR - Since the presented case study is very simple, given the similarity of the MSC description with UML Sequence Diagram one, we omit the case study description here and we refer to Section 2.1.5.

2.1.5 UNIFIED MODELING LANGUAGE

All the above discussed notations (except MSC) are formal specification languages so they have an exact semantics, but as a drawback, most of them are hard to use in ordinary software engineering practice. This is overcome by less precisely defined formalisms like the Unified Modeling Language [6].

UML, specified by the Object Management Group (OMG), is a notation to describe software at different levels of abstraction. It defines several types of diagrams that can be used to model different system views. Models are usually described in a visual language, which makes the modeling work easier. Even if their semantics is not formally defined, UML diagrams are well accepted because they are flexible, easy to maintain and to use. However, not everything is suitable for a visual description, some information in models is best expressed in ordinary text.

UML diagrams allow us to describe systems either statically or dynamically in an object-oriented style. *Use case diagrams* emphasize the interaction between a user and a system. *Class diagrams* show the logical view of the system by means of classes and their relationships. *Interaction diagrams* represent system objects and how they interact. UML defines two types of interaction diagrams: sequence diagrams and collaboration diagrams. Analogously to MSC, the former emphasizes the lifetime of each object and when interaction between objects occurs, the latter focuses on the system layout to indicate how objects are statically connected. Moreover, sequence diagrams allow us to specify conditions on message sending, to use iteration marking (that identifies multiple sending of a message to receiver objects), and to define the type of communication (synchronous or asynchronous). *State diagrams* show the state space of a given computational unit, the events that cause a transition from one state to another, and the actions that result. *Activity diagrams* show the flow of system activities, *Component Diagrams* specify the decomposition of the system in software components by highlighting their dependencies in terms of required and provided interfaces, and *Deployment diagrams* show the configuration of runtime processing elements associating the software components with hardware platform.

Recently OMG has improved the UML notation releasing a new version of the language that is UML 2.0 [148] that strengths the expressiveness of some diagrams (such as Sequence diagrams), better specifies other ones (such as Component diagrams) and introduces new ones (such as Timing diagrams).

We do not introduce other UML diagrams, since they are not used in our context.

As listed above, UML provides many features to describe system behavior. The dynamics of a software system can be specified by using interaction diagrams which describe the message exchange among instances, or by using state diagrams to specify the internal behavior of each software entities, or by using activity diagrams to show the flow of the activities performed by all the components involved in the computation of interest or by using any combinations of the above diagrams.

UML FOR THE XML TRANSLATOR - The behavior of the XML Translator is described by the UML Sequence diagrams shown in Figure 2.6. It is worthwhile noting that this representation is extremely synthetic thanks to the different semantics associated to the graphical notation of the arrows representing the communication. For example, the open arrow head denotes asynchronous communication (such as for example the `Markup` interaction between the `StructureBuilder` and the `Marker-up` components), the filled arrow head denotes synchronous communication (such as for example the interaction between the `Users` the `StructureBuilder` component), and the dashed arrow denotes return of control. The user requests a service to the XML Translator system by $markuprequest$ method invocation. *StructureBuilder* component catches the request and processes it. When it has finished it calls the $markup$ method on the *Marker-up* component, that, after some $refinement$ steps, returns the control to the user.

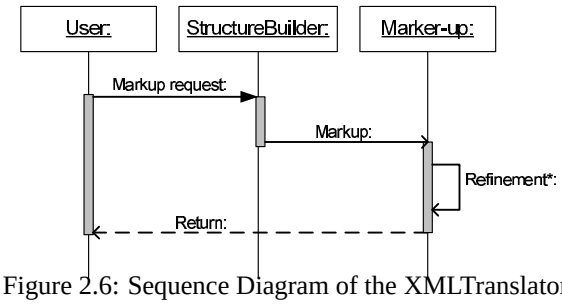


Figure 2.6: Sequence Diagram of the XMLTranslator.

2.1.6 USE CASE MAPS

Use Case Maps (UCM) [49] is a graphical notation allowing the unification of the system use (Use Cases) and the system behavior (Scenarios and State Charts) descriptions. UCM is a high-level design model to help humans express and reason about system large-grained behavior patterns. However, UCM does not aim at providing complete behavioral specifications of systems. At requirement level, UCM models components as black boxes, and at high-level design refines components specifications to exploit their internal parts.

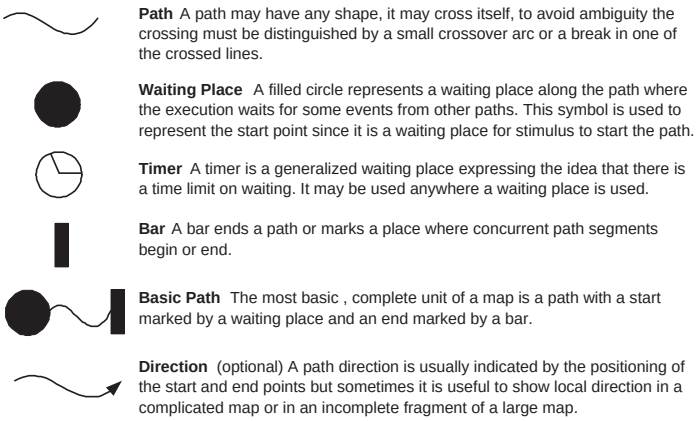


Figure 2.7: Basic Symbols of the UCM Notation.

Figure 2.7 shows the basic elements of UCM notation. UCM represent scenarios through path scenarios. The *start* point of a map corresponds to the beginning of a scenario. Moving through the path, UCM represents the scenario in progress till its end. Paths may traverse components and are routes along chains of causes and effects propagate through the system.

UCM FOR THE XMLTRANSLATOR - Figure 2.8 shows the UCM model of the XML Translator. A user, that is represented as a component, triggers the path scenario by formulating a request. Then the path traverses the component that represents the XML Translator and eventually returns to the user where it has its end point. The two system components are contained in the XML Translator. The UCM model shows their responsibilities and how the path scenario traverses them.

2.2 PERFORMANCE MODELS

In this section we briefly introduce three main classes of stochastic performance models: Queueing Network (QN) models [109, 111, 151], Stochastic Timed Petri Nets (STPN)[17, 22, 16] and Stochastic Process Algebras (SPA) [41, 94, 91].

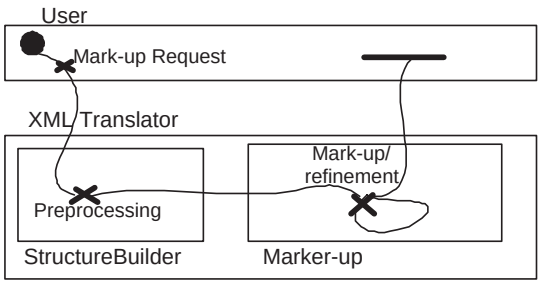


Figure 2.8: UCM Model for the XMLTranslator.

Each type of performance model can be analyzed by analytical methods or by simulation in order to evaluate a set of performance indices such as resource utilization, throughput and customer response time. Simulation is a widely used general technique whose main drawback is the potential high development and computational cost to obtain accurate results. On the other hand, analytical methods require that the model satisfies a set of assumptions and constraints. These models are based on a set of mathematical relationships that characterize the system behavior. The analytical solution of the performance models relies on stochastic processes, usually discrete-space continuous-time homogeneous Markov chains (MC) [104]. Hence, we also present a short description of Markov processes.

Besides being a solution technique for performance models, simulation can be a performance evaluation technique itself [34]. It is actually the most flexible and general analysis technique, since any specified behavior can be simulated. This technique is based on simulation models that we describe at the end of this section.

Performance analysis of complex systems feasible can be based on structuring or factorizing techniques in order to simplify the analysis. Several decomposition and aggregation techniques [103, 22, 104] and hierarchical modeling methodologies [47, 56] have been defined for various classes of performance models.

In line with hierarchical modeling in engineering context, describing a complex system with a hierarchical performance model means applying a top-down decomposition technique: starting from an abstract model of the system, each step defines a more detailed model of the same system, which is composed of interacting submodels that can be further refined in successive steps. The performance analysis of a hierarchical model starts instead from the most detailed model and requires the application of a bottom-up aggregation technique. In other words, the model of each level is analyzed by first solving all its submodels in isolation, and then by aggregating them with respect to the model of the preceding level.

Having specific techniques for the hierarchical definition of performance models allows for the use of a hybrid approach to the solution of submodels: *hybrid models* are obtained when mixed (analytical and simulation) techniques are applied. Moreover, some authors propose also the combined use of different performance models for describing submodels of a given system [23, 24].

In this section we do not deal with hierarchical and hybrid models, and the performance modeling on the three main and basic performance notations that are QN, SPA and STPN.

2.2.1 MARKOV PROCESSES

A stochastic process is a family of random variables $X = \{X(t) : t \in T\}$ where $X(t) : T \times \Omega \rightarrow E$ defined on a probability space Ω, an index set T (usually referred as time) with state space E. Stochastic processes can be classified according to the state space, the time parameter, and the statistical dependencies among the variables $X(t)$. The state space can be discrete or continuous (processes with discrete state space

are usually called *chains*), the time parameter can also be discrete or continuous, and dependencies among variables are described by the joint distribution function.

Informally, a stochastic process is a Markov process if the probability that the process goes from state $s(t_n)$ to a state $s(t_{n+1})$ conditioned to the previous process history equals the probability conditioned only to the last state $s(t_n)$. This implies that a process is fully characterized by these one-step probabilities. Moreover, a Markov process is homogeneous when such transition probabilities are time independent.

Due to the memoryless property, the time that the process spends in each state is exponential or geometrically distributed for the continuous-time or discrete-time Markov process, respectively.

Markov processes can be analyzed and under certain constraints it is possible to derive the stationary and the transient state probability. The stationary solution of the Markov process has a time computational complexity of the order of the state space E cardinality.

Markov processes play a central role in the quantitative analysis of systems, since the analytical solution of the various classes of performance models relies on a stochastic process which is usually a Markov process.

2.2.2 QUEUEING NETWORKS

Queueing Network (QN) models have been widely applied as system performance models to represent and analyze resource sharing systems [109, 111, 103, 151]. A QN model is a collection of interacting *service centers* representing system resources and a set of *customers* representing the users sharing the resources. Its informal representation is a direct graph whose nodes are service centers and edges represent the behavior of customers' service requests.

The popularity of QN models for system performance evaluation is due to the relative high accuracy in performance results and the efficiency in model analysis and evaluation. In this setting the class of product-form networks plays an important role, since they can be analyzed by efficient algorithms to evaluate average performance indices. Specifically, algorithms such as convolution and Mean Value Analysis have a computational complexity polynomial in the number of QN components. These algorithms, on which most approximated analytical methods are based, have been widely applied for performance modeling and analysis.

Informally, the creation of a QN model can be split into three steps: *definition*, that include the definition of service centers, their number, class of customers and topology; *parameterization*, to define model parameters, e.g., arrival processes, service rate and number of customers; *evaluation*, to obtain a quantitative description of the modeled system, by computing a set of figures of merit or performance indices such as resource utilization, system throughput and customer response time. These indices can be *local* to a resource or *global* to the whole system.

Extensions of classical QN models, namely Extended Queuing Network (EQN) models, have been introduced in order to represent several interesting features of real systems, such as synchronization and concurrency constraints, finite capacity queues, memory constraints and simultaneous resource possession. EQN can be solved by approximate solution techniques [111, 103].

Another extension of QN models is the Layered Queuing Network (LQN) which allows the modeling of client-server communication patterns in concurrent and/or distributed software systems [133, 163, 77]. The main difference between LQN and QN models is that in LQN a server may become client (customer) of other servers while serving its own clients requests. A LQN model is represented as an acyclic graph whose nodes are software entities (or *tasks*) and hardware devices, and whose arcs denote service requests (through synchronous, asynchronous or forwarding messages). A task has one or more *entries* providing different services, and each entry can be decomposed in two or more sequential *phases*. A recent extension

of LQN allows for an entry to be further decomposed into *activities* which are related in sequence, loop, parallel (AND fork/join) and alternative (OR fork/join) configurations forming altogether an activity graph. LQN models can be solved by analytic approximation methods based on standard methods for EQN with simultaneous resource possession and Mean Value Analysis or they can be simulated.

Examples of performance evaluation tools for QN and EQN are RESQ/IBM [135, 134], QNAP2 [153] and HIT [37], and for LQN the LQNS tool [163, 78].

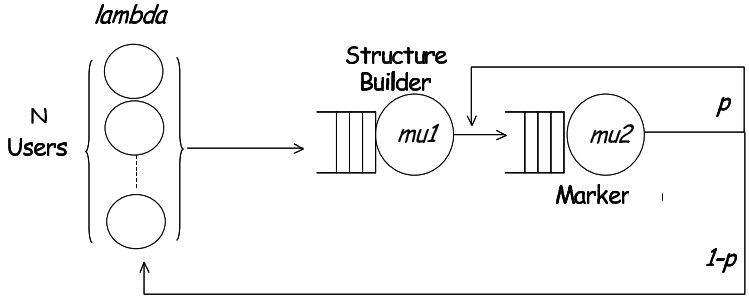


Figure 2.9: QN Model for the XMLTranslator.

QN MODEL FOR THE XMLTRANSLATOR - Figure 2.9 shows a QN consisting of three service centers corresponding to the Structure Builder, the Marker components and the users. We assume exponential service time distribution and FCFS queueing discipline in order to have a product-form QN. The workload of the QN is a closed one produced by the infinite servers center named $Users$ where N indicates the number of service requests (or QN customers) present in the system. The QN topology represents the routes that a user request follows to accomplish its task, where label $1 - p$ represents the probability that the Marker component has completed the iterative marking of the text. Service rates $mu_i, 1 \leq i \leq 2$, $lambda$ and number of customers N are the QN parameters.

2.2.3 STOCHASTIC PROCESS ALGEBRAS

Stochastic Process Algebra (SPA) are extensions of Process Algebras, aiming at the integration of qualitative-functional and quantitative-temporal aspects into a single specification technique [41, 94, 91]. Temporal information is added to actions by means of continuous random variables, representing activity durations. Such information enriches the Label Transition System (LTS) semantic model, hence making it possible the evaluation of functional properties (e.g. liveness, deadlock), temporal indices (e.g. throughput, waiting times) and combined aspects (e.g. probability of timeout, duration of action sequences) of the modeled systems.

The quantitative analysis of the modeled system can be performed by constructing, out of the enriched LTS, the underlying stochastic process. In particular, when action durations are represented by exponential random variables, the underlying stochastic process yields a Markov Chain. Various attempts have been made in order to avoid the Markov Chain state space explosion, which soon makes the performance analysis unfeasible. Some authors propose a syntactic characterization of process algebra terms whose underlying Markov Chain admits a product-form solution that could allow more efficient solution algorithms [90, 95, 138].

Example of performance evaluation tools for SPA are the TIPP tool [93], Two Towers [39, 40], and PEPA Workbench [82].

AN SPA MODEL FOR THE XMLTRANSLATOR - Figure 2.10 shows a SPA model defined by using TIPP process algebra [91]. This model has been obtained from the model presented in Section 2.1.2 by adding performance related information to some actions. Therefore we have simple actions (e.g., enq1) and rated actions, whose rates can be used to express their execution times (e.g., (markup,mu_2)) and their relative execution frequencies (e.g., (refinement,p)).

```
specification System

behaviour

        (User|||User|||User)|[enq1,arrival]|(Queue1(0,3)|[deq1]|
        StructureBuilder|[enq3]|Queue2(0,3)|[deq2,enq2]|Marker)
where
        process User := (work,lambda); enq1; arrival; User endproc

        process Queue1(n,k) := [n>0] -> (deq1; Queue1(n-1,k)) []
                               [n<k] -> (enq1;Queue1(n+1,k)) endproc

        process Queue2(n,k) := [n>0] -> (deq2;Queue2(n-1,k)) []
                               [n<k] -> (enq3;Queue2(n+1,k)) []
                               [n<k] -> (enq2;Queue2(n+1,k)) endproc

        process StructureBuilder := deq1; (processing,mu1); enq3; StructureBuilder     endproc

        process Marker := deq2; (markup,mu2); (((refinement,p); enq2; Marker) []
                                                ((backtousers,100000-p);  arrival;  Marker))
endproc
endspec
```

Figure 2.10: TIPP Process Algebra Model for the XMLTranslator.

2.2.4 STOCHASTIC TIMED PETRI NETS

Stochastic Timed Petri Net (STPN) are extensions of Petri nets. Petri nets can be used to formally verify the correctness of synchronization between various activities of concurrent systems. The underlying assumption in PN is that each activity takes zero time (i.e. once a transition is enabled, it fires instantaneously). In order to answer performance-related questions beside the pure behavioral ones, Petri nets have been extended by associating a finite time duration with transitions and/or places (the usual assumption is that only transitions are timed) [16, 103, 22].

The *firing time* of a transition is the time taken by the activity represented by the transition: in the stochastic timed extension, firing times are expressed by random variables. Although such variables may have an arbitrary distribution, in practice the use of non memoryless distributions makes the analysis unfeasible whenever repetitive behavior is to be modeled, unless other restrictions are imposed (e.g. only one transition is enabled at a time) to simplify the analysis.

The quantitative analysis of a STPN is based on the identification and solution of its associated Markov Chain built on the basis of the net reachability graph. In order to avoid the state space explosion of the Markov Chain, various authors have explored the possibility of deriving a product-form solution for special classes of STPN. Non polynomial algorithms exist for product-form STPN, under further structural constraints. Beside the product-form results, many approximation techniques have been defined [22].

A further extension of Petri Nets is the class of the so called Generalized Stochastic Petri Nets (GSPN), which are continuous time stochastic Petri Nets that allow both exponentially timed and untimed (or immediate) transitions [17]. Immediate transition fires immediately after enabling and have strict priority over timed transitions. Immediate transitions are associated with a (normalized) weight, so that, in case of concurrently enabled immediate transitions the choice of the firing one is solved by a probabilistic choice. GSPN admit specific solution techniques [22].

There are many performance evaluation tools for STPN and other extended classes: a database on (stochastic) Petri net tools can be found at [4].

AN STPN MODEL FOR THE XMLTRANSLATOR - Figure 2.11 shows a STPN model for the XML Translator system, obtained from the PN model of Section 2.1.3 by introducing performance related information. In particular we are here considering a GSPN model. We assign a time attribute to all the transitions modeling the service of each component. Hence the timed transitions are: $work_i, 1 \leq i \leq n$, representing that the $i-th$ user is formatting a text; $Preproc$, the *StructureBuilder* component is processing a request; $Mark-up$, the *Marker* component is processing a request. All the remaining transitions are immediate. The transitions outgoing place M_3 have a probability associated, in order to model the relative frequency of the $refinement$ and $back$ alternatives.

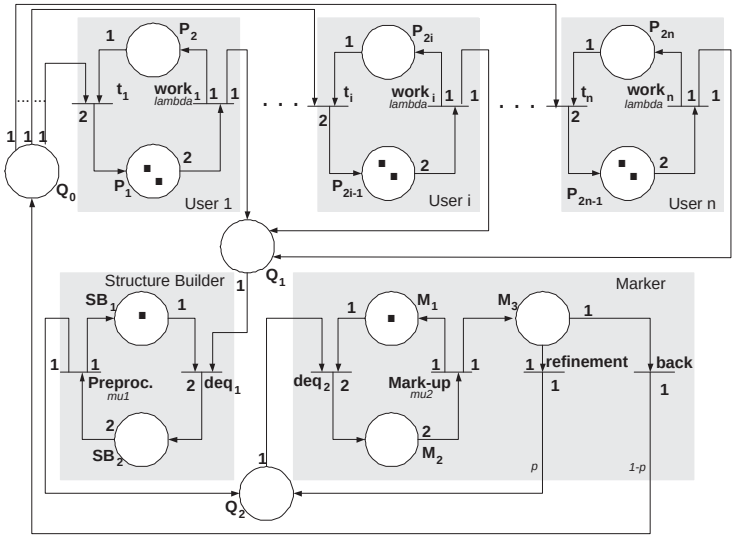


Figure 2.11: STPN Model of the XMLTranslator.

2.2.5 SIMULATION MODELS

Besides being a solution technique for performance models, simulation can be a performance evaluation technique itself [34]. It is actually the most flexible and general analysis technique, since any specified behavior can be simulated. The main drawback of simulation is its development and execution cost.

Simulation of a complex system includes the following phases:

- building a simulation model (i.e., a conceptual representation of the system) using a *process oriented* or an *event oriented* approach;
- deriving a simulation program which implements the simulation model;
- verifying the correctness of the program with respect to the model;
- validating the conceptual simulation model with respect to the system (i.e. checking whether the model can be substituted to the real system for the purposes of experimentation);
- planning the simulation experiments, e.g. length of the simulation run, number of run, initialization;

- running the simulation program and analyzing the results via appropriate output analysis methods based on statistical techniques.

A critical issue in simulation concerns the identification of the system model at the appropriate level of abstraction.

Existing simulation tools provide suitable specification languages for the definition of simulation models, and a simulation environment to conduct system performance evaluation, e.g., CSIM [2], C++Sim [1] and JavaSim [3].

2.3 Performance Model Comparison from a Software Designer Perspective

To deal with performance analysis of software systems at the early phases of the software life cycle, several model notations can be used. Except for the simulation models, these model notations fall into two main categories: notations like Queueing Networks that were initially proposed to represent performance features of actual systems, typically hardware or manufacturing systems; notations like (Stochastic) Petri Nets or Process Algebras that were first proposed in the software specification field and then exported in the whole performance domain.

Early in the lifecycle, the choice of the performance model notation is still open. From the software designer perspective, there can be a relevant difference between the above choices. For example, while Queueing Networks are apparently quite far from the software developer knowledge, Process Algebras (or Petri Nets) they seem to be closer to the developer viewpoint. Nevertheless, it can be observed that, in the last few years, Queueing Networks constitute the favorite target for performance assessment [27], even in the early lifecycle phases where the software model is still based on abstract interacting components. Moreover, "Queuing Network modeling is a top-down process. The underlying philosophy is to begin by identifying the principal *components* of the system and the ways they *interact*, then supply any details that prove to be necessary" (quoting from [111]). This suggests a very intuitive and natural mapping of Queueing Network with early in the software lifecycle artifacts, like software architectures descriptions.

On the other hand PA and PN have the advantage of importing performance analysis almost for free in their modeling, thus making the performance model construction straightforward at the expense of the behavioral model construction.

Based on these observations, the study described in this section originates from our interest to investigate the impact that a performance model notation may have on the software development. The lack of studies in this direction has been outlined recently in [123]. To this extent we consider the three major notations at work on the XT case study, namely Queuing Networks (QN), Generalized Stochastic Petri Nets (GSPN), Stochastic Process Algebras (SPA). The question we would like to answer is: which notation may be more acceptable for a software designer? and under which assumptions on the designer skill and on the software development environment is this true?

To address these questions,we discuss these model notations, by means of the modeling and the analysis of a simple case study, along two dimensions:

1. adequacy to embed and manage performance relevant aspects (e.g., workloads) at the design architectural level;
2. easiness to model, adjust and modify the architectural aspects (e.g., number and type of components) taking into account the possible feedback obtained by means of performance validation.

The aim is to highlight the suitability of such notations from a software designer perspective basing on the case study modeling and analysis. Even from the analysis of a simple case study, relevant differences can be devised among performance models along the sketched dimensions.

The comparison of the considered notations relay on the experience of six people with not a deep knowledge of any of the notations but with good software engineering principles. They used the three tools to model the case study, and they reported on the dimensions above.

In Figure 2.12 the design process of a software architecture considered in the experiment is shown; it is enriched by the feedback coming out from the performance validation. At this level, performance is estimated with low knowledge of the hardware platform where the software system will be executed. Therefore, the expected performance feedback consists of the identification of "critical" software components/subsystems whose design needs to be revised.

The primary step consists of building, from an abstract description of the software system, a software architecture model that embeds performance aspects. The output of the performance assessment on the software architecture consists of a set of indices of interest (e.g., component throughput, mean queue length). From the output analysis, some issues may come out.

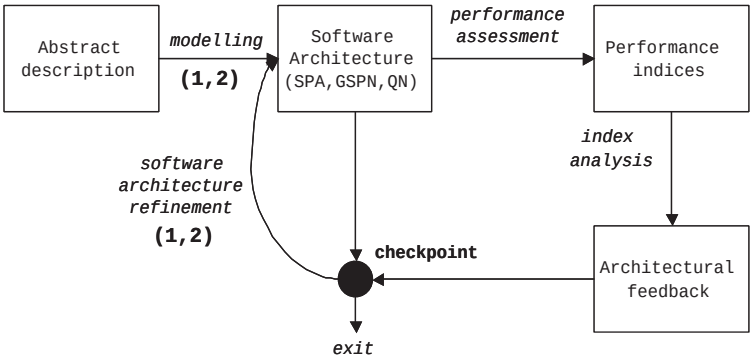


Figure 2.12: Architecture Design Process

In response to these performance issues, a range of alternative solutions can be suggested, and these constitute the *architectural feedback* of Figure 2.12. Being at the architectural level, the techniques to produce alternatives may affect either the components or the communications between them. The techniques we are interested to are the ones closely affecting the components and their workload, which essentially may fall in three categories: splitting, merging and duplication.

- *Splitting* an overloaded component in two or more components means to distribute the set of services provided from the component over a set of newly introduce components. The way of splitting such component is driven by several criteria, including the operational profile. For example, let us suppose that the component provides three services, namely s_1, s_2 and s_3, and their operational profile ([1]) is expressed by the tuple (f_1, f_2, f_3), where $f_1 \cong f_2 + f_3$. In this scenario a natural alternative could be splitting the component in two new ones, one providing just the s_1 service, and the other providing s_2 and s_3 services.

- *Merging* means to distribute the set of services provided from an underloaded component over a set of existing components. For example, let us suppose that the utilization of each component ([2]) is among the indices analyzed in the performance assessment step, and let us suppose that one

[1] By operational profile, for this specific case, we mean the distribution of frequencies of service invocations.
[2] By utilization we mean the percentage of time the component is busy.

component utilization is under a certain threshold (e.g., 40%). In this scenario a natural alternative could be distributing the services of this component over other ones whose utilizations allow such an overhead.

- *Duplication* of a component trivially means to create one or more new occurrences of the same component. This type of technique may be used every time the component can not be split, for example because it is a minimal component (i.e. it provides only one basic service) or because of some design constraints that force the software structure.

At the checkpoint in Figure 2.12 the choice among alternatives is performed considering all software product/process requirements. In practice, the developer must make a tradeoff analysis in order to decide whether and how it is worth to refine the architecture according to the performance feedback.

The numerical labels on the edges of Figure 2.12 refer to the above introduced dimensions (i.e., adequacy and easiness) and indicate the steps we concentrate to observe and compare the three model notations. The remaining steps may be affected from those dimensions as well, but for the scope of this work we are more interested in the designer viewpoint rather than to the performance evaluator one.

To compare the three notations we use the XT system introduced at the beginning of the chapter and its modeling in the considered frameworks. The QN, SPA, GSPN and models of the case study (XT) introduced in Section 2.2 are the ones reported before in Figure 2.9, Figure 2.10 and Figure 2.11, respectively.

Across the three models we use a set of common parameters, that are introduced to characterize performance aspects. In particular:

- $lambda$ represents the inverse of the user average thinking time, that is the average interval of time between a system response and the following user request;
- $mu1, mu2$ are the service rates of the *StructureBuilder* and *Marker* components, respectively;
- p models the probability of document refinement (namely the heuristic condition in Section 2).

Note that $mu1$ and $mu2$ are intrinsic parameters of the software system (i.e. they depend on the internal design of the software components), whereas $lambda$ models the types of users and p the types of documents to be processed. All of them assume the same meaning, independently of the notation adopted to model the system. So, they were used as reference values to configure the made experiments.

In the following three Sections specific considerations about the tool used as well as general thoughts about the notations are summarized. In Section 2.3.4 we show the application of the design process of Figure 2.12 by considering the three performance models separately in order to compare the three notations. Consideration on the model comparison are reported in Setion 2.3.5.

2.3.1 On Stochastic Process Algebra Modeling

We modeled our case study by using the *TIPP* stochastic process algebra [92], which fits our modeling requirements and is supported by a stable design and evaluation tool (namely *TIPPtool* [108, 93]). TIPPtool is a free downloadable software, which allows to edit the model and perform qualitative and quantitative analysis. These tasks are supported by a user-friendly GUI.

The TIPP specification of the XT system is shown in Figure 2.10. As discussed in Section 2.2.3, it includes two processes, one for each software component, i.e. the *StructureBuilder* process and the *Marker* process.

To properly model the whole system, we introduce two additional processes, *Queue1* and *Queue2*, that represent buffers to store asynchronous service requests addressed respectively to *StructureBuilder* and to *Marker*. An external user is modeled by a special process, *User*, that generates text formatting requests for the system.

The internal behavior of each process is modeled using a standard process algebra semantics. Process actions are nondeterministically composed by the *[]* operator, and each action may be guarded with a boolean expression (e.g., *[n>0]*); action sequencing is expressed by the semicolon operator.

We have simple actions (e.g., *enq1*) and rated actions, whose rates can be used as measures of either their execution times (e.g., *(markup,mu2)*) or their relative execution frequencies (e.g., *(refinement,p)*). The execution time is straightforwardly obtained from inverting the rate value (e.g., *markup* execution time is given by $1/mu2$). The same rate may be also used to transform a nondeterministic choice among actions into a stochastic one. For example, in Figure 2.10, *refinement* and *backtousers* are rated actions, but since they are placed as heads of two nondeterministic alternatives, their rates also give (besides the standard time meaning) the relative frequency of each alternative. In other words, the *refinement* alternative will be selected with a $p/(p + (100000 - p))$ frequency (while its execution time will be $1/p$), and the *backtousers* alternative will be selected with a $(100000 - p)/(p + (100000 - p))$ frequency (while its execution time will be $1/100000 - p$).

Finally, the whole system behavior is specified in the topmost part of Figure 2.10, where the basic system processes are composed by using the parallel operator $|||$ and the synchronization actions (e.g., *[enq1,arrival]*). For the sake of the example, we show in Figure 2.10 a system configuration with 3 users.

TIPP Tool Specific Considerations The particular choice of rates p and $100000 - p$ is due to the need of introducing relative frequencies over two alternatives introducing almost no further delays to their execution times. In fact, by varying the interval of values for p from 10000 to 90000 by 10000, we are able to model different stochastic distributions with negligible delays. This artifice is strictly related to the semantics underlying the TIPP Process Algebra. It can be overcome by using a different algebra/tool. For example, the $EMPA_{gr}$ Process Algebra [44] permits to associate priorities and execution frequencies to immediate actions (corresponding to the simple ones of the TIPP algebra).

Process Algebras general thoughts PA allow a natural mapping between processes and architectural components. This helps the software designer to describe the software architecture. However, in order to quantify the component behavior the PA specification requires more details on the internal behavior of the components (in terms of the actions each process performs) often not available in the early stages of a development process. With regard to component interactions, PA allow one to easily specify synchronous interactions. In order to introduce asynchronism in communication some additional structures (e.g. processes) are needed, one example is processes that model waiting queues and scheduling policies over queues. On the positive side, SPA allow the specification of a performance model with a notation that is not distant from the one used for software specification, hence attaining the feature of easiness to use from software designers.

2.3.2 On Generalized Stochastic Petri Net Modeling

We modeled the XT system by using a GSPN, and we used for performance analysis the HiQPN tool [36]. HiQPN is a free downloadable software, which (like *TIPPtool* does for SPA) permits the model definition and certain types of analysis, also supported by a user-friendly GUI.

Our GSPN model of the XT system is shown in Figure 2.11. As discussed in Sectionsec:STPN, the lower

shaded areas highlight the sub-nets modeling the *StructureBuilder* and the *Marker* components. The higher shaded areas represent the system users that provide text formatting requests to the system.

The generic $i-th$ user is made up of two places: P_{2i-1} represents a busy user (formulating a request), and P_{2i} represents an idle user (waiting for a reply). With the user busy two tokens appear in P_{2i-1}; upon a $work_i$ transition firing, one token goes into P_{2i} to move the user in a waiting state, and one token enqueues to Q_1. Q_1 models the waiting queue of the *StructureBuilder* component.

The same logic applies to the *StructureBuilder* (*Marker*) component, since a token into SB_1 (M_1) represents the idle state of the component, whereas a pair of tokens into SB_2 (M_2) models its busy state. Service requests processed by *StructureBuilder* enqueue to Q_2, which models the waiting queue of the *Marker* component. Service requests processed by *Marker* may be either refined from the same component (enqueued in Q_2) or sent back to the users as replies of accomplished service (enqueued in Q_0).

We assigned a time attribute to every transition modeling the service execution of a component. The timed transitions of the model are: $work_i$, the $i-th$ user is formatting a text; $preproc$, the *StructureBuilder* component is processing a request; $markup$, the *Marker* component is processing a request. All the remaining transitions are immediate. A probabilistic selection rule is applied to the transitions outgoing M_3, in order to model the relative frequency of the $refinement$ and $back$ alternatives.

HiQPN Tool Specific Considerations Since our intent is to consider basic modeling notations, the model of Figure 2.11 has been built using a minimal Petri Net notation that allows the modeling of performance related features, that are timed transitions and stochastic distributions on nondeterministic behaviors. However, the HiQPN tool permits to build models in extended Petri Net notations, such as Colored PN and Hierarchical Queueing PN [36].

The complexity of our model would be lower by using extended notations, but this would mean also a higher PN skill in the software designer that we want instead to keep minimal. However, the choice of adopting such a powerful PN tool (i.e., HiQPN) leaves open the possibility, in future, of considering more complex and demanding models.

Petri Nets General Thoughts With basic Petri Nets (PN) the system is modeled from a functional viewpoint, making it difficult to identify components within the model. Indeed there is no direct mapping between PN facilities (places, transitions and tokens) and software components, rather a software component may correspond to a Petri sub-net. The PN notation was originally created to model concurrent systems, so it is especially suited for modeling systems with several loosely coupled components. In cases of highly interacting components, synchronous interactions are obviously modeled, whereas asynchronous ones may require (as for Process Algebras) additional structures. For example a simple priority based scheduling on a waiting queue requires the usage of an extended PN notation, such as Colored Petri Nets. As for PA, out of the above limitations, extensions of PN (such as GSPN) allow to specify a performance model with a notation that is not distant from the one used for software specification.

2.3.3 On Queueing Network Modeling

Queueing Networks are a well-known notation for modeling system performance [111].

In Figure 2.9 we have shown the Queueing Network Model of XT system. It reflects very closely the SA description in Figure 2.1. Each component is modeled as a queued service center, while the group of users is modeled as an *Infinite Servers* center. Timing attributes are assigned in a straightforward manner to the service centers. A probabilistic selection rule is applied to the paths outgoing the *Marker* service center to model the relative frequency of the $refinement$ and $back-to-users$ alternatives.

To solve the QN model of the XT system we used the Mean Value Algorithm (MVA [111]), since the model is in product form. Our MVA implementation takes as input a text file containing all the parameters needed to the computation (such as number of users, service rates and the QN topology) and gives as output four text files, each containing values of a performance index: utilization, throughput, mean queue length and response time, respectively.

MVA specific considerations We just like to remark that the possibility of evaluating the QN model by means of the MVA algorithm is due to the simple nature of our case study, which results in a product form model.

Queuing Networks General Thoughts Queuing Networks embed an intuitive mapping between components and service centers. For software modeling at the architectural level, they also provide an immediate way to connect components, that is by means of connections among service centers. Of course, communications among service centers are all asynchronous, based on the queues associated to service centers. Being queues explicitly modeled, different scheduling policies are easy to introduce. Limitations arise in QN when synchronous interactions have to be modeled. In these cases, QN modeling has to add atypical features such as null length queues and service blocking policies for the servers, and their evaluation may become much more complex. Besides, QN are not well suited to describe details of internal behavior of the components in terms of the actions each component performs. Therefore, in late lifecycle phases, when the software modeling requires more details, QN may not be powerful enough to support software design, so resulting far from common software notations.

2.3.4 Models at Work: Results and Comments

We apply the architecture design process (shown in Figure 2.12) to the XT system. This experiment is aimed at comparing the ability of the three considered notations to embed feedback coming from performance validation. We remind that, in response to performance issues, a range of alternative solutions can be suggested from the three techniques presented in Chapter 1: splitting, merging and duplication.

We have used two different configurations of model parameters, that in this section we identify respectively as ([3]):

- **Fast StructureBuilder:** $lambda = 1.5$**,** $mu1 = 1.0$**,** $mu2 = 0.5$**;**
- **Fast Marker:** $lambda = 0.5$**,** $mu1 = 1.0$**,** $mu2 = 1.5$**.**

For both configurations we assume that other software systems represent the users of the XT system, and this assumption allows such low values for $lambda$. Each configuration has been evaluated with the probability of document refinement assuming the following values: $p \in \{0.1, 0.5, 0.9\}$.

The comparison is carried on two performance indices that are the mean queue length (i.e. average number of documents waiting to be processed) and the throughput (i.e. average number of documents processed per time unit) of each software component building up the XT system. We study the index trends while growing the number of XT users.

We like to remark that the complexities of the model evaluation processes may sensibly differ from each other, and they may also introduce some approximation errors in the index values. It is out of the scope of

[3] All the parameter values are expressed in $documents/msec$.

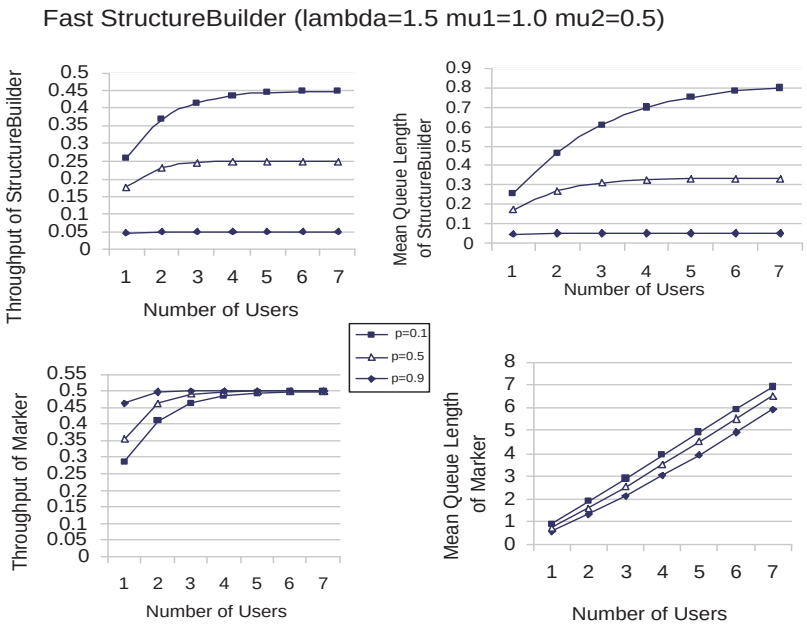


Figure 2.13: Fast StructureBuilder Performance Indices.

this study to compare the model notations along this dimension, because many other factors would enter into the picture (e.g. product forms, solution tool features).

Due to the low complexity of the XT system, full convergence has been experienced over the performance index values obtained for the considered notations. The result values are shown in the following sections. Observe that the full convergence of the results validates the three models of XT system and assesses their semantic equivalence.

Fast StructureBuilder Figure 2.13 shows the performance indices for the Fast StructureBuilder configuration. The results analysis in this case brings the straightforward consideration that the workload of the Marker component is too high. In fact, overall the document types (modeled by p), the Marker throughput saturates for a number of users that goes from 2 to 5.

Note that the Marker component is minimal that is it provides only one basic service. Therefore the only suitable feedback alternative consists of duplicating the component itself. At the checkpoint of the architecture design process (see Figure 2.12), we suppose that the developer opts to refine the architecture by duplicating the Marker component.

The refinement implementation obviously depends on the model notation (i.e. SPA, GSPN, QN). The modifications required to duplicate the Marker component are shown in the following, and all of them require the duplication of the Marker waiting queue as well.

SPA - No new process definition is introduced. The only modification concerns the *behaviour* part of the specification shown in Figure 2.10, where the subsystem composed by the *Queue2* and *Marker* processes has been duplicated. The new instance of this subsystem runs in parallel to the existing one, and it also synchronizes with the remaining part of the XT system by means of the *enq3* action. The service requests sent to the subsystem are now routed to each instance with a probability of 0.5.

GSPN - To model the new XT architecture, the subsystem composed by the Q_2 place and the Marker sub-net

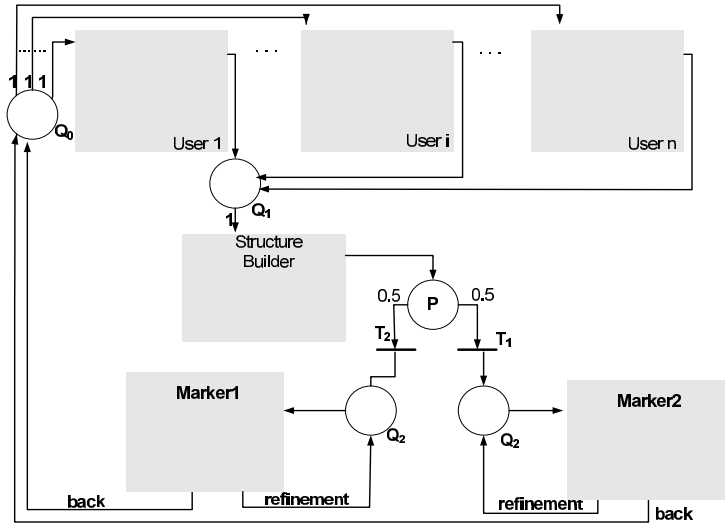


Figure 2.14: Petri Net Refinement.

(i.e. the one in the Marker shaded area in Figure 2.11) must be duplicated. As shown in Figure 2.14, in order to connect these subsystems with the remaining part of the system, a new place (namely *P*) and two new transitions (namely T_1 and T_2) have to be introduced in the GSPN model. The *Preproc* transition now goes into the new place *P* instead of going in the Q_2 place. T_1 and T_2 outgo the place *P*, and each one enters an instance of the Q_2 place. T_1 and T_2 are immediate transitions and we associate a 0.5 relative frequency to each one in order to model the same workload for each subsystem instance.

QN - The Marker service center with its waiting queue has to be duplicated. The paths outgoing the StructureBuilder service center enter with 0.5 probability each Marker instance.

New results (shown in Figure 2.15) are obtained from evaluating the new models. From a quick analysis we observe that the throughput of any Marker instance has decreased and the performance of the whole system has improved, because no evident bottleneck appears over the range of values considered for the number of users. The StructureBuilder component shows a quite high throughput which is still far from saturation.

We judge these results satisfactory for the software designer, thus exiting the process at the checkpoint.

Fast Marker Figure 2.16 shows the performance indices for the Fast Marker configuration. Even here the StructureBuilder and Marker components experience some saturation phenomenon for extreme values of p (i.e., $p = 0.1$ for the former component and $p = 0.9$ for the latter one). The designer may consider acceptable, in this case, the XT behavior since for all the intermediate p values the system seems to perform sufficiently well, thus he/she exits the architecture design process.

2.3.5 SUMMING UP: MODEL COMPARISON

In this section we discuss the lessons learned from the experiment. In Table 2.1 we show the results that come out from the general thoughts on the considered performance notations, and from the experiment report. Of course, the interpretation of the results also take into account the limitations derived from the case study we used. In fact, the case study presents some peculiar aspects, such as all asynchronous communications, small architectural size (i.e., limited number of components), and lack of external sources/sinks of requests (i.e., it is a closed system), that might promote a notation versus the other ones.

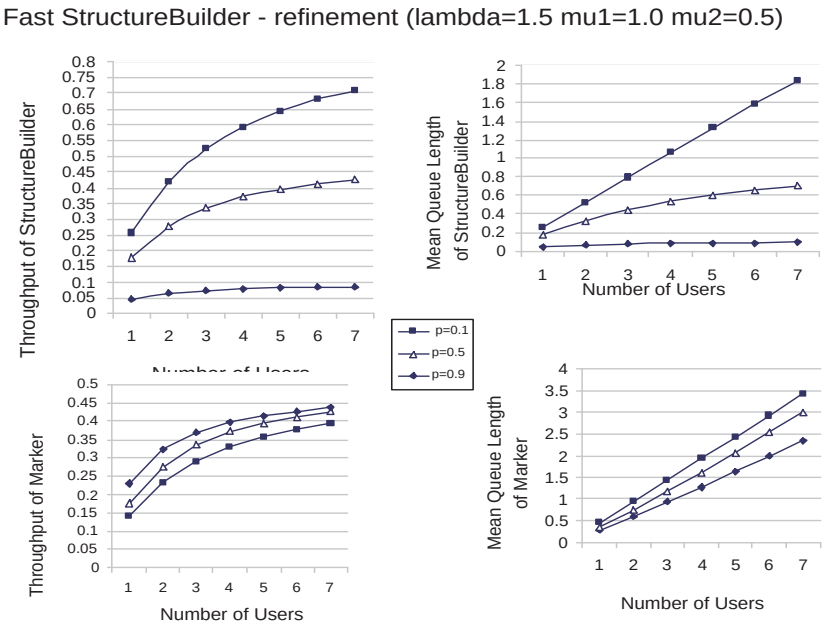


Figure 2.15: Performance indices of Fast StructureBuilder refinement.

Notation	Easiness		Adequacy
	To model	To resize	
SPA	Medium	High	Medium
GSPN	Medium	Low	Medium
QN	Medium	High	High

Table 2.1: Classification of the Considered Notations

In Section 2.3 we devised the capability of each modeling notation as a combination of ability to describe and refine typical architectural aspects and adequacy to embed and manage performance relevant aspects. In Table 2.1 these two macro-dimensions are identified as *easiness* and *adequacy*. Easiness divides into *easiness to model* which considers the difficulty to provide the initial model and its refined versions (in terms of their topology), and *easiness to resize* which considers the difficulty to change the system configuration, i.e. to change the number of instances of a component. Adequacy refers to the capability of the model to embed and manage performance aspects, e.g. to express service times. We use a coarse grain numerical scale for these dimensions, with only three ordered values: low, medium and high.

Easiness to model holds medium for QN because although they are quite distant from commonly used design notations, in the early software lifecycle phases there is a natural correspondence with architectural concepts ([4]). Easiness to model holds medium also for SPA and GSPN even though they may be considered notations familiar to software designers. Their drawback is that as soon as the system architecture becomes more complicated the complexity of the models sensibly increases.

Generalized Stochastic Petri Nets result difficult to resize. Let us consider, for example, the users issue. In order to modify the number of considered users the sub-net corresponding to the user has to be singled out, duplicated and suitably connected to the network. Stochastic Process Algebras and Queueing Networks are instead easy to resize. In SPA it is sufficient to compose new user (process) instances in parallel, and in QN

[4]This value is not set as high in order to mitigate the particular suitability of our case study to be modeled with QN, due to its asynchronous nature.

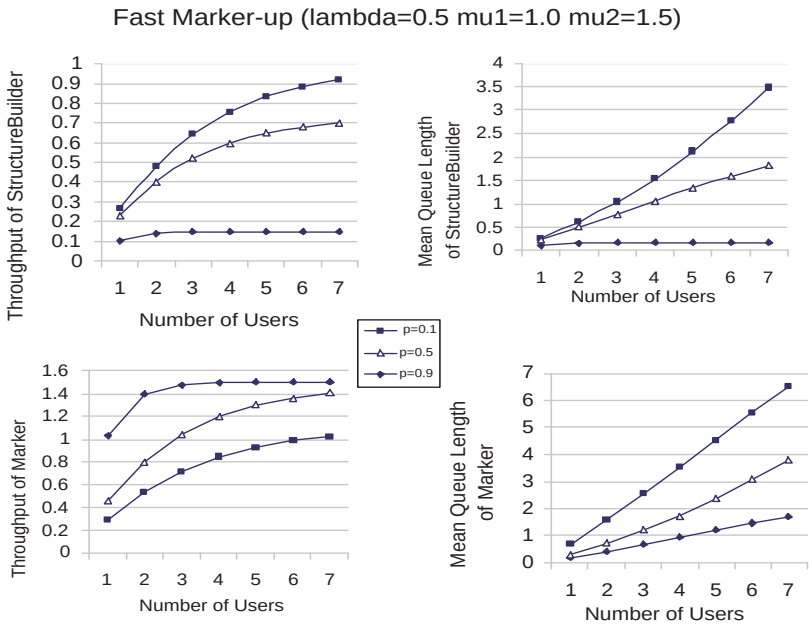


Figure 2.16: Fast Marker Performance Indices.

only an input parameter needs to be changed.

Adequacy is high for QN where performance indices, input parameters and routing probabilities are explicitly considered and managed. GSPN and SPA instead provide performance information less directly. For example, in both cases in order to represent the routing probability, notational tricks needed to be adopted: in the PA two extra actions were introduced, while in GSPN an extra place and two immediate transitions were introduced.

In summary QN seemed to behave better with respect to all the considered dimensions despite their performance analysis aptitude. This should not surprise, as sketched at the beginning of this section, since we are using the QN notation at the architectural level, where behavioral details can be hidden. The limitation of QN lays in their potential distance from the behavioral model. The more behavioral details (possibly internal to components) the software model requires, the more lack of expressiveness the QN notation suffers.

CONSIDERED DIMENSIONS VS REFINEMENT TECHNIQUES - The feedback obtained from a performance analysis consists in several suggestions of architectural refinements. In the design process we have devised three categories of architectural refinements, i.e. *splitting*, *merging* and *duplication*. From a practical viewpoint the characteristics of the adopted notation (i.e., easiness and adequacy) affect the complexity of the implementation of the architecture refinements suggested from the performance results.

Let us separately consider the three refinement techniques. Each component splitting changes the architecture topology, therefore a notation with a high value of *easiness to model* would be suited to this goal. A similar consideration can be made for merging operations, whereas it is evident that component duplications are better supported from notations with high values of *easiness to resize*. Besides, the application of any refinement technique leads to changes in performance aspects such as workload distribution and routing probabilities among components. Therefore the *adequacy* of the notation is better being high in any case.

Of course these considerations cannot affect our classification of the considered notations. In fact, before starting the architecture design process (Figure 2.12) it is virtually unattainable to predict what types of

refinements will be suggested from the performance results, so the choice of the modeling notation cannot be affected from these considerations.

FINAL CONSIDERATION - The three performance model notations (and their variants) we presented, have been and are largely used. Normally the choice of one of them, as basis of a performance validation approach, is due to several factors which do not consider the user/software-designer perspective. The aim of our experiment was to look at these model notations in order to assess their suitability to support software designers.

From the software designer point of view, QN provide the most abstract/black-box notation, thus allowing easier feedback and model comprehension, especially in a component-based software development framework. For QN the problem remains to easily obtain the model from the behavioral descriptions, especially when a certain level of behavioral detail is required. This is not a problem in the other two models once the designers use the same notations for the behavioral descriptions. Therefore in cases where performance and behavioral analyzes are both needed PA and PN notations evidently take advantage.

If we assume a standard development process, with standard software artifacts, like UML-based ones, the effort to produce the performance model from the behavioral description is comparable for all the three notations. In this context, it becomes relevant the existence of algorithms and tools that allow the creation of performance models from standard software artifacts at whatever level of detail. Several automated methodologies have been recently introduced for QN [27] but, to our knowledge, do not yet exist complete methodologies for GSPN and SPA, even if several approaches have been outlined (see Chapter 3). In order to make performance analysis widely used, future research must focus on the automatization and engineering of existing approaches which integrates standard behavioral modeling with performance model generation, and on the availability of user-friendly frameworks to carry on the analysis.

2.4 SUMMARY

In this Chapter we have briefly review the main software and performance notations. For all of them we showed an example of modeling by considered the XML Translator system. We also compared three performance notations, that are Queuing Networks, Stochastic Process Algebras and Stochastic Timed Petri Nets, through an experiment that involved six people expert in software engineering but having not experience in performance modeling and analysis.

The motivations for such an experiment come from the work in the field of software performance and software architectures we carried on in the last few years. The three performance model notations (and their variants) we considered, have been and are largely used. Normally the choice of one of them, as basis of a performance validation approach, is due to several factors which do not consider the user/software-designer perspective. The aim of our experiment was to look at these model notations in order to assess their suitability to support software designers. From the reported results we do not intend to induce general assessments on this field, due to the limitations of the case study and the experimental setting. We rather aim at setting a framework for a campaign of significant experiments in this direction.

Part I

Predictive Performance Analysis: From Software Models to Performance Models

CHAPTER 3

SOFTWARE PERFORMANCE ENGINEERING: STATE OF THE ART

Software performance aims at integrating performance analysis in the software domain. Historically the approach was to export performance modeling and measurements from the hardware domain to software systems. This was rather straightforward when considering the operating system domain, but it assumed a new dimension when the focus was directed towards software applications. Moreover, with the increase of software complexity it was recognized that software performance could not be faced locally at the code level by using optimization techniques since performance problems often result from early design choices. This awareness pushed the need to anticipate performance analysis at earlier stages in software development [139, 111].

In the research community there has been a growing interest in the subject and several approaches to early software performance predictive analysis have been proposed. Although several of these approaches have been successfully applied, however, we are still far from seeing performance prediction integrated into ordinary software development. Economic reasons (such as short time to market and special skills required) and practical reasons (specific information that is often available in the late in the software life cycle) contribute to the reluctance of the software development to adopt an engineered approach to validate the performance requirements.

In this Chapter we review the main approaches in literature which fall into the category that proposes the use of performance models to characterize the quantitative behavior of software systems. These approaches aim at filling the gap between the software development process and the performance analysis by generating performance model ready to be validated, from the software models. The review we carry out analyzes the approaches with respect to a set of relevant dimensions such as software specification, performance model, evaluation methods and level of automated support for performance prediction, in order to make a comparison of the reviewed methodologies.

The work this chapter discusses has been outlined in [27] and it is described here in details.

3.1 SOFTWARE PERFORMANCE ENGINEERING

We mean by *software performance* the process of predicting (at early phases of the life cycle) and evaluating (at the end), based on performance models, whether the software system satisfies the user performance goals.

This definition outlines two basic features of the methods we review: the existence of a performance model suitably coupled with the system software artifacts, and the evaluation of the performance model as the means to obtain software performance results. In our opinion these are necessary conditions for a method to aim at a seamless integration in ordinary software development environments since they both exhibit a good degree of automation.

From the software point of view, the software performance predictive process is based on the availability of software artifacts that describe suitable abstraction of the final software system. Requirements, software architectures, specification and design documents are examples of artifacts. Since performance is a run time attribute of a software system, performance analysis requires suitable descriptions of the software run time behavior, from now on referred also to as dynamics. For example finite state automata and message sequence charts are largely used behavioral models. As far as performance analysis is concerned, we concentrate on model-based approaches that can be applied to any phase of the software life cycle to obtain figures of merit characterizing the quantitative behavior of the system. The most used performance models are queueing networks, stochastic Petri nets, stochastic process algebra and simulation models.

Analytical methods and simulation techniques can be used to evaluate performance models in order to get performance indices. These can be classical resource-oriented indices such as throughput, utilization, response time and/or new figures of merit such as power consumption related to innovative software systems, e.g., mobile applications.

Consistent efforts have been spent in the last few years in order to fill the gap between software development and validation versus the performance requirements. Beyond every approach to the problem, two common issues can be envisaged: (i) determine the amount and the type of missing information to embed in a software design in order to enable its validation with respect to the performance requirements; (ii) introducing algorithms to translate the software description language/notation (enriched by additional information) into a model ready to be validate.

Various approaches have been recently introduced for both the issues. Two attributes appear today crucial to make any approach acceptable by the software community, that are: transparency, i.e. minimal affection on the software notation and the software process adopted (to cope with issue (i)), and effectiveness, i.e. low complexity algorithms to annotate and transform software models (to cope with issue (ii)).

All the software performance analysis approaches need additional information to carry on the performance analysis that is generally missing in the software models and has to be annotated in them. Several proposal on which information and how they should be annotated over the software models have been introduced. However, the most complete and formal one is the UML profile for Schedulability, Performance and Time (SPT) [87] by OMG. This profile defines the stereotypes and tag values the designer might use to annotated UML diagrams with the missing information needed for predictive performance analysis. We introduce such a profile in Section 3.2.

A key factor in the successful application of early performance analysis is automation. This means the availability of tools and automated support for performance prediction in the software life cycle. Although complete integrated proposals to software development and performance prediction are not yet available, several approaches provide automation of portions of it. From software specification to performance modeling to evaluation, methods and tools have been proposed to partially automate this integrated process.

From Section 3.3 to Section 3.7 we review approaches that propose a general methodology for software performance focusing on early predictive analysis. The approaches we review refer to different specification languages and performance models, and consider different tools and environments for system performance evaluation. Besides these, other proposals in the literature present ideas on model transformation through examples or case studies (e.g., [97, 107, 129, 128]). Although interesting, these approaches are still preliminary thus we will not treat them in the following comparison. Background material concerning notations to specify software dynamics and performance models is summarized in the previous chapter. The various methodologies are grouped together and discussed on the basis of the type of underlying performance model. In the same logical grouping, approaches are discussed chronologically. For each methodology we describe the software development phases in which it collects, receives or supplies the information required for performance analysis. Such information is available in Table 3.2, which is presented and commented in Section 3.8.1.

3.2 THE SCHEDULABILITY, PERFORMANCE AND TIME UML PROFILE (SPT)

In this section we introduce the performance modeling allowed by the Schedulability, Performance and Time UML Profile [87]. This is the OMG response to the transparency issue of the early performance validation. This profile defines stereotypes and tag values a software designer can use to annotate the missing information in the UML diagrams. The UML Profile for Scheduling, Performance and Time was described in [87] and has been adopted as an official OMG standard in March 2002. In general, a UML profile defines a domain-specific interpretation of UML; it might be viewed as a package of specializations of general UML concepts that capture domain-specific variations and usage patterns. Additional semantic constraints introduced by the UML profile must conform to the standard UML semantics. To specify a profile UML extensibility mechanisms (i.e. stereotypes, tagged values, constraints) are used.

The main aims of the UML Profile for Scheduling, Performance and Time (Real-time UML standard) are to identify the requirements for enabling performance and scheduling analysis of UML models. It defines standard methods to model physical time, timing specifications, timing services and logical and physical resources, concurrency and scheduling, software and hardware infrastructure and their mapping. Hence, it provides the ability to specify quantitative information directly in UML models allowing quantitative analysis and predictive modeling. This profile has been defined to facilitate the use of analysis methods and to automate the generation of analysis models and of the analysis process itself. Analysis methods considered in the profile are scheduling analysis and performance analysis based on queueing theory.

This profile is based on a performance analysis domain model that defines the main entities in the performance analysis.

In the follow, we give details on the the performance analysis domain model and we present the PAprofile package with the stereotypes and tags used to annotate the UML diagrams.

DOMAIN MODEL

In general, performance analysis is inherently instance-based and it applies to models that capture either actual or hypothetical execution runs of systems consisting of sets of instances. In Figure 3.1 a general performance model that identifies the basic abstractions and relationships used in performance analysis is depicted.

The concepts in this model are fully consistent with the conceptual framework defined in the generic resource model. This allows the performance sub-profile to take advantage of the mechanisms (e.g., modeling styles and stereotypes) that are provided for that framework. The domain model is fully based on the General Resource Modeling (GRM). The relationship of the performance modeling concepts to corresponding GRM concepts is depicted in Figure 3.2.

As the shown in the Figure 3.1, a performance context specifies one or more scenarios useful to explore various dynamic situations involving a specific set of resources and whose performance could be critical. Hence it is composed by a set of scenarios and the relative workloads, and a set of resources. The QoS values considered here are load intensity and various measures of response delay.

In performance-related models, each scenario is executed by a job class or user class with a load intensity, and these classes are either open or closed . We call such a class workload. An *open workload* has an infinite arrivals of requests which enter in the system at a given rate in some predetermined pattern (such as Poisson arrivals), and population that varies over time. Customers that have completed service leave the model. A *closed workload*, instead, has a fixed number of jobs (`population`) which cycle between

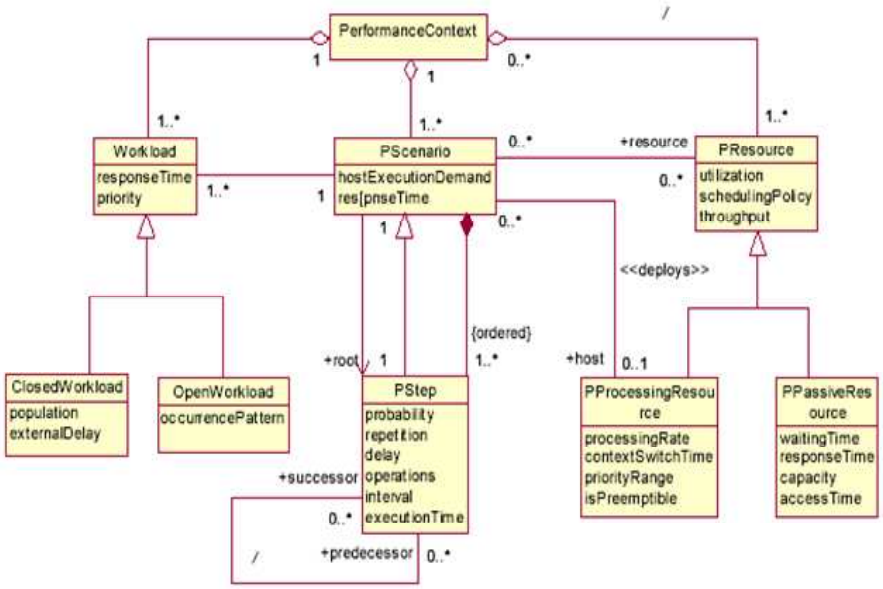


Figure 3.1: The Performance Analysis Domain Model.

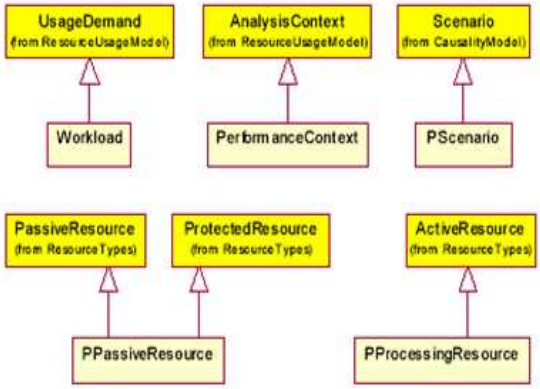


Figure 3.2: The Relationship Between Performance Concept and the General Resource Model

executing the scenario, and spending an external delay period (called a Think Time) outside the system, between the end of one response and the next request (`externalDelay`).

Attributes on the scenario entity are the `hostExecutionDemand` and the `responceTime` representing the total execution demand of the scenario on its host resource, if defined.

Scenarios are composed by (scenario) steps with predecessor-successor relationships which may include forks (a step with more `successors`), joins (a step with more `predecessors`) and loops. A step may be an elementary operation (at the finest granularity), or, it may be defined by a sub-scenario. A scenario step represent an increment in the execution of a scenario and it may use resources to perform its function. In general, a step takes finite time to execute (`executionTime` or `delay`), it may have a probability to be executed, a repetition number and an optional time interval between two repetitions. Finally a scenario step may have QoS properties and may specify the resource demands to the resources involved in the step achievement (characteristics inherited from the scenario entity).

`Resource` models an abstraction view of passive or active resource, which participates in one or more scenarios of the performance context. Resources are modeled as servers and maintain information about their `utilization`, `throughput`, and `schedulingPolicy`.

Active resources are the usual servers in performance models, and have service times. A `ProcessingResource` is an active resource, such as a processor or a storage device. It has a `processingRate` indicating its speed factor, it can be preemptive and can require some time to switch from the execution of one scenario to a different one (`contextSwitchTime`) and finally it could indicate a set of valid priorities used to define the scheduling priorities of the resource actions.

Passive resources are acquired and released during scenario. Additionally to the characteristics it inherits from the `Resource` entity, it has a `capacity` indicating the number of concurrent users and some holding time (`accessTime` and `waitingTime`).

Performance measures for a system include resource utilizations, waiting times, execution demands and response time that is the actual or wall clock time to execute a scenario step or scenario. For performance analyses to be meaningful, we have to identify the semantics of the provided numerical values for performance-related characteristics. Each measure may be: a required value, coming from the system requirements or from a performance budget based on them (e.g., a required response time for a scenario); an assumed value, based on experience (e.g., for an execution demand or an external delay); an estimated value, calculated by a performance tool and reported back into the UML model; a measured value.

STEREOTYPES AND ASSOCIATED TAGS OF THE PERFORMANCE ANALYSIS PROFILE

Based on the modeling of the performance analysis domain identified before, we here describe how the domain concepts can be represented in UML by introducing the UML extensions defined for this purpose. these extension are defined by stereotypes and tags.

For each stereotypes we report the base class they can extend, the list of the tags and the constraints they must satisfy.

$<< PAcontext >>$ This stereotype models a performance analysis context. The base classes it can extend are:

Stereotype	Base Class
$<< PAcontext >>$	Collaboration CollaborationInstanceSet ActivityGraph

This stereotype does not present any tags.

Constraints to be satisfied:

- A performance analysis context must contain at least one element that is stereotyped as a kind of step.
- A performance analysis context based on collaborations must have exactly one model element stereotyped as a workload.
- Only a top-level performance context can have a workload defined.

$<< PAclosedLoad >>$ This stereotype models a closed workload.

In the following two tables it is reported the base classes the stereotype can extend and the definition of the tags it could have, respectively.

Stereotype	Base Class	Tags
$<< PAclosedLoad >>$	Message Stimulus Action State SubactivityState Action ActionExecution Operation Method Reception	PArespTime PApriority PApopulation PAextDelay

Tag	Type	Multiplicity	Domain AttributeName
PArespTime	PAperfValue	[0..*]	Workload::responseTime
PApriority	Integer	[0..1]	Workload::priority
PApopulation	Integer	[0..1]	ClosedWorkload::population
PAextDelay	PAperfValue	[0..1]	ClosedWorkload::externalDelay

Constraint: This stereotype can only be applied to be the first step in a performance context..

$<< PAopenLoad >>$ This stereotype models an open workload.

In the following two tables it is reported the base classes the stereotype can extend and the definition of the tags it could have, respectively.

Stereotype	Base Class	Tags
$<< PAopenLoad >>$	Message Stimulus Action State SubactivityState Action ActionExecution Operation Method Reception	PArespTime PApriority PAoccurrence

Tag	Type	Multiplicity	Domain AttributeName
PArespTime	PAperfValue	[0..*]	Workload::responseTime
PApriority	Integer	[0..1]	Workload::priority
PAoccurrence	RTarrivalPattern	[0..1]	OpenWorkload::population

Constraint: This stereotype can only be applied to be the first step in a performance context.

$<< PAhost >>$ This stereotype models a processing resource.

In the following two tables it is reported the base classes the stereotype can extend and the definition of the tags it could have, respectively.

Constraint: This stereotype can only be applied to be the first step in a performance context.

Stereotype	Base Class	Tags
$<< PAhost >>$	Classifier Node ClassifierRole Instance Partition	PAutilizzation PAschdPolicy PArate PActxtSwT PAprioRange PApreemptable PAthroughput

Tag	Type	Multiplicity	Domain AttributeName
PAutilizzation	Real	[0..*]	Resource::utilization
PAschdPolicy	Enumeration: {FIFO,HOL,PR,PS,PPS,LIFO}	[0..1]	ProcessingResource::schedulingPolicy
PArate	Real	[0..1]	ProcessingResource::processingRate
PActxtSwT	PAperfValue	[0..1]	ProcessingResource::contextSwitchTime
PAprioRange	Integer range	[0..1]	ProcessingResource::priorityRange
PApreemptable	Boolean	[0..1]	ProcessingResource::isPreemptable
PAthroughput	Real	[0..1]	Resource::throughput

$<< PAresource >>$ This stereotype models a passive resource.

In the following two tables it is reported the base classes the stereotype can extend and the definition of the tags it could have, respectively.

Stereotype	Base Class	Tags
$<< PAresource >>$	Classifier Node ClassifierRole Instance Partition	PAutilizzation PAschdPolicy PAcapacity PAaxTime PArespTime PAwaitTime PAthroughput

Tag	Type	Multiplicity	Domain AttributeName
PAutilizzation	Real	[0..*]	Resource::utilization
PAschdPolicy	Enumeration: {FIFO,Priority}	[0..1]	PassiveResource::schedulingPolicy
PAcapacity	Integer	[0..1]	PassiveResource::capacity
PAaxTime	PAperfValue	[0..n]	PassiveResource::accessTime
PArespTime	PAperfValue	[0..n]	PassiveResource::responceTime
PAwaitTime	PAperfValue	[0..n]	PassiveResource::waitTime
PAthroughput	Real	[0..1]	Resource::throughput

Constraint: This stereotype can only be applied to be the first step in a performance context.

$<< PAstep >>$ This stereotype models a passive resource step in a performance analysis scenario.

In the following two tables it is reported the base classes the stereotype can extend and the definition of the tags it could have, respectively.

Stereotype	Base Class	Tags
$<< PAstep >>$	Message Stimulus Action State SubActivitySate	PAdemand PArespTime PAprob PArep PAdelay PAexOp PAinterval

Tag	Type	Multiplicity	Domain Attribute Name
PAdemand	PAperfValue	[0..*]	Step::hostExecutionDemand
PArespTime	PAperfValue	[0..*]	Step::responceTime
PAprob	Real	[0..1]	Step::probability
PArep	Integer	[0..1]	Step::repetition
PAdelay	PAperfValue	[0..*]	Step::delay
PAextOp	PAextOpValue	[0..*]	Step::operations
PAinterval	PAperfValue	[0..*]	Step::interval

3.3 Queueing Network Based Methodologies

In this section we consider a set of methodologies which propose transformation techniques to derive Queueing Network (QN) based models —possibly Extended QN (EQN) or Layered QN (LQN)— from Software Architecture (SA) specifications. Some of the proposed methods are based on the Software Performance Engineering (SPE) methodology introduced by Smith in her pioneer work [139].

3.3.1 Methodologies Based on the SPE Approach

The SPE methodology [139, 145] was the first comprehensive approach to the integration of performance analysis into the software development process, from the earliest stages to the end. It uses two models: the software execution model and the system execution model. The first takes the form of Execution Graphs (EG) that represent the software execution behavior; the second is based on QN models and represents the system platform, including hardware and software components. The analysis of the software model gives information about the resource requirements of the software system. The obtained results, together with information about the hardware devices, are the input parameters of the system execution model, which represents the model of the whole software/hardware system.

M1: The first approach based on the SPE methodology was proposed by Williams and Smith in [159, 145]. They apply the SPE methodology to evaluate the performance characteristics of a software architecture specified by using the Unified Modeling Language (UML) diagrams, that is Class and Deployment diagrams, and Sequence Diagrams enriched with ITU Message Sequence Chart (MSC) features. The emphasis is in the construction and analysis of the software execution model, which is considered the target model of the specified SA and is obtained from the Sequence Diagrams. The Class and Deployment diagrams contribute to complete the description of the SA, but are not involved in the transformation process. This approach was initially proposed in [142], which describes a case study and makes use of the tool SPE•ED for performance evaluation. In [161] the approach is embedded into a general method called PASA (Performance Assessment of Software Architectures) which aims at giving guidelines and methods to determine whether a SA can meet the required performance objectives.

SPE•ED is a performance modeling tool specifically designed to support the SPE methodology. Users identify the key scenarios, describe their processing steps by means of EG, and specify the number of software resource requests for each step. A performance specialist provides overhead

specifications, namely the computer service requirements (e.g., CPU, I/O) for the software resource requests. SPE•ED automatically combines the software models and generates a QN model, which can be solved by using a combination of analytical and simulation model solutions. SPE•ED evaluates the end-to-end response time, the elapsed time for each processing step, the device utilization and the time spent at each computer device for each processing step.

M2: An extension of the previous approach was developed by Cortellessa and Mirandola in [66]. The proposed methodology, called PRIMA-UML, makes use of information from different UML diagrams to incrementally generate a performance model representing the specified system. This model generation technique was initially proposed in [65]. The technique considered OMT-based object-oriented specification of systems (Class diagrams, Interaction diagrams and State Transition diagrams) and defines an intermediate model, called Actor-Event Graph, between the specification and the performance model.

In PRIMA-UML, SA are specified by using Deployment, Sequence, and Use Case diagrams. The software execution model is derived from the Use Case and Sequence diagrams, and the system execution model from the Deployment diagram. Moreover, the Deployment diagram allows for the tailoring of the software model with respect to information concerning the overhead delay due to the communication between software components. Both Use Case and Deployment diagrams are enriched with performance annotations concerning workload distribution and parameters of hardware devices, respectively.

M3: In [86] the PRIMA-UML methodology was extended in order to cope with the case of mobile SA by enhancing its UML description to model the mobility-based paradigms. The approach generates the corresponding software and system execution models allowing the designer to evaluate the convenience of introducing logical mobility with respect to communication and computation costs. The authors define extensions of EG and EQN to model the uncertainty about the possible adoption of code mobility.

M4: In [60] Cortellessa et at. focus on the derivation of a LQN model from a SA specified by means of a Class diagram and a set of Sequence diagrams, generated by using standard CASE tools. The approach clearly identifies all the supplementary information needed by the method to carry out the LQN derivation. These include platform data, e.g., configuration, resource capacity, and the operational profile which includes the user workload data. Moreover, in case of distributed software, other input documents are the software module architecture, the client/server structure, and the module-platform mapping.

The method generates intermediate models in order to produce a global precedence graph which identifies the execution flow and the interconnections among system components. This graph, together with the workload data, the software module architecture and the client/server structure, are used to derive the extended EG. The target LQN model is generated from the extended EG and the resource capacity data.

Other SPE-based approaches have been proposed by Petriu and coauthors. Since they are also based on patterns we describe them in the next subsection.

3.3.2 Architectural Pattern Based Methodologies

The approaches described hereafter consider specific classes of systems, identified by architectural patterns in order to derive their corresponding performance models. Architectural patterns characterize frequently used architectural solutions. Each pattern is described by its structure (what are the components) and its behavior (how they interact).

A first approach based on architectural patterns for client/server systems is presented in [84, 85], where Gomaa and Menascé investigate the design and performance modeling of component interconnection patterns,

which define and encapsulate the way client and server components of SA communicate with each other via connectors. The authors, rather than proposing a transformational methodology, describe the pattern through Class and Collaboration diagrams and directly show their corresponding EQN models. It is worth mentioning that the approach has been the first to deal with component-based SA.

M5: In [125, 124, 88] Petriu et al. propose three conceptually similar approaches where SA are described by means of architectural patterns (such as pipe and filters, client/server, broker, layers, critical section and master-slave) whose structure is specified by UML Collaboration diagrams and whose behavior is described by Sequence or Activity diagrams.

The three approaches follow the SPE methodology and propose systematic methods of building LQN models of complex SA based on combinations of the considered patterns. Such model transformation methods are based on graph transformation techniques. We now discuss the details of the various approaches.

- In [125] SA are specified by using UML Collaboration, Sequence, Deployment and Use Case diagrams. Sequence diagrams are used to obtain the software execution model represented as a UML Activity diagram. The UML Collaboration diagrams are used to obtain the system execution model, i.e., a LQN model. Use Case diagrams provide information on the workloads and Deployment diagrams allow for the allocation of software components to hardware sites. The approach generates the software and system execution models by applying graph transformation techniques, automatically performed by a general-purpose graph rewriting tool.
- In [124] the authors extend the previous work by using the UML performance profile [87] to add performance annotations to the input models, and by accepting UML models expressed in XML notation as input. Moreover the general-purpose graph rewriting tool for the automatic construction of the LQN model, has been substituted by an ad-hoc graph transformation implemented in Java.
- The third approach proposed in [88] uses the eXtensible Stylesheet Language Transformations (XSLT), to carry out the graph transformation step. XSLT is a language for transforming a source document expressed in a tree format (which usually represents the information in a XML file) into a target document expressed in a tree format. The input contains UML models in XML format, according to the standard XML Metadata Interchange (XMI) [121], and the output is a tree representing the corresponding LQN model. The resulting LQN model can be in turn analyzed by existing LQN solvers after an appropriate translation into textual format.

M6: Menascé and Gomaa presented in [116, 117] an approach to the design and performance analysis of client/server systems. It is based on CLISSPE (CLIent/Server Software Performance Evaluation), a language for the software performance engineering of client/server applications [115]. A CLISSPE specification is composed of a *declaration section* containing clients and client types, servers and server types, database tables and other similar information, a *mapping section* allocating clients and servers to networks, assigning transactions to clients, etc., and a *transaction specification section* describing the system behavior.

The CLISSPE system provides a compiler which generates the QN model, estimates some model parameters, and provides a performance model solver. For the specification of a client/server system the methodology uses UML Use Case diagrams to specify the functional requirements, Class diagrams for the structural model and Collaboration diagrams for the behavioral model. From these diagrams the methodology derives a CLISSPE specification. A relational database is automatically derived from the Class diagram. The CLISSPE transaction specification section is derived (not yet automatically) from Use Case diagrams and collaboration diagrams, and references the relational database derived from the structural model. Moreover, the CLISSPE system uses information on the database accesses (of the transaction specification section) to automatically compute service demands and estimate CPU and I/O costs.

In order to complete the CLISSPE declaration and mapping sections, the methodology requires the specification of the client/server architecture with the related software/hardware mapping annotated with the performance characteristics such as processor speeds, router latencies, etc.

3.3.3 METHODOLOGIES BASED ON TRACE-ANALYSIS

In this subsection we consider approaches based on the generation and analysis of traces (sequence of events/actions) from dynamic description of the software system.

M7: The first approach we consider proposes an algorithm that automatically generates QN models from software architecture specifications described by means of MSC [18] or by means of Labeled Transition Systems (LTS) [19]. A key point of this approach is the assumption that the SA dynamics, described by MSC or LTS, is the only available system knowledge. This specification models the interactions among components. This allows the methodology to be applied in situations where information concerning the system implementation or deployment are not yet available.

The approach analyzes the SA dynamic specification in terms of the execution traces (sequences of events exchanged between components) it defines, in order to single out the real degree of parallelism among components and their dynamic dependencies. Only the components that actually can behave concurrently will correspond to service centers of the QN model of the SA description. In the QN model the dynamic activation of software components and connectors is modeled by customers' service time. Concurrent component activation of software components is represented by customers' concurrent activity that possibly compete for the use of shared resources. Synchronous communication of concurrent software components are modeled by service centers with finite or zero capacity queues and an appropriate blocking protocol. The analysis of the QN model of the SA leads to the evaluation of a set of performance indices, like throughput and mean response time, that are then interpreted at the SA level. The model parameter instantiations correspond to potential implementation scenarios. Performance results are used to provide insights on how to carry out refinements to the SA design.

M8: Woodside et al. describe in [162] a methodology to automatically derive a LQN model from a commercial software design environment called ObjecTime Developer [10] by means of an intermediate prototype tool called PAMB (Performance Analysis Model Builder). The application domain of the methodology is real-time interactive software and it encompasses the whole development cycle, from the design stage to the final product.

ObjecTime Developer allows the designer to describe a set of communicating actor processes, each controlled by a state machine, plus data objects and protocols for communications. It is possible to "execute" the design over a scenario by inserting events, stepping through the state machines, and executing the defined actions. Moreover, the tool can generate code from the system design. The approach in [162] takes advantage of such code generation and scenario execution capabilities for model-building. The prototype tool PAMB, integrated with ObjecTime Developer, keeps track of the execution traces, and captures the resource demands obtained by executing the generated code in different execution platforms. Essentially, the trace analysis allows the building of the various LQN sub-models (one for each scenario) which are then merged into a global model, while the resource demand data provide the model parameters. After solving the model through an associated model solver, the PAMB environment reports the performance results by means of performance annotated MSC and graphs of predictions.

M9: More recently Woodside et al. present in [126] an approach to performance analysis from requirements to architectural phases of the software life cycle. This approach derives LQN performance models from system scenarios described by means of Use Case Maps (UCM).

The UCM specification is enriched with performance annotation. The approach defines where and how the diagrams have to be annotated and the default values to be used when performance data are missing.

The derivation of LQN models from annotated UCM is quite direct, due to the close correspondence between UCM and LQN basic elements. The derivation is defined on a path by path basis, starting from UCM start points. The identification of the component interaction types, however, is quite complex since the UCM notation does not allow the specification of synchronous, asynchronous and

forwarding communication mechanisms. The approach describes an algorithm which derives such information from the UCM paths, by maintaining the unresolved message history while traversing a UCM path.

The UCM2LQN tool automatizes the methodology and it has has been integrated into a general framework called UCM Navigator. This allows the creation and editing of UCM, supports scenario definitions, generates LQN models, and exports UCM specifications as XML files.

3.3.4 UML-Based Modeling Approach

In this subsection we consider efforts that have been pursued entirely in the UML framework in order to make performance analysis possible by starting from UML software descriptions.

In this section should be included the SPT Profile that we already presented in Section 3.2. The reader can refer to such a section for more details on it.

M10: The approach introduced by Kähkipuro in [102] is quite different from the others described in this section. The proposed framework consists of three different performance model representations and of the mappings among them. The starting representation is based on UML. The key point of this approach is the use of UML as a new way to represent performance models. This approach proposes a UML-based performance modeling notation (i.e., a notation compatible with the UML design description) which can be used in the UML specification of a system, in order to specify performance elements besides the pure functional ones. The UML representation is then automatically mapped into a textual representation, which retains only the performance aspects of the system, and it is further translated into an extended QN model representing the simultaneous resource possessions, synchronous resource invocations and recursive accesses to resources. Such a model can be solved by using approximate or simulation techniques, and the results can be translated back to the textual representation and the UML diagrams, thus realizing a feedback mechanism. The approach has been partially implemented in a prototype tool called OAT (Object-oriented performance modeling and Analysis Tool).

Note that this approach does not really propose a transformation methodology from a SA specification to a performance model, since the three steps (or representations) of the framework just give three equivalent views of the modeled system with respect to performance. In this approach a designer must have performance skills, besides UML knowledge, to produce a correct diagram. However, the approach lifts up the transformation step to the specification level. In fact, in order to obtain a real model transformation methodology, it would be sufficient to add a further level on top of the whole framework in order to produce extended UML diagrams annotated with performance information out of purely functional oriented UML diagrams.

3.3.5 Approaches for Component-Based Software Systems

Component Based Software Engineering (CBSE) is the emerging software development process that aims at maximizing the re-use of separately developed components. It promise to yield cheaper and higher quality assemble large systems. The basic principle here is that individual components with own properties are released once, and the properties of the composed software system can be obtained from the properties of all involved in a compositional way.

In this section we report two software performance approach for component-based software system.

M11: Mirandola and Bertolino in [45, 46] propose an automated compositional approach for component-based performance engineering called CB-SPE. This approach is applied at two level, namely the

component layer and the *application layer* managed by the component development and by the system assembler respectively. At the component layer the goal is to obtain components with predicted performance properties (to be used later at the application layer) that are explicitly declared in the component interfaces. This implies that the component developer has to introduce and validate the performance requirements of the component considered in isolation. Such analysis must be platform independent. Later, at the application level, it will be instantiated on a specific platform. The component developers is expected to fill a "component repository" with components whose interface explicitly declare the component predicted performance properties. The performance analysis of the assembled system is obtained by combine the performance properties of the pre-selected components, instantiated over a specific hardware platform.

The approach uses the UML Sequence diagrams to model the SA behavior in terms of component interactions and the UML Deployment Diagram to describe the specific hardware platform where the application will run. These diagrams are annotated with performance information by means of UML SPT profile. Such an information is extracted from the component repository previously filled. The approach, according to the SPE principles, provides two different models: a stand-alone performance model namely Execution Graph (derived from the sequence diagrams) and a contention based performance model namely a QN model (from the deployment diagram). The authors implemented such a methodology in a tool, namely the CB-SPE Tool.

M12: Woodside et al. in [164, 69] define an approach to predict the performance of component-based applications that assumes the existence of performance LQN sub-models for the considered software components. Such performance sub-models are stored in a library and properly combined to generate the performance model of the whole application. The way the software components are assembled to compone the final software system is described by the UML 2.0 component diagram [148]. The performance sub-models of the software components are combined all together as the component diagram specifies in order to obtain the LQN model of the whole software system. This step is supported by a new XML-based language, namely Component-Based Modeling Language that Woodside et al. define in [69]. Such language specifies new features of the Layered Queuing Modeling language that ease the hierarchical performance modeling of components based on the UML component diagram.

The author implemented the component performance sub-model assembly in a tool called Component Assembler.

3.4 Process Algebras Based Approaches

M13: Several Stochastic extensions of Process Algebras (SPA) have been proposed in order to describe and analyze both functional and performance properties of software specifications within the same framework. Among these we consider TIPP (TIme Processes and Performability evaluation) [93], EMPA (Extended Markovian Process Algebra) [44, 42] and PEPA (Performance Evaluation Process Algebra) [94, 82] which are all supported by appropriate tools (the TIPP tool, PEPA Workbench and Two Towers for EMPA).

All of them associate exponentially distributed random variables to actions, and provide the generation of a Markov chain out of the semantic model (LTS enriched with time information) of a system. Beside exponential actions, also passive and immediate actions are considered. The main differences among PEPA, EMPA and TIPP concern the definition of the rate of a joint activity which arises when two components cooperate or synchronize, and the use of immediate actions. Different choices on these basic issues induce differences to the expressive power of the resulting language. We refer to [91] for an accurate discussion about this topic.

The advantage of using PEPA, EMPA or TIPP for software performance is that they allow the integration of functional and non-functional aspects and provide a unique reference model for software specification and performance. However, from the performance evaluation viewpoint, the analysis usually refers to the numerical solution of the underlying Markov chain which can easily lead to

numerical problems due to the state space explosion. On the software side, the software designer is required to be able to specify the software system using process algebras and to associate the appropriate performance parameters (i.e., activity rates) to actions.

In order to overcome this last drawback, Pooley describes in [128] some preliminary ideas on the derivation of SPA models from UML diagrams. The starting point is the specification of a SA by means of a combined diagram consisting of a Collaboration diagram with embedded Statecharts of all the collaborating objects. The idea is then to produce a SPA description out of each Statechart and then to combine the obtained descriptions into a unique model.

Balsamo et al. introduced in [24] a SPA based Architectural Description Language (ADL) called Æmilia (an earlier version called ÆMPA can be found in [42]), whose semantics is given in terms of EMPA specifications. Æmilia aims at facilitating the designer in the process algebra-based specification of software architectures, by means of syntactic constructs for the description of architectural components and connections. Æmilia is also equipped with checks for the detection of possible architectural mismatches. Moreover for Æmilia specifications a translation into QN models has been proposed in order to take advantage of the orthogonal strengths of the two formalisms: formal techniques for the verification of functional properties for Æmilia (SPA in general), and efficient performance analysis for QN.

3.5 Petri Net Based Approaches

M14: Like SPA, Stochastic Petri Nets (SPN) are usually proposed as a unifying formal specification framework, allowing the analysis of both functional and non-functional properties of systems. There are a lot of stochastic Petri net frameworks (e.g., GreatSPN, HiQPN, DSPNExpress 2000, and many others which can be found in [4]) allowing the specification and the functional/quantitative analysis of a Petri Net model.

Recently, approaches to the integration of UML specifications and Petri Nets have been proposed [107, 38]. In [107] the authors present some ideas on the derivation of a GSPN from Use Cases diagrams and combined Collaboration and Statecharts diagrams. The idea is to translate the Statechart associated with each object of the Collaboration diagram into a GSPN (where states and transitions of the Statechart become places and transitions in the net, respectively), and then to combine the various nets into a unique model.

M15: In [38] the authors propose a systematic translation of Statecharts and Sequence Diagrams into GSPN. The approach consists of translating the two type of diagrams into two separate labelled GSPN. The translation of a Statechart gives rise to one labeled GSPN per unit where a unit is a state with all its outgoing transitions. The resulting nets are then composed over places with equal labels in order to obtain a complete model. Similarly, the translation of a Sequence diagram consists of modeling each message with a labeled GSPN subsystem and then composing such subsystems by taking into account the causal relationship between messages belonging to the same interaction, and defining the initial marking of the resulting net. The final model is obtained by building a GSPN model by means of two composing techniques. In [113] the authors extend the methodology by using the UML Activity diagrams to describe activities performed by the system usually expressed in a statechart as *doActivity*. Again, the activity diagrams are translated in labeled GSPN. Such targets model are then combined with the labeled GSPN modeling the statecharts that use the *doActivity* modelled by the activity diagrams.

The authors implemented a java module to implement the GSPN generation.

3.6 Methodologies Based on Simulation Methods

We shall now consider three approaches based on simulation models. They use simulation packages in order to define a simulation model whose structure and input parameters are derived from UML diagrams.

M16: The first approach, proposed by de Miguel et al. in [119], focuses on real time systems, and proposes extensions to UML diagrams to express temporal requirements and resource usage. The extension is based on the use of stereotypes, tagged values and stereotyped constraints. SA are specified using the extended UML diagrams without restrictions on the type of diagrams to be used. Such diagrams are then used as input for the automatic generation of the corresponding scheduling and simulation models via the Analysis Model Generator (AMG) and Simulation Model Generator (SMG), respectively. In particular, SMG generates OPNET models [12], by first generating one sub-model for each application element and then combining the obtained sub-models into a unique simulation model. The approach provides a feedback mechanism: after the model has been analyzed and simulated, some results are included into the tagged values of the original UML diagrams. This is a relevant feature, which helps the SA designer in interpreting the feedback from the performance evaluation results.

M17: The second approach, proposed by Arief and Speirs in [21], presents a simulation framework named Simulation Modeling Language (SimML) to automatically generate a simulation Java program (by means of the JavaSim tool [3]) from the UML specification of a system that realizes a process oriented simulation model. SimML allows the user to draw Class and Sequence diagrams and to specify the information needed for the automatic generation of the simulation model. The approach proposes a XML translation of the UML models, in order to store the information about the design and the simulation data in a structured way.

M18: The third approach, proposed by Balsamo and Marzolla in [28, 45], generates a process-oriented simulation model of a UML software specification describing the software architecture of the system. The used UML diagrams are Use Case, Activity and Deployment diagrams. The diagrams are annotated according to a sub-set of the UML SPT profile [87]. Such annotations are used to parameterize the simulation model. The approach define an (almost) one-to-one correspondence between the entities expressed in the UML model and the entities or processes in the simulation model. This correspondence allows easy report of the performance results back to the software specification that are annotated by means of UML SPT tag values. The approach has been implemented in the prototype tool UML-Ψ (UML Performance SImulator). In [30] they extended the previous approach to deal with mobile systems. Here the main contribution is the modelling of the mobility of a user. They model the physical mobility of a user by means of UML activity diagrams that they called ”high-level” activity diagrams. Each mobility user has associated a set of such ”high-level” activity diagrams describing their physical mobility behavior.

3.7 Methodology Based on Stochastic Processes

All the approaches presented in this section so far consider QN, SPA, SPN or simulation based performance models. In this subsection we describe an approach which considers generalized semi-Markov processes, i.e., stochastic processes where general distributions (and not only memoryless ones) are allowed and the Markovian property is only partially fulfilled.

M19: The approach in [112] proposes a direct and automatic translation of system specifications given by UML State diagrams or Activity diagrams, into a corresponding discrete-event stochastic system, namely a generalized semi-Markov process. The first step of the approach consists of introducing

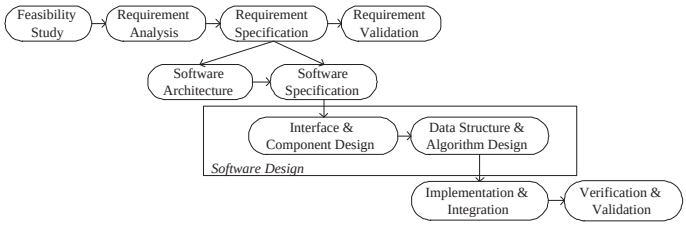


Figure 3.3: Generic software life cycle model.

extensions to UML State diagrams and Activity diagrams to associate events with exponentially distributed durations and deterministic delays. The enhanced UML diagrams are then mapped onto a generalized semi-Markov process by using an efficient algorithm for the state space generation.

The approach has been implemented using the tool DSPNexpress 2000 (information in [4]) which supports the quantitative analysis of discrete-event stochastic processes. DSPNexpress imports UML diagrams from some commercial UML design package, and adds the timing information by using a graphical user interface.

The approach has two main drawbacks, namely, the use of non standard UML notation for the additional timing information and the possible state space explosion in the generation of the state transition graph out of the UML diagrams.

3.8 Classification of the Existing Approaches

In this section we classify and compare the reviewed methodologies. We focus on the integration of performance analysis at the earliest stages of the software life cycle, namely software design, software architecture and software specification. We will review the most important approaches in the general perspective of how each one integrates into the software life cycle.

To carry out the classification and the comparison we consider a generic model of software life cycle as presented in Figure 3.3.

We identify the most relevant phases in the software life cycle. Since the approaches we consider aim at addressing performance issues early on in the software development cycle, our model is detailed with respect to the requirement and architectural phases. Moreover since performance is a system run time attribute we will focus on dynamic aspects of the software models used in the different phases.

The various approaches differ with respect to several dimensions. In particular we consider the software dynamics model, the performance model, the phase of the software development in which the analysis is carried out, the level of detail of the additional information needed for the analysis, and the software architecture features of the system under analysis, e.g., specific architectural patterns such as client-server, and others. All these dimensions are then synthesized through the three following indicators: the *integration level of the software model with the performance model*, the *level of integration of performance analysis in the software life cycle* and the *methodology automation degree*, as shown in Figure 3.4. We use them to classify the methodologies as reported in Section 3.8.1. The level of integration of the software and of the performance models ranges from syntactically related models to semantically related models to a unique comprehensive model. This integration level qualifies the mapping that the methodologies define to relate the software design artifacts with the performance model. A high level of integration means that the performance model has a strong semantic correspondence with the software model. Syntactically related models permit the definition of a syntax driven translation from the syntactic specification of software artifacts to

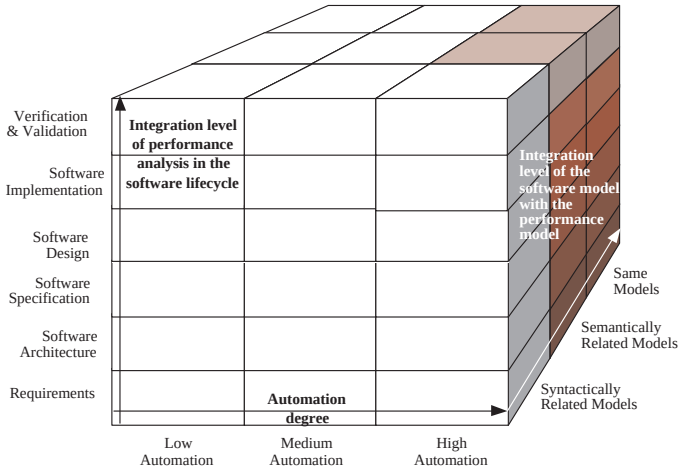


Figure 3.4: Classification Dimensions of Software Performance Approaches.

the performance model. The unique comprehensive model allows the integration of behavioral and performance analysis in the same conceptual framework. With the level of integration of performance analysis in the software life cycle we identify the precise phase at which the analysis can be carried out. In the context we are dealing with, the earlier the prediction process can be applied the better the integration with the development process is obtained. For each methodology we also single out the software development phases in which the specific information required to perform the analysis is collected, received or supplied. This information is synthesized in Table 3.2, discussed in Section 3.8.1, and it is used to explicitly state the requirements that each methodology puts to software designers in terms of additional information required for performance analysis. It is worth noting that these requirements can be very demanding thus restricting the general applicability of the methodology. We associate each required information to the phase of the software life cycle where it would be naturally available. There exist methods that assume the availability of this information at earlier phases of the software life cycle in order to carry out the predictive analysis. In this case their underlying assumption is that this information is available somehow, for example through an operational profile extracted from similar systems or by assuming hypothetical implementation scenarios. We will mention this kind of assumption for each presented methodology. The last dimension refers to the degree of automation that the various approaches can support. It indicates the potentiality of automation and characterizes the maturity of the approach and the generality of its applicability.

The classification schema illustrated in Figure 3.4 is inspired by Enslow's model of distribution [76].

The ideal methodologies fall at the bottom of the two rightmost columns, since they should have high integration of the software model with the performance model, high level of integration of performance analysis with the life cycle, i.e., from the very beginning, and high degree of automation.

3.8.1 Comparison and Classification

In this section we provide a synthesis of the methodologies previously surveyed, in light of the several features outlined above. The Table 3.1 summarizes all the methodologies and their characteristics. Each row refers to a methodology. The first column indicates the methodology label that will be used in Figure 3.5 while the second column refers to the section presenting the methodology. The third and fourth columns indicate the behavioral and performance models, respectively. The former is the starting point of the methodology and the latter represents the target model for the analysis. The fifth and sixth columns indicate potential constraints implied by the methodology. These can be related to the software system

architecture or to a specific application domain, e.g., real time systems. The life cycle phase column reports the first software development phase in which the methodology can be applied. The eighth, ninth and tenth columns are quality attributes that we derive from the study of the various methodologies. Information level represents the amount of information that has to be provided in order to make the methodology applicable. This encompasses information like operational profile, resource workloads, as detailed in Table 3.2. Feedback indicates that a methodology has explicitly addressed the issue of providing some sort of feedback from performance analysis to the software designer. This means, for example, that from a bad throughput figure it is easy to single out which are the software components and/or interactions responsible. Automation degree refers to the suitability for automation of a methodology. It is worth noting that this does not indicate the current achieved automation, which is instead considered in the last column. The tool column mentions the tools that support the automated portion of the methodology, if any.

Note that there is a dependency between the information level and the life cycle phase. The more information needed, the later in the software development cycle the analysis is performed. This means that some methodologies, although starting from abstract software descriptions, are applicable only from the design level onward, as far as the life cycle phase is concerned. The kind of additional performance information required by each methodology can be found in the Table 3.2. Note that one could also imagine that this information do somehow exist, e.g., can be predicted by using execution scenarios. However, for classification purposes we consider the development phase in which this kind of information would naturally be available and relate the methodology to that phase. More precisely, the Table 3.2 depicts the various phases of the software life cycle in the columns and the considered methodologies in the rows. Each row is relative to one methodology and reports the information needed for performance analysis purposes, with respect to the significant phases of the software life cycle.

Looking at Table 3.1 we can make the following observations.

- Most of the methodologies make use of UML or UML-like formalisms to describe behavioral models. This indicates a general tendency which is driven by the need to integrate with standard practice development environments.
- QN are the preferred performance models. This depends on two factors: the abstraction level of the QN formalism which makes it suitable to describe software architecture models and high level software design and the availability of efficient algorithms and tools to evaluate the model. Moreover QN models can be extended to better reflect software characteristics like communication patterns and hierarchical structure (e.g., EQN, LQN).
- Architectural and application domain constraints represent a serious attempt to make an integrated approach to performance prediction analysis efficient and applicable. Identifying specific application domains and architectural structures can be the means for achieving scalability and/or modularity in performance analysis. This research area is even more relevant considering the development of component-based systems. In this setting, components are plugged into fixed underlying architectural structures in order to produce different systems.
- As far as the additional information required for performance analysis is concerned, Table 3.2 summarizes the kind of information needed by each approach. Performance requirements are obviously always assumed to exist. All the methods explicitly based on SPE, see Section 3.3.1, assume to have available performance related information at each phase of the software life cycle, till the detailed design is obtained. This obviously prevents their use at early stages. The same consideration applies to M5, which although based on architectural patterns follows the SPE approach and requires detailed information such as resource usage and loop repetitions that can only be available at detailed design level. Differently M6, also working with architectural patterns, requires information on the hardware platform that in many cases can be either available at requirement phase or supplied as possible alternative platform scenarios.

  The three methods based on trace analysis, M7, M8 and M9 see Section 3.3.3, are less demanding in terms of actual information and two of them, M7 and M9, can be applied at software architecture and

Approach	Reference in Section	Behavioral Model	Performance Model	Architectural Constraint	Application Domain	Life cycle Phase	Information Level	Feedback	Automation Degree	Tool
M1	3.3.1	Annotated MSC	QNM	—	General	Design	High	No	High	SPE●ED
M2	3.3.1	UML Sequence	EQNM	—	General	Design	High	No	High	—
M3	3.3.1	UML Diagrams	EQNM	—	Mobility Code	Design	High	No	High	—
M4	3.3.1	UML Sequence	LQN	Client/Server	General	Design	High	No	Medium/ Low	—
M5	3.3.2	UML Activity Collaboration	LQN	Architectural Patterns	General	Design	High	No	High	Implementations (not available)
M6	3.3.2	UML Collaboration	QNM	Client/Server	General	Design	High	No	Medium	CLISSPE System (not available)
M7	3.3.3	Annotated MSC	QNM	—	General	Architecture Design	Low	No	High	—
M8	3.3.3	ROOM Notation: State Machine	LQN	—	Real-Time Interactive	Design	High	Yes	High	PAMB
M9	3.3.3	UCM	LQN	—	General	Reqs and Architectural Design	High	No	High	UCM2LQN UCM Navigator
M10	3.3.4	UML Diagrams	QN (closed multiclass product form)	—	General	Design	High	Yes	High	OAT
M11	3.3.5	UML Diagrams	EQNM	Component based systems	General	Architecture Design	High	Yes	High	CB-SPE tool
M12	3.3.5	UML 2.0 Component Diagram	LQN	Component based systems	General	Architecture Design	High	No	High	Prototype tool
M13	3.4	Process Algebra	SPA	—	General	Specs	Low	No	High	Two Towers PEPA Workb. TIPP Tool
M14	3.5	Petri Net	GSPN	—	General	Specs	Low	No	High	HiQPN GreatSPN DSPNexpress2000
M15	3.5	UML Diagrams	GSPN	—	General	Design	High	No	High	Prototype tool and GreatSPN
M16	3.6	UML Diagrams	Simulation	—	Real-Time Systems	Design	High	Yes	High	SMG and OPNET
M17	3.6	UML Sequence	Simulation	—	General	Design	High	No	High	SimML, JavaSim
M18	3.6	UML Diagrams	Simulation	—	General Mobility Code	Architectural Design	High	Yes	High	UML-Ψ
M19	3.7	UML Activity State Diagrams	Semi-Markov Process	—	General	Design	High	No	High	DSPNexpress2000

Table 3.1: Summary of the Methodologies.

requirement level, respectively.

M10 as discussed in Section 3.3.4, suggests a UML-based design to achieve performance modeling. Therefore all the performance related information is integrated into design artifacts.

M11 and M12 are two approaches dealing with performance analysis of component-based software systems. They re-design two software performance approaches to integrate the predictive analysis in the reuse-based software development process. Both the approaches suppose a previous analysis on the selected components from which the performance properties (in M11) or the performance model (in M12) of the components are collected. This allows their application since the software architecture level in a reuse-based software development process.

M13 and M14 fall into a different group, since they apply at software specification time. All the required information to carry out performance analysis has to be interpreted in terms of action/transition execution time. Thus either all the performance information is available at specification time or the specification has to be properly refined in order to reflect further design choices.

M15 is the only complete approach that generate a GSPN model from UML description of the software system. Since it requires detailed design of the components it can be used from the design level.

The last four methodologies, M16, M17, M18 and M19 propose different approaches based on simulation techniques and stochastic processes, respectively. The first two generate the simulation models based on architectural and design artifacts enriched with temporal information. In order to produce the stochastic process model the last needs very detailed information at design level, like associating duration and delays to events. This implies that at design level there must be a clear picture of the performance effects of high level design concepts.

- Few approaches provide feedback information. In general, this requires a direct correspondence between the software specification abstraction level and the performance model evaluation results. Note that the existence of a unique behavioral and performance model does not necessarily imply the existence of this correspondence. Let us consider a process algebra specification, whose syntax provides the description of the software structure in terms of processes and their composition, and the behavioral and performance evaluation is carried out the global state transition model. Like in traditional debugging approaches it is possible to build a direct correspondence between the syntactic and the execution level, but it can be highly complex and costly. Obviously methods based on simulation techniques can more easily provide feedbacks, because there can be a direct correspondence between the software specification abstraction level and the performance model evaluation results.

- Most approaches exhibit high automation potentiality. This clearly indicates that there is a strong concern about automation as a key factor for the applicability of the approach. In general, automation is considered at any level of the methodology, from the specification of the software artifacts to the derivation of performance model to the evaluation phase. It is worth noting that there is a common trend towards achieving automation through integrating with existing tools both at software specification level, e.g., UML CASE tools, and at the performance analysis level, e.g., QN analyzers.

From this detailed information we derive the three main indicators introduced before to classify the various approaches. The integration level of software model with the performance model, takes into account the architectural constraints and application domain. The integration of performance analysis in the software life cycle accounts for life cycle phase, information level and feedback. Automation is kept as an autonomous dimension which also comprises tool support.

According to the above discussion, in Figure 3.5 we classify the methodologies. In each internal small cube, label ordering has no meaning. Differently, when a label lies at the intersection of more cubes, like M4, this indicates that the corresponding methodology has characteristics of both classes.

Most of the methodologies fall in the groups of syntactically or semantically related models and in the layer referring to software design. Let us recall that the dimension concerning the integration level of

Approach	Reference in Section	Req. Analysis	Requirement Specification	Software Architecture	Software Specification	Interface and Component Design	Data Structure and Alg. Design
M1	3.3.1	Performance Requirements	Operational Profile	Execution Env., Scenarios Frequencies		Process View, Service Rates, Resource Contention Delay	Resource Usage, Loop Repetitions, HW Config.
M2	3.3.1	Performance Requirements	Operational Profile	Execution Env., Scenarios Frequencies		Service Rates	Resource Usage, HW Config., Loop Repetitions,Occurrence Time of Interaction
M3	3.3.1	Performance Requirements	Operational Profile	Execution Env., Scenarios Frequencies, Time of Interaction		Service Rates	Resource Usage, Loop Repetitions, HW Config.
M4	3.3.1	Performance Requirements	Operational Profile			Module-Platform Mapping	Platform Configuration, Resource Capacity
M5	3.3.2	Performance Requirements	Operational Profile	Workload			Resource usage, Loop Repetitions, HW Config.
M6	3.3.2	Performance Requirements	Operational Profile	Workload			HW Configuration
M7	3.3.3	Performance Requirements		Implementation Scenario Alternatives			
M8	3.3.3	Performance Requirements		Critical Performance Scenarios		Env. & Deployment Info.	HW Configuration
M9	3.3.3	Performance Requirements	Number of Calls & Loop Iterations, Sw Comp. to Devices Mapping, Fork Probabilities, Devices Speed-up Factor, System Workload Intensity, Default Values	Number of Calls & Loop Iterations, Sw Comp. to Devices Mapping, Devices Speed-up Factor, Fork Probabilities, System Workload Intensity, Default Values			
M10	3.3.4	Performance Requirements		Classes for Resources		Workload, Running Config., Scheduling Policy for resources	Service Demand
M11	3.3.5	Performance Requirements	Operational Profile	Scenarios Frequencies, Component Service Rates/Times and Resource Usage, Loop Repetitions			HW Config.
M12	3.3.5	Performance Requirements	Operational Profile	Scenarios Frequencies Service Rates/Times of Comp. Interfaces, Comp. Resource Usage, Loop Repetitions, Level of Replicas and Multi-Threading			HW Config.
M13	3.4	Performance Requirements			Action Execution Time		
M14	3.5	Performance Requirements			Transition Execution Time		
M15	3.5	Performance Requirements				Time Rate of *do activities*, Stochastic Behavior on *outgoing* on *transitions*	
M16	3.6	Performance Requirements		Temporal Information		Temporal Information	
M17	3.6	Performance Requirements				Performance Information	
M18	3.6	Performance Requirements	Operational Profile	Workload, Step Execution Probabilities		Delay, Repetitions Service Execution Time	HW configuration
M19	3.7	Performance Requirements				Service Time Distributions & Deterministic Delays	

Table 3.2: Performance Information Required by the Methodologies.

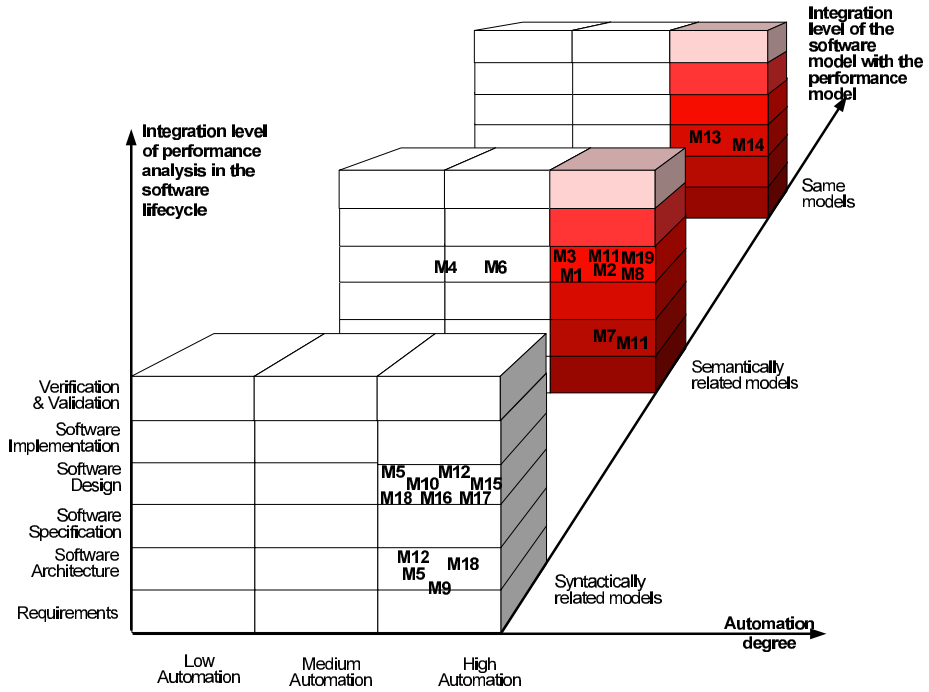


Figure 3.5: Classification of Considered Methodologies.

software models with the performance models refers to the way the mapping between software models and performance models is carried out. In the group of syntactically related models we put the methodologies that integrate the design and the performance models based on structural-syntactical mappings. For example in M9 there is an explicit correspondence between syntactic elements at the design level and elements of the performance model, e.g., UCM components and LQN tasks.

The group of semantically related models refers to methodologies whose mapping between design and performance models is mainly based on the analysis of the dynamic behavior of the design artifacts. For example, in M8 the mapping is based on the analysis of the state machines model of the system design.

The last group singles out the methodologies that allow the use of the same model both for design description and performance analysis. This group can actually be seen as a particular case of the semantically related models when the mapping is the identity modulo timing information. However, we have decided to keep it separate because this ideal integration can be obtained only if the software designer has the skill to work with sophisticated specification tools.

The analysis of the integration level of software models with the performance models shows from a different perspective the tendency to use separate formalisms and/or models. As we already pointed out, the rationale for this choice is to offer software designers tools at the same level as their expertise and skills in order to increase the acceptance and the use of the methods. The same reason motivates the high automation degree offered by most of the methods.

The integration level of performance analysis in the software life cycle shows that most of the methods apply to the software design phase in which a good approximation of the information needed to performance analysis is usually provided with a certain level of accuracy. At design level many crucial design decisions of the software architecture and programming model have already been taken and it is thus possible to extract accurate information for performance analysis, e.g., communication protocols and network infrastructure, process scheduling. At a higher abstraction level of the software system design there are many more degrees of freedom to be taken into account that must be suitably approximated by the analyst

thus making the performance analysis process more complicated. Of course there is a trade-off here and approaches operating at different abstraction levels rather than conflicting can be seen as complementary since they address different needs. Performance prediction analysis at the very early stages of design allows for choice from different design alternatives. Later on, performance analysis serves the purpose of validating such choices with respect to non-functional requirements.

Another consideration embracing all the dimensions regards the *complexity* of the methods as far as their ability to provide a performance model to be evaluated is concerned. In this respect we consider the complexity of the translation algorithms to build the performance model and the information to carry out the analysis. This notion of complexity does not consider the complexity of the analysis itself, although there can be a trade-off between complexity of deriving the performance model and complexity of evaluating the model. For example, highly parameterized models can be easily obtained but very difficult to solve. Of course the efficacy of a methodology, and therefore its success also depends on the analysis process, thus this issue should be taken into account as well. We do not consider it now, since an accurate analysis of the complexity of the analysis process depends on a set of parameters that can require extensive experimentation and use of the methodologies and this kind of analysis is out of the scope of the present Chapter.

Roughly speaking, with respect to the three dimensions of Figure 3.5, the possible methods which occur at the bottom of the second group, on the leftmost side of the automation dimension, exhibit a high complexity. This complexity decreases when moving out along any of the three dimensions. This is due to different reasons. Towards syntactically related models or towards the same models, the mapping between behavioral and performance models simplify or disappears. Moving up along the integration level of performance analysis into the software life cycle dimension, the accuracy of available information increases and simplifies the production of the performance model.

The last issue we discuss concerns the automation of the methodologies and in particular tries to summarize the portions of the software performance process that have been most automated.

Figure 3.6 shows the three stages of any software performance analysis process and highlights some automation tools that can be applied in these stages. A tool can support one or more stages as indicated by the arrow entering the stage or the box enclosing the set of stages. The dashed arrow going from performance model evaluation to software specification represents the analysis of potential feedback. It is dashed since, so far, not all the tools pointing to the largest box automatically support this feature.

The picture shows that most of the tools apply to the whole process providing a comprehensive environment. On the other hand there exist several tools with different characteristics that automate single stages of the process. This might suggest the creation of an integration framework where the tools automating single stages of the process can be plugged in, in order to provide a best-fit analysis framework.

Among the tools cited in Figure 3.6 there exist commercial tools and academic tools. In particular, most of the tools applying to the whole software performance process have been developed by academic institutions. Some of them are just prototype tools (even not available, in some cases), while others do have a commercial version, like GreatSPN, TwoTowers, and DSPNexpress 2000. The tool SPE•ED is the only purely commercial tool. The correspondence between the considered methodologies and their supporting tools is shown in the last column of Table 3.1. We refer to the presentation of the various methodologies for a brief description of the main characteristics of their supporting tools.

3.9 Final Considerations and Summary

In this Chapter we have reviewed the state of the art in model-based software performance prediction. We have taken a software-designer perspective in order to classify and evaluate the growing number of

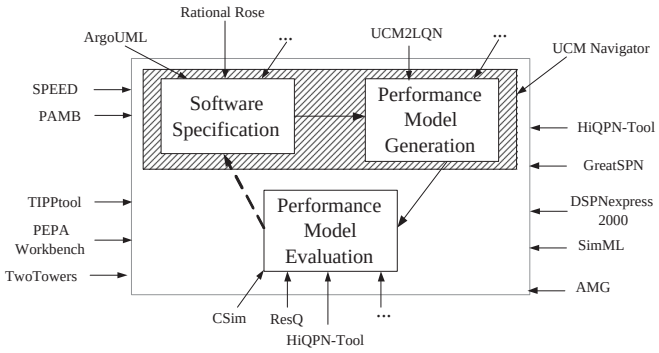


Figure 3.6: Tools and Performance Process.

approaches that have lately appeared. Our choice is driven by the generally acknowledged awareness that the lack of performance requirement validation in current software practice is mostly due to the knowledge gap between software engineers/architects and quality assurance experts rather than due to foundational issues. Moreover, short time to market requirements make this situation even more critical. In this scenario expressive and intuitive notations, coupled with automated tools for performance validation, would allow quantitative numerical results to be interpreted as design feedback, thus supporting quick and meaningful design decisions.

This is confirmed by the classification we carried out. Referring to Figure 3.5 it clearly appears that almost all methodologies try to encompass the whole software life cycle starting from early software artifacts. It is also meaningful that most methodologies are tightly coupled with tool support that allows the (partial) automation of them. There is no methodology which is fully supported by automated tools, and at the same time there is no methodology that does not provide or foresee some kind of automatic support. Most approaches try to apply performance analysis very early, typically at the software architecture level. So far, most of them still require much detailed information from the implementation/execution scenarios in order to carry out performance analysis. Nevertheless there is a growing number of attempts that try to relax implementation/execution constraints in order to make the analysis applicable at abstract design levels.

Three indications are highlighted from this survey. The first concerns software design specifications, the second performance models and the third one is related to the analysis process.

For software design specifications we believe that the trend will be to use standard practice software artifacts, like UML diagrams. Queuing Networks and their extensions are candidates as performance models. QN provide an abstract/black-box notation, thus allowing easier feedback and model comprehension, especially in a component-based software development process. As far as the analysis process is concerned, besides performance analysis, feedback provision is a key success factor for a widespread use of these methodologies.

Obviously, several problems still remain to be studied and solved. Software notations should allow for easily expressing performance related attributes and requirements. The more abstract the software notation, the more difficult it is to map the performance model. For QN this is a problem that can only be amended by the existence of algorithms and tools that permit the creation of performance models from standard software artifacts. This is not a problem for GSPN and SPA provided that the designers use these same notations for the behavioral descriptions, which, in software practice, is rarely the case. Therefore, if we assume a standard development process, with standard software artifacts, like UML-based ones, the effort to produce a performance model from the behavioral description is comparable for all three models.

Another problem concerns the complexity of the obtained performance model. This sometimes might prevent efficient model evaluation. The existence of a strong semantic mapping between the software

artifacts and the performance model may suggest strategies to reduce the performance model complexity still maintaining a meaningful semantic correspondence. Complexity problems can also be addressed by using simulation techniques besides analytical ones.

Last but not least the whole analysis process asks for automatic support. How to provide useful feedback is an important issue that deserves more study and experiments.

Summarizing, we believe that from this survey it emerges that although no comprehensive methodology is at present available, the field of model-based software performance prediction is mature enough for approaches that can be profitably put into practice.

CHAPTER 4

A NEW APPROACH FOR PREDICTIVE PERFORMANCE ANALYSIS OF COMPONENT-BASED SOFTWARE ARCHITECTURES

Early performance analysis based on Queueing Network (QN) models has been often proposed to support software designers during the software development process (see Chapter 3). These approaches aim at addressing performance issues as early as possible in order to reduce design failures. All of them try to adapt a system performance analysis methodology to software systems and they assume as available, at design time, information about the hardware platform the software will run on.

Few approaches can be used at the Software Architecture (SA) level, when a first description of statics and dynamics of the software system is provided. At the time of the software architecture specification, designers have to take crucial decisions on the design of the developed system. These decisions, mostly based on their experience, may affect functional and performance (in general non functional) aspects of the software system. Hence the performance evaluation at the software architectural level becomes an important task.

In this chapter we present our methodology that permits quantitative reasoning on performance aspects at the software architecture level. This approach, differently from other methodologies, does not require the specification of the hardware platform aspects. By filling the knowledge gap between the software and the performance worlds, our methodology provides the designers an easy and quick validation technique of the software architecture against the performance requirements. The approach, in fact, systematically generates a QN model representing the software architecture ready to be evaluated by performance solvers.

When a new performance model generation approach is defined several decisions should be taken: the software notation, the target notation, the additional information needed to carry on the analysis (such as operational profile and workload) , and where and how such an information should be specified. We detail in this chapter each of these issues.

4.1 PREMISES

The methodology we present in this chapter integrates the performance analysis process and the software life-cycle at the Software Architecture level. The Figure 4.1 shows such an integration. In the Figure, ovals represent activities of the process whereas rectangles indicate the output of each process step. From the performance requirements, performance figures of interest and a set of performance scenarios are defined. This information is an input for the performance evaluation step. From the description of the software architecture enriched by additional information a target QN model is generated. After that, the target model is evaluated and measures for the defined figures are calculated. The designers decide whether and how the software architecture should be refined from the analysis of the results of the evaluation step. If the designers do not experience any performance problems after the interpretation of the performance results, they proceed to develop the software system. Otherwise, to meet the performance requirements, they modify the software architecture and re-iterate the performance analysis process over the new software architecture. In

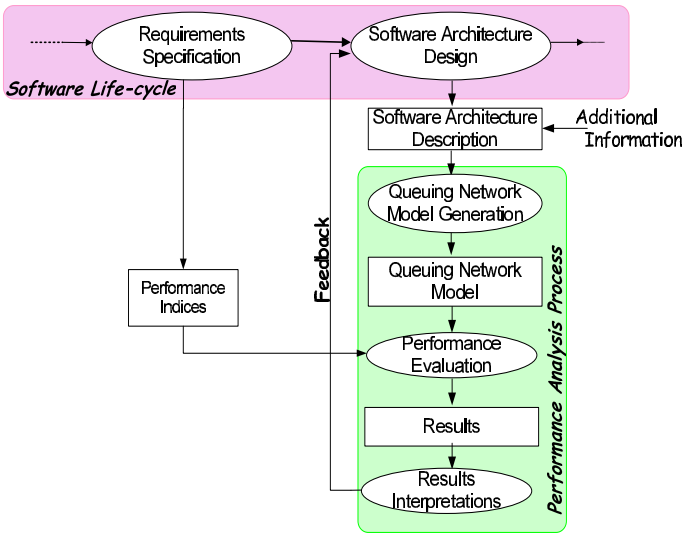


Figure 4.1: Software Performance Analysis Process and its Integration into the Software Life Cycle.

the software architecture modification the designers will consider the insights gained from the performance analysis. The way the software architecture is modified specifies the feedback the performance analysis process reports at the software architecture level.

The choice of software architecture as starting point is motivated by the observation that performance is a run time attribute and the software architecture is the first artifacts describing the behavior of a software system. Errors in such definition could compromise the success of the whole project (see [89] for an example). There are several definitions of Software Architecture and we consider as reference definition the following one (quoted from [35]): ”The software architecture of a program of computing system is the structure or structures of the system, which comprise software components, the externally visible proprieties of those components, and the relationships among them”. The architecture description hence defines the structure of the software system by identifying the software components, its behavior in terms of component interactions and ”externally visible” properties on the components that refers to assumptions made of an element, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on.

Also the choice of QN model is not casual. We guess that the following paragraph,extracted from [111], leads an ideal support for this choice: ”QNModelling is a top-down process. The underlying philosophy is to begin by identifying the principal *components* of the system and the ways they *interact*, then supply any details that prove to be necessary”. From this definition and the one for the software architecture we reported above, it is simple to identify several aspects that the queuing networks and software architectures have in common.

We anticipated before that our methodology does not consider hardware platform information. Differently from the existing QN-based approaches, in fact, the QN service centers do not represent hardware devices (such as disk and processor), but they represent software components. The QN topology describes how the software components are combined to form the software system and how a system service request is accomplished through the software components interactions. The underlying assumption is that there exist some logic devices having particular characteristics/properties and each software component is deployed on one of such logic devices. All the logical devices have the same processing power but they could have different waiting queue characteristics (capacity and scheduling policy). This implies that the service requiring the minimum resource demand (or workload) will always be faster. The device speed factor cannot reduce the time a request spends to be accomplished. In our approach, hence, we do not distinguish service time from the service demand for a class of requests (or jobs).

The performance analysis we carry on aims at identifying (potential) performance problems due to logical structure of the functionalities of the software system, and/or identifying critical software components whose design has to proceed carefully. In other words, the predictive analysis should point out portions in the software architecture that could raise performance problems.

Our predictive performance analysis fails when the software components are deployed on hardware devices showing high-variance speed factors. This happens because our assumption about the uniform processing power of the logical devices is not valid any more. However, in these cases, the predictive analysis we propose can be useful (i) to support the designers in the software architecture development when components are identified, in fact, in this step the analysis can suggest how distribute the functionalities among the components in order to have good performance, and (ii) to identify critical software components that must be carefully developed in the subsequent life cycle phases.

The target model obtained can be solved or simulated. There exist many algorithms and tools able to simulate and solve QN models [153, 134]. A QN model can be solved in an exact way if it is a product-form QN [111, 109], otherwise several approximate solution techniques are available in literature [26, 13, 136, 9, 7, 163].

Since whenever the QN model is specified well-established techniques are used in the performance evaluation step to solve it, we believe that the automation of the QN model generation is a key point for a wider usage of the predictive performance analysis. In the following we mainly concentrate on the definition of the QN model generation step.

The preliminary version of the QN model generation approach [18, 19, 72] we propose allows automatic specification of a QN model [111] from the representation of a Software Architecture by means of a set of Message Sequence Charts (MSC) [137]. However, unfortunately, its application to complex case studies points out that the approach is not able to consider all their aspects. We briefly overview the main aspects of such approach version in Section 4.2

In a more recent work [73] we extend this methodology to encompass a compositional approach to QN model generation for a software architecture described by means of UML2.0 diagrams [148]. The new version of the methodology improves the original one in several directions: it introduces a multi-chain QN model [111] to deal with software systems that export multiple services and it handles the QN parametrization. Moreover it is compositional and driven by architectural patterns making easier the architectural feedbacks generation. The target model here is an Extended QN.

4.2 An Introductory Approach to Software Architecture Performance Analysis

Before to move into the details of the proposed approach in this section we sketch the main characteristics of the previous work from which the new one derives. In [18, 19] it has been defined a methodology that, starting from a Software Architecture (SA) description given by means of Message Sequence Charts (MSC) [137], automatically derives a performance evaluation model, based on a Queueing Network (QN) model [111]. In the following, we refer to this methodology as MSC2QN.

The MSC2QN step allows the generation of the QN topology. Very briefly we recall that the topology of a QN model is represented by a set of service centers, which are independent entities, suitably connected. The translation algorithm aims at deriving a QN model as close as possible to the SA model where the software components that are strongly synchronized are represented by means of a single service center, and independent components are, instead, associated to individual service centers. Service centers are then connected in the QN model depending on how the SA components interact with each other (see [19] for more details).

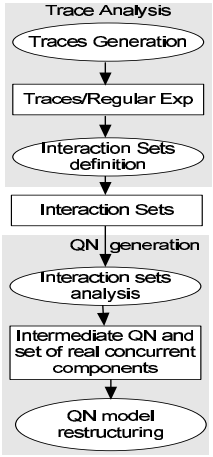


Figure 4.2: QN Generation in MSC2QN Methodology.

Figure 4.2 outlines the macro steps of the MSC2QN methodology which are:

1. the `Trace Analysis` phase that starts encoding the MSC by means of regular expressions. These regular expressions are analyzed in pair to find out their common prefix and to identify interaction pairs that can give information on the real concurrency between SA components. These interactions pairs form the Interaction sets.
2. the `QN Generation` step that, by analyzing the Interaction sets built previously, identifies the components or sets of components that should be modelled as service centers of the final QN, and the interconnections among the service centers, i.e. the topology of the QN.

To apply this methodology, the set of MSC must be representative of major system behaviors and for each SA component there must exist at least an MSC describing its interactions with other components.

The methodology requires that the MSC description must have state information of each component before interactions occur. This information on the component state allows the identification of concurrent behaviors in a software component. Let us suppose that the methodology identifies two interactions I_1 and I_2 having the same sender component. They represent for such a component parallel behaviors only if the states the component reaches before executing I_1 and I_2 are not in conflict. Conflict is a relation on states components that holds when, given a component, two states represent two mutually exclusive component behaviors. Otherwise the identified pairs of interactions do not give information on the degree of parallelism of the involved component but they just belong to two not concurrent execution traces.

In this version of the methodology a crucial assumption is made: all the MSC of the SA description start from the same initial state of the software system. In other words there is no hierarchy among MSC, and every one can only be triggered, in practice, from an external event. This constraint may be overcome by making use of HMSC [137]. Moreover, to simplify the approach, the primitive basic communication between components is supposed to be one to one. This assumption is not restrictive since both multiple-senders and multiple-receivers interactions can be modelled by a set of one to one communications.

4.2.1 MSC Features used in MSC2QN Step

The MSC2QN methodology uses several features of the MSC notation, that we have analyzed in [72] and that we summarize in Figure 4.3.

MSC2QN uses, according to the standard MSC notation, different arrow heads to model synchronous and asynchronous interactions : full arrow heads for synchronous communications, and half arrow heads for asynchronous ones.

The MSC setting condition facility, instead, is used to model conditions on the state of each component, before interactions occur. A setting condition sets or describes either the current system global state or the components state in order to restrict the traces that an MSC can take. Its graphical representation is an hexagon placed on one or more instance lifelines. The approach requires that, if a component is designed as a multi-thread component, state information for that component is specified by a tuple of setting conditions, one for each thread. Figure 4.3 shows setting condition for single thread component such as $comp1$ and the one for multi-threads component such as $comp3$ component that is composed by two threads.

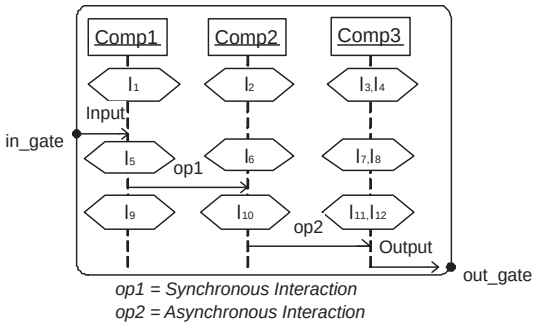


Figure 4.3: State Information and Interaction Types in MSC Notation.

Finally, MSC gates are used to model incoming and outgoing requests/data for the software system. The software system may receive service requests from external software components and it may give back information. These external entities are called environment. In a QN, the interactions of the system with its environment are modelled by input and output flows respectively, and in such cases the QN is called open. Instead, when interactions between the software system and its environment do not exist, the corresponding QN is called closed. The methodology can build both the QN model typologies. However, to generate an open QN, the approach needs information on the interactions with the environment. Gates in the MSC notation represent the interface with the environment. Any message attached to the MSC frame constitutes a gate. If the arrow starts from the MSC frame and ends in the lifetime line of an object it will correspond to an input flow. Analogously, if the arrow starts from the lifetime line of an object and ends to the MSC frame it will correspond to an output flow. Figure 4.3 shows two gates, *in_gate* and *out_gate*, referring respectively to an input stream and to an output stream.

MSCs permit to represent many aspects of the SA dynamics useful to produce QN models from SA descriptions. However some extensions to the notation must be defined.

First of all, interactions specification must be extended. In general the interactions involve objects. In our approach the interactions occur between architectural components. Although the difference is mostly conceptual, in practice our extension allows the designers to abstract from (possibly unknown) behavioral details. Obviously, on the other hand, we loose notation expressiveness, for example, dynamic creation of objects can not be modelled with MSC.

Secondly, we require to extend the MSC notation by introducing blocks of interactions that can occur many times in order to represent iteration cycles, as shown in figure 4.4.

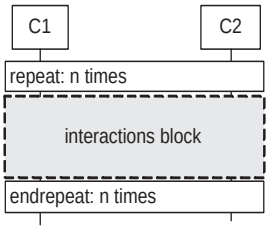


Figure 4.4: MSC with a Repeat Block.

4.2.2 LIMITS OF MSC2QN TECHNIQUE

This approach has many limits. In fact, by applying it to complex case studies we experienced that the approach is not able to consider all their aspects. First of all, it is not able to properly manage complex systems having more than one functionality. The approach loses information about the type of exchanged messages maintaining information only on the components involved in the communication. Two errors arise when more software system services are considered: we cannot distinguish different types of communications having the same sender and receiver and, as a consequence, we do not know the subsequent evolution of the execution trace and the service rate to be applied when the communications are very different from each other. This can result in an inconsistency problem in the obtained QN model with respect to the original software architecture.

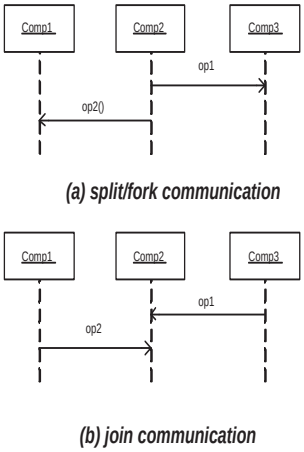


Figure 4.5: Some Patterns the MSC2QN Approach does Not Deal with.

Moreover, the actual approach does not deal with some particular architecture behavioral patterns, such as, for example, the ones in Figure 4.5 where a component communicates with two different ones (split/fork communication in the figure) or where it receives two messages from different senders component (join communication in the figure). Again, it does not cover the QN parameterization phase that requires the knowledge of domain specific information such as system operational profile and resource workload. To this aim many efforts have been done to extend scenario notations to allow performance data specifications. For example, UML profile for Schedulability, Performance and Time has been defined to allow the embedding of performance related data in the UML diagrams.

Finally, due to its monolithic nature, the approach makes the architectural feedbacks generation difficult.

4.3 Software Architecture Performance Analysis: an Advanced Approach

In this section we present the new approach to Software Architecture Performance analysis (that we call SAP•one) that generates a multi-chain QN model from a software architecture description based on UML 2.0 [73]. It deals with the QN parameterization and it is defined over component-based software systems requiring detailed information about interactions among components. SAP•one derivers from a re-engineering of the MSC2QN to make it systematic. The approach, in fact, defines translation rules that map architectural patterns into QN patterns. The target model is generated by composing the identified QN patterns suitably instantiated to the particular case.

To carry on the performance analysis we need additional information on the software system generally missing in the software architecture description. Such data are strictly related to the performance aspects and are used in the QN parameterization and in the workload definition. They are the operational profile of the system (modelling the way the system will be used by the users), the workload entering the system, the service demand required by a request (job) to the system components it visits, and performance characterization of the system components (service rate, scheduling policy, waiting queue capacity). We annotate the UML Diagrams with such an information by using the UML Profile for the Schedulability, Performance and Time [87].

We use UML 2.0 as an Architecture Description Language (ADL) to describe the software system architecture (as we reported in [63]). Software Architecture describes the system at a very high level of abstraction by specifying its (statical) structure and its (dynamical) behavior. We use the component diagram to model the statics structure of the system in terms of the software components and connectors. The description of the behavior of the software system is, instead, provided by the sequence diagrams where the lifelines represents the software component instances and the arrows model their interactions. Finally, we use Use Case Diagrams to specify the services provided by the software system.

The QN modelling process is composed of three steps: (i) *QN topology definition* where the service centers and their interconnection are identified. In this step, also QN chains describing the path crossed by a request (or a software functionalities) among the QN service centers (the software components) are specified; (ii) *QN parametrization* where the characteristics of each service center are specified, such attributes are service rate, scheduling policy and waiting queue capacity if the center is a queueing center, or the latency if it is a delay center); (iii) *workload intensity definition* that defines the characteristics of incoming traffics (open, close or mixed workload).

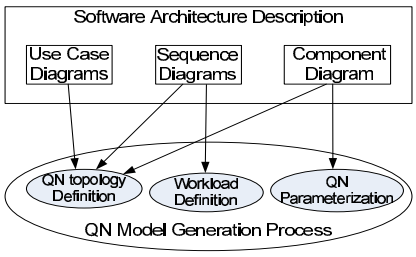


Figure 4.6: UML Diagrams Contribution in QN Model Generation.

Figure 4.6 shows how the considered UML diagrams contribute in the generation of a QN model. All the three diagrams are involved in the QN topology definition, sequence and component diagram also provides information for the specification of the workload and the QN parameterization, respectively.

In the following, motivations about the migration towards UML 2.0 and about the usage of a multi-chain QN models are given.

MIGRATION TOWARDS UML 2.0

UML has become a standard de fact in software modelling due to its capabilities to represent different aspects and views of a software system. Its wide diffusion is also due to the proliferation of open source tools that can be freely used and work on UML representation.

We decided to migrate from MSC notation to UML because the new release of UML specification, that is UML 2.0 [148], improves the expressiveness of the UML notation providing the definition of all facilities we need for our QN model generation. In particular, it strengthens the definition of Sequence Diagrams, by introducing all the facilities we identified and summarized in subsection 4.2.1. Moreover, UML 2.0 defines new operators on Sequence Diagrams that can facilitate and improve our approach. For example, operators such as `alternatives, parallel` and `loop` make more concise the software behavioral description allowing the representation of several execution traces in a single Sequence Diagram. The new operator `ignore`, instead, allows the slicing of the software system behavior. By means of it the simplification of the target model can be done by reasoning at the software designers level. This is extremely useful when it is necessary to simplify the target QN model in order to permit a reasonable performance evaluation. Quite often, in fact, the software system model describes details that are useless for the performance analysis and that make the target model too complex to be evaluated in an acceptable time. Details on the considered sequence operators are reported in Section 4.3.7 where we introduce the translation rule that maps them in QN patterns.

UML 2.0 better specifics the Component diagram. In such a diagram a component specifies a formal contract of the services that it provides to its clients and those that it requires from other components or services in the system in terms of its provided and required interfaces. We use the component diagram to specify the structure of the software system and we annotate it with service rates for the provided interface and the scheduling policies of each software components.

Finally, the specification of UML profiles, such as, for example, the UML profile for Schedulability, Performance, and Time [87], provides common facilities to annotate performance information on the UML diagrams.

MOTIVATIONS OF USING MULTI-CHAINS QUEUING NETWORK

In the new approach, a QN model represents the software architecture we want to analysis and a service center models an architectural software component. Since in general, the software system provides several services to its users, as shown by its use case diagram, the requests entering the software system are of several types. Hence, the customers entering to the QN model are of different types. Moreover, the software system has different behaviors according to the different types of requests, and we suppose that these behaviors are specified by a set of Sequence diagrams. This implies that the QN customer types representing the users requests should also have different behavior in the QN model. The behavior of a customer in a QN model is defined by the chain of services it requires to the service centers.

Informally, a chain in a queueing network model describes how a request type is served by passing through the QN service centers. Hence, it defines the routing of the request (QN customer) between the service centers and the time the request spends in them. Since different user requests (QN customers) might require different works to a software component (QN service center), the software component should provide several services that can have different complexity and service time.

To cope with this situation we use multi-chains QN model where the service center can be a queueing center or a delay center. Customers at a queueing center compete for the use of the server. Thus the time spent by a customer at a queueing center has two components: time spent waiting, and time spent receiving service. Customers at a delay center are allocated to an own server, so there is no competition for service. Thus the

residence time of a customer at a delay center the service demand the customer requires there. In our model the queueing centers have one non preemptive server able to do different jobs with different service time. Also delay centers can impose different latencies against different jobs to be executed. The approach maps a software component into a delay center when it is represent a logical resource dedicated to each system customer and hence it does not represent a shared resource.

4.3.1 UML 2.0 DIAGRAMS

UML 2.0 is the new release of the OMG for the Unified Modelling Language Specification that improves the definition of some diagrams (such as Component Diagram), strengths others (such as Sequence Diagrams) and introduces new ones (such as Timing Diagrams) [148].

As discussed in the previous section, in our methodology we use UML diagrams to model the system at the software architecture level. In particular, we consider Use Case diagrams to model the system services, the Sequence Diagrams to describe the software architecture dynamics, and the Component Diagrams to describe its static structure. UML 2.0 does not modifies the specification of the Use Case diagrams, whereas it improves the Sequence and Component diagrams. In the following, we consider only the changes UML 2.0 presents for such diagrams.

SEQUENCE DIAGRAMS

Sequence Diagrams describe interactions. An interaction, in an UML terminology, is a unit of behavior that focuses on the observable exchange of information between elements (such as for example objects) in form of messages. Interacting elements are represented by means of lifelines and messages through arrows. An interaction in a Sequence Diagram is represented by a solid-outline rectangle. The keyword `SD` followed by the interaction name and parameters, is in a pentagon in the upper left corner of the rectangle. For an example of interaction described by a sequence diagram please refer to the Figure 4.9 where it is reported a Sequence Diagram for the `usecase1 sequence` interaction.

The main evolutions in the interaction specification UML 2.0 introduces are the concepts of `InteractionFragment` and `CombinedFragment`. The former is a piece of an interaction. The latter, instead, defines an expression of interaction fragments. A combined fragment is defined by an interaction operator and the corresponding interaction operands. Through the use of `CombinedFragment` the user will be able to describe a number of traces in a compact and concise manner.

The semantics of a `CombinedFragment` dependents on the semantics of its interaction operator. The introduced interaction operators are:

Alternatives : the interaction fragments represent behavior alternatives; *Option*: the combined fragment represents a choice of behavior where either the (sole) operand happens or nothing happens;

Break : the combined fragment designates a breaking scenario in the sense that the operand is a scenario that is performed instead of the remainder of the enclosing interaction fragment;

Parallel : the combined fragment represents a parallel merge between the behaviors of the interaction fragments;

Weak/Strict Sequencing : the combined fragment represents a weak/strict sequencing between the behaviors of the operands;

Negative : the traces that are defined in this combined fragment are invalid;

Critical Region : this operator models a critical region;

Ignore/Consider : *ignore* designates that there are some message types that are not shown within this combined fragment. These message types can be considered insignificant and are intuitively ignored if they appear in a corresponding execution. Alternatively one can understand ignore to mean that the messages that are ignored can appear anywhere in the traces. Conversely the interaction operator *consider* designates which messages should be considered within this combined fragment.

Assertion : this operator representing an assertion;

Loop : this operator designates that the interaction fragment has to be iterated for a number of times.

The notation for a CombinedFragment in a Sequence Diagram is a solid-outline rectangle. The operator is shown in a pentagon in the upper left corner of the rectangle.

UML 2.0 introduces also the concept of gate that represents the syntactic interface between the combined fragment and its surroundings, which means the interface towards other interaction fragments.

In Figure 4.9 the sequence diagram is composed by several (nested) combined fragments, that are *weak sequencing*, *alternatives* and *loop*, and it shows two gates, namely `in_gate` and `out_gate`.

COMPONENT DIAGRAMS

A Component Diagram in UML 2.0 is defined by a sets of components connected each other by means of their interfaces. An interface defines a set of operation involving the component. A component is a modular unit with well-defined interfaces; it can be replaced by any unit that has the equivalent functionalities and compatible interfaces. The interfaces of a component are classified as provided interfaces and required interfaces. Provided interfaces defines a formal contract of services that the component provides to other components while required interfaces have defined the services that it requires from outside in the system in order to function. These interfaces may optionally be organized through ports. The replacement of a component may take place at either design time or run-time. The substituting component should be able to interact with other components or its environment provided that the constraints of the interfaces are followed.

In UML 2.0, a component can have two different views, external view and internal view. The external view is also known as a ”black-box” view in which it exhibits only the publicly visible properties and operations which are encapsulated in the provided and required interfaces. The wiring between components is specified by dependencies or connectors between component interfaces. The internal view is a sort of ”white-box” view which shows the component internals that realize the functionality of the component. An external view is mapped to an internal view by using dependencies which are usually shown on structure diagrams, or by using delegation connectors that connect to the internal parts which are shown on composite structure diagrams.

In our work we consider the only external view of the components. The component in UML 2.0 is represented by a rectangle while the provided interfaces are represented by circle connected to the rectangle by a line. The required interfaces, instead, are represented by a semi-circle.

Figure 4.8 shows a component diagram with three components an their required and provided interfaces.

4.3.2 USAGE OF SPT TO ANNOTATE UML 2.0 DIAGRAMS

We use UML Profile for Schedulability, Performance and Time (SPT) [87] to annotate additional information to Sequence and Component diagrams. We use the SPT Profile stereotypes as notes in the diagrams and the semantics of numerical values annotated in the diagrams was *assumed* since they are based on the designers' experience since such information is not available at the software architecture level. We were able to use this profile without extensions even if it has been defined on UML 1.x.

ANNOTATION IN THE COMPONENT DIAGRAMS

The annotated component diagram provides information on the parameterization of the service centers of the QN, such as the type of the service center (e.g. servers with waiting queue or a delay), the rates of the services they provide and the scheduling policies they use to extract jobs from their waiting queues.

We use the `<<PAhost>>` stereotype to annotate the components in the diagram. `<<PAhost>>` stereotype models an active resource. As we introduced before, we assume that each component is deployed on an own logical active device. Hence there is a strong correspondence between the software components and the active (logical) resources processing them. This assumption supports our usage of the `<<PAhost>>` stereotype to annotate components.

Since we assume that all logical devices have the same processing power, we do not specify any tag values for it, except for the `PAschdPolicy` tag value that indicates the scheduling policy of the waiting queue of the components.

We also annotate the component diagram with the execution time required to a software component by a service request. This time represents the total execution time required locally to the component in order to satisfy the request. This information is specified by either `<<PAdemand>>` or `<<PAdelay>>` tag values of the `<<PAstep>>` stereotype. The approach associates such stereotype to each component interface.

At the moment, for simplicity, we assume that an interface contains just an operation from which it inherits the name. This assumption eases the annotation of the service demands in component diagram.

See Figure 4.25 for an example of the annotated component diagram.

4.3.3 ANNOTATION IN THE SEQUENCE DIAGRAMS

According to the UML profile, the workload intensity, provided to the system by a customer class, is specified through the `<<PAopenLoad>>` or `<<PAclosedLoad>>` stereotypes. The former is used when the customer type provides an open workload, whereas the latter is used in case of closed workload. These stereotypes annotate the first message of the sequence diagram describing the system behavior for such workloads. For example, in Figure 4.9 the `<<PAclosedLoad>>` stereotype defines the workload intensity for the customer class (or system use case) `usecase1`. In this case, the workload is defined by the number of users (`PApopulation` tag value) and the time needed to generate a request (`PAextDelay` tag value).

We use the `PAprob` tag value of the `<<PAstep>>` stereotype to annotate the probability of execution an alternative behavior when the sequence diagram presents either `alt`, `break` or `option` fragment operator. Such stereotype annotates the first message of all the interaction fragments that are operands of the introduced operators.

Sometimes it can be useful to explicitly identify the time a component spends to accomplish partial tasks. We use the `<<PAstep>>` stereotype to give such an information. The stereotype is associated to the

interactions in the sequence diagram that identify the start point of such partial task. For an example of this usage of the <<PAstep>> see the e-commerce case study in Section 4.4.

4.3.4 Overview of the Approach

The SAP•one approach extracts from the annotated UML diagrams all the information needed to generate and parameterize the QN model. It is a three steps methodology:

- *Identification of the QN customers*. Use Case Diagram describes the software system at a very high level of abstraction by identifying its functionalities. These functionalities corresponds to the requests (or customers) entering the system. Thus this type of diagram gives information on the type of traffics incoming in the system. As required from the approach, for each type of customer the use case diagram associates a set of sequence diagrams describing the corresponding behavior inside the system. For each customer type, hence, we know which sequence diagrams describe its behavior. For example, in Figure 4.7 the diagram shows three use cases that are `usecase1`, `usecase2` and `usecase3`. It specifies that `usecase1` and `usecase3` behaviors are described by a sequence diagram each, while the behavior of `usecase2` is described by three sequence diagrams. From such a diagram the approach identifies three customer types and the relative behavior inside the system.

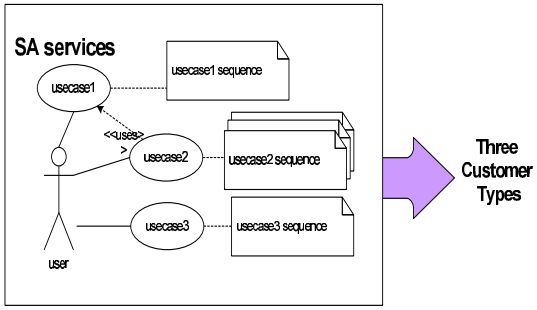


Figure 4.7: SAP•one Customer Types Identification Step.

- *Identification of the service centers and their characteristics*. This information is stored in the Component Diagram: the approach maps to each component a QN service center with one non-preemptive server, or a delay center. The component interfaces define the classes of job processable by the related component-service center. Since the software component can provide many interfaces (or services), the corresponding service center may have several service classes, one for each interface in the component diagram. The service time of the identified service class (or job class) is extracted from the `PAdemand`/`PAdelay` tag value of the <<PAstep>> stereotype.

  The center is a waiting center if its interfaces are annotated with the `PAdemand` tag value, otherwise, if `PAdelay` tag value is used, the center represents a delay center. Obviously, the interfaces of a component cannot be annotated by both tag values at the same time.

  Finally, from the `PAschdPolicy` tag value of the <<PAhost>> stereotypes the approach extracts the scheduling policy of the service centers.

  In Figure 4.8 the diagram shows three components: `Comp2` and `Comp3` are queueing centers with FIFO scheduling policy, component `comp1` is a (FIFO) delay center. In fact, the formers show interfaces with service demand annotation while `Comp1` has interfaces with delay annotations.

- *Definition of the QN chains and the relative workloads*. Sequence diagrams show how a customer of a given type moves among the service centers asking for services when it enters in the system. From

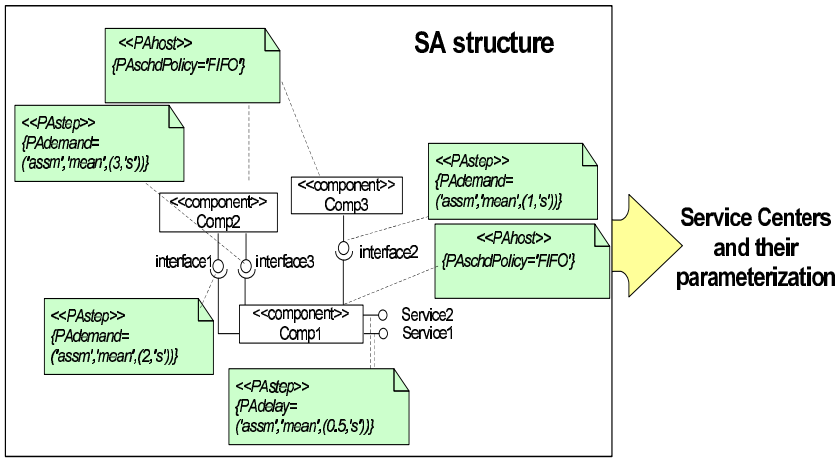


Figure 4.8: SAP●one Service Centers Identification Step.

each set of scenarios associated to a customer, the approach generates a QN chain. The sequence diagrams indicate the workload such chain imposes to the system and report routing probabilities when there are alternatives in the behavior description. For the sake of simplicity, without loosing generality, we can assume that the behavior of a customer is described by a single sequence diagram. Indeed, the new interaction operators UML 2.0 introduces allow the modelling of complex behavior with several alternatives, in only one scenario. Figure 4.9 shows an example of sequence diagram the approach deals with. The diagram describes the behavior of the `usecase1` customer. The annotation at the top of the figure specifies that the workload of such type of customer is a closed one. It specifies for such load the population size (`PApopulation` tag) and the think time the source spends to generate the next request (`PAextDelay` tag). The sequence diagram models an alternative behavior by using the `alt` operator. This operator points out a branching point where the customer can behaves as one of the alternatives indicated, but not both. The `PAstep` annotations bring information on the probability in executing one of the alternatives described.

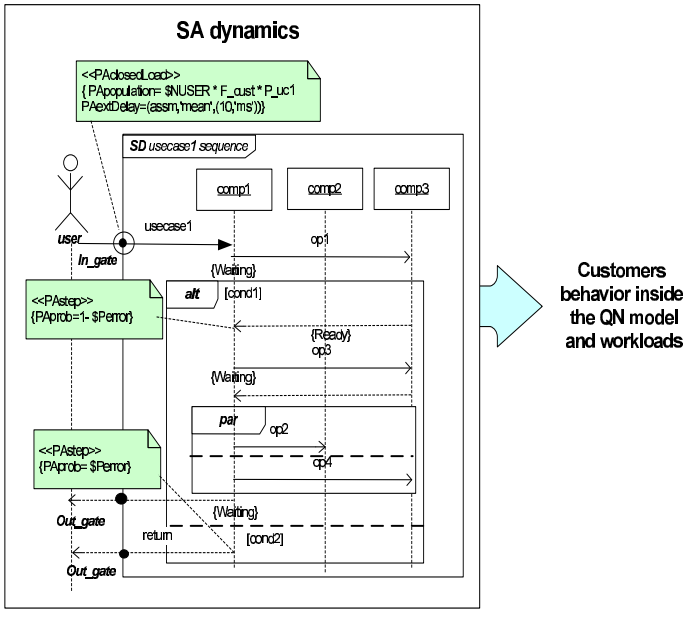


Figure 4.9: SAP●one Chains Identification Step.

4.3.5 CHAINS GENERATION: A COMPOSITIONAL APPROACH

In this section we present the technique to generate a multi-chain QN topology. As we already said, the approach introduces a chain for each use case of the system. The use cases are considered separately, one by one. All the sequence diagrams describing the software system behavior of an use case are considered.

The output of a chain generation is a sub-QN model that describes the behavior of the mapped QN customer type.

The approach models the software components interaction (arrow in the sequence diagram) according to the previous approach. It associates a connection in the QN topology from the service center of the component sender to the one of the receiver. The service center of the receiver has an infinite (or finite with a given capacity) waiting queue if the interaction is asynchronous, or a waiting queue with zero capacity if the interaction is synchronous.

As the previous approach does, the new version uses the *gate* facilities to identify the environment and the incoming and outgoing system traffic. A gate in a sequence diagram is the point of the most external interaction fragment crossed by an arrow. The gate is an output gate if the interaction is an outgoing arrow from the frame. Whereas it is an input gate if the interaction is entering into the frame.

The approach is compositional. Statically, we define a library of architectural patterns (sequence diagram fragment) the approach deals with and we relate them to QN patterns. Thanks to the new operators for the sequence diagrams, it is easy to identify and to model architectural patterns of interest. Then, the approach suitably combines the involved QN patterns by instantiating them to the particular scenario the sequence diagrams describe.

The compositional feature of the approach helps in the architectural feedbacks derivation. The separation of system behaviors and the identification of patterns helps to point out software system behavior parts that result in performance problems.

In the following, we introduce the translation rules for basic interactions (called *basic rules*), and the rules to translate the UML sequence operators into the QN pattern. The presentation of the identified rules is supported by figures having all the same two parts structure: the left-hand side part shows the behavioral pattern (described by UML sequence notation) the rule deals with, and the one at the right hand side reports the corresponding target sub-model

4.3.6 BASIC RULES

We intend as *basic rules* the rules we defined to translate a components interaction into a QN sub-model. From an initial study we identify two types of interaction: the synchronous and the asynchronous one. In Figure 4.10 it is reported identified translation rules for such interaction types.

A components interaction can be interpreted as a connection from the QN service center representing the sender component to the one corresponding to the receiver. If the interaction is asynchronous, the waiting queue of the receiver is infinitive (see the top level of the Figure 4.10) and the request is buffered into the queue, leaving the sender free to continue its execution on the next request. If the interaction is synchronous, in order to simulate the blocking in the sender execution until the receiver becomes free, we model the interaction by using finite waiting queues and appropriate blocking protocols [25, 19].

In QN with finite capacity queues, when a queue reaches its maximum capacity, the flow of customers into the service center is stopped, both from other service centers and from external sources in open network, and the blocking phenomenon arises. Various blocking mechanisms have been defined. We model synchronous

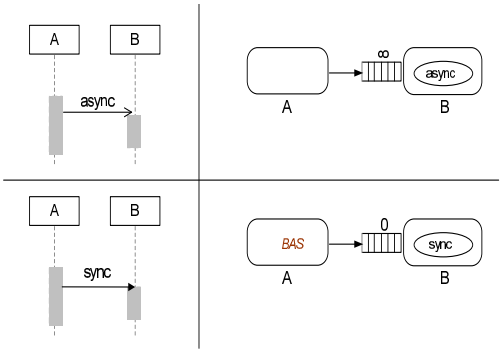


Figure 4.10: Translation Patterns for Synchronous and Asynchronous Interaction.

interaction with the type called Blocking After Service (BAS). Under this policy when a customer, after having received the service at the service center k, tries to enter a saturated center j, it is forced to wait at the node k until the destination service center j releases a job. When more than one node is blocked by the same saturated node, a scheduling discipline must be considered to define the unblocking order of the blocked nodes. We refer to First Blocked First Unblocked (FBFU) discipline to manage this problem.

To model synchronization among software components (see bottom level of the Figure 4.10) we associate to the receiver a service center with a zero capacity queue and we force a BAS blocking policy to the sender.

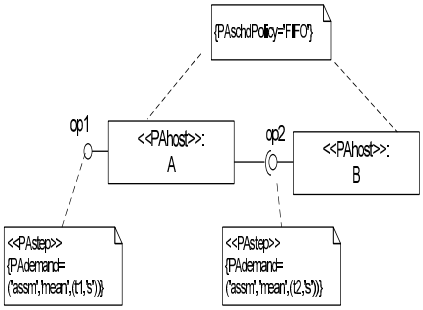


Figure 4.11: Component Diagram.

In general, the synchronous and asynchronous interaction could represent a signal or a call action, where we distinguish a call action from a signal when the component interaction imposes a (control/data) return from the receiver. From this observation we can characterize four more complex cases that we show in Figures 4.12, 4.13, 4.14 and 4.15 where the interaction modeling is referred to the *op2*. In the following, to describe the basic translation rules we refer also to the information annotated in the Component Diagram of Figure 4.11.

SYNCHRONOUS SIGNAL

In Figure 4.12 we show the rules to deal with synchronous signal. Synchronous signals are synchronous interactions which do not required a control/data return. We identify two situations depending on when the considered interaction occurs: rule *(a)* that can be applied when the *op2* interaction is performed during the execution of the *op1*, before its execution time *t1* is elapsed; and rule *(b)* that can be applied when the *op2* interaction is performed at the end of the execution of the *op1*. In rule *(b)* (the simpler one) the QN customer enters in the *A* service center (software component) by asking *op1* service. When *A* ends its work, the customer enters in the *B* service center to receive the *op2* service.

The rule *(a)* instead is more complex, it has to model the parallel provision of a portion of the *op1* service (*Op1-b* in the Figure) and the *op2* service. This is modelled by means of the fork feature of the QN modeling that we represent in the figure as a triangle. Of course, due to the synchronous nature of the *op2* interaction, *op1* service is a service with the BAS blocking policy. If the service center *B* is busy the BAS policy throws and the service center *A* remains blocked until *B* becomes free again. When a job leaves *A*, it forks into two new jobs, one entering *B* and the other going back to *A* with a different job type and higher priority. The usage of low and high priorities for the customer entering in *A* guarantees that *A* terminates the current request before to serve a new one.

However, the rule *(a)* can be applied when it is possible to estimate the times *t1'* and *t1"* (such that $t1 = t1' + t1''$), otherwise we have to approximate the model by using the second rule. The same approximation can be fairly done when we plan to use an analytical analysis technique based on MVA algorithm to evaluate the QN model since all analytical MVA-based techniques implicitly make the same approximation. In fact, they do not consider the detailed customer routing but only the total workload a customer type imposes to the service centers.

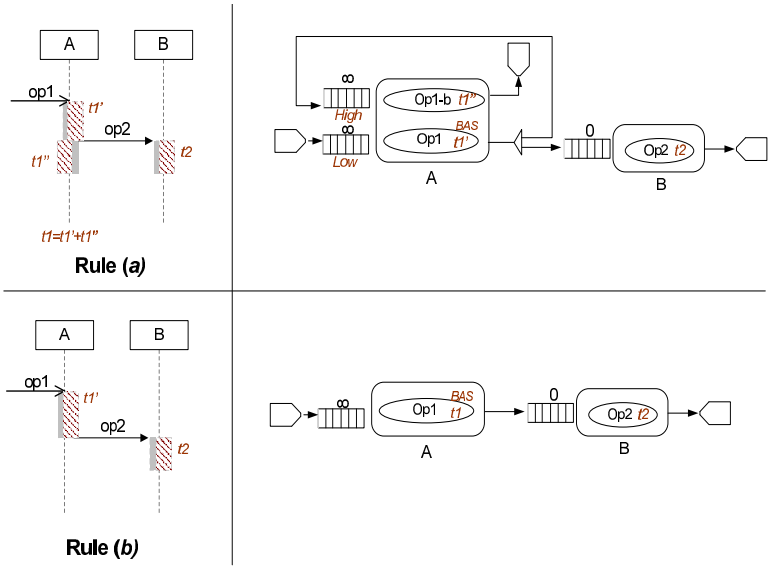


Figure 4.12: Translation Patterns for Synchronous Signals.

ASYNCHRONOUS SIGNAL

In Figure 4.13 we show the rules to deal with asynchronous signal. Asynchronous signals are asynchronous interactions which do not required a control/data return. As before, we identify two rules: rule *(c)* that can be applied when the *op2* interaction is performed during the execution of the *op1*, before its execution time *t1* is totally spent; and, rule *(d)* that can be applied when the *op2* interaction starts at the end of *op1*.

The modeling of both situations follow the same guidelines of the previous ones. In rule *(d)* the QN customer enters in the *A* by asking *op1* service. When *A* ends its work, the customer enters in the *B* to receive the *op2* service. The rule *(d)* models the parallel execution of part of the *op1* service and the *op2* service by means of the fork feature of the QN modeling. Again, whenever a job leaves *A*, it forks into two new jobs, one entering *B* and the other going back to *A* with different job type and a higher priority. As before, this guarantees that *A* terminates the current request before to serve a new one. Since the interface is asynchronous we do not model the wait status of A if B is busy. Indeed, we associate an infinite waiting queue to B to represent asynchrony.

The rule *(c)* can be applied when it is possible to estimate the times *t1'* and *t1"*, otherwise we have to

approximate the model by using the second rule. The same approximation can be fairly done whenever we plan to use an analytical analysis technique based on MVA algorithm to evaluate the QN model for the same reasons reported before.

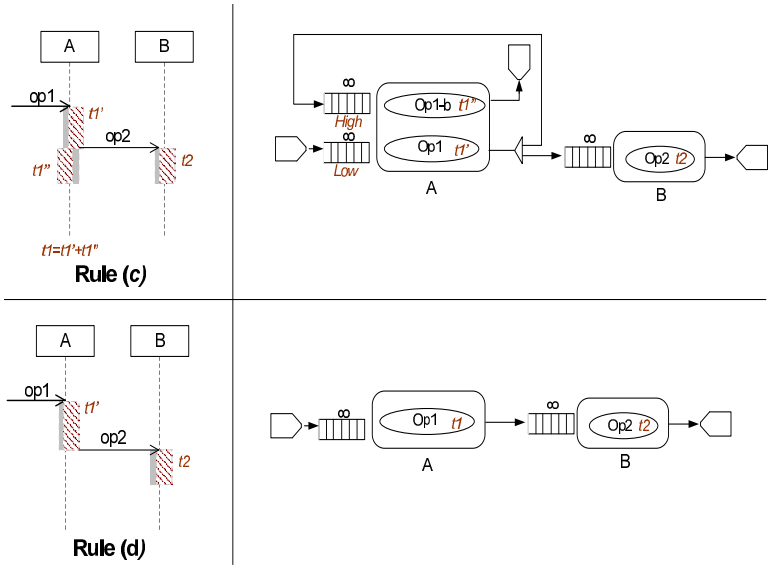


Figure 4.13: Translation Patterns for Asynchronous Signals.

SYNCHRONOUS CALL ACTION

A call action implies a control/data return to the sender. Due to its characteristics the corresponding QN sub-model is more complex compared to the one generated for a signal. As shown by the Figure 4.14, synchronous call action has only one behavioral description. The *A* component, during its activation, calls the action *op2* of *B* and waits for some results or simply for the action termination. When it receives back the control, it continues its task. The time *t1* needed for *op1* can hence be split into three portions $t1', t1''$, and $t1'''$, representing the time elapsed between the *op1* receipt and *op2* invocation, the waiting phase, and the time between the control return and the *op1* termination, respectively. Let us observe that $t1''$ time is equal to $t2$.

The *op2* interaction is synchronous and we model this interaction as before by associate a zero capacity queue to the *B* service center and the BAS blocking policy to *A*. Moreover, to model the active waiting of *A*, we fork the job outgoing *A* into two jobs, one entering *B* and the other going back to *A*. When the waiting time is elapsed, the job exits the QN. Finally, when *B* terminates its task a job is re-directed to *A* as a *return* request in order to model the control return to *A* software component. To guarantee the sequence of actions we introduce three different priorities to *op1*, *wait* and `return` actions.

When it is unknown the time fragments *t1',t1", and t1"'*, we use the second QN sub-model in the Figure to model the call action (that is the one shown in rule (b)). The differences here is that the time *t1* is spent by *A* completely before to send a request to *B*. When *B* terminates the required task the job exits the QN sub-net. As before, this approximation can be done every time the QN model is resolved by a MVA-based algorithm.

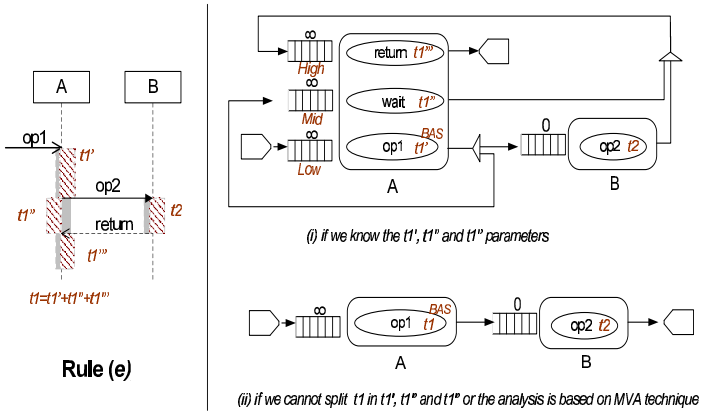


Figure 4.14: Translation Patterns for Synchronous Call Actions.

ASYNCHRONOUS CALL ACTION

For this type of interaction we introduce three different corresponding QN patterns. This type of interaction and the corresponding QN modeling for the first and the third options are the same of before except for the asynchronous communication mechanism. The component *A* continues to work on its task after having requested some elaborations to *B*. *op1* can be split into three parts having $t1', t1''$ and $t1'''$ execution time, such that $t1 = t1' + t1'' + t1'''$. The only difference here with respect to the rule *(e)* is the asynchronous nature of the *op2* interaction. This implies that B has a non null waiting queue capacity and A is not a BAS server. We use again different probabilities in A job classes to guarantee the accomplishment of a request before a new one is served by A.

When it is unknown the time fragments *t1',t1", and t1"'*, we use the second QN sub-model in the Figure to model the call action (that is the one shown in rule (d)). The differences here is that the time *t1* is spent by *A* completely before to send a request to *B*. When *B* terminates the required task the job exits the QN sub-net. As before, this approximation can be done every time the QN model is resolved by a MVA-based algorithm.

For the asynchronous call action we also introduce a QN pattern of the intermediate complexity and accuracy, where we remove the fork a join facilities (option (ii) in the Figure). We modelling introduce this modellig that can be used when the A sender can serve other requests while it waits for a B response.

4.3.7 QN PATTERNS FOR THE FRAGMENT OPERATORS IN SEQUENCE DIAGRAMS

REFERENCE OPERATOR

The `Reference` operator allows the definition of a reference to a behavior already defined by another sequence diagram. In Figure 4.16 a `reference` operator, identified by the *ref* keyword, is used to indicate that the behavior of the *Scenario* includes the behavior described in *Scenario2* sequence diagram. The `Reference` operator models a change of chain in a multi-chain QN. This means that the chain of the *Scenario* use case, has a sub-chain corresponding to the one modelling the behavior of the *Scenario2* customer type.

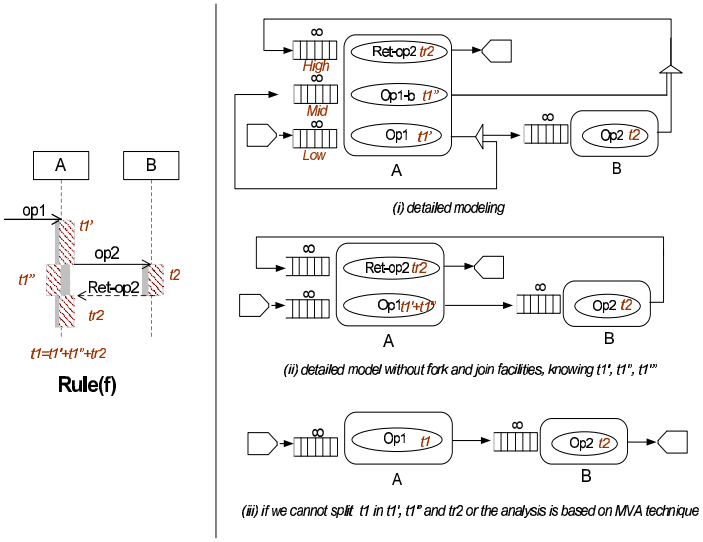


Figure 4.15: Translation Patterns for Asynchronous Call Actions.

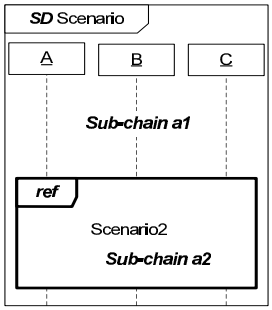


Figure 4.16: *Reference* Sequence Operator translation rule.

ALTERNATIVE OPERATOR

"The `Alternative` operator designates that the interaction fragment it represents models a choice of the behavior. At most one of the operands will executes. The operand that executes must have an implicit or explicit guard expression that evaluates to true at this point of interaction" (quoted from [148]). An `alternative` operator, identified by the *alt* keyword, models several mutual exclusive alternative behaviors. The `Alternative` operator models a branching in a QN. The alternative behaviors in the interaction operator represents the different routings of the customers among the services of the service centers. The routing probabilities among the alternatives are annotated in the sequence diagram by the `<<PAstep>>` stereotype and their sum must be equal to 1. As an example, let us consider Figure 4.17. The `alt` operator has n alternatives each guarded by a condition (`cond1`, `cond2`,...,`condn`). Each alternative behavior has a probability to be executed (`¶1`, `P2`,...,`Pn`). The alternative behaviors are modelled as different routing in the QN model (or sub-chains) with the same probabilities.

OPTION OPERATOR

The interaction operator `opt` designates that the behavioral fragment it delimits represents a choice of behavior where either the (sole) operand happens or nothing happens. An option is semantically equivalent to an alternative fragment where there is one operand with non-empty content and the second operand is empty [148].

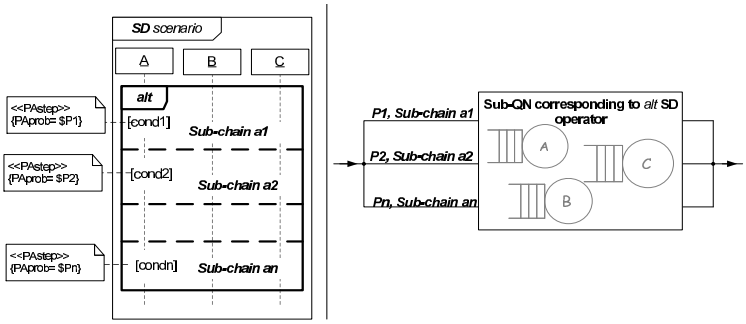


Figure 4.17: *Alternative* Sequence Operator translation rule.

In Figure 4.18 we show at the left-hand side the `opt` operator inside a generic scenario. To be properly processed, it should have annotated the probability it will happen. To this aim we use `<<PAstep>>` stereotype and the `PAprob` tag value. The stereotype annotates the first interaction inside the `opt` frame. The methodology maps to such operator the sub-model at the right-hand side of the Figure 4.18. The sub-model presents a branching point in the correspondence of the *opt* operator. A job can enter in the QN sub-model with the probability `P1`(that is the probability annotates in the sequence diagram) or not with `1-P1` probability. Either it enters in the sub-model or it does not, it will handle according to the behavior described in the sequence diagram subsequent the `opt` fragment.

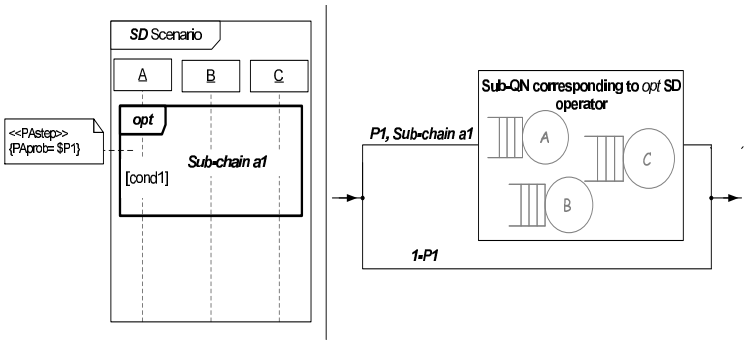


Figure 4.18: *Option* Sequence Operator translation rule.

BREAK OPERATOR

The interaction operator `break` designates that the fragment represents a breaking scenario in the sense that the operand is a scenario that is performed instead of the remainder of the enclosing behavior. Thus the break operator is a shorthand for an `Alt` operator where one operand is given and the other assumed to be the rest of the enclosing behavior. `Break` fragments must be global relative to the enclosing scenario.

The `break` operator models exceptional behaviors, such as, for example, the behavior the system has to take in case some errors occur. The `break` operator has to be annotated with the probability that it happens. We use the `<<PAstep>>` stereotype and the `PAprob` tag value attached to the first interaction inclosed in the `break` operand. The similarities with the `alt` interaction operator are reflected on the QN sub-model as the right-hand of the Figure 4.19 shows. The only difference is that the two alternatives in the QN sub-model correspond to the break behavior and to the remainder of the enclosing behavior respectively.

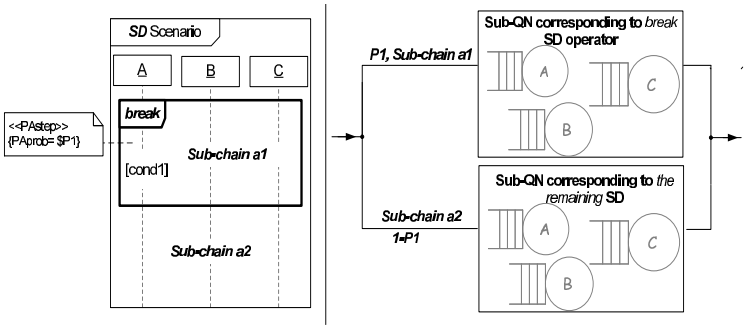


Figure 4.19: *Break* Sequence Operator translation rule.

PARALLEL OPERATOR

"The `Parallel` operator designates that the interaction fragment it represents models a parallel merge between the behaviors of the operands. The event occurrences of the different operands can be interleaved in any way as long as the ordering imposed by each operand as such is preserved" (quoted from [148]). In Figure 4.20 left-hand side a `parallel` operator, identified by the *par* keyword, defines n parallel behaviors. In a QN model, the `parallel` operator represents a fork. In the right-hand side of Figure 4.20 there is reported the QN pattern for the `parallel` operator of the left-hand side of the figure. The QN sub-model has a fork (the triangle at the left) with n different out-coming sub-chains that occur concurrently. A join QN feature (the triangle at the right) is used to join the sub-chains at the end of the traces defined in the `par` frame.

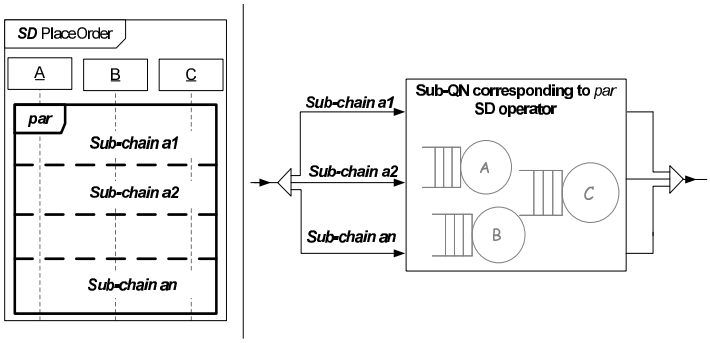


Figure 4.20: *Parallel* Sequence Operator translation rule.

LOOP OPERATOR

The `loop` interaction operator designates that the fragment represents a loop whose operand will be repeated a number of times. The number of repetitions is indicated by the guard of the operator that may include a lower and an upper number of iterations as well as a boolean expression. The semantics is such that a loop will iterate at least the `minint` number of times and at most the `maxint` number of times. After the minimum number of iterations have executed, and the boolean expression is false the loop will terminate.

In Figure 4.21 a sequence diagram enclosing a `loop` operator is shown. Its guard has a `minint` equals to `k`, a `maxint` equals to `m` and the boolean expression `exp`. The behavior of the `loop` operator is modelled as a sequence of the interactions by respecting the basic rules introduced in the previous section. However to model the repetition of such a behavior, we weigh the service times of the component interactions involved

in it by the repetition numbers (minint, maxint) and the boolean expression (exp) in the guard of the `loop` operator. In general we can make a worst case analysis by imposing a weight equal to the `maxint`, or a best case analysis where we consider `minint` as weight, or a mean analysis where we consider the mean value between `maxint` and `minint`.

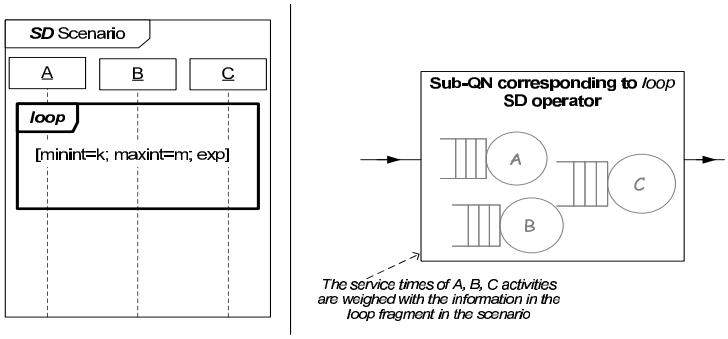


Figure 4.21: *Loop* Sequence Operator translation rule.

IGNORE/CONSIDER OPERATOR

The interaction operator `ignore` designates that there are some message types that are not shown within this combined fragment. These message types can be considered insignificant and are intuitively ignored if they appear in a corresponding execution. Alternatively one can understand ignore to mean that the messages that are ignored can appear anywhere in the traces. Conversely the interaction operator consider designates which messages should be considered within this fragment. This is equivalent to defining every other message to be ignored.

The approach deals with only the `ignore` operator. The semantics considered is as follows: the operator identifies traces or interactions that can be ignored in the performance analysis process. This operator is useful since it allows the slicing of the software system behavior. By means of it the simplification of the target model can be done by reasoning at the software designers level. This is extremely useful when it is necessary to simplify the target QN model in order to permit a reasonable performance evaluation. Quite often, in fact, the software system model describes details that are useless for the performance analysis and that make the target model too complex to be evaluated in an acceptable time or with a given accuracy.

NOT CONSIDERED FRAGMENT OPERATORS

At the moment we do not consider the `Weak` and the `Strict` sequencing,`Negative`, `Critical Region` and `Assertion` interaction operators.

The interaction operator `critical` designates that the fragment represents a critical region. A critical region means that the traces of the region cannot be interleaved by other event occurrences (on those Lifelines covered by the region). This means that the region is treated atomically by the enclosing fragment when determining the set of valid traces. Even if this operator is important for the performance analysis since it limits the sharing of the resources, at the moment the approach does not deal with it.

The `Weak` and the `Strict` Sequencing are not considered since they give information about the weak or strict sequencing of the events. The approach we propose does not make such difference. Indeed, to our aims, we need to know which are the events and the cause-effect relations among them, and we do not mind if the sequencing is strictly respected or not when several behavior are interleaved.

The `negative` interaction operator represents traces that are defined to be invalid. The performance analysis takes into account only valid execution traces, hence the approach wastes all the traces defined invalid.

Finally, the interaction operator `assert` designates that the fragment represents an assertion. Performance analysis cannot take advantages from such information thus it is not considered.

4.3.8 Further Considerations

The new approach, due to the use of the UML profile, requires some knowledge on performance analysis, and on how to determine the additional information needed to carry on the evaluation of the generated QN model. This is not trivial. At the moment, do not exist automatic approaches that assist in the whole process of performance analysis. To this respect a relevant effort is still required to software designers attending in the process itself.

However, it is not common to find specific performance skills in software designers and, due to the high level of abstraction of the software architecture description, the additional information to parameterize the QN model is not easy to determine.

The assumption of using an ideal hardware architecture should help to overcome these drawbacks. As we already discussed in [18], in our framework we consider QN performance models of SA to evaluate and to compare SA specifications. The goal is to derive performance results that can be interpreted in terms of SA design artifacts. Therefore, unlike other approaches that build QN models as a combination of the hardware platform with software requirements, we consider QN models as performance models at the level of the SA. The definition of a QN model at the SA level cannot be completely specified because of the high level of abstraction. If some parameters or characteristics of the QN are not specified, it is then up to the designer their definition. The model parameter instantiations correspond to potential implementation scenarios, and the performance evaluation results can provide useful insights on how to carry on the development process.

Due to the high level of abstraction, we do not model software resources other than the software components presented in the component diagram. This means that buffers are considered if they are explicitly modelled as software components. For what concerns other software artifacts, such as semaphores or locks, in general they are not explicitly modelled in an SA. They contribute in the specification of the communication protocol used by the software component to interact each other, hence, in a SA specification, they are embedded in the connectors.

Of course, the methodology deals with all kind of software systems that can be modelled by means of UML facilities we use to model the software architecture, and that have characteristics that can be modelled by means of a QN model.

At the moment, as feedback, the approach annotate the UML diagrams with the performance indices the analysis determines. For this aim the approach uses the tag values `<<PAutilization>>` and `PAthroughput` of the `<<PAhost>>` stereotypes and the tag value `PArespTime` of the `<<PAstep>>` stereotypes. More complex feedback provision can be design by considering the techniques we introduced in Section 2.3 of Chapter 2. Such techniques suggest to the developers possible changes in the software architecture design that may improve the performance of the whole system.

As final remarks, we want to observe that, even if the new approach allows the generation of a QN model closer to the software architecture, the obtained performance model is more complex. This complexity implies that we have to use more complex techniques to evaluate the new QN model. These techniques, quite often, are not analytic as instead they were in the previous approach. In particular, whenever we have a parallel operator in a sequence diagram, we have a fork in the QN model. A QN with forks (namely Extended QN) can be evaluated only through simulation techniques.

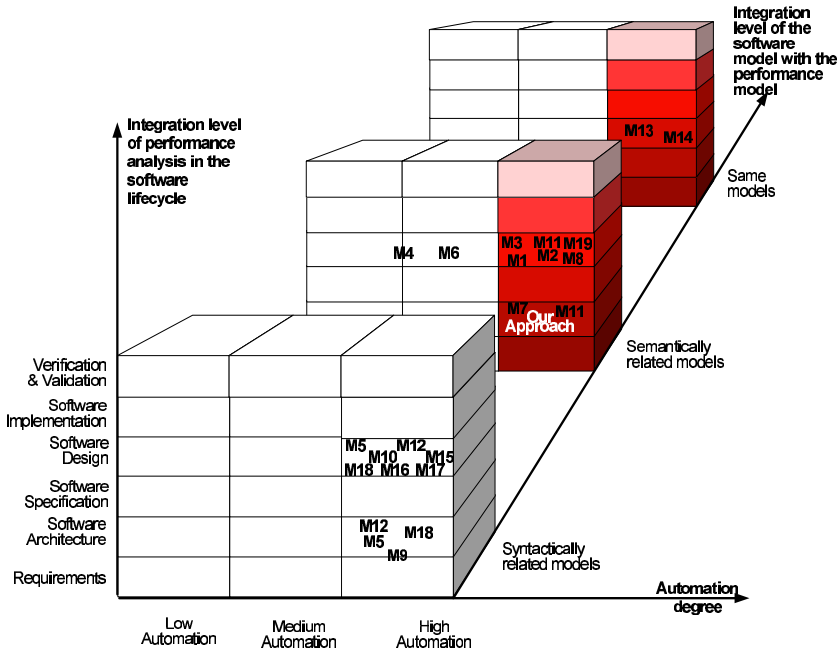


Figure 4.22: Classification of our Approach.

Finally, by considering the classification propose in Chapter 3, we classify the proposed approach among the ones that can be applied at the software architecture level, have high automation degree and present semantically related models generation technique.

Figure 4.22 shows such a classification.

4.4 The Electronic Commerce System

In the electronic commerce system, there is a supplier that publishes his catalogue on the web [83]. The supplier accepts customer orders and delivers the ordered items maintaining all the relevant data. He needs to maintain information on data, on the catalogue and on the orders purchased by his customers. Each registered customer has a cart where he/she can insert or delete items. The customer can order only if the cart is not empty. The system also allows the customer to monitor the order status and to confirm the delivery in order to permit the payment.

In Figure 4.23 we show the Software Architecture of the system, where we identify some databases (Customer DB, Cart DB, Order DB,etc.) and four components (CustomerProcess, SupplierProcess, etc.). Each customer has associated a (individual) CustomerProcess and there is a server for each involved database that permits to communicate with it. The interactions with these servers are asynchronous.

For the sake of presentation, we consider a simplified version of this system. We focus on the customer view by considering only the software system services reported in Figure 4.24. In the use case diagram the association between the use case and the sequence diagram describing the corresponding system behavior is expressed by means of notes. All the use cases for the e-commerce system have associated only a sequence diagram, and both InsertItem and DeleteItem use cases use the BrowseCart use case.

In Figure 4.25 we present the UML 2.0 Component Diagram for the portion of the e-commerce system we consider. This diagram highlights the software components and their required and provided interfaces. The

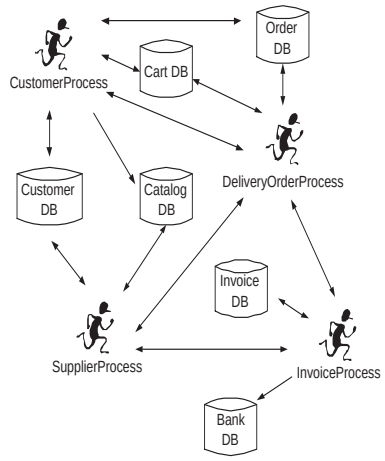


Figure 4.23: SA components of the Electronic Commerce System.

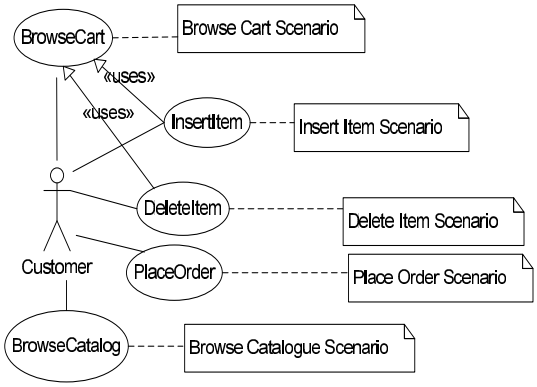


Figure 4.24: UML Use Case Diagram.

provided interfaces are represented by a circle whereas the required ones are represented by a semicircle.

In general, an interface is composed by a set of operations provided or required by a software component. Here, to simplify the presentation of the parameterization phase, we assume that an interface is composed of a single operation from with the interface derives the name. The Component diagram is annotated, according to the UML SPT profile, in order to introduce performance aspects of the software components and of their interfaces. All the values for the execution times are assumed as the `assm` keyword in the tag value indicates. Our intent is to carry on a performance analysis at the software architecture level when such information is not available, whereas it is guessed by the designers on the basis of their experience.

As described by the annotations on the component interfaces, all the components are queuing centers except the CustomerProces that is a delay center. In fact, all the annotations on its provided interfaces show the `PAdelay` tag value.

The Customer Process is a component running at the user side. There is an instance of it dedicated to each used logged in the system. This design choice makes the CustomerProcess component an unshared resource and hence its execution times constitute delays.

From Figure 4.26(a) to Figure 4.28 we report the scenarios for the use case considered. The scenarios are expressed by means of the Sequence Diagrams of UML 2.0, as we detailed before. All the sequence

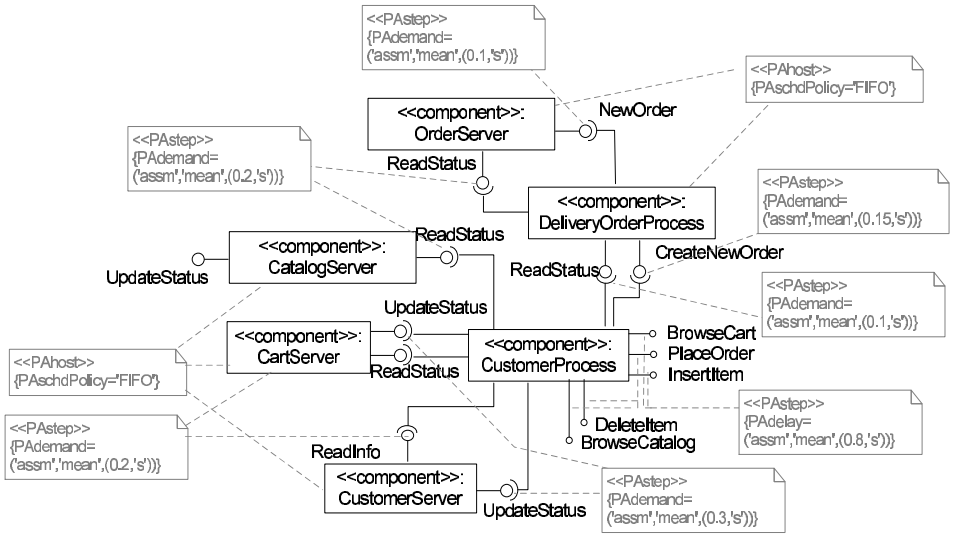
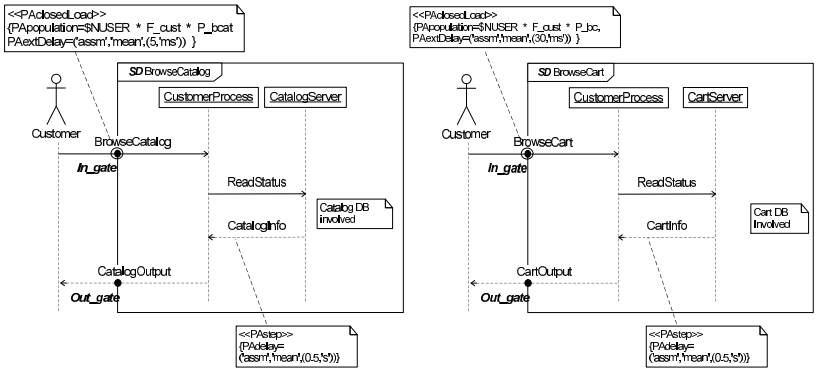


Figure 4.25: Component diagram of the considered portion of the e-commerce system.



(a) Browse Catalogue Scenario. (b) Browse Cart Scenario.

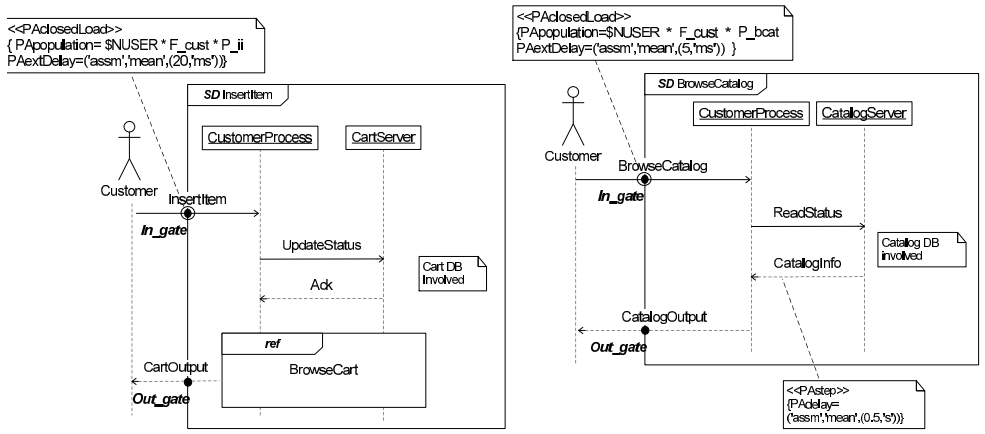
Figure 4.26: E-commerce Scenarios.

diagrams report annotations for the workload they impose to the system. The workload for each of them is closed and it is defined by the number of users (`PApopulation` tag value) and the time needed to generate a request (`PAextDelay` tag value). It is worthwhile to observe that the number of users (`PApopulation` tag value) can be a function of external variables (`$NUSER` in the Figure 4.26(b)) and some other values suitable for the particular case (`F_cust`, `P_bc` values in the Figure 4.26(b), that represent the probability a customer logs in the system and the probability a customer request the BrowseCart service, respectively).

We choose to model all the workload as closed ones since the only the registered customers can access to the services and hence the population is fixed. Indeed, the BrowseCatalog is a service available to all the internet users and not just to the registered ones. Hence its workload should be open. However to simplify the modelling we assumed it closed.

Both the scenarios in Figure 4.26 has an annotation associated to a return message (CatalogInfo and CartInfo, respectively). This annotation models the time the CustomerProcess spends after having received the return message and before sending the following return interaction. Again, this time is a delay since the CustomerProcess is a delay center.

InsertItem and DeleteItem uses the BrowseCart use case. This modelling is reported in the sequence dia-



(a) Insert Item Scenario. (b) Delete Item Scenario.

Figure 4.27: E-commerce Scenarios.

grams of Figure 4.27 where the `ref` interaction operator referring the BrowseCart scenario is used.

We assume that an item can be removed from the cart only if it is in the cart, otherwise an error occurs. To model the errors occurring we use in the DeleteItem sequence diagram the `break` operator. the first interaction in the `break` operand is annotated with the assumed probability under which the error can happen.

As final scenario we present the PlaceOrder sequence diagram in Figure 4.28. It shows an example of alternative behaviors by means of the `alt` operator. The first interaction of the both alternatives herein modelled, have associated the assumed probability that the system behaves as one of the alternatives. Observe that the sum of those probabilities is 1 as required by the approach.

4.4.1 QN MODEL GENERATION FOR THE CASE STUDY

The proposed approach generates a QN model by executing three steps as outlined in Section 4.3.4. In the following we show the three steps on the e-commerce case study.

IDENTIFICATION OF THE QN CUSTOMERS - The use case diagram in Figure 4.24 has five use cases, hence the QN customers are of five different types. The behavior of such customers are described by the corresponding sequence diagrams annotated in the diagram. Such sequence diagrams define five QN chains.

IDENTIFICATION OF THE SERVICE CENTERS AND THEIR CHARACTERISTICS - From the Component Diagrams we identify six QN centers, five are queuing service centers while one, the CustomerProcess, is a delay center. All the service centers have a service type for each interface of the corresponding component in the component diagram. The CustomerProcess component, instead, is modelled by a set of delays, one for each request type and one modelling the internal actions. This choice is conform to the annotations in the component diagram where it is specified that the provided interfaces of the CustomerProcess are delays. We remark here that this modelling respects the architectural constraint concerning the presence of a CustomerProcess instance for each e-commerce customer.

From the `<<PAhost>>` stereotypes the approach extracts the scheduling policy of each centers. From the annotations on the provided component interfaces it extracts the service times or the delays of each service

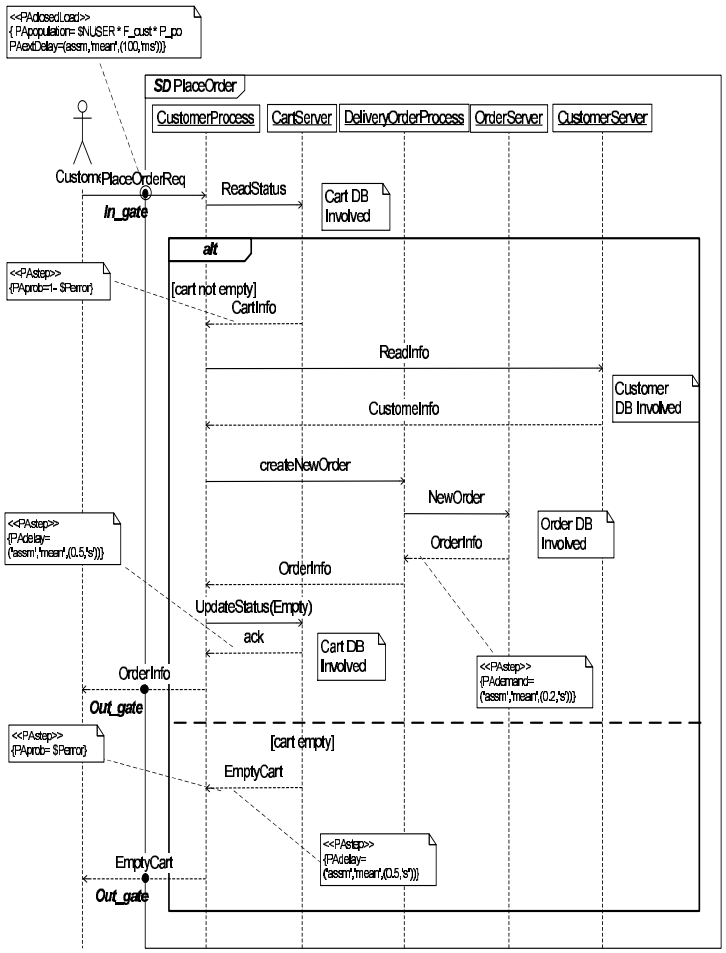


Figure 4.28: Place Order Scenario.

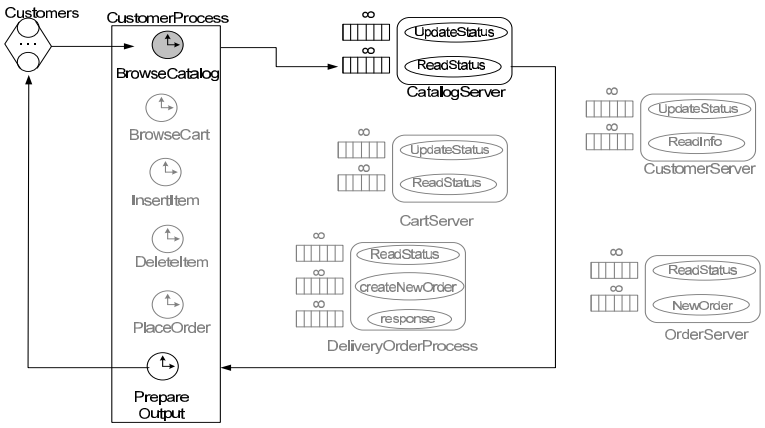


Figure 4.29: Chain in the QN model corresponding to BrowseCatalog Scenario.

of the components needed to parameterize the QN model.

DEFINITION OF THE QN CHAINS AND THE RELATIVE WORKLOADS - In this step, the workloads intensity entering the system is derived by the information annotated in the sequence diagrams. In our case study all the workload are closed workload. Here the traffic generators are the e-commerce users. A generic e-commerce user is modelled by a set of infinite server, one for each QN customer type whose parameterization is extracted by the workload annotations in the sequence diagrams.

In this step, by applying the rules defined in Section 4.3.5 opportunely instantiated, the approach also defines the QN chains for each customer type. Since all the interactions among components are asynchronous call message we apply the rule (f), in its simpler fashions (that are option (ii) and (iii)).

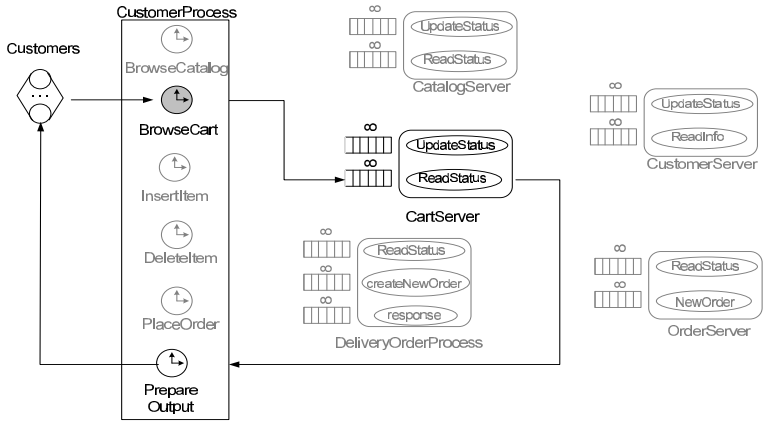


Figure 4.30: Chain in the QN model corresponding to BrowseCart Scenario.

A chain is defined through the sequence diagram describing its behavior. A chain does not necessarily involve all the service centers, but it will visit the centers corresponding to the components involved in the sequence diagram. If we consider, for an example, the chain for the *browseCart* (see Figure 4.30), it involves the CartServer and the CustomerProcess service centers only, as specified in the relative sequence diagram of Figure 4.26.

In Figures 4.29 and 4.30 we show the chains for the *BrowseCatalog* and *BrowseCart* customer classes respectively. These chains are the most simple among the ones presented. Both of them are generated by applying the (iii) and (ii) options to the first and the second asynchronous call actions, respectively. The chains terminate in the infinite servers (Customers) that generate the job types, as closed workloads specify.

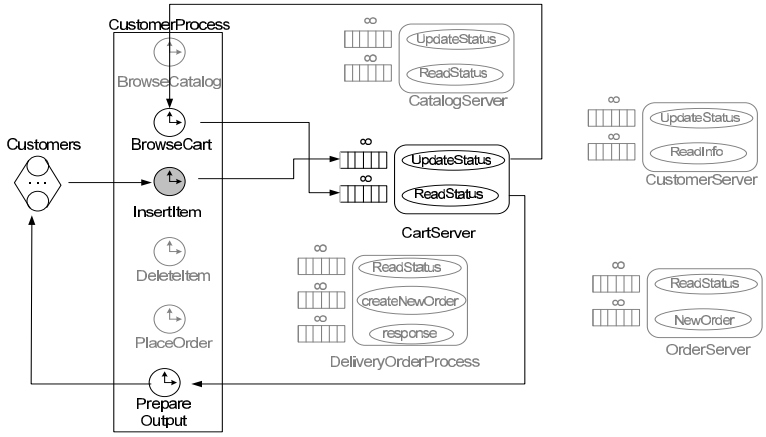


Figure 4.31: Chain in the QN model corresponding to InsertItem Scenario.

In Figure 4.31 we report the chain for the *InsertItem* request type. To generate this chain we used the

translation rule defined for the `reference` operator since this operator appears in the relative sequence diagram (see Figure 4.27(a)). This operator indicates that the behavior of this *InsertItem* QN customer type includes the behavior described in *browseCart* sequence diagram. The `Reference` operator, in fact, models a change of chain in a multi-chain QN. Thus the *InsertItem* chain has a sub-chain corresponding to the one modelling the behavior of the *browseCart* customer type. In Figure 4.31 the point where the *InsertItem* chain becomes the *BrowseCart* chain is when the QN customer enters the CustomerProcess delay center by asking the BrowseCart service.

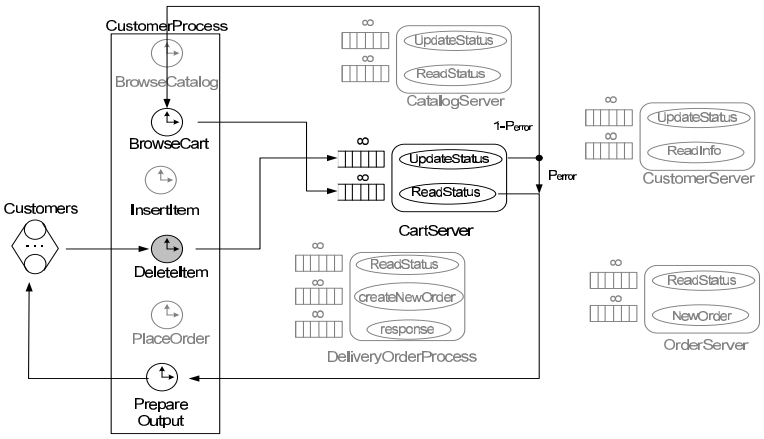


Figure 4.32: Chain in the QN model corresponding to DeleteItem Scenario.

In Figure 4.32 we report the chain for the *DeleteItem* request type. To generate this chain we used the translation rules defined for `reference` and `break` operators since they appear in its sequence diagram (see Figure 4.27(b)). The treatment for the former is the same of before. The `break` operator designates a breaking scenario in the sense that the operand is a scenario that is performed instead of the remainder of the enclosing interaction fragment. In the DeleteItem scenario, the `break` operator models an error that can occur in such scenario. To model this semantics the approach introduces a branching point with two possible routes: one, with probability `Perror`, that breaks the scenario by routing the customer towards the PreperaOutput service of the CustomerProcess, and the other one with probability `1-Perror`, that models the remainder of the scenario, that routes the customer towards the BrowseCart service of the CustomerProcess, where the change of chain is executed. The probabilities of both routes are extracted from the sequence diagram.

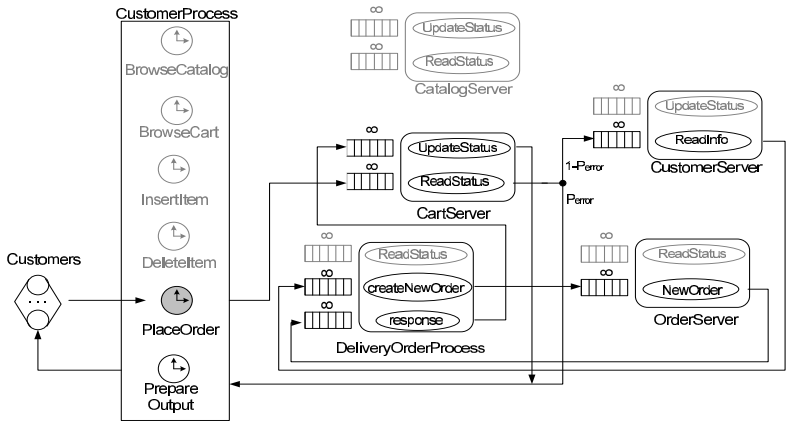


Figure 4.33: Chain in the QN model corresponding to PlaceOrder Scenario.

Finally, in the Figure 4.33, we report the chain for the *PlaceOrder* request type. Here the translation rule for the `alternative` operator is used. As the sequence diagram in Figure 4.28 shows, the `alternative`

operator has two alternative behaviors, one with *not empty cart* guard expression and the other with *empty cart* guard expression. The `Alternative` operator models a branching in a QN. The alternative behaviors in the operator are the possible routings of the customers among the services of the service centers. The routing probabilities among the alternatives are annotated in the sequence diagram. By considering the Figure 4.33, the `alternative` operator defines the branching point having two alternatives one with `1-$Perror` probability and the other with `$Perror` probability to be covered.

4.5 SUMMARY

In this Chapter we have presented the new version of our approach to software system performance analysis. The approach allows the automatic generation of a performance model based on QN models. This new version addresses the problems, discovered in the previous version, with respect to the multiple services software systems, by generating a multi-chain QN model.

Differently to the previous version, the new approach uses UML 2.0 as notation to describe the software system architecture. The migration has been driven by two factors: the more expressiveness of UML 2.0 diagrams that allows us the definition of a compositional approach, and the presence of an UML profile that defines suitable capabilities to annotate the diagrams with information on performance aspects.

The new version also deals with the parameterization phase of performance models by assuming that the software designers suitably label the used UML diagrams with useful information.

Even if the new approach allows the generation of a QN model closer to the software architecture, the obtained performance model is more complex. This complexity implies that we have to use more complex techniques to evaluate the new QN model. These techniques, quite often, are not analytic, but can be approximates or simulative.

CHAPTER 5

APPLICATION OF TWO PREDICTIVE PERFORMANCE ANALYSES TO A COMPLEX CASE STUDY

The approaches on early performance validation recently defined allow software developers to address performance issues since the first phases of software life cycle, when crucial decisions on the software system are taken. However, the lack of completely automated methodologies and the need of special skills prevent the introduction of this type of analysis in real industrial contexts. Both automation and transparency of the approach might, in fact, allow the application of these methodologies on industrial products without delaying the software development process.

In this direction we have done and we here report our experience in the modelling and analysis of a real telecommunication system at the design level. The system is the Naval Integrated Communication Environment (NICE) developed by Marconi Selenia. This system is responsible for managing and monitoring the heterogeneous equipments composing a naval communication system, by providing several secure communication facilities. At the beginning we applied a technique based on stochastic process algebra with the support of the Æmilia Architecture Description Language and the TwoTowers toolset. For some scenarios of interest this approach suffers of the state space explosion problem, therefore we experimented another technique based on simulation.

We discuss the advantages and the disadvantages of the applied techniques and we compare them with respect to a framework specifying suitable characteristics the approaches should have. We also discuss how to take advantage of the integration usage of different methodologies when applied on complex real systems.

The work this chapter discusses has been outlined in [58, 31, 57] and it is described here in details.

5.1 PREDICTIVE PERFORMANCE ANALYSIS AND REAL INDUSTRIAL CONTEXT

Recently many approaches on early performance validation of software artifacts have been defined [35, 27]. However it has been experienced that they are not part of the development process in complex industrial contexts. From this observation we asked ourself which characteristics an approach should show in order to facilitate its integration in the industrial realties.

We tried to integrate the software performance analysis in the Lab.NMS C2 of Marconi Selenia Communications software development environment by applying two approaches to the real telecommunication software system provided. Marconi Selenia Communications is a global communications and information technology company headquartered in Rome.

When we integrate an analysis to a real context we have several constraints to respect. First of all, the choice of the software artifacts and relative description notation is imposed by the development process we would

integrate with. In fact, integration means that if a performance analysis is applied, this should not impose any requirements on the software modeling and on the development. This applies, as a consequence, that we are not totally free in the choice of the approach to use. Rather, we have to apply a predictive performance analysis that takes in input the software artifacts described by the industrial process with the same notation. For example in Marconi Selenia the software development process is based on the UML notation in the first phases of the software life cycle. Hence the analysis we carried on was at the Software Architecture (SA) over UML-based SA description.

Moreover, the performance analysis requires additional information, such as for example operational profile and workloads, that the designer should provide to carry on the analysis. To be used in a real context, the predictive approach should require small effort in providing and annotating the software models with such an information.

Since, the industry has strict and short time to market, the analysis should quickly provide insights on the software models and in an efficient way. To this scope automation is a key factor in the integration of an approach in a real software development context.

Finally, the performance results interpretation and the consequently feedback provision have to be as ease as possible in order to allow software designer, with no special skills and deep knowledge in performance analysis, to pick up and understand the problems the analysis captured. This feature facilitates the feedback provision at the software architecture level.

In the following we introduce the criteria useful to compare different analysis techniques and to evaluate their suitability in complex industrial context applications. Such attributes are:

Transparency We refer as transparency the desiderate property of minimal influence of the performance analysis approach on the development process, preventing also the request to the software designer of special skills in performance modeling. The absence of such a property might prejudice its applicability in concrete and complex situations. We may define the transparency as a combination of the following aspects:

- **Performance Model Derivation** The performance model should be easily derived from the software specification used by the industry. No additional efforts in the software behavioral modelling should be asked to the design team by the predictive performance analysis.
- **Software Model Annotation** In order to obtain a performance model, it is necessary to provide quantitative, performance-oriented information which can be used to build the performance model. There are several ways to provides such an information, but the optimal solution is to annotate them directly on the software models by using their same features or an its extension in order to require less efforts to the designers to learn it. Otherwise the software model annotation task could delay the development process and prevent the usage of the performance analysis. Same conclusions might be made if the amount of the additional information needed to carry on the predictive analysis is too high or if it is difficult to estimate.
- **Performance Indices** The performance indices of interest should be easily specified without asking for the knowledge of the underly theory. The specification of them should be as natural as possible for the software designer. Again, the analysis should provide the accuracy level of the derived quantitative figures.

Automation The availability of the automation could allow the application of these methodologies on industrial products without delaying the software development process. This aspect is fundamental today due to the short time to market the industry has to respect to improve its business. To allow the integration of software performance analysis in the industrial software development process it is required the use of an automated methodology that automatically generates a performance model of the provided specification of the software system, specifies the performance indices of interest, resolves the performance model to obtain values for such indices and hopefully reports such analysis

results on the software models by highlighting (potential) performance problems about the provided design.

Result Interpretation When performance indices have been calculate, from their interpretation the design should guess the goodness of the proposed design. A suitable approach comes up with an easy results interpretation. In contrast to this desiderate characteristic, some approach allows the specification of the performance indices as a function of others measures that the approach is able to determine. Hence, whenever the analysis is done, to obtain the values for the performance indices, the designer has to combine them in a way that could be not intuitive and difficult.

Generality This criteria gives information about the application domains and architectural styles (such as client/server, layered architectures and others) an approach can be applied. More the approach is general more it is powerful. Moreover this attribute can indicate if the approach is able to deal with some particular aspects such as for example to model fork/join systems, simultaneous resource possession, general time distributions and arbitrary scheduling policies.

Feedback Whenever the performance indices have been calculate it should be extremely useful that the approach reports them back into the original software description, identifies parts of the design that could lead to potential performance problem, and suggests some design alternatives to overcome the identified problems. However, in contrast to this desiderate characteristic, some approach applies complex rules to generate the performance model such that at the end of the analysis it is hard also for performance experts to interpret them and derive some feedback upon the software model.

Scalability The selected approach should be scalable, meaning that the complexity of the performance model should increase linearly with the software model size.

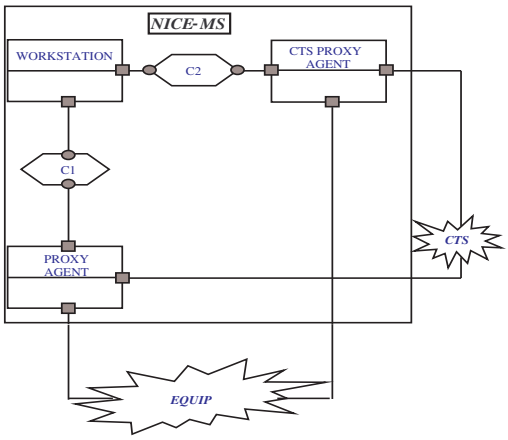


Figure 5.1: NICE Static Software Description

5.2 An Industrial Case Study: The Naval Integrated Communication Environment

The Naval Integrated Communication Environment (NICE) is a project developed by Marconi Selenia. It provides communications for voice, data and video in a naval communication environment. It also provides remote control and monitoring in order to detect equipment failures in the transmission/reception radio chain. It manages the system elements and data distribution services, enables system aided message preparation, processing, storage, retrieval distribution and purging, and it implements radio frequency transmission and reception, variable power control and modulation and communications security techniques. The

NICE-MS complexity and heterogeneity (different hardware and operating systems and relative applications) together with its real time context needs the definition of a precise software architecture to express its coordination structure. The system involves several operational consoles that manage the heterogeneous system equipment including the ATM based Communication Transfer System (CTS) through blind Proxy computers.

SubSystem	Component
WORKSTATION	COORDINATOR P1 P2
CTS PROXY AGENT	CTS PROXY AGENT
PROXY AGENT	PROXY AGENT
EQUIP	EQUIP

Table 5.1: Subsystem Decomposition

On a gross grain its Software Architecture is composed of the NICE Management subsystem (NICE MS), CTS and EQUIP subsystems, as highlighted in Figure 5.1. The WORKSTATION subsystem represents the management entity, while the PROXY AGENT and the CTS PROXY AGENT subsystems represent the interface to control the EQUIP and the CTS subsystems, respectively. Subsystems are connected by means of connectors (C1 and C2) that allow the communication between them.

Each subsystem represented in Figure 5.1 is composed of one or more components. Table 5.1 shows the identified components. In particular, only the WORKSTATION is modelled by three components, that are COORDINATOR, P1 and P2. The COORDINATOR component is the control entity that coordinates all the actions to be performed time by time. Instead, the P1 component interacts with PROXY AGENT subsystem and the P2 component interacts with CTS PROXY AGENT subsystem.

Actually, the running configuration of the software system will be composed by one instance of WORKSTATION subsystem, two instances of CTS PROXY AGENT subsystem, ten instances of PROXY AGENT subsystem and at least twenty EQUIP subsystem instances. In general, a PROXY AGENT instance manages at least two EQUIP instances.

The more critical component is the NICE MS subsystem. It controls both internal and external communications and it satisfies the following class of requirements: fault and damage management, system configuration, security management, traffic accounting and performance management. Each requirements class groups a set of functionalities. All these class of requirements must satisfy some particular performance constraints. For the sake of the presentation, in the following we focus *Recovery* activity. The scenario is described in Figure 5.2. The estimated execution times of the actions in the scenario are exponentially distributed random variables; the mean values were provided by the system developers and are reported in Table 5.2.

RECOVERY SCENARIO - The activity of system recovery belongs to the class of Fault and damage management requirements. This activity reacts to the failure of a remote controlled equipment. The recovery consists in a set of actions, part of which are executed on the equipment in fault and the others are executed on the CTS subsystem.

Figure 5.2 shows the UML sequence diagram relative to the recovery activity in terms of components interactions.

For the sake of the modeling, as shown in Figure 5.2,the WORKSTATION subsystem is decomposed in three main components: COORDINATOR, P1 and P2 where P1 and P2 are auxiliary components interacting with PROXY AGENT subsystem and CTS PROXY AGENT subsystem, respectively, while COORDI-

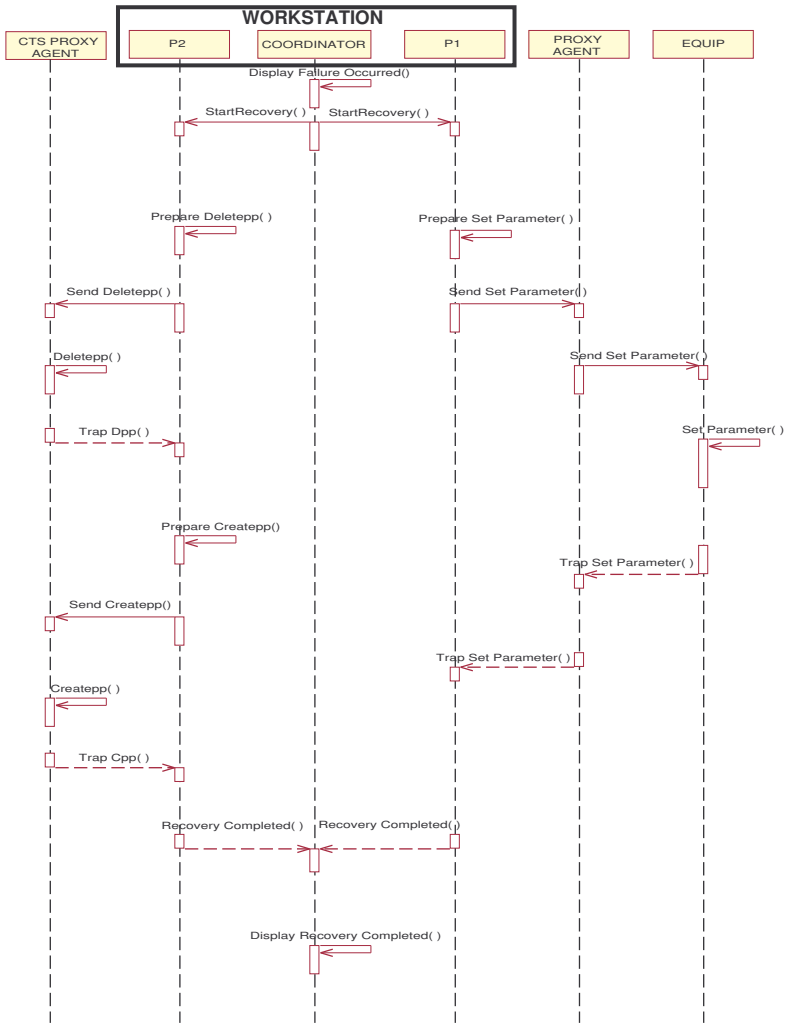


Figure 5.2: Recovery Scenario

NATOR represents the control logics of the WORKSTATION subsystem.

When a failure occurs, COORDINATOR actives two parallel executions (a fork takes place), the one through P1, PROXY AGENT and EQUIP and the other through P2 and CTS PROXY AGENT. When the recovery procedure is completed, COORDINATOR receives a notification from P1 and P2 (a join takes place).

The performance requirement for this activity, as required by the system developers, is: *"The mean execution time of the recovery has to be lower than 6 seconds, when a failure occurs. .*

5.3 Performance Analysis Based on Stochastic Process Algebras

In this section we report our experience in using a performance analysis approach based on SPA. The analysis is carried on at the SA level by specifying a performance model based on the Æmilia Architectural Description Language (ADL). Æmilia is a Stochastic Process Algebra (SPA) based ADL proposed

Action	Mean Execution time (sec.)
Display Failure Occurred	0.00
Start Recovery	0.00
PrepareSetParameter	1.00
SendSetParameter	0.01
SetParameter	2.00
GetParameter	0.01
TrapSetParameter	0.01
TrapCreatepp	0.01
TrapDeletepp	0.01
PrepareDeletepp	1.00
SendDeletepp	0.01
Deletepp	1.00
PrepareCreatepp	1.00
SendCreatepp	0.01
Createpp	2.00
RecoveryCompleted	0.00
DisplayRecoveryCompleted	0.50

Table 5.2: Mean Execution Times for the Actions in the Recovery Scenario

by Bernardo [43]. In order to make the approach as transparent as possible, we define a systematic approach that generates an Æmilia specification from a set of sequence diagrams (and component statecharts) describing the system dynamics at the SA level. The resulting approach does not require performance specific skills to software designers by allowing automatic derivation of SPA-based performance model from standard software artifacts, and the evaluation of performance indices by means of tools.

The approach used in this work is schematized in Figure 5.3. Starting from UML sequence diagrams, an Æmilia textual description is automatically extracted through the derivation of Statecharts diagrams and flow graph.

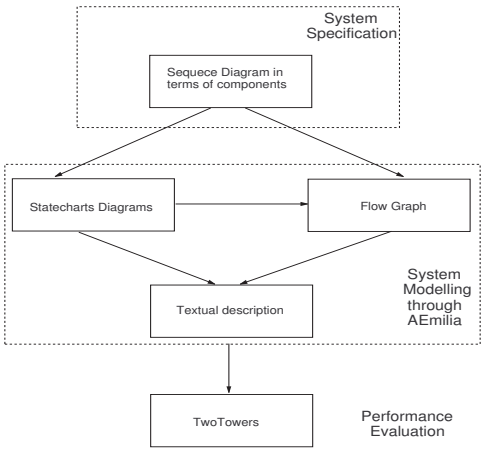


Figure 5.3: Used Approach

The Æmilia textual description can be translated into a performance model of the software system architecture. This translation is performed through the TwoTowers tool [40], that allows the model evaluation and validation against the system performance requirements. The reason for the choice of Æmilia as modelling notation is twofold: on one side, Æmilia being an ADL facilitates the description of the SA of the system; on the other side Æmilia inherits the capability to specify and to evaluate performance model from the SPA

$EMPA_{gr}$ [41] upon which it is defined. This section proceeds to introduce the Æmilia ADL, to sketch the defined approach for performance modeling step, to present the analysis of the obtained performance model and it concludes with some consideration on the used technique.

5.3.1 ÆMILIA: AN ARCHITECTURAL DESCRIPTION LANGUAGE

Æmilia is an ADL based on the SPA $EMPA_{gr}$ [41], introduced by Bernardo et al. in [43] in order to easy the use of Stochastic Process Algebra [91] as software model notation.

SPA is a modelling technique for the analysis of concurrent systems which are described as collections of entities, or processes, executing atomic actions. The processes are used to describe concurrent behaviors and they synchronize in order to communicate. Processes can be composed by means of a set of operators, which include different forms of parallel composition. Moreover, temporal information is added to actions by means of continuous random variables, representing activity durations. The quantitative analysis of the modelled system can be performed by constructing the underlying stochastic process. In particular, when action durations are represented by exponential random variables, the underlying stochastic process yields a Markov Chain.

ARCHI_TYPE	<name and numeric parameters>
ARCHI_ELEM_TYPES	<architectural elements types: behaviors and interactions>
ARCHI_TOPOLOGY	
ARCHI_ELEM_INSTANCES	<architectural elements instances>
ARCHI_INTERACTIONS	<architectural interactions>
ARCHI_ATTACHMENTS	<architectural attachments>

Figure 5.4: Structure of an Æmilia Textual Description

Æmilia provides a formal specification language for the compositional, graphical and hierarchical modelling of software systems, which is equipped with suitable checks for the detection of possible architectural mismatches. A description in Æmilia represents an Architectural Type (AT), that is a family of software architectures whose members must have the same observable functional behavior and topology, while the internal behavior and the performance characteristics can vary. The description of an AT starts with the name of the AT and its numeric parameters, which indicates activity execution rate and action priority level. Each AT defines a set of Architectural Element Types (AET) modelling the software components and complex connectors, and the considered architectural topology.

An AET is defined by its behavior (specified either as a family of $EMPA_{gr}$ sequential terms or through an invocation of a previously defined AT) and its interactions (specified as a set of $EMPA_{gr}$ action types occurring in the behavior). The architectural topology is specified through the declaration of a set of Architectural Element Instances (AEI) and a set of Directed Architectural Attachments (DAA) among the interactions of the AEI. Depending on the SA configuration, one or more AEI for each AET can be specified.

Figure 5.4 shows the structure of an Æmilia textual description.

Every interaction in an AET is declared as an input interaction or an output interaction and the DAA must respect such a classification: every DAA must involve an output interaction and an input interaction of two different AEI.

All the occurrences of an action type in the behavior of an AET have the same execution rate (exponential or immediate with the same priority level or passive with the same priority level) and must comply with

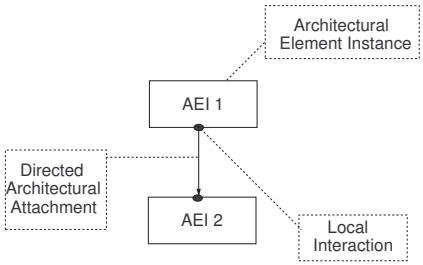


Figure 5.5: Basic Flow Graph

the synchronization discipline of $EMPA_{gr}$. Every chain of DAA contains at most one interaction whose associated rate is exponential or immediate.

It is possible to specify an Æmilia model by means of the TwoTowers tool [40]. TwoTowers is developed by Bernardo at University of Urbino and it translates Æmilia textual descriptions in $EMPA_{gr}$ .The semantics of an AT is obtained by composing in parallel the semantics of its AEI according to the specified DAA [43]. TwoTowers provides functional verifications and performance evaluation capabilities.

Finally, Æmilia provides a helpful graphical notation for the architectural design of complex systems. Such a graphical notation is based on flow graphs [120]. In a flow graph representing an Æmilia architectural description (see Figure 5.5), the boxes denote the AEI, the black circles on the box lines denote the local interactions and the directed edges denote the attachments.

5.3.2 System Modelling

In this section we present the approach to systematically generate an Æmilia textual description from a set of UML sequence diagrams (and component statecharts). To show the modelling process we use the recovery activity scenario. For this scenario the sequence diagram of Figure 5.2 is the only artifact we need in order to derive the Æmilia textual description since it contains also information on the internal actions executed by the components in the scenario evolution. In general, to specify an Æmilia textual description also statecharts are needed.

The considered sequence diagram describes the system behavior in terms of software components and how they interact when a recovery is executed.

The modelling process starts from the sequence diagram and proceeds in three steps. The first step (synthesis) derives the (partial) statechart diagrams for each component present in the sequence diagram. The second step constructs the flow graph, which is the Æmilia graphical representation. Information on the software system configuration (in terms of number of component instances) is needed in this phase to specify the architectural topology. The last step combines information on action service rates (see Table 5.2) and information from the statecharts and flow graph in order to obtain the Æmilia textual description.

Synthesis

In the literature many synthesis algorithms exist, e.g. Uchitel et al. [152] have defined an algorithm that translates a scenario specification into a Finite Sequential Processes (FSP) specification. In our synthesis approach, differently to the existing ones, we do not synthesize the complete behavior specification of the SA components which would require the use of all the available scenarios. Instead, we consider a

single scenario aiming at deriving the partial behavior description of all components involved in it. This simplification is correct for our aim since in the NICE description it is assumed that each SA component is a multi-thread component and, for each scenario, it has a dedicate thread that manages the scenario. This assumption allows us to analyze a scenario independently from the others describing the system behavior, and to synthesize just the part of the components behavior of interest. Thus, from a single scenario our approach derives the partial statechart of each component involved in it.

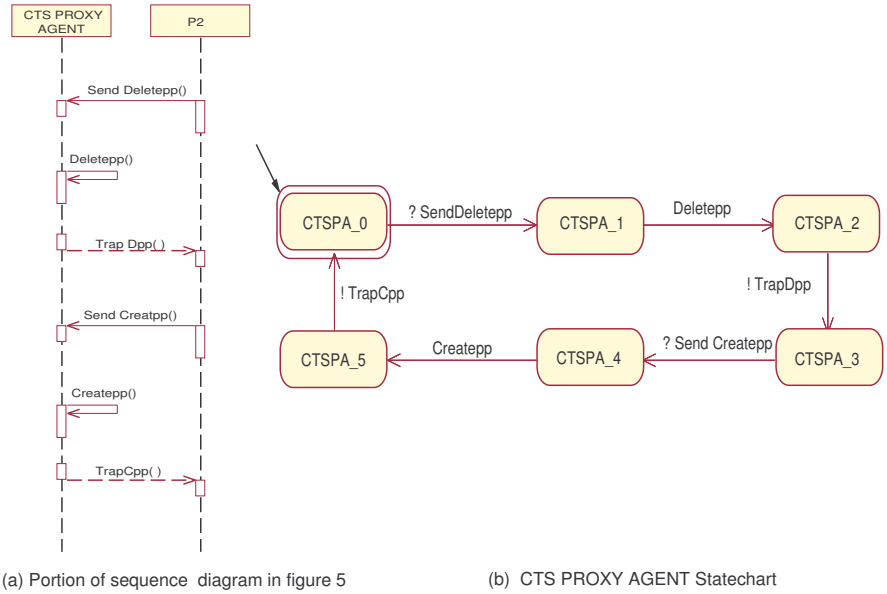


Figure 5.6: CTS PROXY AGENT Component Statechart

A statechart starts from an initial state in which the component is waiting for taking part to the scenario execution and it contains one state transition for each message (sent, received, to itself) that occurs in its scenario lifeline.

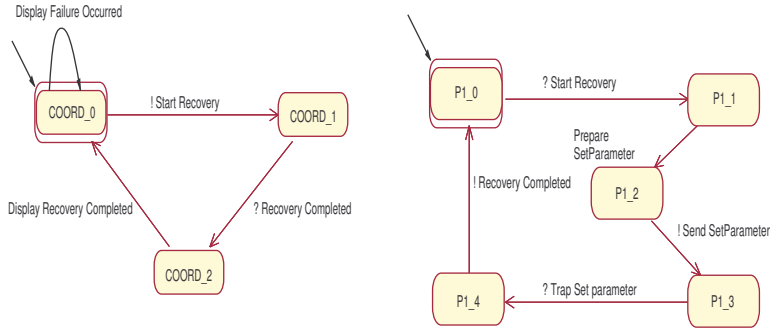
The transition label is composed by the message label and an additional symbol which indicate the message type: the symbol ”?” denotes a received message, the symbol ”!” denotes a sent message, while no symbol denotes a ”message to itself”. The last state transition ends in the initial state to indicate that the component has finished its ”work” in the scenario.

In the left-hand side of Figure 5.6 we report the portion of the sequence diagram of Figure 5.2 that involves the CTS PROXY AGENT component while, in the right-hand side, we show the synthesized CTS PROXY AGENT statechart. An initial state is introduced. By following the lifeline of the CTS PROXY AGENT component we add a new state and a new transition for each interaction beginning from or ending in the CTS PROXY AGENT lifeline, except for the last message that corresponds to a transition going back to the initial state. The transition labels are created using the conventions introduced above.

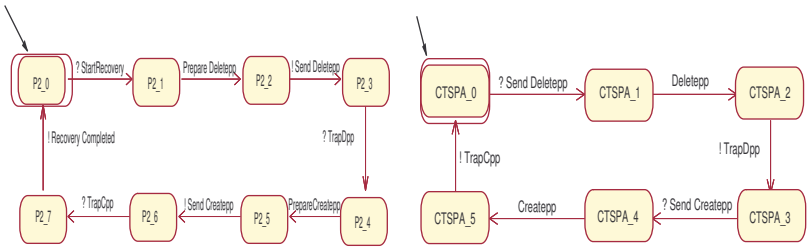
Figure 5.7 reports the stateharts of all the components involved in the recovery scenario obtained by applying the synthesis step.

FLOW GRAPH CONSTRUCTION

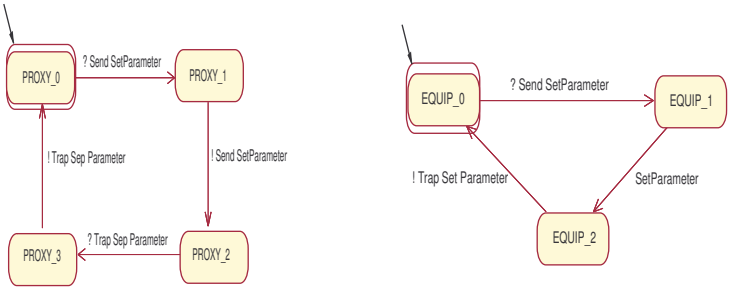
Starting from the sequence diagram and the derived statecharts, the flow graph corresponding to the scenario can be derived through some simple steps:



(a) COORDINATOR and P1 Statecharts



(b) P2 and CTS PROXY AGENT statecharts



(c) PROXY and EQUIP statecharts

Figure 5.7: Component Statecharts

1. *Identification of component instances*: according to the SA configuration, one or more instances of each component together with their names are identified and a box is created in the flow graph representation for each of them;

2. *Identification of local interactions*: for each component, by looking at the transition label in its statechart, a set of input (with "?" prefix in the label) and output (with "!" prefix in the label) interactions is identified. These local interactions are represented by black circles on the box line of the component instance;

3. *Attachment generation*: according to the interactions in the sequence diagram, the component instances are linked by arrows in the flow graph representation. Such arrows are called attachments. Since this step depends on the configuration of the system that is not at the moment depicted by any diagram, this step is made manually by the analyst. This step could be automatized if the SA configuration is described by a UML 2.0 component diagram.

Local interactions and attachments derive from the messages exchanged by components. For each message exchanged, two interactions (an output interaction on the sender instance, an input one on the receiver instance) and an attachment between these instances are identified.

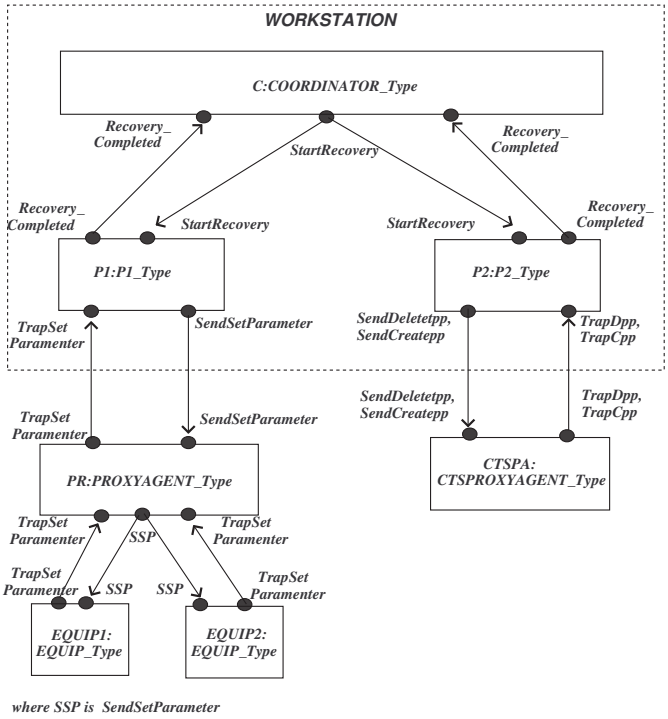


Figure 5.8: Flow graph of Considered Scenario

In order to initially model our system we consider one instance for COORDINATOR, P1, P2, CTS PROXY AGENT and PROXY AGENT components and two instances of the EQUIP component.

The complete flow graph is shown in Figure 5.8.

THE ÆMILIA TEXTUAL DESCRIPTION

The Æmilia textual description is derived from the statecharts, which represents the behavior evolution of components, and from the flow graph, which contains information on the interactions between instances (see Figure 10). Moreover, information on the software system configuration, such as for example the number of component instances, is needed to specify the architectural topology.

The textual description is divided in two parts:

1. The description of the behavior and local interactions of each component (AET definition) which are obtained from the corresponding statechart;
2. The description of the architectural topology, i.e. component instances (AEI) and attachments (DAA) between them, which is obtained from the flow graph.

The behavior of a component is defined as a set of process terms. A process term is a sequence of actions conforming to the Æmilia syntax [43].

In our case a process term starts with an input interaction, follows with a sequence (possibly empty) of

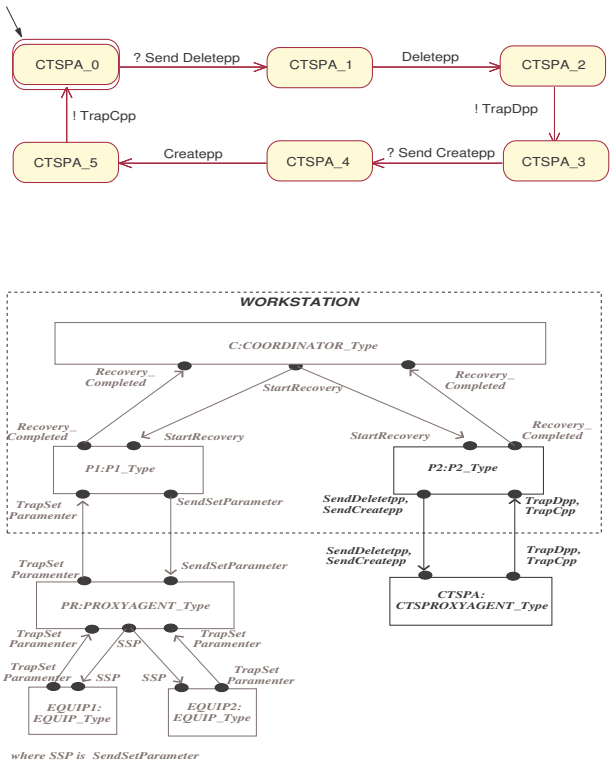


Figure 5.9: Statechart and Flow Graph of the CTS PROXY AGENT Component

internal actions and with an output interaction, and terminates with a new process term or with the first specified one. In this last case the behavioral description of the component is complete.

In the behavioral description the performance related information of the actions is specified as service rates. Figure 5.10 shows the behavioral description of the CTS PROXY AGENT component. The `ELEM_TYPE` keyword indicates the beginning of the CTS PROXY AGENT component type definition. It names `CTSPROXYAGENT_type` and it takes as input three parameters that represent the action service rates. In the following, the behavioral descriptions of the two process terms are specified. In this definition the input parameters are associated to the internal actions specifying that they have an execution time related to the indicated rate. The input and the output interactions are also reported in the correspondent part of the specification.

To specify the architectural topology we have to define instances and attachments. By looking at the flow graph and considering the software system configuration for each box in the flow graph we insert in the textual description a number of instances as specified in the configuration. After that, we define the correspondent architectural attachments for each link in the flow graph. In the architectural attachment specification we have to take care of the link direction. Figure 5.10 shows the definition of a CTS PROXY AGENT instance and the specification of all attachments in which it is involved.

For the complete Æmilia textual description of our case study please refer to the Appendix A at the end of the thesis. The action service rates in the last column of Table 5.2 are used as input parameters in the Æmilia description.

```
        ⋮
      ELEM_TYPE
        CTSPROXYAGENT_Type(void;
          rate a1, rate a2, rate a3)
        BEHAVIOR
          CTSPA(void; void)=
            <SendDeletepp, *>.<Deletepp, a1>.
            <TrapDpp, a3>.CTSPA'();
          CTSPA'(void; void)=
            <SendCreatepp, *>.<Createpp, a2>.
            <TrapCpp, a3>.CTSPA();
        INPUT_INTERACTIONS
          UNI SendDeletepp;SendCreatepp
        OUTPUT_INTERACTIONS
          UNI TrapDpp;TrapCpp

⋮

ARCHI_TOPOLOGY
      ARCHI_ELEM_ISTANCES
        ⋮
       P2: P2_Type(;a1,a3);
       CTSPA: CTSPROXYAGENT_Type(;a1,a2,a3);
        ⋮
      ARCHI_ATTACHMENTS
        ⋮
      FROM P2.SendDeletepp TO CTSPA.SendDeletepp;
      FROM P2.SendCreatepp TO CTSPA.SendCreatepp;
      FROM CTSPA.TrapDpp TO P2.TrapDpp;
      FROM CTSPA.TrapCpp TO P2.TrapCpp;
        ⋮
```

Figure 5.10: Textual Description of CTS PROXY AGENT Component

5.3.3 ANALYSIS

To evaluate the obtained Æmilia textual description by means of TwoTowers, we must specify the performance indices of interest using the rewards technique [40]. The indices specification must respect the TwoTowers input format. After that, the results can be obtained by running TwoTowers on the performance specification. In the following we discuss the performance indices of interest and we report our experience in their evaluation.

PERFORMANCE INDICES - The performance requirement presented in Section 5.2 demands the evaluation of the mean response time of the system when the considered scenario is executed. Applying the Little Law [111], the mean response time of the system is equal to the average number of requests in the system divided by the throughput of the system. Hence, we need to calculate the throughput of the system and the average number of recovery requests in the system. In our case, the throughput of the system corresponds to the throughput of the last executed action in the scenario of Figure 5.2 due to the particular structure of the model whose actions are strongly synchronized. Whereas the average number of recovery requests in the system is always one since the COORDINATOR component is blocked waiting for the recovery termination and can not capture new failures. Therefore in order to evaluate the mean response time of the system we reduce to evaluate the throughput of the system.

In TwoTowers for each performance index of interest we have to define a particular specification where we indicate how to calculate it by following the rewards technique for Markov chain [100]. This technique requires to specify a reward for each state and for each state transition of the Markov chain strictly related to an action. These rewards are then suitably combined with the probability that the Markov chain reaches the particular state and with the probability that the state transition occurs.

PROXY AGENT	EQUIP	States	Transitions	Exec. Time (sec.)	System Throughput	Mean Response Time
1	2	90	171	0	0.1546	6.4683
2	4	468	1326	2	0.141997	7.042402
3	6	3114	12057	789	0.133783	7.4747491
4	8	21636	105696	10824	0.127787	7.8255221
5	10	151290	890823	86592	0.1248814	8.0119247
10	20	Not Enough Memory				

Table 5.3: Performance Evaluation Results of the Simplified Æmilia Model

Throughput-like measures can be specified in a way that does not depend on the specific rate of the actions, using transition rewards with value 1.

To evaluate the throughput of the system for the Recovery scenario, corresponding to the throughput of the action "RecoveryCompleted", we specify:

MEASURE throughput **IS ENABLED**
(COORD.RecoveryCompleted)− >**TRANS_REW**(1.0)

The actual throughput is hence obtained by summing the probabilities of all the state transitions in which the action "RecoveryCompleted" of the COORD instance can be executed, and by multiplying such a sum by the transition reward 1.0.

PERFORMANCE EVALUATION PHASE - Table 5.3 shows the results obtained by of evaluating the Æmilia specification by means of the TwoTowers tool with respect to the specified performance indices and considering a set of system configurations. The results presented in Table 5.3 are obtained through a computer equipped with Pentium IV 1.6 GHz and 512MB RAM. OS used is RedHat Linux 8.0

Unfortunately, for bigger configurations state space explosion occurs. Indeed, we were able to obtain results for system configuration specified than the actual configuration.

The first and the second columns of the table in Table 5.3 show the instance number of the PROXY AGENT and EQUIP components in the configurations we considered. The third and the fourth columns report the dimensions of the Markov Chain built by TwoTowers in terms of the number of states and transitions, respectively. Finally, the last two columns show the system throughput obtained by TwoTowers and the mean response time of the system.

We can observe that we are not able to calculate the mean response time of the real configuration (i.e. 10 PROXY AGENT instances and at least 20 EQUIP instances) since the state space explosion problem happens when the number of instances increases. Anyway, already from these results on reduced configurations we see that the proposed performance constraints on the software components are not satisfied. Hence, according to the predictive performance analysis, the next step will be to define a new reasonable set of performance constraints and then to repeat the evaluation.

5.3.4 CONSIDERATIONS ON THE USED APPROACH

The methodology is based on the Æmilia SPA-based ADL that has nice and useful features inherited both from the process algebra notation and from the ADL. Æmilia expressiveness is high thanks to its the expressiveness of the SPA it is based on. Its ADL-nature makes its use easier for a software engineer. From a software architectural specification it is quite simple to derive an Æmilia textual description since it reflects

the SA structure. The only drawback of Æmilia in order to carry on a performance analysis at SA level, is strictly related to its process algebra aspects. In fact, we need pieces of information on the internal behavior of the components. This drawback is not evident in our case study since this information is contained in the scenario. The scenario contains component interactions and method invocations internal to the components that represent actions that consume time. Every time we have this information, the Æmilia textual description is easily and automatically derivable. These characteristics make the introduction of our approach in an industrial context possible.

However, automation is a fundamental aspect and the state-space explosion reported by TwoTowers during the evaluation step reduces the usage of the approach in a real industrial context. There can be solutions to this problem, for example by reducing the complexity of the performance model obtained. However such approaches reduce the automation and make the integration with the software development process difficult. These solutions require in fact to consider models that are not directly derived from the software specification and thus imply specific skills and expertise.

Another drawback of this performance analysis approach is in the specification of performance indices. Indeed the TwoTowers input format for indices requires a specific knowledge on the underlying reward theory. This aspect reduces the integration of performance analysis in an industrial software development process. However, we believe that this limitation could be overcome by providing TwoTowers with a more friendly interface.

In the following we report the criteria defined in Section 5.1 suitable instantiated for the approach:

Transparency The approach shows good transparency even if it should be improved in several parts, as the following criteria highlight.

- **Performance Model Derivation** The methodology based on Æmilia SPA-based ADL has nice and useful features inherited both from the process algebra notation and from the ADL. Thanks to its ADL nature it is quite simple to derive an Æmilia textual description from a software architectural specification. However the approach requires detailed information on the component internal behavior that is not always available at the SA level. This drawback is not evident in our case study since this information is contained in the scenario that contains method invocations internal to the components with respect to the actions that consume time.
- **Software Model Annotation** In order to obtain a performance model, it is necessary to provide quantitative, performance-oriented information which can be used to build the performance model. Annotations can be expressed using standard UML mechanisms (stereotypes and tagged values) which are supported by default by most UML CASE tools. Many efforts have spent to define a UML profile for such aims (UML profile for Schedulability, Performance and Time [87]). A good aspect in this approach is that the performance model generated in Æmilia is parametric, meaning that each parameter (such as activities durations) must be instantiated before the model is executed by modifying it.
- **Performance Indices** The Æmilia-based approach does not suffer of the accuracy problems, as the model can be solved analytically by the TwoTowers tool, which computes an exact numerical result. However, this approach shows a drawback in the specification of the performance indices. Indeed the TwoTowers input format for indices requires a specific knowledge on the underlying reward theory.

Automation The approach shows high automation in the analysis due to the TwoTowers tool. However, at the moment the step of generation of the textual description is not automatic even if it could be since it is based on a set of systematic transformation rules.

Result Interpretation The results interpretation might be very difficult since the user may combine different performance measures in order to obtain information related to the software model elements.

Generality The approach can be applied to any application domain and architectural pattern (such as client/server, layered architectures and others). However the modeling of fork/join systems, simultaneous resource possession, general time distributions and arbitrary scheduling policies, even if it is possible, is not an easy task since they require some tricks not always trivial.

Feedback Obtaining feedback is quite difficult in this approach, because it may be required to the user to combine different performance measures in order to obtain performance information about the software model elements.

Scalability The scalability of the Æmilia-based approach is somewhat limited by the state-space explosion problem. Even for software models of moderate size, the analytical model becomes too complex and overflows the resources available on the host platform. This problem reduces the applicability of the approach in a real context. There can be solutions to the state-space explosion problem, but these solutions require the user to hand-tune the generated performance model. This requires specific skills and expertise, and reduces the automation of the approach and make the integration with the software development process difficult.

Summarizing, the most critical issues that can compromise the applicability of the approach are automation i.e. the availability of tools that overcome the state-space explosion, the performance indices specification and the consequently results interpretation. These last two aspects should be made more friendly.

5.4 Performance Analysis Based on Simulation

An alternative approach for performance modeling and analysis of the software architecture we applied to the NICE system is based on simulation. This modeling approach allows general system representation of arbitrarily complex real-world situations, which can be too complex or even impossible to represent by analytical models. The simulation approach is implemented in a tool called UML-Ψ that is described in [45, 29]. It is based on a process oriented discrete-event simulation. We apply the UML-Ψ tool to to the NICE system in order to derive the simulation model from annotated UML use case, deployment and activity diagrams specification, as follows.

The section proceeds to briefly recall the main aspects of a simulation model, to present the generation approach of the simulation model, to report the analysis results and it concludes giving some consideration on the used technique.

5.4.1 Simulation Models

Besides being a solution technique for performance models, simulation can be a performance evaluation technique itself [34]. It is actually the most flexible and general analysis technique, since any specified behavior can be simulated. The main drawback of simulation is its development and execution cost.

A simulation model (i.e., a conceptual representation of the system) is generally derived by using a *process oriented* or an *event oriented* approach. From it a simulation program is derived. Such simulation program implements the simulation model.

One fundamental step in deriving a simulation model/program is the verification of the correctness of the program with respect to the model and the validation of the conceptual simulation model with respect to the system (i.e. checking whether the model can be substituted to the real system for the purposes of experimentation).

A critical issue in simulation concerns the identification of the system model at the appropriate level of abstraction.

Whenever a simulation model/program is derived three more tasks have to be accomplished: (i)planning the simulation experiments, e.g. length of the simulation run, number of run, initialization, (ii) running the simulation program and (iii) analyzing the results via appropriate output analysis methods based on statistical techniques.

Existing simulation tools provide suitable specification languages for the definition of simulation models, and a simulation environment to conduct system performance evaluation, such as for example CSIM [2], C++Sim [1] and JavaSim [3].

5.4.2 SIMULATION MODELING

The approach, proposed by Balsamo and Marzolla in [28, 45], generates a process-oriented simulation model of a UML software specification describing the software architecture of the system. The used UML diagrams are Use Case, Activity and Deployment diagrams. Use case diagrams are used to model workloads applied to the system. Actors correspond to open or closed workloads, a workload being a stream of users accessing the system. Each user executes one of the use cases associated with the corresponding actor. Use cases $U_{i,j}$ associated with Actor A_i is given probability $p_{i,j}$ to be chosen. The sum of the probabilities of use cases associated to the same actor must be 1, that is $\sum_{j=0}^{k} p_{i,j} = 1$, for each i. Deployment diagrams are used to describe the physical resources (processors) which are available. Finally, activity diagrams show which computations are performed on the resources. There must be at least one activity diagram associated to each use case. Each action state represents a computation, that is, a request of service from one active resource (processor).

In order to carry on performance analysis in UML-Ψ we add quantitative informations to UML specification, by using stereotyped and tagged values corresponding to a subset of those described in the UML Performance Profile [87].

≪OpenWorkload≫ or ≪ClosedWorkload≫ actors respectively denote unlimited (open) and finite (closed) number of users accessing the system. For actors representing open workloads we specify the interarrival pattern of users with the `RTarrivalPattern` tagged value. For closed workloads we specify the following tagged values:

PApopulation Number of users of the system;

PAextDelay External delay experienced by users.

Each step of an activity diagram is stereotyped as ≪PAstep≫, and can be furtherly annotated with the following tagged values:

PArep the number of times this step has to be repeated;

PAdelay an additional delay in the execution of this step, for example to model a user interaction;

PAinterval the time between repetitions of this step, if it has to be repeated multiple times;

PAdemand the processing demand of the step;

PAhost The name of the host (deployment diagram node instance) to which the service is requested.

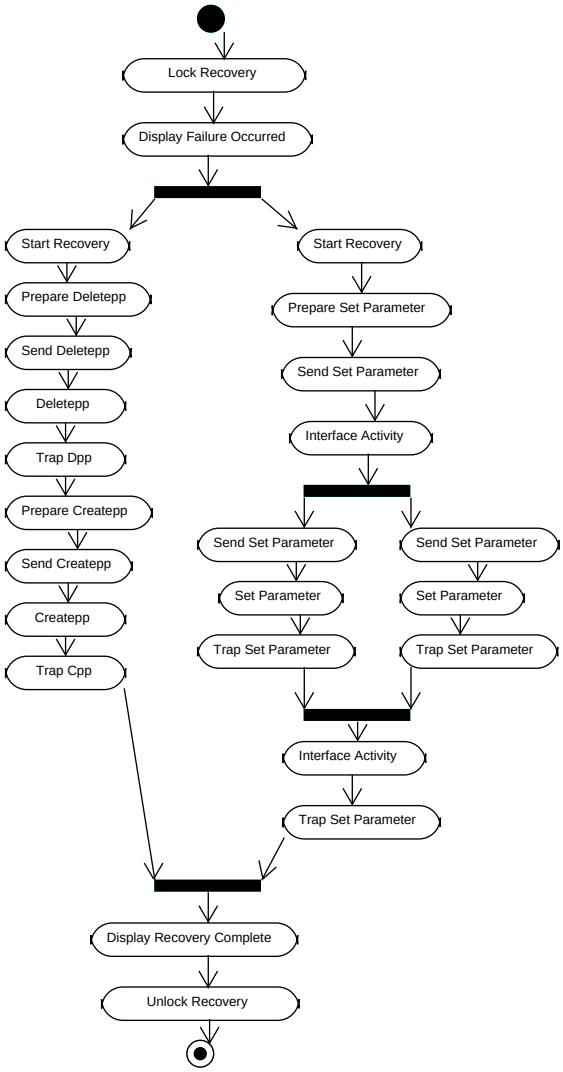


Figure 5.11: Recovery Scenario

Resources are modeled as node instances in Deployment diagrams. Active resources (processors) correspond to nodes stereotyped as ≪PAhost≫. Each node instance can be tagged with the following attributes:

PAschedPolicy (recognized values are “FIFO”, “LIFO” and “PS”) The scheduling policy of the processor, either first-come-first-served (FIFO), last-come-first-served (LIFO) or processor sharing (PS).

PActxSwT the context switch time.

PArate the processing rate of the host, with respect to a reference processor. Thus, a `PArate` of 2.0 means that the host is twice as fast as the reference processor.

Passive resources correspond to nodes stereotyped as ≪PResource≫. Passive resources have a maximum capacity, expressed with the `PAcapacity` tag. Requests of a resource are done by actions stereotyped as ≪GRMacquire≫, while release of a resource is done by actions stereotyped as ≪GRMrelease≫. If the residual capacity of a resource is less than what requested, the requesting action is suspended until enough resource is available. Pending requests are served FIFO.

The UML-Ψ tool parses the XMI representation [121] of the annotated UML model. Currently the XMI variant used by the ArgoUML [20] tool is supported. From the annotated UML model, a process-oriented simulation model is automatically derived. UML elements are mapped directly into simulation processes in the following way. Actors are translated into processes generating the workload. Deployment node instances correspond to processes simulating the resource with the given scheduling policy, processing rate and context switch time. Finally, each action state in the activity diagrams is translated into a simulation process. When a workload user is activated, it chooses the use case to execute. The activity diagram associated with the selected use case is translated into a set of processes, one for each action state. The simulation process associated with the starting activity is finally executed. Each step, once completed, starts the successor step until the end of the activity diagram is reached. At that point the workload user is resumed.

In order to apply the simulation-based modeling technique, it is necessary to translate the sequence diagram of Figure 5.2 into activity diagram. This can be done easily, resulting in the activity diagram depicted in Figure 5.11.

Note that the system we are simulating is synchronous, meaning that when an equipment is being repaired, then no other equipment can be repaired at the same time, but must wait until the current corresponding operation has been completed. In order to simulate this behavior it is necessary to use a passive resource in order to simulate a lock on the scenario. When executing a scenario it is first necessary to get the lock; if no recovery operation is currently running, then the lock is granted immediately. If the lock is not available, the recovery request is put on a queue. We compute the mean execution time of the scenario, including the contention time spent waiting for another running scenario to complete. The service demand for each action state was set as in Table 5.2.

A simplified representation of the resulting simulation model is given in Figure 5.12 where each node represents a simulation process and arrows indicate the process activation order. This graph shows the correspondence between the activity diagram and the simulation process. The dotted arrows indicate requests or releases of passive resources, which in our example are denoted as borderless gray boxes labelled “Lock Recovery Scenario”. As explained above, these resources are used to guarantee that only a single recovery scenario is executed at the same time.

5.4.3 ANALYSIS

UML-Ψ computes the following steady-state performance measures: mean execution time of each action state, mean execution time of each use case, and mean utilization and throughput of the processing resources. These values are computed using the batch means method [32, Chap. 7]. Mean values are expressed in term of confidence intervals, where the confidence level can be specified by the user. The simulation is stopped when the desired accuracy is obtained, that is, when the relative confidence interval widths are less than a given threshold. For more details on the UML-Ψ approach please refer to [114]

For simulating the NICE system we set the confidence level to 95% and a 10% confidence interval relative width. Simulation provides estimated performance measures whose mean values are inserted into the original UML model as tagged values associated with the relevant UML elements.

We simulate the system considering an increasing number N of equipments, for $N = 1 \ldots 6$. We assume that the time between successive recovery operations on the same equipment are exponentially distributed with mean $15s$. Simulation results in terms of average execution times of the Recovery scenarios are shown in Table 5.4. The table displays also the total execution time of the simulations on a Linux/Intel machine running at 900Mhz, with 256MB or RAM.

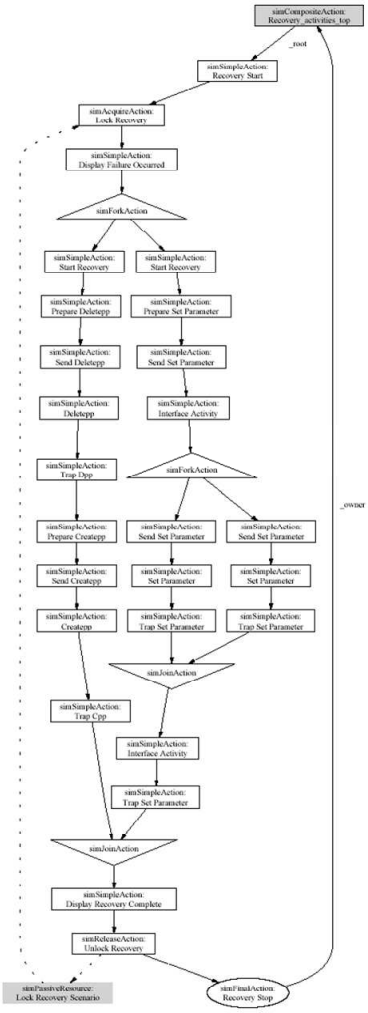


Figure 5.12: Structure of the Simulation Model.

5.4.4 Consideration on the Approach

Transparency The approach shows high transparency as the following criteria highlight.

Performance Model Derivation The simulation-based performance model can be easily derived from the UML specification, as there is an almost one-to-one mapping between UML elements and simulation processes [29].

Software Model Annotation In order to obtain a performance model, it is necessary to provide quantitative, performance-oriented information which can be used to build the performance model. UML-Ψ requires the UML model to be annotated according to a subset of the UML Performance Profile [87]. Annotations are expressed using standard UML mechanisms (stereotypes and tagged values) which are supported by default by most UML CASE tools. The modeler only needs to know the notation used to specify the values, which in our case is based on the Perl language.

Performance Indices The simulation-based approach can derive many different performance metrics. However, the results are expressed in term of confidence intervals. Special care has to

N	**Recovery Scenario**		**Simulation**
	Mean Exec. Time (s)	Requirement Satisfied?	**Exec. Time**
1	6.61	no	$56s$
2	7.64	no	$1m09s$
3	10.26	no	$1m37s$
4	13.70	no	$1m29s$
5	17.66	no	$2m44s$
6	23.97	no	$2m51s$

Table 5.4: Computed mean execution times for the Recovery scenario, for different number N of equipments. The last column on the left reports the execution time of the simulation program

be taken in order to apply the correct statistical techniques to remove the initialization bias and compute means from sequence of observations which are usually correlated, requiring many samples (and thus potentially longer execution times) to be taken [32, 110]. UML-Ψ implements the batch means method described in [33] to compute the mean values. The user only specifies the desired accuracy in term of confidence interval relative width and confidence level.

Automation The UML-Ψ approach shows high automation degree. It allows software model annotation by means of ArgoUML case tool. From the xmi exporting of the UML diagrams UML-Ψ generates and, after having defined performance indices and simulation parameters (confidence intervals), it runs the simulation in order to obtain information of the quantitative figures specified before. At the end of a simulation run UML-Ψ reports them by means of annotations in the used diagrams.

Result Interpretation Due to the provision of the performance indices upon the software models, the interpretation results is not a big deal.

Generality The approach can be applied to any application domain and architectural pattern (such as client/server, layered architectures and others). It allows general software model, that is, without any constraints on the software architecture model in order to derive the performance model. For example, it is possible to simulate fork/join systems, simultaneous resource possession, general time distributions and arbitrary scheduling policies.

Feedback The performance values computed by UML-Ψ are inserted back into the original UML model as tagged values associated to the relevant model elements. In this way the user may get an immediate feedback on the system performances. However the approach does not suggests design alternatives in case some performance problems are experienced.

Scalability The simulation-based approach is scalable, meaning that the complexity of the simulation model increases linearly with the number of UML elements to simulate. Also, the user may perform many different experiments by changing the UML model, performing a simulation run and modifying the UML model to get better performances before iterating the process again. It is then very easy to perform many "what-if" experiments, changing parameters or structure of the model to see what the result is. Unfortunately, the lack of parameterized results can be a limitation for that. Namely, it is not possible to get performance results as a function of an (unknown) parameter, or set of parameters. It is necessary to explore explicitly all the alternatives by running different simulations to collect the results.

5.5 Combined Usage of Tools

As presented before, we applied two different approaches to software performance analysis to the NICE case study from the telecommunication domain. The first technique allows analytical analysis of the performance of the Æmilia model derived from UML sequence (and statechart) diagrams. The second one obtains the

performance indices of interest by executing a simulation model automatically generated from UML use case, activity and deployment diagrams.

Both the approaches showed high generality in terms of performance aspects, software system application domains and software architectural patterns. Both of them require specific information on performance aspects of the system that are then used in the analysis. While the simulation technique allows their insertion through the UML activity diagram annotations, the other approach imposes the analyst to insert them directly in the performance model.

Moreover, the presented methodologies showed, as main drawback, the necessity to specify configuration parameters. The simulative technique requires the user to define confidence intervals and the duration of the simulation, while the Æmilia approach requires expertise for the specification of performance indices since the user must have a deep knowledge on the reward theory and queuing theory.

Differently to Æmilia approach, simulation provides a simple feedback mechanism that reports the obtained performance results on the software architecture level.

Finally, the methodology based on stochastic process algebra may require specific skills in software modeling since, in order to derive an Æmilia model, detailed information on the system behavior have to be provided. However, on the positive side Æmilia specification also allows for behavioral analysis.

The comparison of the two approaches is summarized in Table 5.5.

As largely expected the two methodologies have pro and cons. However, in this work we have experimented the feasibility of a complementary approach at an affordable cost. Different approaches highlight different figure of merits in terms of performance modeling, analysis and indices that can be evaluated, and of feedbacks at the design level that can derive from the performance results interpretation. In this scenario, the use of different techniques can provide to the software designer a more precise and comprehensive picture on the software architecture. Thus, we believe that the concurrent/complementary application of several analysis techniques can overcome the problems of the single techniques and conduct to more faithful analysis results.

A big element toward a combined use of the two approaches is the use of standard software artifacts as system initial documents. This can simplifies the comparison among the results obtained by different approaches.

5.6 SUMMARY

In this Chapter we applied two different approaches to software performance analysis to the NICE case study from the telecommunication domain. The first technique allows analytical analysis of the performance of the Æmilia model derived from UML sequence (and statechart) diagrams. The second one obtains the performance indices of interest by executing a simulation model automatically generated from UML use case, activity and deployment diagrams.

As largely expected the two methodologies have pro and cons. However, in this work we have experimented the feasibility of a complementary approach at an affordable cost. A big element toward a combined use of the two approaches is the use of standard software artifacts as system initial documents.

	Simulation/UML-Ψ	Æmilia/TwoTowers
Perf. Model Derivation	Easy. There is an almost direct mapping between UML elements and simulation processes.	Easy due to the ADL Æmilia notation which allows performance models which tightly reflect the software specification.
Annotations	Uses stereotypes and tagged values to specify performance-oriented parameters. Tag values are written in Perl following the specification for the UML Performance Profile [87]	Parameters should be instantiated by the modeler when the performance model is executed.
Generality	No constraint on the software model.	Modeling some particular aspects of software systems, even if possible, can be difficult.
Computed Performance Indices	Only approximate values are computed, which are given as confidence intervals. UML-Ψ implements a set of statistical functions which are automatically applied to output data analysis in order to compute sound results without any user guidance. For many problems (eg removing the initialization bias) there is no single agreed upon technique which can be applied to every situation.	Results are exact numerical values computed analytically.
Scalability	The size of the performance model increases linearly with the size of the UML model.	The size of the performance model can grow exponentially with the size of the software model, thus making the approach difficult if not impossible to apply but for small models.
Feedback	Performance results are inserted into the original UML model, as tagged values associated to the relevant model elements.	It is not easy to associate the computed performance measures to the software components to which they refer.

Table 5.5: Summary of the Comparison between the Simulation-based and the Analytical Approach.

Part II

Integration of Predictive Functional and Non-Functional Analyses

CHAPTER 6

A TOOL INTEGRATION FRAMEWORK

Software analysis has always been a non-trivial activity along the software development process, due both to special skills required to developers and to short time-to-market. On the other end, as the software systems progress and find new application environments (e.g. heterogeneous platforms, mobile devices, etc.) the analysis of functional and non-functional properties is becoming a primary concern to meet customer requirements.

Major efforts have been spent in the past for the functional analysis of software systems, that brought nowadays to offer quite sophisticated (formal and semiformal) methodologies and tools to verify and validate the functional behaviour of a software system since the early phases of its lifecycle [8]. On the contrary, non-functional attributes (such as performance, security, etc.) have not received the same consideration. Only in the last few years the idea of integrating such type of analysis along the whole software lifecycle has been supported from new methodologies aimed at filling the gap between the software development process and its non-functional validation [11].

The software evolution seems to ask for the integration of functional and non-functional analysis, and the automation in embedding the feedback resulting from analysis into the software models. The need of integration has been repeatedly claimed in the recent past, with particular focus on the software architectural level [74]. In this chapter we introduce a framework to cope with the integration issues at that level.

The work this chapter discusses has been outlined in [62] and it is described here in details.

6.1 MOTIVATIONS AND GOALS

Major efforts have been spent in the past for the functional analysis of software systems, that brought nowadays to offer quite sophisticated (formal and semiformal) methodologies and tools to verify and validate the functional behaviour of a software system since the early phases of its lifecycle [8]. On the contrary, non-functional attributes (such as performance, security, etc.) have not received the same consideration. Only in the last few years the idea of integrating such type of analysis along the whole software lifecycle has been supported from new methodologies aimed at filling the gap between the software development process and its non-functional validation [11].

The rationale behind this Chapter is that the software evolution seems to ask for advances in two major topics of software analysis: (i) the integration of functional and non-functional analysis, and (ii) the automation in embedding the feedback resulting from analysis into the software models. The need of integration has been repeatedly claimed in the recent past, with particular focus on the software architectural level [74]. The availability of such integration will provide a tool that coordinates several analises that can be made over a software architecture. We introduce here a framework to cope with the integration issues at that level. We started from the goal of evidencing inter-relationships between functional and non-functional aspects that would not necessarily emerge from separate analysis. For example, it is intuitive that upon detecting

a deadlock in a software model, a critical component may be split in two components, and this refinement may heavily affect the software performance. Viceversa, a security analysis may lead to introduce additional logics to components (that work as firewalls) in a subsystem, thus the behaviour of the subsystem needs to be validated again. In many cases analysis methodologies are based on a translation of the software model into a different notation to be analyzed (e.g., a formal language for functional verification, or a Petri Net for performance validation). Our intent is to place an intermediate representation (based on XML) of software models that may work as a common ground to apply functional and non-functional analysis as well as to feed back the analysis results on the software models.

In a collaborative working several teams can concurrently work on the development and analysis of a software system. This means that it should be necessary a design environment that timely notifies to all the involved teams changes made by one of them. Let us suppose for example that at the SA level one team is checking the reliability of the software system design, while other two teams are validating the SA design against the functional and performance requirements. The first team to improve the reliability of the system makes some changes on the SA. If these changes are made in a uncontrolled way and are not propagated to the views of the other two teams, these will work on different software system. The consistence of the different views of the SA is not guaranteed any more. To address with this consistency issue there should be designed a design environment that is able to reflects changes made on a views over the others active at the same time. Our intend is to propose a framework that guarantees consistency views of a SA design when different design/analysis teams are working on it. The XML-based intermediate representation help us to meet this aim.

To show our idea, in this Chapter we integrate two existing methodologies for software analysis: CHARMY [101] that translates scenarios and state machine diagrams into Promela language [98] for software formal verification, and a methodology that generates a Markov model [150] from a software specification based on the *Æmilia* architectural description language [43], to validate software performance. Both the methodologies work at an architectural level.

Our long term goal is to incrementally implement such framework by embedding other methodologies for software analysis at the architectural level. Obviously this goal lays on the ability of the XML core to be extended. As we will show in Section 6.3, we devise this ability as the result of two tasks: keeping the XML core general enough to embed new software notations; introducing rules and relationships across different software notations that allow to automatically propagate an analysis feedback from a functional world to a non-functional one, and viceversa.

The framework implementation will be based on the *Eclipse* platform, namely a software tool for tool implementation. This is a relevant aspect of our work, in that by embedding structures, rules and algorithms into *Eclipse*, we exploit the well-known integration features of this technology.

Loosely related work to this can be found in the research done in the field of independent verification and validation of software systems (e.g., see [68]). The contribution reported in this chapter is the integration of software analysis by proposing the design (as well as an idea of implementation) of an actual integration framework. A similar approach can be found in [67], where a framework has been introduced to integrate non-functional requirements in conceptual models. The integration work in [67] is performed at a requirement elicitation level, and is based on Entity-Relationship models. Our approach differs from the one in [67] because we assume that the (functional as well as non-functional) requirements have been formulated and modeled in some software notation. Then, we do not constrain the developers to build ad-hoc models (such as ER), rather our integration is based on XML models that can be automatically generated from the software models.

6.2 XML TECHNOLOGIES AS INTEGRATION SUPPORT

The wide success of XML for tool interoperability has opened promising perspectives on the integration of software and performance tools. The definition of XML Schema for the interchanged data will simplify the task.

In this Section we give some preliminary notions on the key concepts of XML and XML Schemas. The information reported here is not complete, but enough for the reader to understand the following sections.

6.2.1 THE EXTENSIBLE MARKUP LANGUAGE

The XML language is a standard meta–markup language defined by the W3C Web Consortium [155]. The XML itself is a simple but powerful idea, derived with many simplifications from SGML. SGML has been used for years to formally specify document and data formats, but its complexity kept it a "tool for professionals".

The basic syntax of XML is quite simple and should be familiar to anyone that knows HTML. Well–formed XML documents must obey to some simple rules, such as, e.g. each document must have an unique *root* element. In addition, XML languages may also be associated to a specific "grammar", defined through a *Document Type Definition* (*DTD*). or through an XML Schema Definition (XSD).

The idea of XML comes from the success of markup languages like HTML, focusing on a particular class of documents that show a strong hierarchical structure. The number of supporting technologies developed around XML made its success in few years. Today, XML is becoming a well–founded standard for the description of document formats.

However, the use of XML and DTD/XSD is not limited to document format definition. In the last years, XML languages are being adopted also for data description and manipulation. Indeed, the hierarchical structure of markup documents is very suitable to represent a wide variety of data, especially objects and similar structures. XML fragments are used to contain data structures and make them more portable and open–format. Applications can easily interface with XML–structured data streams or packets using XML parsers and querying tools (like XPath [157], XSLT [158]) to manipulate their content.

At the present XML and DTD/XSD are being used much more as a data–definition formalism than as a document–definition language. Indeed, most of the XML languages currently defined and standardized are explicitly designed to contain and give formal structure to data that has no "document" format.

6.2.2 THE EXTENSIBLE SCHEMA DEFINITION

DTD is a notation to define the XML language grammar. DTDs have many limitations and most of them derive from the initial definition of XML as a language used to structure textual documents. Moreover, they are specified by means of a language that is not XML related. To overcome these drawbacks, a new XML–grammar definition language has been developed. The new standard is itself an XML language, called eXtensible Schema Definition (XSD) [156], and it has many features lacking to DTD.

The XML Schema specification uses the concept of *type* as its basic mechanism to define the structure of XML documents. XSD distinguishes among *simple* and *complex types*: the former are built-in and include the most common data types used in DBMS and programming languages; the latter are used to describe the structure of elements, i.e. their *content model*.

```
<element name="E">
 <complexType>
  <sequence>
   <element name="F" type="integer" minOccurs="1"
     maxOccurs="unbounded"/>
   <choice minOccurs="0" maxOccurs="1">
    <element name="G" type="string"/>
    <element name="H" type="a"/>
   </choice>
  </sequence>
 </complexType>
</element>
```

Figure 6.1: A Complex Type Definition

```
<simpleType name="b">
 <restriction base="string">
  <minLength value="0"/>
  <maxLength value="20"/>
  <whiteSpace value="collapse"/>
  <pattern value="A(B|C){2;4}D*"/>
 </restriction>
</simpleType>
```

Figure 6.2: A Simple Type Definition

A content model defines the children (elements or attributes) of a particular element and optionally the order in which they must appear.

Content models are created by freely composing three basic models: sequence (sequence), choice (choice), and set (all). A cardinality constraint can be assigned both to the elements and to the basic models, through the minOccurs and maxOccurs attributes. Finally, an element must be defined as *mixed* complex type every time it can contain both other elements and text.

Fig. 6.1 shows an example of element declaration, that is read as follows:

> *The element E contains a sequence of at least one element F, optionally followed by an element G or H. Element F contains an integer, element G contains a string and element H contains a value belonging to the user–defined type a.*

Schemas also offer a rich sub–language for the definition of derived types. Types can be derived by *extension* or *restriction*. Elements of derived types can be declared compatible and arranged in an object-oriented fashion by means of *substitution groups*.

For instance, Fig. 6.2 illustrates the definition of a restriction of the string build–in type and Fig. 6.3 shows an element J whose type is obtained as an extension of the complex type element eType by adding some attributes.

Schemas allow declarations of types, elements and attributes to be *global* or *local*, i.e. nested inside another declaration. In the former case, the declared type, element or attribute has a distinctive name and can be referenced (possibly many times) in other parts of the schema.

```
<complexType name="eType">
 <sequence>
  <element reg="F" maxOccurs="unbounded"/>
  <choice minOccurs="0">
   <element ref="G"/>
   <element ref="H"/>
  </choice>
 </sequence>
</complexType>

<element name="J">
 <complexType>
  <complexContent>
   <extension base="eType">
    <attribute name="Attr1" type="boolean"
      use="required"/>
    <attribute name="Attr2" type="double"
      use="optional" default="1.0"/>
   </extension>
  </complexContent>
 </complexType>
</element>
```

Figure 6.3: A Complex Type Derived by Extension

Elements and their structure are defined through complex types. XML Schema allows to define non-hierarchical relations between elements using *identity constraints*. The key construct is used to declare that a particular element instance in a valid XML document is uniquely identified through a set of its attributes and/or sub–elements. The unique construct has the same meaning and syntax of key, but it also allows the specified attributes and/or sub–elements to be empty.

Both key and unique constraints have a scope, i.e. it is possible to declare different keys for the same element from different nesting levels.

Finally, the keyref construct defines a relation between two elements, requiring the matching of a set of attributes and/or sub–elements of an element with those in the key of another element.

6.3 A FRAMEWORK FOR SOFTWARE ANALYSIS INTEGRATION

In this section we introduce the framework we propose to integrate several functional and non-functional analysis of software architectures.

Figure 6.4 shows the architecture of such a framework.

Rounded boxes on the top side of the figure represents software notations adopted for the software development (e.g., the *Unified Modeling Language* or whatever *Architectural Description Language*). Let us assume that a software architecture has been built using one of these notations. Several methodologies are nowadays available to take as input a software model and to produce the same model in a different notation, ready to be validated by automated tools with respect to either functional or non-functional properties. On the bottom side of Figure 6.4 are presented some examples of methodologies (i.e. CHARMY and *TwoTow-*

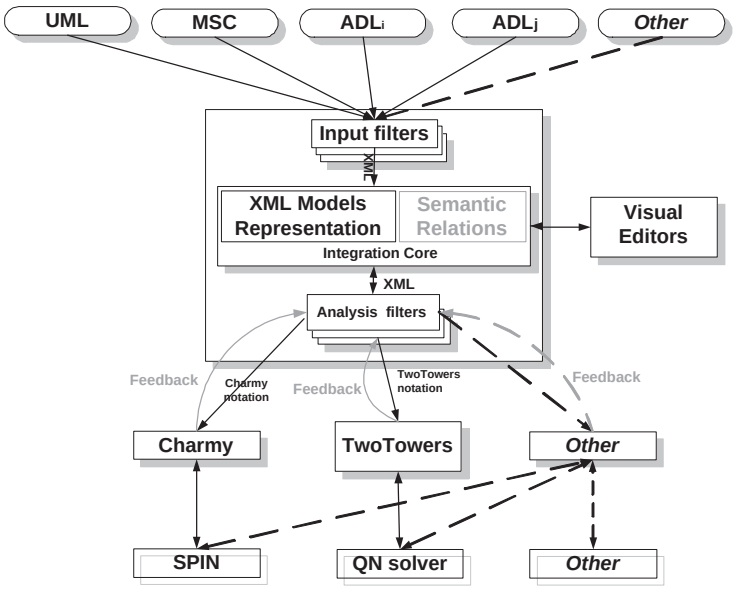


Figure 6.4: The Framework Architecture.

ers) as square boxes, and some examples of automated tools for software analysis (e.g., the *SPIN* model checker).

In the scenario of a stand-alone analysis of a software architecture, the sequence of steps to validate the architecture, for example from a functional viewpoint with the CHARMY approach, would be as follows: the UML diagrams of the software model from the rounded box directly flow to the CHARMY square box, where they are translated into the *SPIN* specification language (i.e., *Promela*) and forwarded to the *SPIN* model checker (a square box in the bottom side of Figure 6.4). *SPIN* runs the model and produces results that have to be examined by the developer in order to embed the analysis feedback into the software architecture.

In our framework the validation steps are different, due to the integration of such activity with other potential software analysis as well as to a certain automation in the interpretation and provision of result feedback. In Figure 6.4 a big square box has been placed between the topmost software notations and the bottommost analysis methodologies. It contains some filters and the XML *Integration Core*, which is the main component of our framework.

Each *Input filter* translates the software model from its original notation to a XML-based common representation, namely the *XML Models Representation* box in Figure 6.4. An appropriate *Analysis filter* translates the XML representation into the input notation to the desired analysis methodology (e.g., CHARMY *notation* in Figure 6.4). The latter notation obviously depends on the methodology, and it can go from a subset of the XML representation to a completely different language defined within the methodology. From this point on the steps are the same as for a stand-alone analysis, up to obtain results from the automated tool.

The feedback resulting from a specific analysis (e.g., splitting a component upon a deadlock detection) propagates, through an *Analysis filter*, up to the XML *Integration Core* where the model is either updated or a hint is given on how to modify it.

In the *Semantic Relations* box the rules that link entities to entities are expressed (in XML) and allow to transfer analysis feedback from a notation to another. In fact, the model changes inferred by the analysis results in the XML *Integration Core*, have to be reflected in the other analysis methodologies. This is the way we conceive analysis integration.

In the *Visual Editors* box on the right side of Figure 6.4 there can be any editor able to take a XML representation of a software model and display it. In practice, this is an additional graphical capability of our framework that may extend analysis tools with graphical user interfaces providing either a way to interact with the software models or a way to operate with the analysis methodology tools.

In the next section we give some details of the XML *Integration Core* and how it works for analysis integration.

6.3.1 Architecture of the XML *Integration Core*

The Integration Core purpose is twofold. It contains an easy and manageable representation of all the notations taken in input from the considered analysis approaches by means of a common language (XML). It allows the integration of the analysis in terms of the analysis results and the produced feedbacks at the software architecture level.

To reach this last aim we introduce the concept of semantic relations among the entities of the considered notations. Semantic relations are built every time it is possible to semantically relate the concepts in different notations. This means that, in general, there is not necessarily a relation between every pair of entities of two different notations; sometimes, those relations could not exist at all.

The semantic relation between two elements of two different notations strongly depends from the used approaches. This implies that the semantic relations are given by considering the approaches pairwise. We define the structure rules specifying the relations between concepts of the considered notations. Of course, when a particular software system is analyzed, these structure rules have to be instantiated on it. The rules instantiation is performed by a dedicated engine containing the needed logic to do so. We point out that an engine has to be built for each pair of approaches.

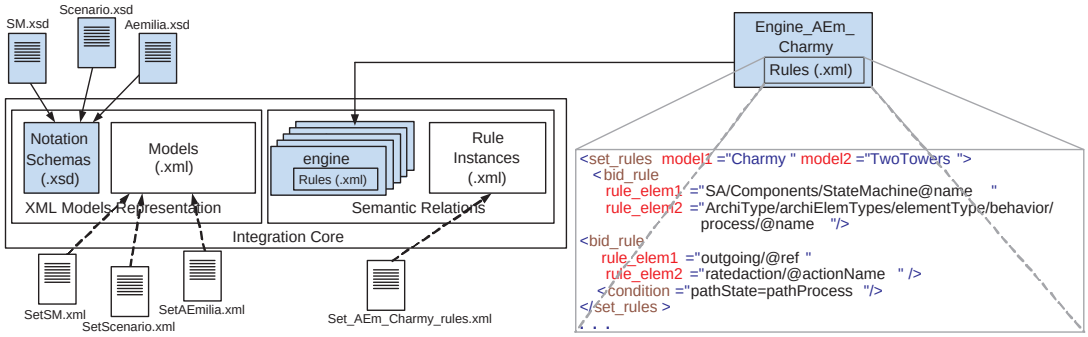


Figure 6.5: Structure of the Integration Core

From the previous considerations, we split the *Integration Core* into the *XML Models Representation* and the *Semantic Relations* as shown in Figure 6.5.

Given an analysis approach to be integrated it is necessary to provide an XML schema for each notation taken in input. A schema describes how to produce the XML representation of the software architecture under analysis. The schemas are stored in the *Notation Schemas* repository (see Figure 6.5) while the *Models* repository contains the software architecture description of the system, represented in XML. For example, let us consider the CHARMY and TwoTowers approaches, and a Set-Counter application which will be introduced later. The schemas of the State Machines, Scenarios and Æmilia notation (in Figure 6.5, SM.xsd, Scenario.xsd and Æmilia.xsd, respectively) are stored in the *Notation Schemas* repository. The *Models* repository contains the XML version of the State Machines, the Scenarios and the Æmilia

textual description modeling the Set-Counter application (in the Figure 6.5, SetSM.xml, SetScenario.xml and SetÆmilia.xml, respectively).

Semantic Relations, instead, contains, for each pair of approaches, a set of structural rules which define relationship classes (in Figure 6.5 *Rules* (.xml)), an engine which instantiate the relationships defined by the structural rules on the current architecture. The instantiated relationships are stored within this component (in Figure 6.5 *Rule Instances* (.xml)). The engine role is to instantiate the structure rules for a particular software architecture of the system that we want to analyze.

In Figure 6.5 we show some structure rules for CHARMY and TwoTowers approaches. In particular we show two rules: the first states that there exist a semantic relation between the CHARMY state machine model and the Æmilia process; the second one claims that a transition over a State Machine is related to a rated action of an Æmilia process if they belong to the same execution trace. The relation defined by the rules are specified by using XPath expression [157] over the notation schema. In this way we are able to unambiguously relate the notation elements.

6.4 FIRST IMPLEMENTATION OF THE XML INTEGRATION CORE

We devise an incremental approach to the framework implementation. The analysis approaches are considered pairwise, and for each pair we intend to introduce only the missing schemas and rules needed to integrate the approach into the framework. Therefore the *Notation Schemas* repository as well as the XML *Rules* may be extended every time two approaches have to be related.

As first step, we have implemented (input and analysis) filters, XML schemas, rules and engine for Charmy and TwoTowers approaches. We found the XML characteristics fairly suitable for such project, and we are nowadays working on integrating other approaches (e.g., the PRIMA-UML approach for Queueing Network-based performance analysis from UML diagrams [66]).

Although some similarities can be found, we remark that our integration task differs from the UML 2 project [148] because we do not intend to push software developers to use a specific notation (such as UML), rather we want to provide tools to make the software analysis as much transparent as possible to the software development process. We figure out developers not necessarily being constrained to a specific notation, and only if (and when) the need of an integrated software analysis comes up during the software lifecycle they can consider to enter the framework by providing the necessary filters, schemas and rules.

We share with the UML 2 project the idea of having an XML intermediate format as a basis for the analysis tasks. Indeed, since the XMI standards for UML 2 are not yet out up to this date, it has been and it is being our concern to make our XML schemas as much compliant as possible to the XMI standards for UML 1.x. However, even UML 2 is not wide enough to embed formal notations such as Process Algebras which are widely used, for example, in performance analysis. Therefore we try to work towards an integration framework that embeds UML as a software development notation (see Figure 6.4), relations among UML entities as rules already embedded into the notation and, in addition, provides tools to integrate whatever analysis approach.

In practice, our rules work at the metamodel level, as they relate notation concepts to notation concepts. The need of rules to integrate software analysis is supported from an OMG call for proposals [14].

6.4.1 THE CONSIDERED SOFTWARE ANALYSIS METHODOLOGIES

CHARMY (CHECKING ARCHITECTURAL MODEL CONSISTENCY) CHARMY is a framework that, since from the earlier stage of the software development process, aims to assist the software architect in designing

software architectures and in validating them against functional requirements. State machines and scenarios are used as the source notation for specifying software architectures and their behavioral properties. Model checking techniques, and in particular the model checker *SPIN* [98], are used to check the consistency between the software architecture and the functional requirements by using a Promela specification and Büchi Automata [48] which are both derived from the source notations. The former is the *SPIN* modeling language, while the latter is the automata representation for Linear-time Temporal Logic (LTL) formulae [127] that expresses behavioral properties.

CHARMY currently offers a graphical user interface which aids the software architecture design and automates the machinery of the approach.

Technical details on CHARMY may be found in [101], and an approach to integrate CHARMY into a real software development life-cycle can be found in [101, 59].

TWOTOWERS: ÆMILIA TO MARKOV MODELS The TwoTowers (3.0) tool [40] allows the validation of performance requirements at the software architecture level. It takes as input an Æmilia textual description, builds the corresponding Markov model (which can be both a Continuous and a Discrete Markov Chain) and evaluates the performance indices of interest.

Æmilia is an architectural description language (ADL) based on the Stochastic Process Algebra $EMPA_{gr}$ [41]. It was introduced by Bernardo et al. in [43] with the aim of making the Stochastic Process Algebra a more familiar software model notation to software engineers.

Stochastic Process Algebras (SPA) permit to analyze the performance of concurrent systems which are described as collections of entities, or processes, executing atomic actions. The processes are used to describe concurrent behaviors and they synchronize in order to communicate. Temporal information is added to actions by means of continuous random variables, representing activity durations. The quantitative analysis of the modelled system can be performed by constructing the underlying stochastic process. In particular, when action durations are represented by exponential random variables, the underlying stochastic process yields a Markov Chain.

A model description in Æmilia represents an Architectural Type (AT) defined as a function of its Architectural Element Types (AET) and its architectural topology. An AET is defined by its behavior, specified either as a family of $EMPA_{gr}$ sequential terms or through an invocation of a previously defined AT, and by its interactions, specified as a set of $EMPA_{gr}$ action types occurring in the behavior. The architectural topology is specified through the declaration of a set of Architectural Element Instances (AEI) and a set of Directed Architectural Attachments (DAA) among the interactions of the AEI. Depending on the SA configuration to be specified by means of Æmilia, it can be necessary one or more AEI for each AET.

6.4.2 AN EXAMPLE: THE SET-COUNTER APPLICATION

To illustrate details on the *Models* and *Rule Instances* repositories, we use a simple example introduced in [79]. The application is made of two components: a Set and a Counter. If a User adds or removes an element to/from the Set ($insert(e)$ and $delete(e)$ respectively) the Set increments or decrements (inc and dec respectively) the number of stored elements into the Counter component. In the following, we refer to the Set-Counter architecture containing one User instance, one Set instance and one Counter instance ([1]).

In Figure 6.6 we report the CHARMY State Machines and the Æmilia specification for the Set-Counter application (Figure 6.6.b and 6.6.c, respectively) and a sketch of the XML representations for the User

[1]For lack of space, we ask the reader to refer to [149] for details about the schemas and the XML files for the Set-Counter application

component (Figure 6.6.a and 6.6.d). These representations are built by using the appropriate input filters (interested reader can found details in [149]).

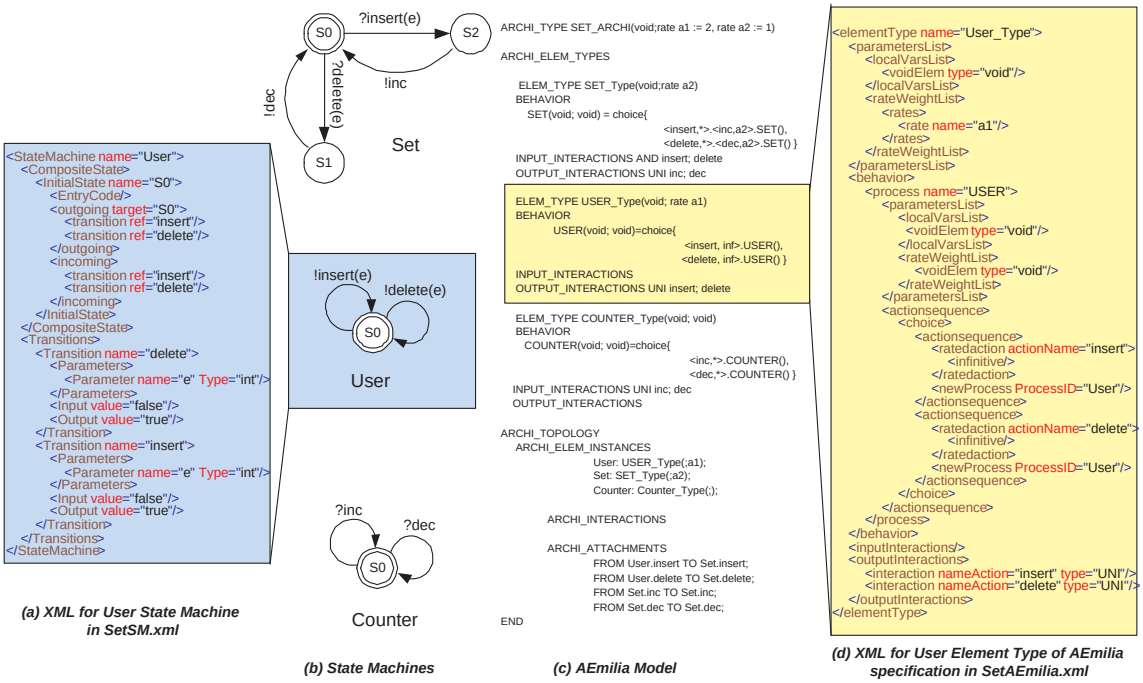


Figure 6.6: Architecture of Set-Counter Application

A fragment of the rules instances expressing the semantic relations between the CHARMY and TwoTowers approaches is shown in Figure 6.7. At the rightmost side of the figure, you can find the rule instance that realizes the relation between the `insert` transition in the User State Machine and the `insert` rated action of the User process inside the Æmilia description. Both these rule members are identified by means of the XPath expressions specifying their positions in the XML representations.

6.5 THE ECLIPSE PLATFORM

The Eclipse platform is an Integrated Development Environment (IDE) for anything and nothing in particular [131]. Eclipse is a framework for building integrated development environments for creating applications, and its main role is to provide mechanisms and rules to create seamlessly integrated development tools, and more. The Eclipse platform supports the construction of tools from an unrestricted variety of tool providers, and facilitates the *integration* of such tools, that usually manipulate different content types, and are developed by different providers.

The Eclipse platform architecture is shown in Figure 6.8(a) and is made of the following components: i) *Platform runtime*: Provides all the low level Application Programming Interfaces (API) to the functionalities of the platform, which can be used by tool providers. ii) *Workspace*: Provides a consistent and efficient way to organize the data used by the tools deployed in the platform. iii) *Workbench*: Enables the tool provider to display a graphical view of the data stored in the workspace, and to provide a graphical user interface to interact with the installed tools. iv) *Help, Team*: Provides an integrated help system and the team working capabilities for sharing the data in the workspace between multiple users.

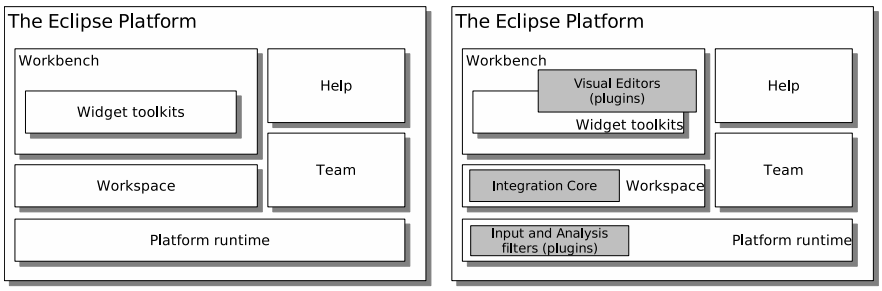
Figure 6.7: Rules Instance for the Set-Counter Application in TwoTowers-CHARMY Integration

6.5.1 DESIGNING THE FRAMEWORK IN ECLIPSE

The framework described in Section 6.3 maps very well to the Eclipse platform, which can be used to profitably implement most of the components of such a framework as extensions to the platform itself, by means of plugins.

As regard to the framework architecture depicted in Figure 6.8(b), we give an outline of how the architectural components can be implemented using the Eclipse framework:

- *Integration Core*: It can be directly mapped to the Eclipse platform workspace, where all the data used by the tools deployed in the platform are saved. Since XML is the target language for storing information about the structures represented by models in the Integration Core, the Eclipse platform



(a) The Eclipse Platform Architecture (b) The Integration Framework over *Eclipse*

Figure 6.8: *Eclipse* Platform and its Instantiation for Our Framework

runtime API may be profitably used to handle such information. In fact the whole Eclipse platform relies on XML for handling data files and, as a consequence, it provides a complete support for XML processing.

- *Visual Editors*: The workbench, with its graphical capabilities and widget toolkits, may be used to export graphical representation of the integration core, in order to spot model sections which need to be revised (as a consequence of some analysis) or even to provide a visual way to act directly on the model by changing it. Analysis methodologies which lacks of visual tool, may also take advantage of the workbench in order to provide visual editors that can give a methodology-oriented view of the models they require.
  Methodologies which already have their set of modeling tools, may provide visual editors which expose a sort of control panel for those tools. In this way, external tools can be used as before, but in a more integrated way, from a single environment.

- *Input and analysis filters*: These filters may be implemented as plugins which interact directly with the integration core, modifying it as needed. Notification mechanisms provided by the Eclipse platform may be also used for triggering some application logic which acts on the integration core as a consequence of an input filter importing of new models, or an analysis filter feedback which modifies the model itself (directly or indirectly).

As a side effect, all the previously cited architectural components, may take advantage of the other Eclipse framework components, especially from the team working capabilities which enables the versioning of the data stored in the workspace, and therefore in the integration core; this could be useful for keeping track of the changes made to the models and, in case, provide an effective way to rollback such changes.

It is also important to point out that many companies are developing modeling tools using the Eclipse framework. Notably UML modeling tools [122] are already available and can be readily used also in the context of our integration framework. Moreover, the fact that development environments for Java and C++ have already been integrated with the Eclipse framework may be useful for bridging the gap between the models and code, for example by automatically generating skeleton code from the models stored in the integration core.

6.6 SUMMARY

In this Chapter we have introduced a framework to support the integration of functional and non-functional analysis of software systems at architectural level. This work originates from the activities in our software engineering lab, where we experienced the crucial need of merging results from different software analysis approaches in order to better refine software architectures.

Our framework lays on an XML-based integration core, where software models and semantic relations between the models are represented. The aim is to provide a seamless integration of different analysis methodologies. To this regard we have sketched guidelines to allow embedding new methodologies in our framework. We have also shown how such a framework can be implemented using the Eclipse platform.

CHAPTER 7

INTEGRATION OF A SOFTWARE PERFORMANCE ENGINEERING METHODOLOGY IN THE TOOL INTEGRATION FRAMEWORK

The aim of this Chapter is twofold: to integrate two SPE approach tools in order to realize a fully automated predictive performance analysis, and to show its integration in the framework tool we presented in the previous chapter. The SPE approaches we considered are the one proposed by Cortellessa and Mirandola in [66] and the other from Smith and Williams discussed in [145] in detail. The first defines a process to generates Execution Graphes and Queuing Networks from software architecture design described by means of annotated UML Use Case, Sequence and Deployment Diagrams. This approach is implement in the XPRIT tool [64]. The second instead determines the workload proposed to the systems by reducing the Execution Graph and uses such an information to properly parameterize the queuing network. This second approach interacts with QNAP tool to evaluate the obtained model and gain the performance indices. This second approach is implemented in the SPE•ED tool.

To accomplish our goals we reviewed the S-PMIF meta-model proposed by Smith et al. in [160]. It formally defines the Execution Graph, in terms of entities and relationships among them. Moreover, we defined upon that meta-model an XML Schema that specifies the syntax of the language we used to integrate the two tools. To integrate the two tools we also made usage of the PMIF meta-model proposed by Smith and Williams in [160] and reviewed by Smith and Lladó in [141] to represent the QN model.

The effort spent in the realization of the integration of the two approaches provided us a new piece of the XML Integration Core and a first step towards the insertion of the SPE approach into the tool integration framework has been done. In particular, we made the XML schema of the S-PMIF part of the XML Model Representation in the XML Integration Core. Indeed the approach insertion in the framework can be made at two levels: at the UML notation level and at the Execution Graph level.

In the following, we first show the xml schema definition step and the integration of the two tools and later, in the last section, we map the implemented modules in the integration tool framework entities discussed in the Chapter 6.

7.1 TOWARDS A FULLY AUTOMATION OF THE SPE PROCESS

The SPE process uses multiple performance assessment tools depending on the state of the software and the amount of performance data available. This Chapter describes two XML based interchange formats that facilitate using a variety of performance tools in a plug-and-play manner, thus enabling the use of the tool best suited to the analysis. A Software Performance Model Interchange Format (S-PMIF) is a common representation that can be used to exchange information between (UML-based) software design tools and software performance engineering tools. Using it, a software tool can capture software architecture and design information along with some performance information and export it to a software performance engineering tool for model elaboration and solution without the need for laborious manual translation from one tool's representation to another, and the need to validate the resulting specification. S-PMIF enables the following Software Performance Engineering (SPE) tasks:

1. Developers can prepare designs as they usually do and export the data to SPE tools where performance models can be constructed automatically.
2. The model transformation can be used to check that the resulting processing details are those intended by the UML specification.
3. Data available to developers can be captured in the development tool - other data can be added by performance specialists in the SPE tool.
4. Rapid production of models makes data available for supporting design decisions in a timely fashion. This is good for studying architecture and design tradeoffs before committing to code.
5. Developers can do some of this on their own without needing detailed knowledge of performance models.

The performance model interchange format (PMIF 2.0) is a common representation for system performance model data that can be used to move models among system performance modeling tools that use a queueing network model paradigm [141]. A user of several tools that support these formats can create a model in one tool and easily move models to other tools for further work.

This Chapter first defines an XML based S-PMIF based on the meta-model of software performance model information requirements in [160]. Then it demonstrates the feasibility of using both the S-PMIF and the PMIF 2.0 to automatically translate an architecture description in UML into both a software performance model and a system performance model to study the performance characteristics of the architecture. The software performance model provides best and worst case performance data for an architecture/design. If the predicted performance results do not meet performance requirements, the model identifies critical areas and makes it easy for an analyst to study alternatives for correcting problems and quantify the performance improvement of each. Once an appropriate architecture/design is selected, the PMIF can be used to transfer the model to a system execution model to study additional facets of the operating environment and look for problems due to contention, locking, etc., and to study the effect of changes in the computer or network environment.

This overall process is beneficial because no single tool is good for everything. Early in development one needs to quickly and easily create a simple model to determine whether a particular architecture will meet performance requirements. Precise data is not available at that time, so simple models are appropriate for identifying problem areas. Later in development, when some performance measurements are available, more detailed models can be used to study intricacies of the performance of the system. At that time, different tools are desirable that provide features not in the simpler models. These ”industrial strength” modeling tools are seldom appropriate earlier in development because the models take additional time and expertise to construct and evaluate, and it is seldom justified when performance details are sketchy at best.

A common set of XML based interchange formats lets one use a variety of different tools as long as they support the interchange. Each tool must either provide an explicit import and export command, or provide an interface to/from a file and an XSLT translation can convert between the interchange format and the file. The translation can be relatively easy.

Earlier work defined both a meta-model for software performance models and a PMIF using an EIA/CDIF (Electronic Industries Association/CASE Data Interchange Format) paradigm for transferring information between CASE tools [144, 160]. The PMIF was subsequently enhanced and implemented in XML [141]. An exchange takes place via a file and internal tool information is translated to and from the file’s transfer format. The transfer format in the original CDIF standard used LISP as the implementation language. Today, XML is a more logical choice for a transfer format because it was designed for this purpose and there are many tools available to support the exchange of information in XML.

This project uses the SPE meta-model as a starting point, and contributes the following to the interchange process:

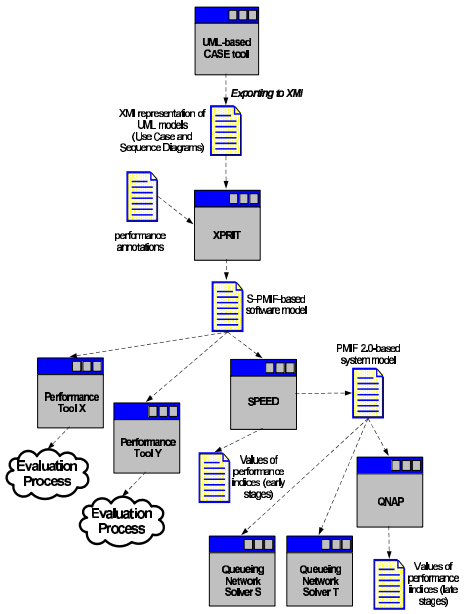


Figure 7.1: The SPE process.

- An updated SPE meta-model
- Definition of the XML schema based on the meta-model
- Implementation of extensions to the XPRIT software to export UML models into the S-PMIF
- Implementation of a new function in the SPE•ED software to import S-PMIF models
- Demonstrated feasibility with an experimental proof of concept that uses both interchange formats to combine the use of software performance engineering models and system performance models to predict performance from a UML specification.

After discussing related work, this Chapter describes the SPE meta-model and the XML schema based on it. Then it presents the SPE process for model exchanges and the required extensions to XPRIT and SPE•ED. The SPE process and the experimental proof of concept are presented. Plans for future work and conclusions complete the presentation.

Differently from other similar work, this follows the software performance engineering approach where from an annotated UML software specification, a software performance model is first derived and evaluated using a software modeling tool, like SPE•ED [142, 143], which outputs are normally enough in early stages of design. When more specific performance measures are needed, the model can be exported as a Queuing Network model and analyzed with a system modeling tool, like Qnap. Furthermore, our approach proposes and uses common XML based interchange formats, S-PMIF and PMIF 2.0, which allow multiple tools to be used to solve the models. Tools may be used in a ”plug and play” fashion to select the tool best suited for a particular problem. It simplifies the implementation of an interchange process because tools only need to interface with the interchange format and need not develop custom interfaces to each other. The whole process that we devise is illustrated in Figure 7.1.

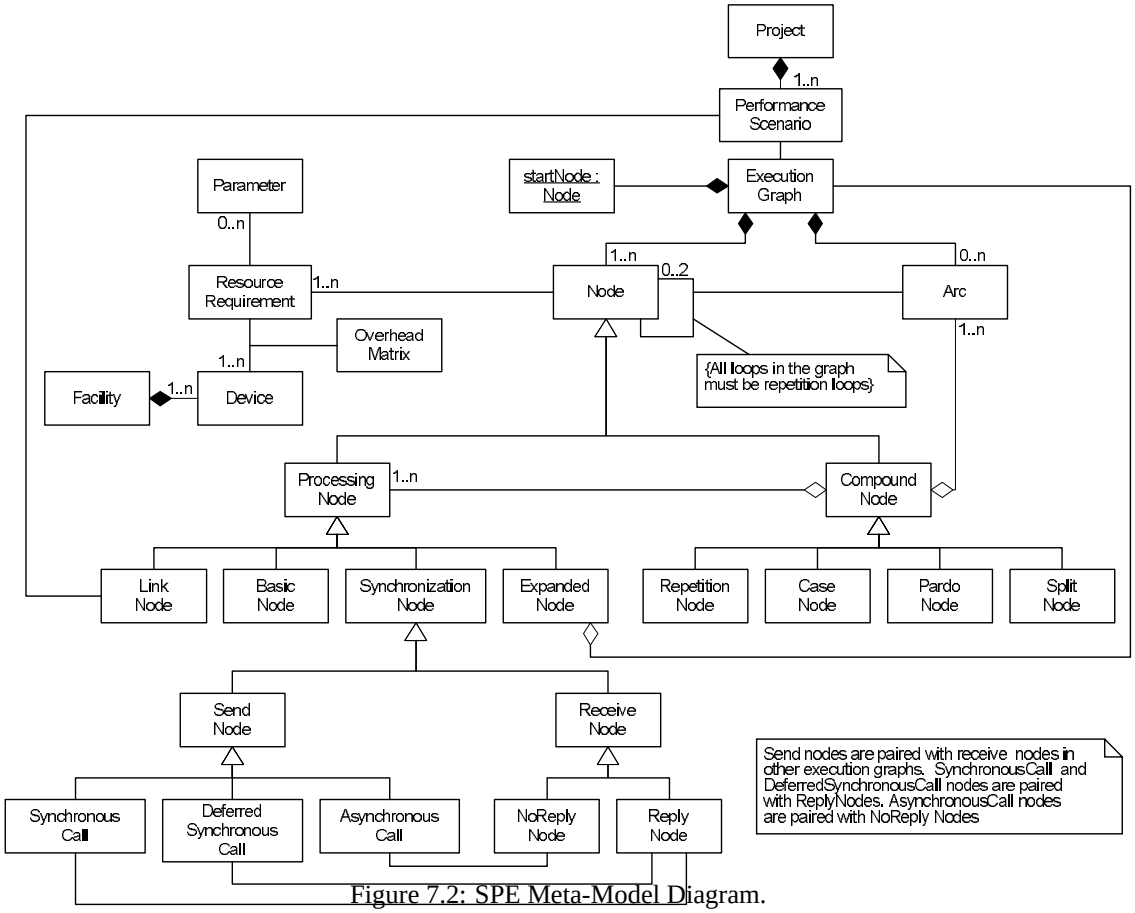


Figure 7.2: SPE Meta-Model Diagram.

7.2 SPE META-MODEL

The SPE meta-model formally defines the information required to perform an SPE study. This model is known as the SPE meta-model because it is a model of the information that goes into constructing an SPE model. Note that this meta-model is different from the Performance Model Interchange Format (PMIF) discussed in [140, 141, 144]. The PMIF defines information exchanged between queueing network modeling tools (QNM) while the meta-model defines information to be exchanged between UML software design tools and performance tools. Additional information, such as the mapping of components to processing locations as well as the internal characteristics of software locations may be exchanged between UML and performance tools. This exchange may lay on PMIF or an extension of it where needed.

7.2.1 SPE META-MODEL 2.0

This meta-model defines the essential information required to create the software and system performance models as defined in [139, 145]. The SPE meta-model class diagram is shown in Figure 7.2. Figure 1b shows the attributes of each object. (Note: Object attributes are typically defined as part of the class diagram. They are shown in Figure 7.3 here to conserve space.) The following paragraphs describe the classes and their relationships. The complete definition is in [147]. An SPE study is based on Projects which contain one or more PerformanceScenarios. Each PerformanceScenario is modeled by an ExecutionGraph. An ExecutionGraph is composed of one or more Nodes and zero or more Arcs. A Node may be connected to

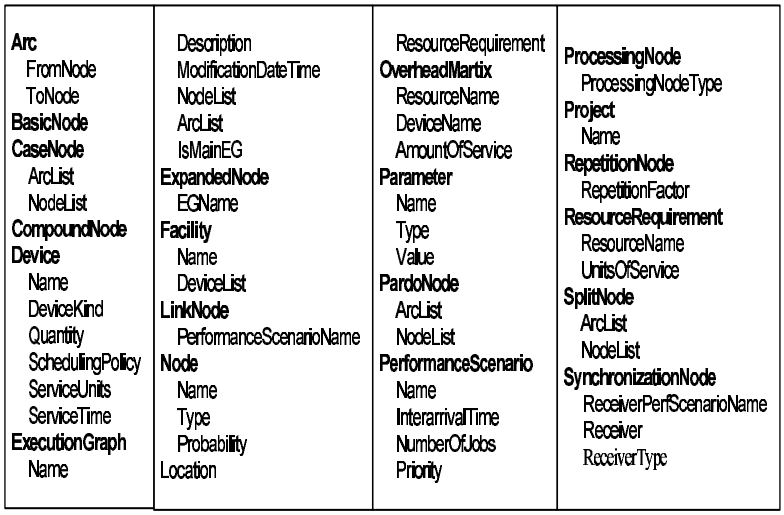


Figure 7.3: Meta-model Attributes.

0, 1, or 2 other Nodes via an Arc [1]. Several types of Nodes may be used in constructing an ExecutionGraph:

ProcessingNode: ProcessingNode: represents processing steps at an appropriate level of detail. There are four types of ProcessingNodes:

1. `BasicNode`: represents a software processing step at the lowest level of detail appropriate for the current development stage.
2. `ExpandedNode`: indicates that processing details are expanded in a subgraph at the next level of detail. The subgraph is, itself, another ExecutionGraph.
3. `LinkNode`: represents a component whose execution requirements are specified in a previously saved performance scenario.
4. `SynchronizationNode`: represents communication and synchronization with a SynchronizationNode in another PerformanceScenario. A SynchronizationNode may be a SendNode or ReceiveNode.

   4.1 SendNode represents a call from one process to another. There are three types of SendNodes:

   4.1.1 SynchronousCall: represents a call in which the caller waits for a reply before proceeding

   4.1.2 DeferredSynchronousCall: represents a call in which the caller continues to execute and later requests the reply. If the reply is not available at that time then the caller waits.

   4.1.3 AsynchronousCall: represents a call with no reply.

   4.2 ReceiveNode: represents the receipt of a request from another process. There are 2 types of ReceiveNodes:

   4.2.1 ReplyNode: represents receipt of request that requires a reply. It can be used with either a SynchronousCall or a DeferredSynchronousCall.

   4.2.2 NoReplyNode: represents receipt of a request for which a reply cannot be sent (i.e., an AsynchronouCall).

CompoundNode: CompoundNode: represents special processing structures, such as Case constructs, repetition, and parallel execution. There are four types of CompoundNode:

[1] Note that some CompoundNodes may be connected to more than 2 attached nodes, but Arcs are not defined for those connections. So Nodes can be connected to at most one predecessor and one successor Node by an Arc.

1. `RepetitionNode`: represents processing that is repeated and a repetition factor specifies the number of repetitions.
2. `CaseNode`: represents conditional execution of components, each with a probability of execution.
3. `PardoNode`: represents parallel execution paths, each with a probability of being initiated. The parallel execution paths join when they finish.
4. `SplitNode`: indicates the initiation of concurrent processes, each with a probability of being initiated, that need not join.

A CompoundNode is also composed of one or more ProcessingNodes and one or more Arcs. The resources used by a Node are specified by one or more ResourceRequirements. A ResourceRequirement may be described by an optional Parameter. A Facility is a collection of Devices. A ResourceRequirement is executed on one or more Devices. A Device represents a unit that provides some processing service. ResourceRequirements are associated with Devices by an OverheadMatrix which specifies the amount of service that each resource type requires from various devices. The current version of the meta-model does not include performance requirements. Currently, performance requirements are defined informally, based on the type of problem and expert judgment. Inclusion of performance requirements in the meta-model will require that they be more formally defined. This is a topic for future research.

The OverheadMatrix merits some additional explanation. It is based on a concept in [145] and the SPE product, SPEoED [142, 143] used in this demonstration. The OverheadMatrix is an associative entity; it describes the relationship between a ResourceName and a Device. An individual instance of OverheadMatrix contains a ResourceName a DeviceName and an AmountOfService. For example, the ResourceRequirement may specify the number of instructions to be executed. The OverheadMatrix would specify the CPU processing time per instruction as the AmountOfService for the CPU Device. The class may be viewed as a table with each instance corresponding to a row that specifies a distinct ResourceName/DeviceName pair such as:

- instructions and the CPU processing time per instruction,
- database updates and the CPU processing time per update
- database updates and the Disk device visits per update.

The use of the overhead matrix makes it possible to separate the portion of the model that describes the software from the portion that describes the execution environment. This is important for the SPE approach because developers are often able to specify the software resource requirements such as the number of database updates or messages transmitted, but are

7.2.2 Adjustments to the Meta-Model

The following changes were made to the original SPE meta-model to reflect more recent information in [145]:

- The StateIdentification node was deleted and the SynchronizationNode was added a subclass of ProcessingNode
- Facility was added
- Project was added

- Device definitions were modified to specify the specific kind of device (such as CPU, Disk, etc.) rather than the generic terms FCFS, NonFCFSDemandSpec, and NonFCFSTimeSpec.

Other minor changes were made to class attributes for the XML implementation. For example, XML schemas allow names to be used as IDs and ID references, so NodeIds were eliminated. We changed the specification for names to match XML names in `http://www.w3.orgTR/2004/REC-xml-20040204/#id`. Other changes are similar to those made in [141].

7.2.3 S-PMIF XML SCHEMA

The diagram of a portion of the XML schema corresponding to the S-PMIF meta-model is shown in Figure 7.4. The complete schema is at `www.perfeng.com/pmif/s-pmifschema.xsd`. The following excerpt shows the schema definition for an ExecutionGraph:

```
<xs:complexType name="EG_type">
        <xs:sequence>
            <xs:choice maxOccurs="unbounded">
                <xs:sequence>
                    <xs:choice>
                        <xs:element name="BasicNode" type="
                           BasicNode_type"/>
                        <xs:element name="ExpandedNode" type="
                           ExpandedNode_type"/>
                        <xs:element name="LinkNode" type="LinkNode_type
                           "/>
                        <xs:element name="SynchronizationNode" type="
                           SynchroNode_type"/>
                    </xs:choice>
                    <xs:element name="ResourceRequirement" type="
                       ResourceRequirement_type" "minOccurs="0"
                       maxOccurs="unbounded">
                    </xs:element>
                </xs:sequence>
                <xs:element name="CompoundNode" type="CompoundNode_type
                   "/>
            </xs:choice>
            <xs:element name="Arc" type="Arc_type" minOccurs="0"
               maxOccurs="unbounded"/>
        </xs:sequence>
```

unable to specify the device requirements for them. The overhead matrix thus provides a mechanism to separate the two and to obtain the ResourceName and UnitsOfService from the software specification and the OverheadMatrix from other sources such as measurement tools or computer experts

```
    <xs:attribute name="EGname" type="xs:ID" use="required"/>
        <xs:attribute name="IsMainEG" type="xs:boolean" use="required"
           />
        <xs:attribute name="StartNode" type="xs:IDREF" use="required"/>
        <xs:attribute name="ModificationDateTime" type="xs:dateTime"
           use="optional"/>
        <xs:attribute name="SWmodelname" type="xs:string" use="optional
           "/>
    </xs:complexType>
```

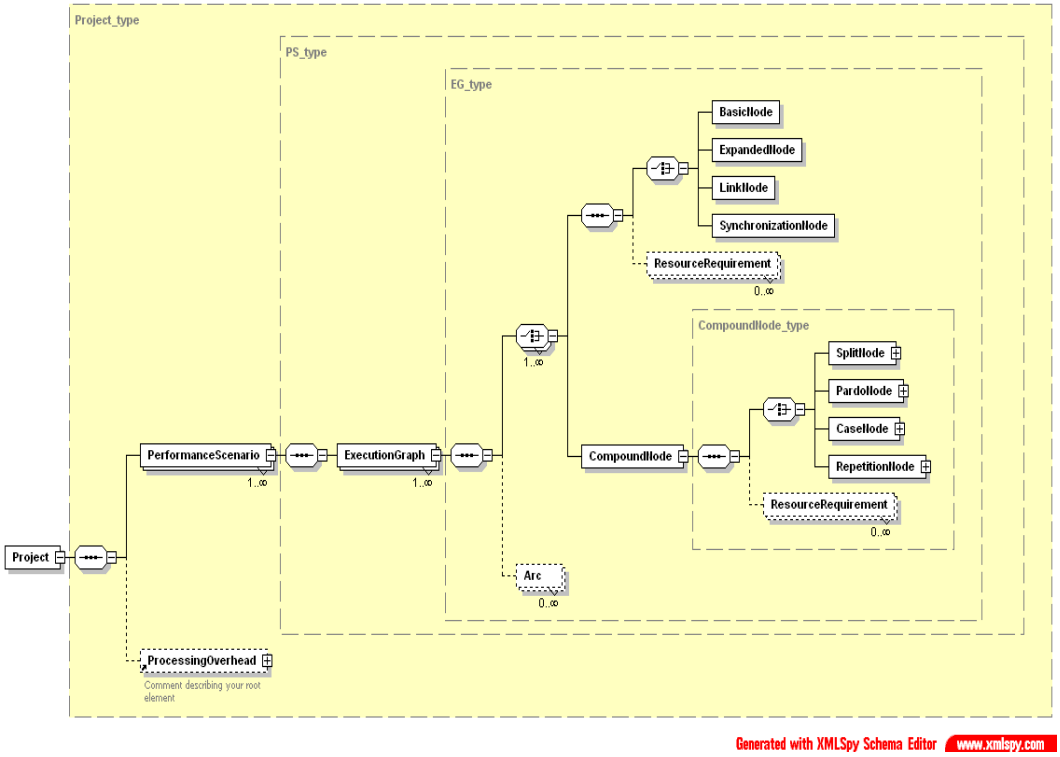


Figure 7.4: Portion of the XML schema corresponding to the S-PMIF meta-model.

The following is a sample s-pmif.xml ExecutionGraph specification:

```
<ExecutionGraph EGname="drawmod_1_EG" IsMainEG="true" StartNode="
    create_Model">
        <BasicNode NodeName="create_Model"/>
        <BasicNode NodeName="draw_Model"/>
        <CompoundNode>
            <RepetitionNode NodeName="r1">
                <ExpandedNode NodeName="e1" EGname="e1_ref"/>
            </RepetitionNode>
        </CompoundNode>
        <BasicNode NodeName="close"/>
        <Arc FromNode="create_Model" ToNode="draw_Model"/>
        <Arc FromNode="draw_Model" ToNode="r1"/>
        <Arc FromNode="r1" ToNode="close"/>
    </ExecutionGraph>
```

The schema has 2 differences from the meta-model. First, we flattened the hierarchy in several areas to simplify the xml. For example, both Nodes and ProcessingNodes are eliminated from the schema and their attributes are moved to the nodes that inherit those attributes. Second, we made some elements and attributes optional in the schema even though they are not optional in a software performance model. For example, a workload intensity such as interarrival time is necessary to solve a software performance model; however, the developer of the UML software diagrams may not know that information so it won't be required in the xml. Similarly, we made resource requirements, overhead matrix and device specifications optional. We discuss this issue further in the next section. We also created three separate schemas for the meta-model: Topology, $Overhead_Matrix$, and Device. They can be combined by including the appropriate schemas. Thus, Topology may include $Overhead_Matrix$ which includes Device. This is useful because one may use

any of the schemas without using the others. For example, if the overhead matrix specification is coming from another source it does not need to be included in the topology, and vice-versa.

7.3 SPE Model Interchange Process

Our vision for the SPE model interchange process is:

1. A software architect, designer, or developer would use a UML tool to create their model of the software and when ready for the assessment, export the model into S-PMIF.
2. A software performance engineer would then import the S-PMIF into a software performance modeling tool such as SPE•ED. They would likely need to supplement the information received from S-PMIF to add one or more of the following: resource requirements, facility and device characteristics, and the overhead matrix. The latter task may be skipped when the original UML model is annotated with all the additional performance information needed (using, for example, the UML SPT profile [87]), and the translation tool is able to process this additional information.
3. The software performance engineer would conduct performance studies, and if problems are found, modify the software performance model accordingly.
4. After resolving any serious problems with the software architecture and/or design, they may export the model into PMIF.
5. A performance engineer would import the PMIF into a system or network modeling tool for further investigation of performance properties of the network and computer system such as the effect of locking and contention with other work in the environment.
6. Results would then be exchanged in the reverse direction and ultimately the software specialist would be able to view suggestions for performance improvements and automatically update the UML to reflect selected changes.

This process differs from that proposed by other authors primarily because we envision the use of a software performance modeling tool such as SPE•ED between the UML and the system performance modeling tool. In our experience, we find many software problems that must be corrected before detailed study of the system performance is feasible. The case study described later illustrates. When problems are detected, it isn't enough to know that the system is saturated. It is also necessary to determine which parts of the software contribute to the problem and how much, in order to determine options for solving the problem. For example, the case study has a problem due to excessive disk usage. A software performance model can identify which portions of the software use the disk and enable the evaluation of different software alternatives that use less I/O. A system performance model, however, will be limited to hardware improvement alternatives such as more or faster devices. The best solution may be a combination of the two. Our model interchange process enables the evaluation of all those options.

PHILOSOPHY - The model interchange strategy that we adopted from CDIF [75] is "export everything you know and provide defaults for other required information"; and "import the parts you need and make appropriate assumptions for required data that is not in the schema and thus the interchange file." We started with a use case for the SPE interchange process in which developers did not have resource requirement specifications, the facility or device information, etc. So it was necessary to fill in many default values such as equal probabilities for Case nodes, etc. Our PMIF experience led us to the realization that everything you know is not necessarily everything you use. For example, SPE•ED uses visits to specify routing, but it knows about probabilities, and it is relatively easy to calculate them. We created an "import-friendly"

PMIF; that is, we include both visits and probabilities to make it easy on the import side. It is easy to do on output and it lets many importers use simple tools like XSLT rather than requiring custom code to do the import. The redundant specifications are currently optional.

EXPORTING UML -MODELS TO S-PMIF - This is a two-steps task: (i) exporting UML diagrams from a CASE tool representation to an XML format, (ii) transforming the exported result into a S-PMIF model. For what concerns the first step, the XMI standard specifications [121] have been adopted by almost all UML CASE tools to export UML diagrams in XML. Actually XMI does not represent a specific Schema for UML diagrams, but gives formal specifications to build standard Schemas for UML diagrams. This is the reason for small differences among the XMI exporting results of UML tools. For the sake of this Chapter experiments we have used the Poseidon tool [5]. The XPRIT tool performs the second step [64]. XPRIT is made of two components: UML2EG, that allows to annotate Use Case and Sequence Diagrams and generate from the annotated diagrams an Execution Graph; UML2QN, that allows to annotate a Deployment Diagram and generate from the annotated diagram a Queueing Network representing the hardware platform where the software shall run. For the sake of these experiments we have used only UML2EG, as the generation of a Queueing Network has been delayed in the process. In particular, we have exploited the XPRIT capability of producing the structure of an Execution Graph (owing S-PMIF) from one or more UML Sequence Diagrams (represented in XMI). The translation algorithm is based on visiting the Sequence Diagram and recognizing elementary patterns. For each pattern in a Sequence Diagram a corresponding pattern of an Execution Graph is associated. The whole structure of the Sequence Diagram is used to interconnect elementary patterns in the Execution Graph. For example, UML2EG is able to recognize sequential and parallel patterns, synchronous and asynchronous communications. Some accommodations were needed on the UML diagrams to make XPRIT work on these experiments:

1. In order to avoid XPRIT considering the paths that depart one after the other from the same SD axis (see draw()'s leaving Beam in Figure 7.5) to all be parallel paths, we added return arrows to the diagrams;
2. XPRIT does not cope with object creation, as all the names of components acting in a diagram need to be known in advance; therefore object creation has been modeled as a standard synchronous message between two existing components;
3. Software loops are not part of the UML 1.x standards (which is the basis of XPRIT), so message labels have been exploited to delimitate the starting and the ending messages of a loop in a Sequence Diagram.

Note that the last limitation will disappear with UML 2 Sequence Diagrams, where frames have been introduced to delimitate special interaction patterns. A new XPRIT release is being implemented based on UML 2, so many translation steps will become straightforward.

IMPORTING S-PMIF MODELS INTO SPE•ED - SPE•ED uses the Document Object Model (DOM) to import the s-pmif.xml. It first loads and parses the document, then uses DOM calls to walk through each execution graph and create the corresponding nodes and arcs in SPE•ED.

SPE•ED required a custom interface because, rather than reading input from a file, it provides a graphical user interface that enables a user to quickly draw a model. When the input comes from an S-PMIF, there is currently no provision for location coordinates for the nodes. Therefore another special routine is required to "reformat" a graph and assign nodes to locations.

EXPORTING A PMIF.XML MODEL FROM SPE•ED -

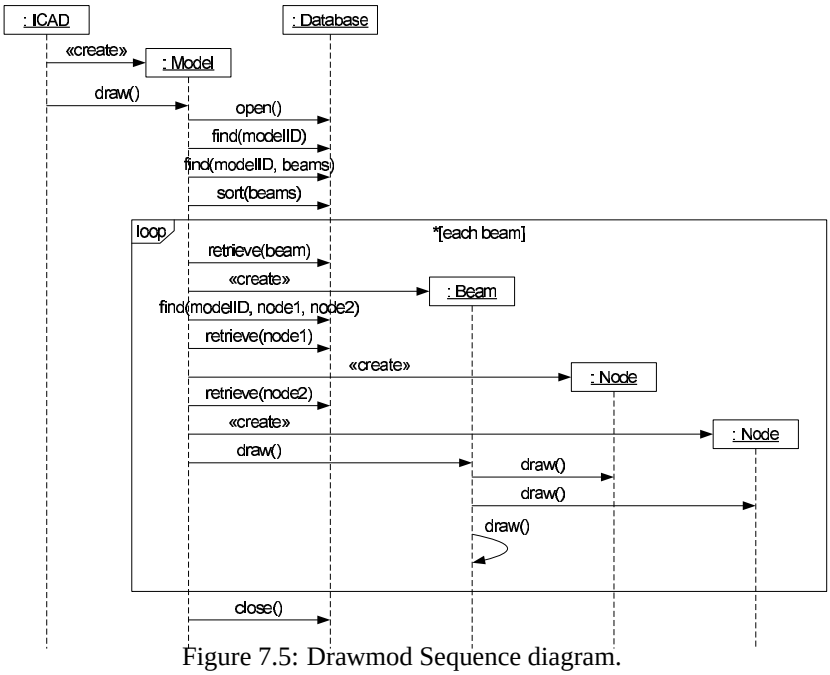


Figure 7.5: Drawmod Sequence diagram.

SPE•ED also uses the Document Object Model (DOM) to export the pmif.xml. It creates the entire document in memory, then writes it to a file. This facilitates the export because elements and attributes can be added in any order as long as they are added in the correct location. It is a relatively small file, e.g., 2-3K for the example in section 5, so the memory requirements are modest. SPE•ED uses a standard topology for models. Each facility contains a CPU and one or more other types of devices. Within a facility the QNM is assumed to be a central server model. Workloads begin execution on the CPU and upon completion transit to one of the other devices, then back to the CPU until completion. A model can contain multiple facilities, each with this central service topology. Several other cases required special handling, such as generating source, sink, and think nodes, transit probabilities, generating separate servers when quantity of servers is greater than one, name substitutions, etc. Details are in [141].

IMPORTING A PMIF.XML MODEL INTO QNAP - Qnap reads the input (QNM specification and solving parameters) from a file. Ultimately, Qnap would have an interface that would read from its standard file OR the pmif.xml file. However, we did not have access to Qnap source code and we could not implement such an interface directly. Therefore, we translated the pmif.xml file into a file in Qnap's format. The model translation from a pmif.xml file into a Qnap input file was done using XSLT. We generated a specific XSLT file that transforms a pmif.xml file into a file that can be directly read and executed by Qnap. The direct use of XSLT was feasible due to the possibility of specifying the stations by parts in the Qnap input file. This might not be possible for some other tools with stricter ordering in the input file, in which case two possibilities would arise: The use of DOM (as used by SPE•ED to export pmif.xml) or the use of XSLT together with a conventional programming language. The use of XSLT is fairly simple, therefore we would recommend XSLT when possible for the translation into a tool's file format. For the case of a real implementation (i.e.,implementing an interface from the tool that would read from the xml file directly), the use of DOM would be necessary since XSLT can only transform an XML file into another file. It would probably be advisable to read the entire pmif.xml file into memory then interpret and insert parameters into appropriate internal data structures because of the ordering in the XML schema. That is, some transformations may require information from elements that have not been read yet.

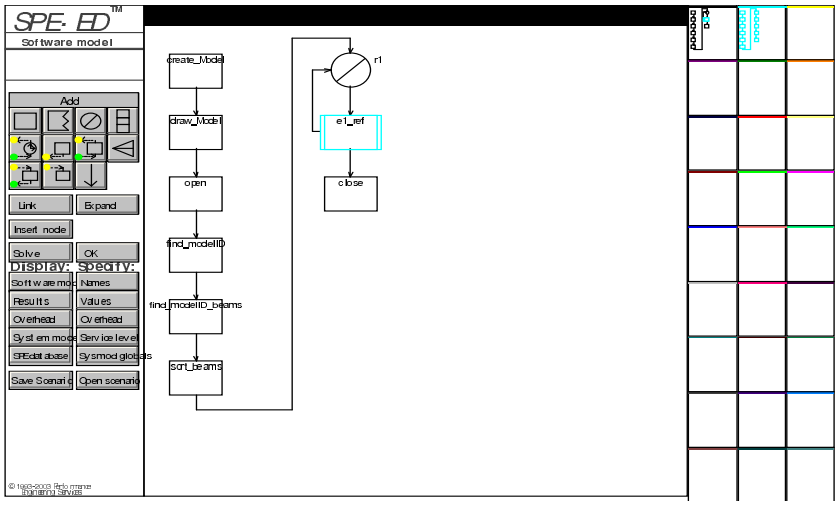


Figure 7.6: Generated SPE•ED Model.

7.4 EXPERIMENTAL RESULTS

For the proof of concept we used the Drawmod Architecture 1 model described in Chapter 4 of Smith and Williams' book ([145]). The sequence diagram for the model is in Figure 7.5.

The following is part of the XML file resulting from the XPRIT translation of the sequence diagram:

```
<PerformanceScenario ScenarioName="drawmod_1"
SWmodelfilename="drawmod_1_SD.xmi">
        <ExecutionGraph EGname="drawmod_1" IsMainEG="true" StartNode="
           create_Model">
            <BasicNode NodeName="create_Model"/>
            <BasicNode NodeName="draw_Model"/>
            <BasicNode NodeName="open"/>
            <BasicNode NodeName="find_modelID"/>
            <BasicNode NodeName="find_modelID_beams"/>
            <BasicNode NodeName="sort_beams"/>
            <CompoundNode>
                <RepetitionNode NodeName="r1">
                    <ExpandedNode NodeName="e1" EGname="e1_ref"/>
                </RepetitionNode>
            </CompoundNode>
            <BasicNode NodeName="close"/>
            <Arc FromNode="create_Model" ToNode="draw_Model"/>
            <Arc FromNode="draw_Model" ToNode="open"/>
            <Arc FromNode="open" ToNode="find_modelID"/>
            <Arc FromNode="find_modelID" ToNode="find_modelID_beams"/>
            <Arc FromNode="find_modelID_beams" ToNode="sort_beams"/>
            <Arc FromNode="sort_beams" ToNode="r1"/>
            <Arc FromNode="r1" ToNode="close"/>
        </ExecutionGraph>
        <ExecutionGraph EGname="e1_ref" IsMainEG="false" StartNode="
           retrieve_beam">
                    <!--Details omitted-->
        </ExecutionGraph>
```

```
</PerformanceScenario>
```

The Execution Graph has a Boolean attribute (IsMainEG) that indicates whether it is the main graph in the file or a sub-graph. It is represented from a sequence of nodes followed by a sequence of arcs between nodes. As long as the Sequence Diagram follows a sequential execution, all Basic Nodes are generated. Upon finding a loop, a Repetition Node is appended that refers to a subgraph identified from the EGname attribute $e1_ref$. The complete file is in [147]. Next the s-pmif.xml model was imported into SPE•ED and the software model was created. The generated software model is shown in Figure 7.6. Note that the text does not fit into the execution graph nodes because the operating system routines use spaces to insert line breaks; however, the XML names cannot contain spaces. Some translation of names will be necessary to create "prettier" models.

Next, we added the resource requirements (from the Drawmod example in [145]), then the model was solved. In general, software performance engineers will need to use the techniques in [145] to estimate requirements that are not in the interchange file. That is an important step in the overall process, but it is beyond the scope of this Chapter. The model was solved and problems were identified in the architecture. After making the architectural changes we produced Drawmod Architecture 3 (also described in [145]) and confirmed that it resolved the performance problems. Note that in this case, SPE•ED has the ability to solve the system execution model both analytically and with simulation to quantify the response time, utilization, etc. for computer resources so it isn't necessary to export the model to get those results. There are other reasons why one might want to export the model, such as:

- to compare solutions
- to get additional metrics such as queue lengths
- to study additional facets of the environment that might not fit the central server assumptions mentioned in section 7.3.

So the next step in the proof of concept is to export the model from SPE•ED into pmif.xml. The following shows an excerpt containing the generated service request (produced from SPE•EDs conversion of the software performance model into the system performance model):

```
<ServiceRequest>
    <DemandServiceRequest WorkloadName="Drawmod_Architecture_3" ServerID="
       CPU" ServiceDemand="3.574195E-03" TimeUnits="sec" NumberOfVisits="
       2219">
        <Transit To="Disk_A" Probability="4.867057E-02"/>
        <Transit To="Disk_B" Probability="4.867057E-02"/>
        <Transit To="Display" Probability="0.9022082"/>
        <Transit To="UserThink" Probability="4.506535E-04"/>
    </DemandServiceRequest>
    <WorkUnitServiceRequest WorkloadName="Drawmod_Architecture_3" ServerID=
       "Disk_A" NumberOfVisits="108">
        <Transit To="CPU" Probability="1"/>
    </WorkUnitServiceRequest>
    <WorkUnitServiceRequest WorkloadName="Drawmod_Architecture_3" ServerID=
       "Disk_B" NumberOfVisits="108">
        <Transit To="CPU" Probability="1"/>
    </WorkUnitServiceRequest>
    <WorkUnitServiceRequest WorkloadName="Drawmod_Architecture_3" ServerID=
       "Display" NumberOfVisits="2002">
        <Transit To="CPU" Probability="1"/>
    </WorkUnitServiceRequest>
</ServiceRequest>
```

The pmif.xml is then imported into Qnap. In this specific implementation the import consists of an XSLT

translation from a file in pmif's format into a file in Qnap's format. The generated Qnap input file for the Drawmod Architecture 3 is shown below. It can be seen that the stations need first to be declared and then they can be modified as many times as is wanted, so when reading the file sequentially, the last information read is the one that is taken. This makes the use of XSLT very convenient.

```
/DECLARE/ QUEUE UserThin , CPU;
          QUEUE Disk_A , Disk_B , Display;
          CLASS Drawmod_;
          REAL TDrawmod;
/STATION/  NAME= UserThin;
           TYPE = INFINITE;
/STATION/  NAME= CPU;
           SCHED = PS;
/STATION/  NAME = Disk_A;
           SERVICE = EXP(0.03);
           SCHED = FIFO;
/STATION/  NAME = Disk_B;
           SERVICE = EXP(0.03);
           SCHED = FIFO;
/STATION/  NAME = Display;
           SERVICE = EXP(0.001);
           TYPE = INFINITE;
/STATION/  NAME = UserThin;
           INIT(Drawmod_) = 10;
           SERVICE(Drawmod_) = EXP(60);
           TRANSIT(Drawmod_)= CPU, 1 ;
/STATION/  NAME = Disk_A;
          TRANSIT(Drawmod_) = CPU, 1 ;
/STATION/  NAME = Disk_B;
          TRANSIT(Drawmod_) = CPU, 1 ;
/STATION/  NAME = Display;
          TRANSIT(Drawmod_) = CPU, 1 ;
/STATION/  NAME = CPU;
           SERVICE(Drawmod_) = EXP(0.000001610723298783236);
           TRANSIT(Drawmod_) = Disk_A , 4.867057E-02,
                               Disk_B , 4.867057E-02,
                               Display , 0.9022082,
                               UserThin , 4.506535E-04 ;
```

The Qnap model is then solved and used for further study. The results of the initial solution are reported in [141] and are not shown here. This proof of concept illustrates the feasibility of the SPE process using XML based interchange formats for using multiple tools, rather than the particular results obtained from the models.

LESSON LEARNED - We learned several lessons while conducting the experimental proof of concept that are described in the following paragraphs. We found that there may be different interpretations of a UML sequence diagram and it may not be clear which is the proper interpretation. For example, the sequence of draw()s in Figure 7.5 were interpreted by XPRIT to be parallel steps because they did not have return arrows. We often find that, for convenience, developers do not specify return arrows from calls, and we do not want to require this specification just so the models can be exported. For this exercise, we just inserted the return arrows. In UML 2 there is a specific construct for parallel execution so this issue will no longer be a problem. In general, the interchange shows the value of viewing processing steps in different notations to confirm that the processing is specified the way the developer intended. Note that the translated model in Figure 4 is far more detailed than the Drawmod model in Figure 4-18 of [145]. Many of the processing steps in the automatically generated model are not interesting from a performance standpoint, and the extra steps tend to "clutter" the model. This is a departure from the simple model strategy described earlier. This is a common problem with automatic translation of designs. In many cases it may be easier to just create a new model and omit those details initially. Some techniques for "pruning" an automatically generated model would make it better suited for SPE. This proof of concept illustrates one pass from UML to Qnap. The SPE process will actually be iterative and there will be a need to exchange multiple models in the forward as well as the reverse direction. Thus, we will need to be able to retain information that was added by tools during the evaluation so that it won't have to be re-created each time, such as resource requirements, location coordinates, etc. We envision using the S-PMIF to transfer this information to the design tool where it will need to be imported, saved, and exported the next time this SPE interchange process is used.

7.5 THE SPE APPROACH IN THE TOOL INTEGRATION FRAMEWORK

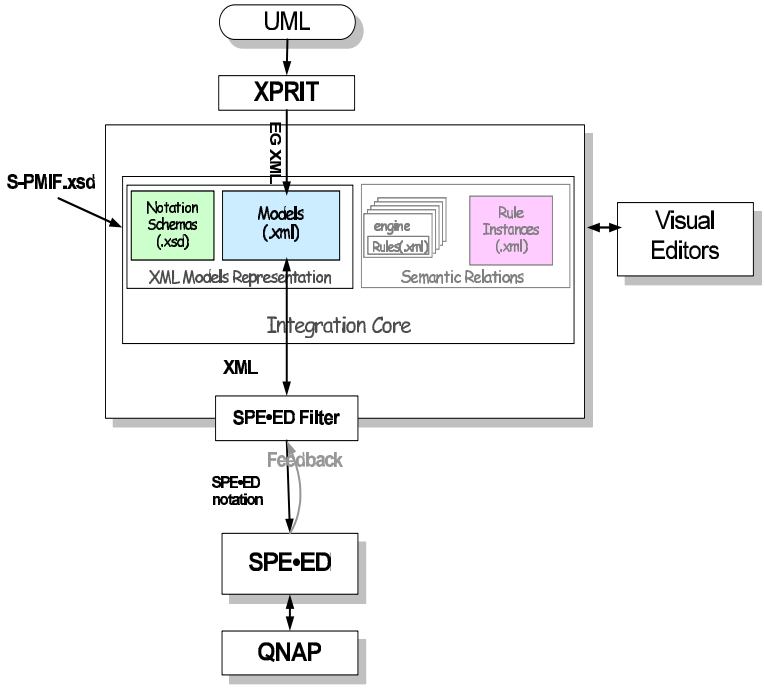


Figure 7.7: First Mapping between the SPE Approach Components and the Tool Integration Framework.

In this section we discuss how insert the above SPE approach(es) in the tool integration framework. We show the possible mapping among the SPE approach and the integration framework entities introduced in the Chapter 6. There are three are three ways to insert to SPE approach in the framework: (i) integration at the Execution Graph notation level by means of XPRIT (see Figure 7.7); (ii) integration at the UML notation level by means of the XMI exporting of the UML diagrams (see Figure 7.8); integration at the Execution Graph notation level by means of S-PMIF exporting (see Figure 7.9).

All the three possibilities have the SPE•EDtool as analysis tool. SPE•ED makes use of QNAP tool to evaluate the performance model form which gain values for the performance indices of indices.

In the first option, reported in Figure 7.7, the XML Schema for the S-PMIF meta-model becomes part of the *XML Integration Core* in the *XML Model Representation*. The `Input Filter` component role in the framework is covered by XPRIT tool that generate the Execution Graph from the UML diagrams. The origin notation is the UML language describing the design of the software system we want to analyze. The `Analysis Filter` here is the module we implemented to input the S-PMIF format in the SPE•ED tool.

In this case, the analysis integration with other techniques requires the definition of `sematic relations` between the EG notation entities and the notation of the other considered analysis.

In the second option, reported in Figure 7.8, it is XML Schema for the UML language that becomes part of the *XML Integration Core* in the *XML Model Representation*. The `Input Filter` component role in the framework is covered by the module of the UML Case tool used to design the software system, that implements the XMI exporting of the UML diagrams. The origin notation is again the UML language. The `Analysis Filter` here is the combination of the XPRIT tool that generates the EG of the software system dynamics we want to analyze, and the module we implemented to input the S-PMIF format in the SPE•ED tool.

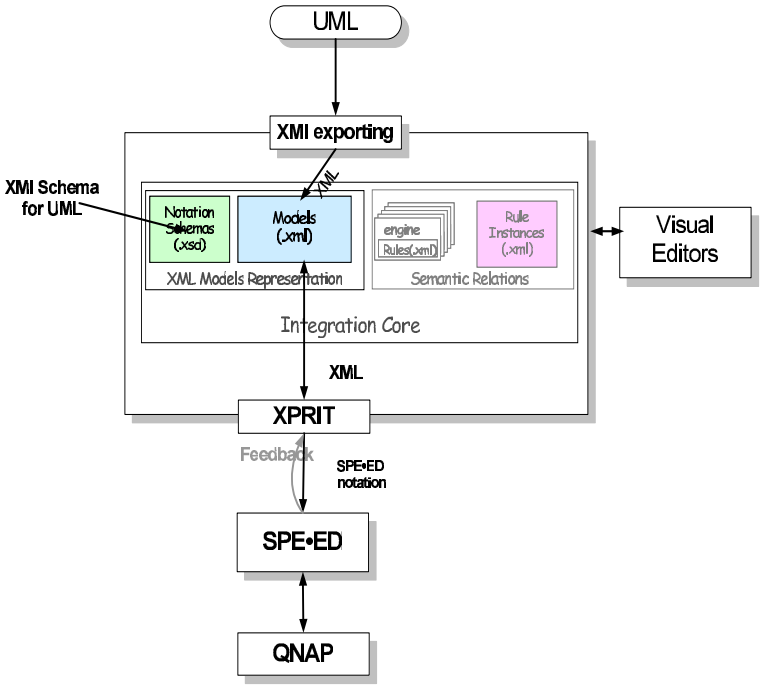


Figure 7.8: Second Mapping between the SPE Approach Components and the Tool Integration Framework.

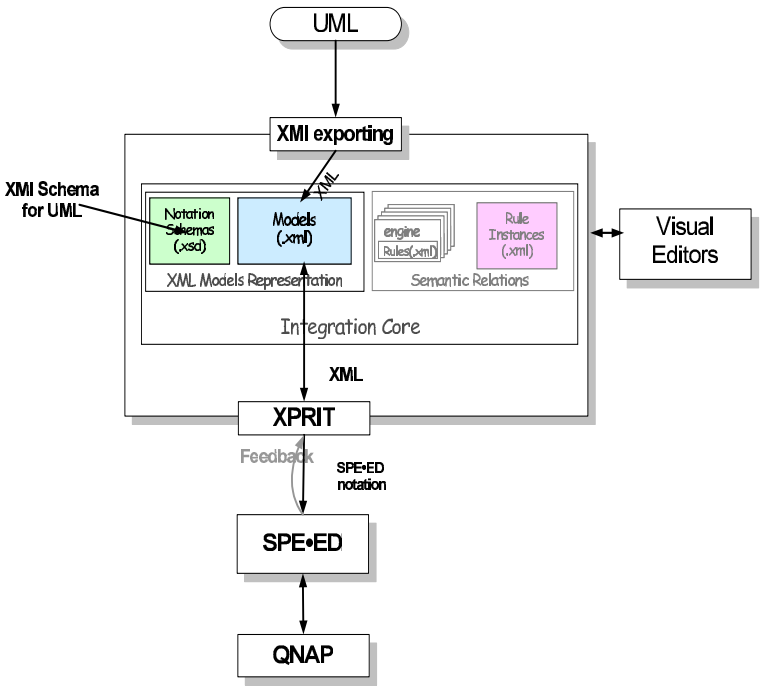


Figure 7.9: Third Mapping between the SPE Approach Components and the Tool Integration Framework.

In this case, the analysis integration with other techniques requires the definition of `sematic relations` between the UML notation entities and the notation of the other considered analysis.

In the last option, reported in Figure 7.9, the XML Schema for the S-PMIF meta-model becomes part of

the *XML Integration Core* in the *XML Model Representation.* The `Input Filter` component role in the framework is covered by a new module we have to implement that exports the S-PMIF format from the Execution Graph drawn by some graphical editor (we can use the one implemented in SPE•ED) . The origin notation is the EG notation describing the dynamics of the software system we want to analyze. The `Analysis Filter` here is the module we implemented to input the S-PMIF format in the SPE•ED tool.

In this case, the analysis integration with other techniques requires the definition of `sematic relations` between the EG notation entities and the notation of the other considered analysis.

7.6 SUMMARY

In this Chapter we defined the S-PMIF XML format in order to automate an SPE approach by integrating two tools: the XPRIT tool and the SPE•ED tool. We also showed how the defined S-PMIF format allows us the integration of the automated approach in the tool integration framework. Actually we discussed three different ways to accomplish this goal where in all the SPE•ED tool cover the role of analysis tool whereas the XPRIT covers the role of an input filter,in the first case, and the role of the analysis filter in the second case. In both of them the origin notation is the UML language. The last case does consider only the SPE•ED tool and the origin notation is the Execution Graph.

Part III

Model-based Performance Analysis in System Dynamic Reconfiguration

CHAPTER 8

DYNAMIC RECONFIGURATION TO MANAGE THE *SIENA* MIDDELWARE PERFORMANCE

Recently, growing attention is focused on run-time management of Quality of Service of complex software systems. In this context, self-adaptation of applications based on run-time monitoring and dynamic reconfiguration is considered a useful technique to manage QoS in complex systems. Many frameworks for dynamic reconfiguration have been recently proposed for this aim. These frameworks rely on monitoring, reconfiguration and on–line model-based analysis to manage/negotiate QoS level of software systems at run time. They share the idea of modifying the application configuration when the threshold of a critical QoS index is crossed. The choice of the new configuration for improving the QoS of the system is based on the current status of the managed software application.

In a previous work [55], we defined a framework able to dynamically reconfigure an application in order to manage the performance of the software system at run-time. The framework monitors the performance of the application and, when some problem occurs, it decides the new application configuration on the basis of feedback provided by the on–line evaluation of performance models of several, pre-defined feasible alternatives. The choice of the new system configuration might consider several factors, such as, for example, security, reliability of the application, and resources needed to implement the new configuration.

In this chapter we report our experience in implementing such an approach on the *SIENA* middleware. We discuss the feasibility of the approach in a real context and point out its power/limits with respect to its generality and reusability in different contexts.

The work this chapter discusses has been outlined in [55, 51] and it is described here in details.

8.1 BACKGROUND

In [130], Porcarelli et al. describe a framework which provides fault tolerance of component-based applications by detecting failures through monitoring, and by recovering through system reconfiguration. The system reconfiguration is decided at run time by using online evaluation of a stochastic dependability model of the system.

In [81], Garlan et al., presented an approach of self-adaptation to manage application performance based on *monitoring - interpretation - reconfiguration*. In this approach the Software Architecture (SA) plays a central role. The SA of the managed system is specified in terms of components and connectors using Acme [80]. The architectural model is enriched with the runtime information provided by the *gauges* and finally it is evaluated by using AcmeStudio. If the current SA violates the specified constraints on the system performance, it is dynamically modified by the framework.

Menascé et al. presented in [118] a framework that dynamically reconfigures an application, called Q-application, in order to provide services with a given QoS level. A Q-application service can be implemented by combining the services provided by registered components, also called QoS-aware components.

Several registered QoS-aware components can provide similar services with different QoS levels. These components have several identical capabilities including QoS negotiation, registration and QoS monitoring. When a Q-application receives a service request with a given QoS-level, it determines whether it can provide it and which registered QoS-aware components are involved. To make this decision, it evaluates the performance models built on the possible configurations parameterized with the monitored data.

8.2 THE RECONFIGURATION PROCESS

In a previous work [55] we have defined a framework that, by using the dynamic reconfiguration, manages the performance of complex systems. That approach is based on monitoring the running system to collect data, dynamic reconfiguration to change the running configuration whenever some performance problems occur, and model-based performance analysis to decide the next system configuration among the ones available.

A first performance model represents the software system behavior at a Software Architecture (SA) level. The other models are generated on–the–fly through the application of suitable reconfiguration policies, defined for the managed software system. Allowed reconfigurations may change only internal parameters of software components (such as the number of threads or other features preventively defined by the component developer), or the application topology by adding/removing component and/or connector instances. However, a reconfiguration must not change the application functionalities (e.g. the substitution of a component with a new one providing different services). A change in the application behavior would in fact imply a re-design of the performance model, and not just its topology or some parameters, and consequently the process could not be automatically carried out.

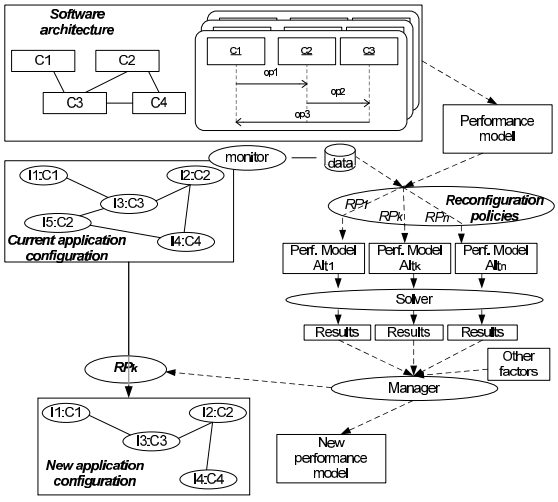


Figure 8.1: The Reconfiguration Process.

Figure 8.1 shows the reconfiguration process. The framework monitors performance attributes of the software application. Whenever the established performance constraints are no longer satisfied, a reconfiguration process will start. Given the set of reconfiguration policies RP_i defined for the application, a set of reconfiguration alternatives is generated. These alternatives are evaluated by a performance model, created on–the–fly on the basis of the performance model which reflects the current running configuration. Each model is initialized with the monitored data, and then evaluated by using a solver tool. Finally, the evaluation results are compared, and the most rewarding configuration is selected and applied to the system.

The online characteristics of the approach require a *reasonable* time of execution. This implies that only models allowing analytical/numerical solution should be considered, whereas simulative models are not

admissible, since the accuracy of the analysis depends on the simulation running time. Moreover, in order to control the state space explosion of the analytical/numerical solution, the model of the system should be as simple as possible, omitting useless details about components behavior. The choice of the Software Architectural abstraction for the application behavior description permits to address such problem. In fact, SA is the minimum detailed description of the application having all the behavioral information needed to carry on a predictive performance analysis, and it allows lightweight and fast models evaluation. Of course, there is a tradeoff between the simplicity of the model and the support of the feedback provided for online decision making.

8.3 *SIENA*

As a concrete example of dynamic application, we have chosen *SIENA*, a distributed, content-based, publish/subscribe event notification service [53, 52]. As depicted in Figure 8.2, the *SIENA* architecture defines

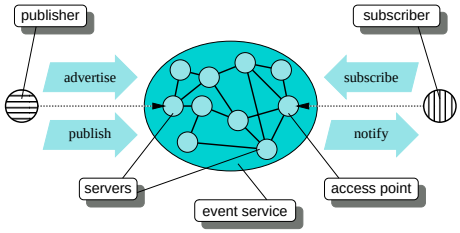


Figure 8.2: *SIENA* Architecture

two main entities: (i) the *clients* and (ii) the *event-service* that is responsible for delivering messages.

Clients may be both *publishers* (i.e. event generators) that publish notifications and, *subscribers* (i.e. event consumers) that express their interest in such kind of events by supplying a *filter*. The event-service, composed of one or more servers interconnected in a hierarchical fashion, forms a store-and-forward network that is responsible for delivering events from publishers to those subscribers that have been submitted a filter matching such notifications.

The performance of a *SIENA* network basically depends on both performance of the single router within the event-service and whole network topology. In this Chapter we are interested in architectural adaptation driven by entities (both clients and routers) performance monitoring.

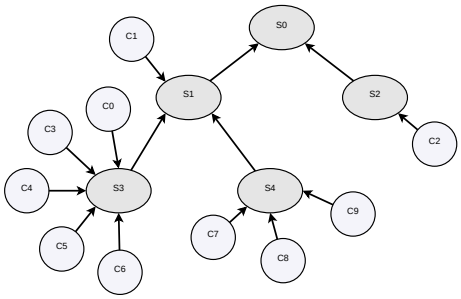


Figure 8.3: A Possible Configuration for the *SIENA* Network

Moreover, we are interested in dynamically modifying the *SIENA* topology depending on several performance attributes, such as *utilization* and *throughput* of *SIENA* Routers.

These attributes depend on the performance of the every SIENA router and on the network topology. The performance of a SIENA router depends on the number of stored filters, as well as on the number of clients connected to it. The SIENA network topology, instead, impacts the internal management of subscription and published messages, and then the global performance of the middleware.

8.4 THE LIRA FRAMEWORK

LIRA [54] (Light-weight Infrastructure for Reconfiguring Applications) is a system that enables the reconfiguration of Component Based Applications in the Large Scale Distributed Systems context. This system has been designed and implemented to manage dynamic and automatic reconfigurations, using a remote interface. LIRA, in fact, dynamically changes the topology of the managed application, by installing new components, adding and removing component instances, or simply changing their connections. In particular, LIRA performs two different kinds of reconfigurations: (i) a *Component Reconfiguration* that permits internal parameters tuning and, (ii) *Application Reconfiguration* that permits to change the overall application by means of architectural properties (i.e. number of components and connections).

As depicted in Figure 8.4, the LIRA Software Architecture specifies three main entities: (i) *Agents* which manage the components and implement both local and global reconfiguration logic, (ii) the agent *Management Information Base* (MIB), that contains the list of variables and functions exported by the agent itself for remotely management and (iii) the *Management Protocol*, that allows the interaction between the agents.

LIRA defines three different kind of agents:

Component Agent: It is strictly coupled with a component and allows for monitoring and reconfiguring the component in terms of internal parameters. The component agent manages the life cycle of the component, being able to start, stop, suspend and resume the managed element.

Host Agent: It is associated with an host in the network and is responsible for installing and activating (deactivating) components on such host.

Manager: It is the agent that manages one or more components by interacting with their agents. Since each Manager is an agent itself, it may be managed by an Higher-level agent. Managers can then communicate with each other by forming a hierarchical agents network. The agent root of the hierarchy, called *Application Manager*, controls all the agents present in the environment and coordinates the reconfigurations following the specified adaptation policies. The hierarchy allows the monitoring and reconfiguration at component, sub-system and application level.

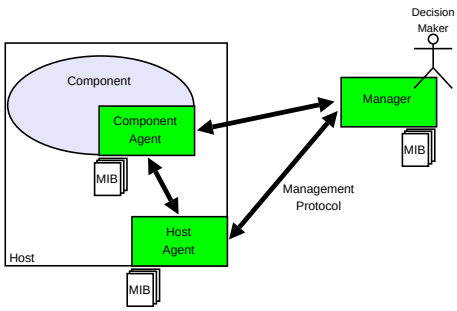


Figure 8.4: LIRA Architecture

LIRA does not require any additional infrastructural support and the management activities are easily programmed by using the *Lira Reconfiguration Language*. Moreover, by using the Lira Reconfiguration Language it is possible to program different reconfiguration activities:

Startup: It is the script that the manager executes at startup. It can be used to instantiate and activate the application components.

Proactivity: It is possible to define a proactive behavior, by means of a set of actions, that the manager continuously executes.

Reactivity: It is possible to define a reactive behavior, by means of a set of actions, that the manager executes as reaction to an external stimulus.

Reconfigurations: Defines the reconfiguration logic by means of a set of *reconfigurations* expressed as functions.

The reconfiguration activities are stored in four different files that the *Application Manager* opportunely reads and interprets when it receives an event. In such a way, it is possible to replace those files online to have a dynamic behavior of LIRA framework.

8.5 RECONFIGURING SIENA

*S*IENA middleware is a network composed by a set of event dispatchers (*S*IENARouters) and clients publishing and subscribing for events (*S*IENA Clients). As introduced in 8.3, a *S*IENA network has a hierarchical structure rooted in a *S*IENA Router, where the leaves are *S*IENA Clients, and all intermediate nodes are *S*IENA Routers.

Due to the event dispatcher rules and the dynamism of a *S*IENA network, it can happen that one or more *S*IENA Routers are overloaded and degrade the performance of the whole network. For example, a *S*IENA Router might be the access point of many Clients or it might be the root of a large *S*IENA sub-network, while other *S*IENA Routers are unloaded (in this case the hierarchical structure is not balanced).

When a *S*IENA Router is overloaded (i.e. its utilization is high) we would reconfigure the *S*IENA network in order to prevent critical performance scenarios. The possible alternatives to reconfigure a *S*IENA network are listed in Section 8.5.1.

Figure 8.5 depicts the Software Architecture of the reconfiguration framework of *S*IENA based on LIRA. Each *S*IENA Router and *S*IENA Client has a LIRA component agent attached to it. Among their tasks, a LIRA component agent has to monitor the associated *S*IENA component. A LIRA Application Agent (called also Manager) is connected to each component agents and it manages the whole *S*IENA network.

When a *S*IENA Router is overloaded, i.e. its utilization is greater than a given value, the associated LIRA agent generates an external stimulus, or reconfiguration event, captured by the LIRA Application Manager and the reconfiguration process will start. The Application Manager executes the reactive actions associated to the event, written in the `reactivity` file. Among the reactive actions, the Application Manager has to collect all monitored data regarding the routers and the clients of the current *S*IENA configuration. It has also to write the current *S*IENA configuration model filled with the collected data in the `Model DB`. Finally, it calls the `reconfigure` service of the `Performance Manager`.

The Performance Manager reads the current configuration model from the DB and, by analyzing the information in it, it generates several reconfiguration alternatives based on the rules described in section 8.5.1. For each alternative it predicts performance indices and, based on them, it decides the next reconfiguration. Then, the Performance Manager writes the reconfiguration function in the `Reconfigurations` file, and, finally, it returns the control to the LIRA Application manager. The latter, by executing the reconfiguration function, reconfigures the *S*IENA network.

During the execution of the reconfiguration process *S*IENA will continue its tasks and the dispatching services provided by it will not interrupt. However to prevent unwanted thrashing between configurations we

disable the reconfiguration process for a fixed period of time (some minutes) after a reconfiguration has taken place. More precisely, we implemented the framework in such a way that, after a reconfiguration event is notified to the Application Manager, the Manager disables all interrupts requiring a reconfiguration. It re-establishes the interrupt mechanism later when reconfiguration has been executed and some time has passed.

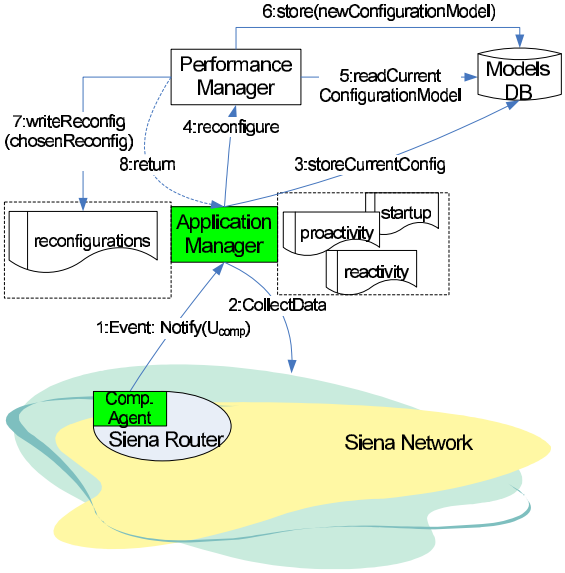


Figure 8.5: Software Architecture of the Reconfiguration Framework Using LIRA.

As said before, the reconfiguration process starts when a LIRA component agent discovers that its associated SIENA Router has the utilization higher than a given value. The choice of such value is critical. If it is too high the reconfiguration process will start late and probably it finishes when a SIENA Router, or worst a SIENA subnet, is saturated. On the other hand, if the chosen value is too low the reconfiguration process will occur too frequently and many efforts and resources will be uselessly spent to reconfigure. Moreover such value should be related to the time needed to reconfigure the software system.

In our experiments, we have chosen two different values for utilization, one for the root of the SIENA hierarchy and one for the other SIENA Routers. The first value is greater than the second one since the root has to manage all publications traffic flowing in the network and it should be the more overloaded SIENA Router. We fixed such values to 0.8 and 0.7, respectively. Our decision is not supported by a deep study and more efforts should be spent to define more suitable values.

8.5.1 SIENA RECONFIGURATION ALTERNATIVES

We identify four different reconfiguration rules for the SIENA middleware.

Moving SIENA Clients - One or more SIENA Clients are moved from the overloaded SIENA Router to the unloaded one(s) . This alternative aims at balancing the workload among the SIENA Routers. Note that, in order to obtain a significant improvement, the receiving SIENA Routers must not belong to the sub-hierarchy of the overloaded Router.

Let us consider as an example the configuration in Figure 8.2 and suppose that the SIENA Router S_3 is overloaded. A reconfiguration alternative here could be moving the SIENA Clients C_3 and C_4 from S_3 to S_2.

Changing *SIENA* Routers internal parameters - The *SIENA* Router implementation allows the modification of the number of its internal threads satisfying service requests by external software entities. In this way, it is possible to add (software) processing capabilities to each *SIENA* Router[1].

Changing *SIENA* Routers topology - One or more *SIENA* Routers are switched from the overloaded *SIENA* Router to the unloaded ones. This alternative aims at balancing the workload among the *SIENA* Routers switching them from one master to another. Again, to reach an improvement, the reconfigured *SIENA* Routers must be attached to a master that does not belong to the sub-hierarchy of the overloaded router.

Let us consider again the configuration in Figure 8.2 and suppose now that the *SIENA* Router S_1 is overloaded. For a greater improvement, the routers receiving a router children of S_1 are S_0, the root of the hierarchy, and S_2. Since the root is the more critical node in the *SIENA* topology, it should not be considered. Hence, it is convenient to switch S_1 childrens to S_2. A reconfiguration alternative here could be moving the *SIENA* Router S_3 from S_1 to S_2.

Adding/Removing *SIENA* Routers - The last possible reconfiguration action is to remove/add *SIENA* Router instances in order to increase/decrease the processing capacity of the *SIENA* network. Of course, we add new router instances if we need more (software) processing capacity, whereas we remove router instances whenever there are too *SIENA* routers with respect to the real necessity.

8.5.2 TRADE-OFF ANALYSIS

Moving *SIENA* Clients - Switching a client from a master to another one, is not provided by the *SIENA* API. Then, it has to be implemented as sequence of basic actions: (i) shutdown the client, (ii) create a new client and connect it to the new master, (iii) re-subscribe all filters to the new master. Of course, since performing such actions requires time, then there is an high probability to lose events. A possible solution to this problem would be to use the MOBIKIT framework [50] in conjunction to *SIENA*.

This reconfiguration alternative is the most critical one for two reasons: it can delay the service accomplishment at the user side and it could imply a more expensive cost for him/her. As a consequence of this reconfiguration policy, in fact, a *SIENA* Client could be moved under a *SIENA* Router physically far from it requiring possibly more time and more expenses to access to the system. For such reasons this alternative should be considered only in critical situations for which the user gave explicit agreement.

Changing *SIENA* Routers internal parameters - The time needed to reconfigure the *SIENA* network is almost constant since the only action to perform is to call the correspondent method of the *SIENA* router.

Changing *SIENA* Routers topology - The current *SIENA* Router implementation provides a method for link modification. However, the reconfiguration action has been implemented in two separated steps: *disconnection* from the current master and *connection* to the new one. Also in this case, the time needed to perform such reconfiguration is not null (and not constant too), but it depends on the number of filters the moved routers has to unsubscribe from the old master and re-subscribe to the new master. The use of MOBIKIT would help in avoiding loss of events.

Adding/Removing *SIENA* Routers - Adding a new *SIENA* Router instance requires a constant execution time. However, this reconfiguration is completed when the established *SIENA* components have been moved to compose the *SIENA* sub-net of the new *SIENA* router. In fact, a new *SIENA*Router is added when additional (software) processing capacity is needed to unload the existent *SIENA* Routers. Hence the time needed to complete such reconfiguration depends on the number of *SIENA* components to be moved. Similar considerations can be done for the Removing *SIENA*Routers reconfiguration action since it requires to move all its connected *SIENA* routers and clients before stopping and shutting down it.

[1] The *SIENA* native implementation sets to five the default number of active threads.

It is worth noticing that the *Moving* SIENA *Clients* and *Changing* SIENA *Routers topology* reconfiguration alternatives are basic steps of the Adding/Removing SIENA Routers ones. Hence their implementation complexity is less than the latter. Whereas Changing the SIENA Routers internal parameters reconfiguration is the simplier one. As first step, we decide to consider only the Moving SIENA Clients and Changing SIENA Routers topology reconfiguration alternatives. First because the last presented alternative is based on them and, whenever they are implemented, it should be easy to also embed the *Adding/Removing* SIENA *Routers* alternative. Second because they do not introduce more processing capacity to the SIENA system. The increasing of (software) processing capacity could require more resources (servers, bandwidth, and so on). In this case we should make a on–line costs-benefits trade-off analysis for such alternatives. To achieve such goals we have to introduce different types of models that take into account also other aspects such as for example available budget.

8.6 PERFORMANCE MODEL AND EVALUATION TECHNIQUE

In this section we report the monitored data and the figure we derived from them. Moreover we discuss the performance model and the evaluation technique used to calculate the performance indices.

For a SIENA Client we can monitor the number of both subscription (SUB) and publication (PUB) requests ($\text{SUB}_{c,m}$, $\text{PUB}_{c,m}$) forwarded by the SIENA Client C to its SIENA Router access point.

Figure 8.6-(a) shows the monitored data for a SIENA Router SR_k. For a SIENA Router SR_k, we are able to monitor the number of SUB and PUB requests served by it ($C_{SUB,k}$, $C_{PUB,k}$), and its average service time needed to process such types of requests ($\tau_{SUB,k}$, $\tau_{PUB,k}$). Moreover, we can monitor the number of SUB and PUB forwarded by SR_k to its master SR_m ($SUB_{k,m}$, $PUB_{k,m}$), the number of PUB that it receives from its SubNet ($PUB_{SN,k}$) and the number of notifications (NOT) that it notifies to its SubNet ($Not_{k,SN}$). Finally, it is possible to measure the number of NOT (among the processed PUB and NOT requests) it receives from its master ($Not_{m,k}$).

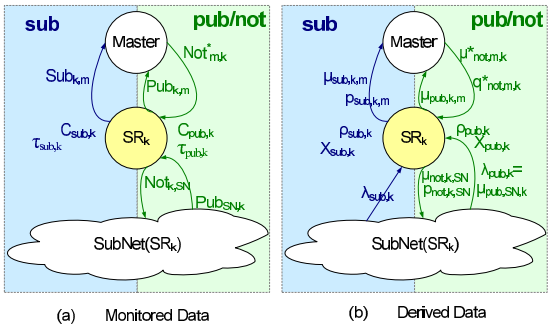


Figure 8.6: Monitored and derived data for a SIENA Router k

From the monitored data, we are able to derive fundamental measures for our approach. The derived data for a SIENA Client C consists of the SUB rate and PUB rate that C forwards to its SIENA Router access point m ($\mu_{SUB,c,m}$, $\mu_{PUB,c,m}$).

Figure 8.6-(b) shows the derived data for a SIENA Router SR_k. For a SIENA Router SR_k we are able to determine the service rate for SUB and PUB requests ($\rho_{SUB,k}$, $\rho_{PUB,k}$), the (total) SUB and PUB arrival rates ($\lambda_{SUB,k}$, $\lambda_{PUB,k}$) and the SR_k throughput for both SUB and PUB ($X_{SUB,k}$, $X_{PUB,k}$). Moreover, we can calculate the SUB and PUB rates that SR_k forwards to its master ($\mu_{SUB,k,m}$, $\mu_{PUB,k,m}$), the NOT rate that SR_k receives from its master ($\mu_{NOT,m,k}$), and the one it forwards to its SubNet ($\mu_{NOT,k,SN}$). Finally, we are able to determine the (observed) probability ($p_{SUB,k,m}$) under which SR_k forwards a SUB

to its master, the (observed) probability that SR_k forwards a notification to its SubNet ($p_{NOT,k,SN}$) and the (observed) probability that the SR_k master forwards a NOT to SR_k ($q_{NOT,m,k}$). For the sake of presentation, the formulae used to derive data have been moved to the appendix B at the end of the thesis.

The performance indices are, instead, calculated by considering the *S*IENA Routers separately. We use the analytical solution techniques for open multi-chain performance models [111]. Due to the particular topology of a *S*IENA configuration, the performance model-based re-configuration consists in re-modulating the incoming traffic to each *S*IENA Router. This is reasonable since we just consider, as possible reconfiguration actions only the *Moving S*IENA *Clients* and *Changing S*IENA *Routers topology* alternatives. Whenever a reconfiguration alternative is generated, the Performance Manager re-modulates the traffic among the *S*IENA Routers in the new *S*IENA topology according to the data monitored at the step before. After that, the performance indices local to each *S*IENA Router are calculated considering the new topology and the related traffic. For details on the formulae used to evaluate the performance indices please refer to the appendix B at the end of the thesis.

Our approach to performance evaluation of *S*IENA Routers in a new configuration, is based on the following assumptions:

1. when we re-modulate the traffic, we assume that the probabilities $p_{SUB,k,m}$, $p_{NOT,k,SN}$ and $q_{NOT,m,k}$ remain constant over the considered reconfiguration alternatives;
2. we assume that the *S*IENA system has sufficient (software) processing capacity to process the given workload. This assumption is necessary to use the open model solution technique considered.

8.6.1 PERFORMANCE INDICES

*S*IENA is a dynamic system where the number of Clients connected to the event service changes frequently with no precise rules, rather it follows the will of the users behind the *S*IENAClients. This dynamism leads us to consider the *S*IENA system as an open system, even if the number of Clients that could connect to the event service is known and finite. Each *S*IENA Router provides two main services (i.e. subscription and the publication services), hence, the *S*IENA middleware is a multi-chain system. Finally, since a *S*IENA Router, as a consequence of a publication, can forward a notification to several *S*IENA Router/Client Siblings, the related performance model contains forks. Hence, in general, *S*IENA leads to an open multi-chain performance model with concurrent behaviors.

For this kind of performance models there exist no analytical solution techniques and, we adopted an approximate solution technique for such model. In fact, in general, they are simulated to obtain performance figures. In our case, due to the strict real-time constraints our approach has, we cannot simulate the model for the reasons discussed in Section 8.2.

However, the reconfiguration process we defined for *S*IENA depends on the performance indices local to each *S*IENA Router and not on performance aspects of the whole system. In fact, for our aims, we mainly need utilization and throughput of each *S*IENA Routers. Hence we can consider a *S*IENA Router separately from the other ones. Moreover, since the *S*IENA Routers and the *S*IENA Clients form a hierarchical topology (i.e. a tree), we are able to approximate the requests arrival rates of each *S*IENA Router in a new configuration, by looking at the new configuration topology and the information derived from the monitoring data on the current running configuration.

The ability of re-modulating the traffic incoming in each *S*IENA Router permits us to consider a *S*IENARouter as an independent service center (or system) and to calculate the performance indices of interest by using the open multi-chain queueing network formulae. Since the *S*IENA Router is a multi-chain system, to obtain performance indices aggregate per service center (i.e. per *S*IENA Router) we have to cal-

culate the ones related to each chain traversing it and then aggregate following rules in [111] as reported in the appendix.

In the following section we report the formulas we used to re-modulate the traffic in order to obtain an approximation of the one flowing in the new SIENA topology defined by a reconfiguration alternative.

8.6.2 TRAFFIC RE-MODULATION

Due to the considered reconfiguration actions, a reconfiguration alternative consists of the definition of a new SubNet possibly for each SR in the SIENA topology, i.e. the sets $SubNet'(SR)$. In the following, the formulae for traffic re-modulation are mainly dependant on such set. Moreover, in defining such formulas, we assume that the previous assumptions are verified.

SUB TRAFFIC RE-MODULATION

- $\lambda'_{SUB,k} = \sum_{i \in SubNet'(SR_k)} \mu'_{SUB,i,k}$ is the new SUB arrival rate for SR_k;
- $\mu'_{SUB,k,m} = X'_{SUB,k} * p_{SUB,k,m}$ where $X'_{SUB,k} = \lambda'_{SUB,k}$ is the new SUB rate that SR_k forwards to its master.

PUB TRAFFIC RE-MODULATION

- $\lambda'_{PUB,k} = \sum_{i \in SubNet'(SR_k)} \mu_{PUB,i,k}$ is the new PUB arrival rate for SR_k;
- $\mu'_{PUB,k,m} = X'_{PUB,k} = \lambda'_{PUB,k}$ is the new PUB rate that SR_k forwards to its master.

NOT TRAFFIC RE-MODULATION

- $\mu'_{NOT,k,SN} = p_{NOT,k,SN} * (\mu'_{NOT,m,k} + \lambda'_{PUB,k})$ is the new NOT rate that SR_k forwards to its SubNet;
- $\mu'_{NOT,m,k} = q_{NOT,m,k} * \mu'_{NOT,m,SN}$ is the new NOT rate that SR_k forwards to its master.

8.7 FRAMEWORK IMPLEMENTATION

In this section we detail the design choices we have made and the overall system implementation.

Figure 8.7 depicts the high-level Software Architecture of the system. Different entities may be distinguished: (i) SIENA router and client with the associated LIRA ComponentAgent; (ii) the LIRA ApplicationManager, (iii) the Performance Manager.

The LIRA ComponentAgents play the role of monitors by profiling the SIENA entities and, in case of anomalies, throwing the alarm to the ApplicationManager. When the ApplicationManager receives the alarm message, the LIRA reactivity script is executed. For each SIENA routers and clients the ApplicationManager retrieves all monitored data and forwards them to the *ConfigMonitor*. Once the ConfigMonitor

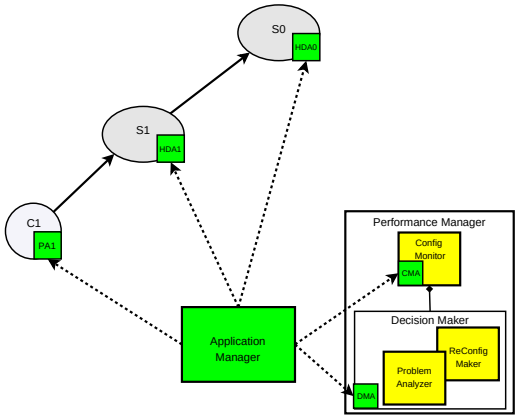


Figure 8.7: Software Architecture

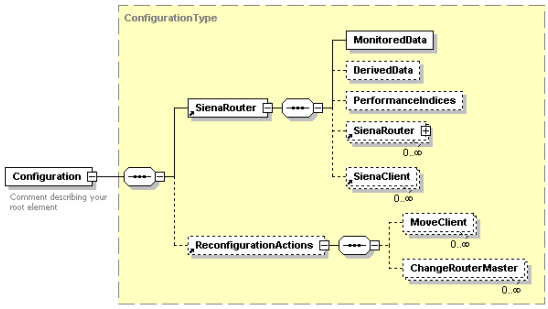


Figure 8.8: The Configuration Schema.

has stored all data sent by the ApplicationManager, it creates a XML file representing the actual system configuration (such file obeys the XSchema depicted in Figure 8.8). At this point the *ProblemAnalyzer* examines this configuration, calculates the derived data and then produces several alternative configurations. The *ReConfigMaker* chooses (driven by heuristics) the best possible reconfiguration and writes the LIRA reconfiguration script. Finally, the ApplicationManager executes this script in order to actuate the produced reconfiguration.

8.7.1 MONITORING

AspectJ [105] is a general-purpose Java extension of the Aspect Oriented Programming paradigm introduced by Kiczales et al [106].

AspectJ extends Java by adding few new constructs: *pointcuts* and *advice* that dynamically affect program flow, *inter-type declarations* that statically affect a program's class hierarchy, and *aspects* that encapsulate these new constructs.

In particular, while a pointcut picks out certain join points (that is a well-defined point in the program flow), and values at those points, a piece of advice is code that is executed when a join point is reached. AspectJ inter-type declarations allow to modify the static structure of a program, namely, the members of its classes and the relationship between classes. Finally, aspects are the units of modularity for crosscutting concerns. They behave somewhat like Java classes, but may also include pointcuts, advice and inter-type declarations. All of these are then automatically combined by using a weaver tool. In particular, given the program source

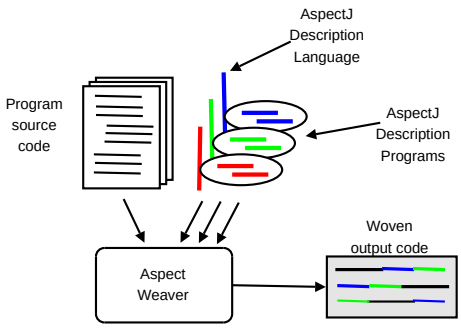


Figure 8.9: The AspectJ Weaving Process

```
public aspect DispatcherMonitor{
 //Attributes Definitions
 int HierarchicalDispatcher.processedPubNum;
 int HierarchicalDispatcher.forwardedPubNum;
 int HierarchicalDispatcher.fromMasterPubNum;
 int HierarchicalDispatcher.fromClientsPubNum;
 ...

 //Pointcuts Definitions
 pointcut processedPUB():
 execution(* HierarchicalDispatcher.publish(SENPPacket));

 pointcut PubMasterForward():
 withincode(* HierarchicalDispatcher.publish(SENPPacket))
 && call(* Interface.send(byte[], int));
 ...

 int HierarchicalDispatcher.getProcessedPubNum(){
   return processedPubNum;
 }
}
```

Figure 8.10: DispatcerMonitor Aspect Source Code

code and the aspects definition as input, the weaving process (showed in Figure 8.9) produces a woven Java output code that will then compiled using traditional Java compiler.

The fact that the application and the aspects are designed and developed separately, makes AspectJ well-suited for defining behavior that ranges from simple tracing, to profiling, as well as to testing of internal consistency within the application. In fact, AspectJ makes it possible to cleanly modularize these kind of functionalities, thereby enabling and disabling them when desired.

An important task in our case study is represented by the monitoring process. In fact, in order to profile the SIENA entities behavior we need to extract information about their internal status. Of course, this requires both to access internal data structures and to watch the execution flow by means of called methods and object interactions.

The use of aspects has easily allowed the servers and clients profiling by avoiding changes/modifications to their source code. In particular, through the use of AspectJ we have been able to add extra attributes and methods to the existing HierarchicalDispatcher class implemented in SIENA. Figure 8.10 reports a piece of the AspectJ DispatcherMonitor source code which shows some monitored attributes, methods declaration and, *pointcuts* of interest. For example, the **processedPubNum** attribute (which counts the

number of processed publications) is incremented every time the **ProcessedPUB** pointcut is reached and can be retrieved at runtime by invoking the added method **getProcessedPubNum()**.

The monitoring operations performed by the aspects are direct accesses to variables with constant computational time $O(1)$. Thus the monitoring overhead on the SIENA HierarchicalDispatcher performance can be considered negligible. On the other hand, retrieving data from the HierarchicalDispatcher during the reconfiguration phase, costs time $O(n_p + n_s)$ where n_p and n_s are the number of processed publications and subscriptions, respectively. In particular, this is the computational complexity necessary for calculating the average processing time of both publications and subscriptions.

8.8 EXPERIMENTS

In this section we describe our experimental setup by detailing both the architectural and implementation choices.

The initial system configuration (showed in Figure 8.3) presents an event-service composed of five routers and ten clients (five subscribers and five publisher). Each router's monitor has been configured to give the alarm when the critical value U_k is greater that 0.7. Since the SIENA root-router is the most overloaded (S_0 referring to Figure 8.3), its critical value U_0 is set up to 0.8. Subscribers (C_0, C_2, C_4, C_6, C_8) submit 100 subscriptions, one each $40ms$. Publishers (C_1, C_3, C_5, C_7, C_9) emit 1000000 events, one each $40ms$. All the experiments have been executed on a Personal Computer with 512 MB of RAM, $1.6GHz$ of CPU, running a Linux Fedora Core III Operating System.

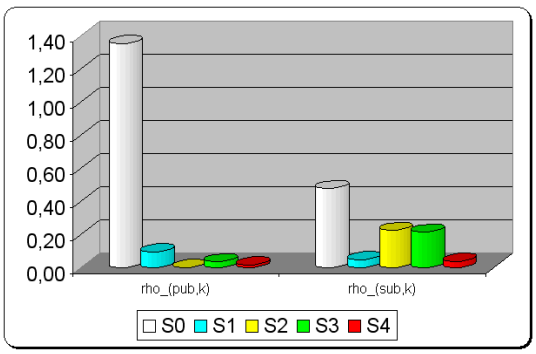


Figure 8.11: Service Rate

Running the test described above, the first alarm has been raised by S_4 after $29849ms$. As described in Section 8.7, the ConfigManager stores all monitored data for what concerns both the clients, and the routers (refer to Figure 8.11 and Figure 8.12). Once all monitored data has been retrieved, the PerformanceManager derives the data concerning the service rate ($\rho_{PUB,k}$ and $\rho_{SUB,k}$), reported in Figure 8.11, and the request arrival rates ($\lambda_{PUB,k}$ and $\lambda_{SUB,k}$) shown in Figure 8.12. In Figure 8.13 we report the utilization of all SIENA routers and we observe that S_3 and S_4 are critical, then the PerformanceManager proposes several alternatives. The one reported in Figure 8.14 is chosen, and the corresponding LIRA reconfiguration script shown in Figure 8.15 is executed reconfiguring the SIENA network as depicted in Figure 8.16. Finally, in Figure 8.17 we report the predicted SIENA routers utilization for the new configuration.

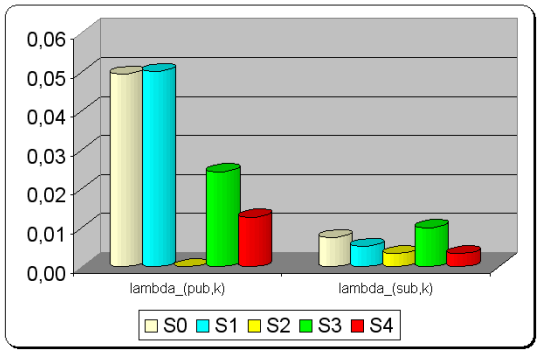


Figure 8.12: Total Arrival rates of SRs

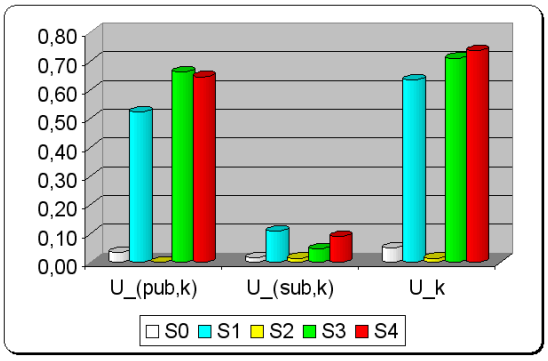


Figure 8.13: Utilization of Siena Routers

```
<ReconfigurationActions>
  <MoveClient SienaClient="C9" From="S4" To="S2"/>
  <MoveClient SienaClient="C3" From="S3"  To="S2"/>
</ReconfigurationActions>
```

Figure 8.14: Reconfiguration Actions

```
reconfiguration globalSystem begin
  println("Reconfiguring Clients");
  connect "C9" to "S2";
  call("CMA", MOVECLIENT, "C9", "S2");
  connect "C3" to "S2";
  call("CMA", MOVECLIENT, "C3", "S2");
end
```

Figure 8.15: LIRA Reconfiguration Script

8.9 FINAL CONSIDERATIONS AND FUTURE WORKS

The implementation we report shows the applicability of the reconfiguration process summarized in Section 8.2 to the SIENA middleware. We were able in fact to reconfigure SIENA network with no human intervention and with no any SIENA service interruption. Moreover, the experiment results reported in Section 8.8 proves the improvement reachable with the framework.

However, some limitations of the approach have been singled out. First of all, the ProblemAnalyzer compo-

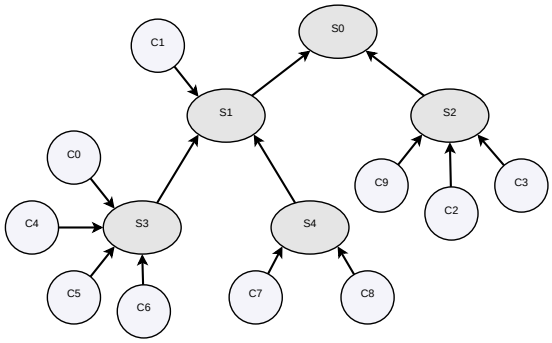


Figure 8.16: The reconfigured SIENA Network

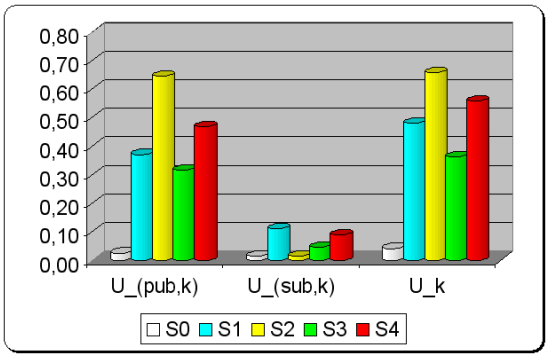


Figure 8.17: Predict Utilization of Siena Routers after Reconfiguration

nent resulted very coupled with the managed system. This means that an ad-hoc implementation should be provided for a given software application to be managed. Moreover, the approach does not cope with reconfiguration that changes application behavior, but it just deals with reconfigurations that modify the topology or internal application parameters. This limitation is strictly related to the necessity of reconfiguring the performance model at run-time. The approach requires an initial effort in designing the performance model from the Software Architecture description of the application. In this modeling step the modeler should translate the software behavioral description into the performance model. Whenever a reconfiguration is required, the suitable reconfigurations must not change this behavior. The execution of a (behaviorally) different application would require a performance model which is not provided.

The current implementation does not deal with all possible reconfiguration alternatives identified in Section 8.5.1, but our aim is to deal with all such cases in our future works. The aim of this first study was to prove the feasibility of the designed reconfiguration process. Other future research goals are to more extensively validate the advantage of the use of such framework by using longer (in terms of execution runs) experiments and on more realistic workloads, and to improve the process decision step. At the moment the next reconfiguration is chosen just on the basis of the predicted performance indices. However many other factors might affect this choice. First of all, the complexity of the reconfiguration and the time needed to implement it. Moreover, the application manager could provide other QoS indices such as security, service availability, dependability. A more complex decision step that takes into account such other constraint should therefore be designed.

We are also planning to measure the overhead introduced by the LIRA framework.

8.10 SUMMARY

In this Chapter we presented our experience in designing and implementing the framework to dynamically manage performance attributes of the *SIENA* publish/subscribe middleware. This has been done in three steps: (i) application monitoring, (ii) model-based performance evaluation, (iii) dynamic reconfiguration.

The monitoring facilities, implemented by using the AspecJ programming language, are needed to extract information about the internal status of the components involved in the system. The use of aspects allowed us to extend the *SIENA* API with profiling functionalities without modifying the *SIENA* source code.

Model-based performance evaluation represents the core of the presented work. Even if the overall process is general, it has been implemented specifically to cope with the characteristics of *SIENA*. In particular, we implemented three main components: a *ConfigManager* that keeps trace of the system configuration; a *ProblemAnalizer* that evaluates the system performances and generates alternative configurations aiming to increase the system performance; and a *ReconfigMaker* that chooses the most rewarding configuration and reconfigures the entire system through the use of the LIRA framework.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

In this thesis we dealt with the model-based performance analysis of software architectures. In particular we focussed on component-based systems modelled at the architecture level for performance validation goals. We used this modelling at two different phases of the software life cycle: design level and run time. In the former we support the designers to take decisions at the software architecture level, whereas in the latter we help to dynamically manage the performance attributes after the system deployment.

To accomplish the performance analysis of software systems, several model notations exist: Queueing Networks, Generalized Stochastic Petri Nets, Stochastic Process Algebras and simulation models. From the software designer perspective, there can be a relevant difference between the above alternatives due to the knowledge gap between the performance analysis and the software development process that we identified in Chapter 2.3. This gap should be fulfilled to reduce the reluctance of the designer to validate the performance along the software life cycle. We made a first study in this direction where the main contribution was the identification of the criteria useful to characterize the performance model notations with respect to the skills of the software designers. The aim of our experiment was to look at these model notations in order to assess their suitability to support software designers.

The thesis is composed by three parts: the first one deals with the fulfillment of the knowledge gap at the software architecture level through the automated performance model generation; the second part defines a framework whose aim is to integrate several predictive analysis of software architectures; the last part, that copes with the last studies of my Ph.D. Program, introduces a reconfiguration process allowing the dynamic performance management of software systems at run time and shows an experiment we did on SIENA middleware.

PART I - PREDICTIVE PERFORMANCE ANALYSIS: FROM SOFTWARE MODELS TO PERFORMANCE MODELS.

In Chapter 3 we have reviewed the state of the art in model-based software performance prediction. Existing approaches propose the use of performance models to characterize the quantitative behavior of software systems. These approaches aim at filling the gap between the software development process and the performance analysis by generating, from the software models, performance models ready to be validated. The review we carried out analyzed the approaches with respect to a set of relevant dimensions from a software-designer perspective.

The resulting classification clearly showed that almost all methodologies try to encompass the whole software life cycle starting from early software artifacts. Most of them are tightly coupled with tool support that allows the (partial) automation of them. However, there is no methodology which is fully supported by automated tools, although at the same time there is no methodology that does not provide or foresee some kind of automatic support. Most approaches try to apply performance analysis very early, typically at the software architecture level. However, most of them still require information from the implementation/execution scenarios in order to carry out performance analysis. Nevertheless there is a growing number of

attempts that try to relax implementation/execution constraints in order to make the analysis applicable at abstract design levels.

For software design specifications we believe that, at least at the next few years, the trend will be to use standard software artifacts, like UML diagrams. On the other hand, Queuing Networks and their extensions are widely used as performance models. QN provide an abstract notation allowing the modeling of software component as black-box entities and easier feedback, especially in a component-based software development process. As far as the analysis process is concerned, feedback provision is a key success factor for a widespread use of these methodologies.

In Chapter 4 we have summarized our methodology to software architecture performance analysis. Our approach generates a QN performance model from a software architecture description based on UML 2.0 diagrams. It derives from a re-engineering process of our previous approach that generates a QN model from MSC describing the dynamics of the software system at the software architecture level.

The new approach uses UML 2.0 to describe the software system architecture. The migration has been driven by two factors: a large expressiveness of UML 2.0 diagrams that allows us the definition of a compositional approach, and the introduction of an UML profile that defines suitable capabilities to annotate the diagrams with additional performance information and with analysis results obtained upon the target model solution.

In this approach we defined two types of translation rules: (i) *basic rules* that deals with the simple interaction fragments (that are component interactions) and the (ii) *rules for fragment operator* that defines QN patterns for each considered fragment operator in the UML 2.0 sequence diagrams.

Even if the new approach allows the generation of a more detailed QN model, the obtained model is however more complex to be complex. This complexity implied that we had to use more complex techniques to evaluate the new QN model. For, example, when we have a parallel operator in a sequence diagram, a fork is generated in the QN model. A QN with forks (namely Extended QN) can be evaluated only through simulation techniques or, in some cases, by means of approximate solutions.

Chapter 5 reports our experience in applying two predictive performance analyses on a real case study. The considered approaches were based on stochastic process algebras and on simulation models. Both of them take as input the description of the software architecture of the system based on the UML notation. We made this experiment to study if these methodologies are suitable to be applied to real systems.

As expected the two methodologies have shown pros and cons. In order to compare the approaches we defined a framework that devises the main criteria a predictive analysis should have. This experience highlighted that the most important features a methodology should have are the automation, the transparency and the feedback provision mechanism.

Moreover, in this work we have experimented the feasibility of a combined usage of approaches at an affordable cost. A key element toward a combined use of the two approaches is the use of standard software artifacts as system initial documents.

PART II - INTEGRATION OF PREDICTIVE FUNCTIONAL AND NON-FUNCTIONAL ANALYSES.

This part of the thesis introduced a framework to support the integration of functional and non-functional analysis of software systems at architectural level. This work originates from the crucial need of merging results from different software analysis approaches in order to better refine software architectures.

Our framework lays on an XML-based integration core, where software models and semantic relations between the models are represented. The aim is to provide a seamless integration of different analysis methodologies. To this regard we have sketched guidelines to allow embedding new methodologies in our framework.

In this direction, in Chapter 7 we showed how to deploy an automated Software Performance Engineering (SPE) approach into the framework. Indeed, we made a first step in this direction by defining the XML schema that describes the Execution Graph notation. Such notation is used to describe the software dynamics enriched by additional information that specifies the workload. The XML schema respects the S-PMIF (Software - Performance Model Interchange Format) meta-model (see Chapter 7).

The defined schema can be part of the XML Integration Core of the framework to allow the integrated analysis. The next step will be to decide which software architecture analyses we want to integrate to, and define the relative semantic rules.

The work shown in this chapter, also allowed us to integrate two implemented tools, XPRIT and SPE•ED in order to build a fully automated SPE process.

PART III - MODEL-BASED PERFORMANCE ANALYSIS IN SYSTEM DYNAMIC RECONFIGURATION.

In this part of the thesis we defined a framework able to dynamically reconfigure a component-based software system in order to manage its performance at run-time. The framework monitors the performance of the application and, when some problem occurs, decides the new application configuration on the basis of feedback provided by the on–line evaluation of performance models of several, pre-defined, feasible alternatives. The choice of the new system configuration might consider several factors, such as resources needed to implement the new configuration.

In this chapter we also presented our experience in designing and implementing the framework to dynamically manage performance attributes of the *SIENA* publish/subscribe middleware.

The monitoring facilities are implemented by using the AspecJ programming language. The use of aspects allowed us to extend the *SIENA* API with profiling functionalities without modifying the *SIENA* source code.

We implemented three main components: a *ConfigManager* that keeps trace of the system configuration; a *ProblemAnalizer* that evaluates the system performance and generates alternative configurations aiming at improving the system performance; a *ReconfigMaker* that chooses the most rewarding configuration and reconfigures the entire system through the use of the LIRA framework.

This implementation showed the applicability of the reconfiguration process with no human intervention and with no *SIENA* service interruption. Moreover, the experimental results reported showed the sensible improvement reachable with the framework. However, some limitations of the approach have been singled out. The *ProblemAnalyzer* component resulted very coupled with the managed system. This means that an ad-hoc implementation should be provided for a given software application to be managed. Moreover, the approach does not cope with reconfiguration that changes application behavior, but it just deals with reconfigurations that modify the topology or internal application parameters. This limitation is strictly related to the necessity of reconfiguring the performance model at run-time. The approach requires an initial effort in designing the performance model from the Software Architecture description of the application. In this modeling step the modeler should translate the software behavioral description into the performance model. Whenever a reconfiguration is required, the pre-defined reconfigurations must not have different behaviors, because a (behaviorally) different application would require a different performance model to be evaluated.

9.1 FUTURE WORK

Due to the wide range of topics the thesis deals with, many directions can be identified about future work. However we believe that the most relevant ones are:

Performance Notations Study. The study we reported in Chapter 2.3 is in its preliminary phase. From the reported results we cannot induce general assessments on the suitability of the considered notations from a software design perspective, due to the limitations of the case study and the experimental setting. Indeed, significant experiments in this direction have still be made. Moreover, early in the life cycle, the choice of the performance model notation is still open and this study can help to identify guidelines in such process.

Software Architecture Early Validation. In the future we should define a mechanism that generate more meaningful feedback at the software architecture level. By meaningful feedback we mean software architecture alternatives that suggest to the software designer how to overcome the performance problems the analysis pointed out. Moreover, since the automation is a key point for the application of the approach to real case studies, our short time goal is to implement a tool prototype of it. This would allow us to apply the methodology to real large scale case studies. Finally, we want to extend this approach to specific application domain, such as mobile and ubiquitous computing.

Functional and Non-Functional Analysis Integration Framework. Our main research direction obviously leads to embed new methodologies for analysis at architectural level in our framework. This task shall bring to enlarge the integration core in terms of software model representations as well as semantic relations. If future results in this direction will appear as promising as the ones obtained from this first setting, a long-term goal will be to extend the scope of the framework to other software life cycle phases.

Moreover, the work in Chapter 7 was an initial step in the overall SPE interchange process. Several additional steps are planned: (i) update XPRIT to export the new constructs in UML 2.0; (ii) export resource requirements specified using the UML Profile for Schedulability, Performance and Time; (iii) define a meta-model and schema for the feedback path, in order to support the transformation of ”abstract” performance results into ”actual” design alternatives for UML or other CASE tools; (iv) define a meta-model and schema for the exchange of performance results from system performance modeling tools back to software performance engineering tools.

Dynamic Reconfiguration Framework. The current implementation does not deal with all possible reconfiguration alternatives identified, but our aim is to deal with a wider set of choices. The aim of this first study was to prove the feasibility of the designed reconfiguration process. Other future research goals are to more extensively validate the advantage of the use of such framework by using longer (in terms of execution runs) experiments on more realistic workloads, and to improve the process decision step. At the moment the next reconfiguration is chosen only on the basis of the predicted performance indices, although many other factors might affect this choice, such as the complexity of the reconfiguration and the time needed to implement it. Moreover, the application manager could provide other QoS indices such as security, service availability, dependability. A more complex decision step that takes into account such other constraints shall therefore be designed.

We are also planning to measure the overhead introduced by the LIRA framework on the application execution.

APPENDIX A

ÆMILIA TEXTUAL DESCRIPTION FOR THE MAXIMAL CONFIGURATION

ARCHI_TYPE *Scenario_Type*(**void**; **rate** $a_1 := 1$,
rate $a_2 := 2$, **rate** $a_3 := 100$,
rate $a_4 := 0.5$

ARCHI_ELEM_TYPES

ELEM_TYPE *COORDINATOR_Type*(**void**; **rate** a_2)

BEHAVIOR

COORD (**void**; **void**)=
<*DisplayFailureOccurred, inf*>.<*StartRecovery, inf*> *COORD'*(),
COORD' (**void**; **void**)=
<*RecoveryCompleted*,*>
<*DisplayRecoveryCompleted*, a_2 >.*COORD*();

INPUT_INTERACTIONS

AND *RecoveryCompleted*

OUTPUT_INTERACTIONS

AND *StartRecovery*

ELEM_TYPE *P1_Type*(**void**; **rate** a_1, **rate** a_3)

BEHAVIOR

P1(**void**; **void**)=
<*StartRecovery*, *>.<*PrepareSetParameter*, a_1>.
<*SendSetParameter*, a_3>.*P1'*();
P1'(**void**; **void**)=
<*TrapSetParameter*,*>.<*RecoveryCompleted*, inf>.*P1*()

INPUT_INTERACTIONS

UNI *StartRecovery*
AND*TrapSetParameter*

OUTPUT_INTERACTIONS

UNI *SendSetParameter*
AND *RecoveryCompleted*

ELEM_TYPE *P2_Type*(**void**; **rate** a_1,**rate** a_3

BEHAVIOR

P2(**void**;**void**)=

<*StartRecovery*, *>.<*PrepareDeletepp*, a_1>. <*SendDeletepp*, a_3>.*P2'*();

P2'(**void**;**void**)=

<*TrapDpp*, *>.<*PrepareCreatepp*, a_1>. <*SendCreatepp*, a_3>.*P2"*();

P2"(**void**;**void**)=

<*TrapCpp*, *>.<*RecoveryCompleted*, inf>.*P2*()

INPUT_INTERACTIONS

UNI *StartRecovery*

AND *TrapDpp*;*TrapCpp*

OUTPUT_INTERACTIONS

UNI *RecoveryCompleted*

AND *SendDeletepp*;*SendCreatepp*

ELEM_TYPE *CTSPROXYAGENT_Type*(**void**; **rate** a_1, **rate** a_2, **rate** a_3

BEHAVIOR

CTSPA(**voidvoid**)=

<*SendDeletepp*, *>.<*Deletepp*, a_1>. <*TrapDpp*, a_3>.*CTSPA'*();

CTSPA'(**voidvoid**)=

<*SendCreatepp*, *>.<*Createpp*, a_2>. <*TrapCpp*, a_3>.*CTSPA*()

INPUT_INTERACTIONS

UNI *SendDeletepp*;*SendCreatepp*

OUTPUT_INTERACTIONS

UNI *TrapDpp*;*TrapCpp*

ELEM_TYPE *PROXY_Type*(**void**;**rate** a_3

BEHAVIOR

PROXY(**void**;**void**)=

<*SendSetParameter*, *>.<*SendSetParameter*, a_3>.*PROXY'*();

PROXY'(**void**;**void**)=
<*TrapSetParameter*, *>.<*TrapSetParameter*, a_3>.*PROXY*()
INPUT_INTERACTIONS
UNI *SendSetParameter*
AND *TrapSetParameter*
OUTPUT_INTERACTIONS
UNI *TrapSetParameter*
AND *SendSetParameter*
ELEM_TYPE *EQUIP_Type*(**void**;**rate** a_2,**rate** a_3)
BEHAVIOR
EQUIP(**void**;**void**)=
<*SendSetParameter*, *>.<*SetParameter*, a_2>.
<*TrapSetParameter*, a_3>.*EQUIP*();
INPUT_INTERACTIONS
UNI *SendSetParameter*
OUTPUT_INTERACTIONS
UNI *TrapSetParameter*
ARCHI_TOPOLOGY
ARCHI_ELEM_ISTANCES
C:COORDINATOR_Type(;a_2); *P1:P1_Type*(;a_1,a_3); *P2:P2_Type*(;a_1,a_3);
CTSPA:CTSPROXYAGENT_Type(;a_1,a_2,a_3);
PR1:PROXY_Type(;a_3); · · ·; *PR10:PROXY_Type*(;a_3);
EQUIP1:EQUIP_Type(;a_2,a_3); · · ·; *EQUIP20:EQUIP_Type*(;a_2,a_3)
ARCHI_ATTACHMENTS
FROM *C.StartRecovery* : **TO** *P1.StartRecovery*;
FROM *C.StartRecovery* : **TO** *P2.StartRecovery*;
FROM *P1.RecoveryCompleted* : **TO** *C.RecoveryCompleted*;
FROM *P2.RecoveryCompleted* : **TO** *C.RecoveryCompleted*;
FROM *P2.SendDeletepp* : **TO** *CTSPA.SendDeletepp*;
FROM *P2.SendCreatepp* : **TO** *CTSPA.SendCreatepp*;

FROM *CTSPA.TrapDpp* : **TO** *P2.TrapDpp*;
FROM *CTSPA.TrapCpp* : **TO** *P2.TrapCpp*;
FROM *P1.SendSetParameter* : **TO** *PR1.SendSetParameter*;
FROM *PR1.TrapSetParameter* : **TO** *P1.TrapSetParameter*;
FROM *PR1.SendSetParameter* : **TO** *EQUIP1.SendSetParameter*;
FROM *PR1.SendSetParameter* : **TO** *EQUIP2.SendSetParameter*;
FROM *EQUIP1.TrapSetParameter* : **TO** *PR1.TrapSetParameter*;
FROM *EQUIP2.TrapSetParameter* : **TO** *PR1.TrapSetParameter*;
. . . .
FROM *P1.SendSetParameter* : **TO** *PR10.SendSetParameter*;
FROM *PR10.TrapSetParameter* : **TO** *P1.TrapSetParameter*;
FROM *PR10.SendSetParameter* : **TO** *EQUIP19.SendSetParameter*;
FROM *PR10.SendSetParameter* : **TO** *EQUIP20.SendSetParameter*;
FROM *EQUIP19.TrapSetParameter* : **TO** *PR10.TrapSetParameter*;
FROM *EQUIP20.TrapSetParameter* : **TO** *PR10.TrapSetParameter*;
END

APPENDIX B

FORMULAE USED IN RECONFIGURING *SIENA*

Along this section $SubNet(SR_k)$ represents the set of *SIENA* Clients and the *SIENA* Routers for which SR_k is the access point or the master in the hierarchy,respectively.

B.1 MONITORED DATA

For a *SIENA* Client *c*we can monitor:

- $Sub_{c,m}$= number of sub requests forwarded by the *SIENA* Client *c* to its *SIENA* Router access point *m* in the observational interval of T ms;
- $Pub_{c,m}$= number of pub requests forwarded by the *SIENA* Client *ct* to its *SIENA* Router access point *m* in the observational interval of T ms;

For what concern the *SIENA* Routers (SR), through monitoring we are able to determine:

- $C_{c,k}$= number of $c \in \{sub, pub\}$ requests served by SR_k in the observational interval of T ms;
- $\tau_{c,k}$= average service time needed by SR_k to process a c request, $c \in \{sub, pub/not\}$;
- $Sub_{k,m}$= number of sub requests forwarded by SR_k to its master;
- $Pub_{SN,k}$=number of pub requests that SR_k receives by its SubNet during the interval of observation time;
- $Pub_{k,m}$=number of pub requests forwarded by SR_k to its master;
- $Not_{m,k}$= number of not request (among the processed pub/not requests) that SR_k receives from its master;
- $Not_{k,SN}$= number of notifications (not) that SR_k notifies to its SubNet (SN);

B.2 DERIVED MEASURES

The derived data for a *SIENA* Client *c* consist of:

- $\mu_{sub,c,m} = Pub_{c,m}/T$ is the sub rate that *c* forwards to its *SIENA* Router access point;

- $\mu_{pub,c,m} = Pub_{c,m}/T$ is the pub rate that *c* forwards to its *SIENA* Router access point;

The derived data for a *SIENA* Router SR_k consist of:

- $\rho_{c,k} = (1/\tau_{c,k})$ is the service rate of SR_k for $c \in \{sub, pub\}$ requests;
- $X_{c,k} = C_{c,k}/T$ is the throughput of SR_k relative to the c $\in \{sub, pub\}$ request;
- $\lambda_{c,k} = \sum_{i \in SubNet(SR_k)} \mu_{c,i,k})$ is the (total) arrival rate for SR_k of $c \in \{pub, sub\}$ requests .
- $\mu_{sub,k,m} = Sub_{k,m}/T$ is the sub rate that SR_k forwards to its master;
- $p_{sub,k,m} = \mu_{sub,k,m}/\lambda_{sub,k}$ is the observed probability that SR_k forwards a sub to its master, of course such probability is zero if SR_k is the root of the *SIENA* Hierarchy;
- $\mu_{pub,k,m} = Pub_{k,m}/T$ is the pub rate that SR_k forwards to its master;
- $\mu_{not,m,k} = Not_{m,k}/T$ is the not rate that SR_k receives from its master;;
- $\mu_{not,k,SN} = Not_{k,SN}/T$ is the not rate that SR_k forwards to its SubNet;
- $p_{not,k,SN} = \mu_{not,k,SN}/(\mu_{not,m,k} + \lambda_{pub,k})$ is the observed probability that SR_k forwards a notification to its SubNet;
- $q_{not,m,k} = \mu_{not,m,k}/\mu_{not,m,SN}$ is the observed probability that the SR_k master forwards a not to SR_k.

B.3 PERFORMANCE INDICES

Formulas for centers performance indices per chain:

1. $U_{c,k} = X_{c,k} * \tau_{c,k}$ is the utilization of SR_k relative to the c $\in \{sub, pub\}$ chain for a *SIENA* configuration;
2. $R_{c,k} = \rho_{c,k}/(1 - \sum_{j \in \{sub,pub\}} U_{j,k})$ is the average response time of SR_k for a request c $\in \{sub, pub\}$;
3. $Q_{c,k} = U_{c,k}/(1 - \sum_{j \in \{sub,pub\}} U_{j,k})$ is the average number of request of type c $\in \{sub, pub\}$ in the waiting queue of SR_k;

1. $X'_{c,k} = \lambda_{c,k}$ is the throughput of SR_k relative to the c $\in \{sub, pub\}$ chain for the new *SIENA* configuration;
2. $U'_{c,k} = \lambda_{c,k} * \tau_{c,k}$ is the utilization of SR_k relative to the c $\in \{sub, pub\}$ chain for the new *SIENA* configuration;

Formulas for performance indices aggregated per *SIENA* router

1. $X_k = \sum_{c \in \{sub,pub\}} X_{c,k}$ is the throughput of SR_k ;
2. $U_k = \sum_{c \in \{sub,pub\}} U_{c,k}$ is the utilization of SR_k;
3. $Q_k = \sum_{c \in \{sub,pub\}} Q_{c,k}$ is the waiting queue dimension of the SR_k;

REFERENCES

[1] C++Sim. http://cxxsim.ncl.ac.uk/.

[2] CSIM-Performance Simulator. http://www.atl.imco.com/proj/csim.

[3] JavaSim. http://javasim.ncl.ac.uk/.

[4] Petri Nets tools database. http://www.daimi.aau.dk/PetriNets.

[5] Poseidon. http://www.gentleware.com.

[6] Unified modeling language (uml), version 1.4. OMG Documentation. At http://www.omg.org/techonology/documents/formal/uml.htm.

[7] *Approximate Mean Value Analysis of Client-Server Systems with Multi-Class Requests* (May 1994).

[8] Proc. of SPIN workshops (1995–today).

[9] *Analysis of Balanced Fork-Join Queueing Networks* (1996).

[10] ObjecTime Ltd., Developer 5.1 Reference Manual, *ObjecTime Ltd.*, 1998.

[11] Proc. of WOSP workshops (1998–2004).

[12] OPNET Manuals, *Mil 3, Inc.*, 1999.

[13] *Special Issue Q̈ueueing Networks with BlockingP̈erformance Evaluation Journal* (2003), vol. 51.

[14] XMOF queries, views and transformations on models using MOF, OCL and patterns (ad/2003-08-07).

[15] ABUHR, R., AND CASSELMAN, R. *Use Case Maps for Object-Oriented Systems*. Prentice-Hall, 1996.

[16] AJMONE, M., BALBO, G., AND CONTE, G. A class of generalised stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems 2* (1984), 93–122.

[17] AJMONE, M., BALBO, G., AND CONTE, G. *Performance Models of Multiprocessor Performance*. The MIT Press, 1986.

[18] ANDOLFI, F., AQUILANI, F., BALSAMO, S., AND INVERARDI, P. Deriving performance models of software architectures from message sequence charts. In *Proceedings of the Second International Workshop on Software and Performance (WOSP00)* (September 2000), pp. 47–57.

[19] AQUILANI, F., BALSAMO, S., AND INVERARDI, P. Performance analysis at the software architecture design level. *Performance Evaluation 45*, 4 (2001), 205–221.

[20] ArgoUML – Object-oriented design tool with cognitive support.

[21] ARIEF, L., AND SPEIRS, N. A UML tool for an automatic generation of simulation programs. In *Proceedings of the Second International Workshop on Software and Performance (WOSP00)* (September 2000), pp. 71–76.

[22] BACCELLI, F., BALBO, G., BOUCHERIE, R., CAMPOS, J., AND CHIOLA, G. Annotated bibliography on stochastic petri nets. In *Performance Evaluation of Parallel and Distributed Systems-Solution Methods* (Amsterdam, 1994), C. Tract, Ed., no. N.105, pp. pp.1–24.

[23] BALBO, G., BRUELL, S., AND GHANTA, S. Combinig queueing networks and generalized stochastic petri nets for the solution of complex models of system behavior. *IEEE Transactions on Computers 37* (1988), 1251–1268.

[24] BALSAMO, S., BERNARDO, M., AND SIMEONI, M. Combinig stochastic process algebras and queueing networks for software architecture analysis. In *WOSP02* (2002), pp. 190–202.

[25] BALSAMO, S., DE NITTO PERSONE, V., AND INVERARDI, P. A review on queueing network models with finite capacity queues for software architectures performance prediction. *Perform. Eval. 51*, 2/4 (2003), 269–288.

[26] BALSAMO, S., DE NITTO PERSONÉ, V., AND ONVURAL, R. *Analysis of Queueing Networks with Blocking*. Kluwer Academic Publishers, 2001.

[27] BALSAMO, S., DI MARCO, A., INVERARDI, P., AND SIMEONI, M. Model-based performance prediction in software development: A survey. *IEEE Transactions of Software Enginnering 30*, 5 (2004), 295–310.

[28] BALSAMO, S., AND MARZOLLA, M. A simulation-based approach to software performance modeling. In *Proc. Joint 9th European Software Engineering Conference (ESEC& 11th SIGSOFT Symposium on the Foundations of Software Engineering FSE-11* (Helsinki, FI, Sept. 1–5 2003), P. Inverardi, Ed., ACM Press.

[29] BALSAMO, S., AND MARZOLLA, M. A simulation-based approach to software performance modeling. Tech. Rep. TR SAH/44, MIUR Sahara Project, Mar. 2003.

[30] BALSAMO, S., AND MARZOLLA, M. Towards performance evaluation of mobile systems in UML. In *Proc. of ESMc'03, The European Simulation and Modelling Conference* (Naples, Italy, Oct. 27–29 2003), B. di Martino, L. T. Yang, and C. Bobenau, Eds., EUROSIS-ETI, pp. 61–68.

[31] BALSAMO, S., MARZOLLA, M., DI MARCO, A., AND INVERARDI, P. Experimenting different software architectures performance techniques: a case study. In *Fourth International Workshop on Software and Performance WOSP 2004* (2004), pp. 115–119.

[32] BANKS, J., Ed. *Handbook of Simulation*. Wiley–Interscience, 1998.

[33] BANKS, J., CARSON, J. S., NELSON, B. L., AND NICOL, D. M. *Discrete-Event System Simulation*, 3rd ed. Prentice Hall, 2000.

[34] BANKS, J., II, J. C., NELSON, B., AND NICOL, D. *Discrete-event System Simulation*. Prentice-Hall, 1999.

[35] BASS, L., CLEMENTS, P., AND KAZMAN, R. *Software Architecture in Practice - second Edition*. SEI Series in Software Engineering. Addison-Wesley, 2003.

[36] BAUSE, F., AND KLAMANN, A. HiQPN user-s guide. Tech. rep., University of Dortmund, 1996.

[37] BEILNER, H., MATTER, J., AND WYSOCKI, C. The hierarchical evaluation tool HIT. In *Proceedings of the /th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation* (Wien, 1994).

[38] BERNARDI, S., DONATELLI, S., AND MERSEGUER, J. From UML sequence diagrams and statecharts to analysable petri net models. In *Proceedings of the Third International Workshop on Software and Performance (WOSP02)* (July 2002), pp. 35–45.

[39] BERNARDO, M. *TwoTowers 2.0 User Manual*. http://www.sti.uniurb.it/bernardo/twotowers, 2002.

[40] BERNARDO, M. Twotowers 3.0: Enhancing usability. In *Proc. of the 11th IEEE/ACM Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecomunication Systems* (Orlando (FL), 2003), pp. 188–193.

[41] BERNARDO, M., AND BRAVETTI, M. Performance measurement sensitive congruences for markovian process algebras. *Theoretical Computer Science 290* (2003), 117–160.

[42] BERNARDO, M., CIANCARINI, P., AND DONATIELLO, L. On the formalization of architectural types with process algebras. In *Proc. of the 8th ACM Int. Symp. on the Foundations of Software Engineering (FSE-8)* (San Diego(CA), 2000), A. Press, Ed., pp. pp.140–148.

[43] BERNARDO, M., DONATIELLO, L., AND CIANCARINI, P. Stochastic process algebra: From an algebraic formalism to an architectural description language. *Performance Evaluation of Complex Systems: Techniques and Tools LNCS 2459* (2002), 236–260.

[44] BERNARDO, M., AND GORRIERI, R. A tutorial on EMPA: A theory of concurrent processes with nondeterminism, priorities, probabilities and time. *Theoretical Computer Science 202*, 1–2 (1998), 1–54.

[45] BERTOLINO, A., AND MIRANDOLA, R. Towards component-based software performance engineering. In *Proc. 6th Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction, ACM/IEEE 25th International Conference on Software Engineering ICSE 2003* (Portland, Oregon, USA, 2003), pp. 1–6.

[46] BERTOLINO, A., AND MIRANDOLA, R. CB-SPE tool: Putting component-based performance engineering into practice. In *CBSE* (2004), pp. 233–248.

[47] BUCHHOLZ, P. *A framework for the hierarchical analysis of discrete evet dynamic systems*. PhD thesis, University of Dortmund, 1996.

[48] BUCHI, J. R. On a decision method in restricted second order arithmetic. In *Proceedings of International Congress of Logic, Methodology and Philosophy of Science* (1960), S. U. Press, Ed., pp. 1–11.

[49] BUHR, R., AND R.S.CASSELMAN. *Use CASE Maps for Object-Oriented Systems*. Prentice Hall, 1996.

[50] CAPORUSCIO, M., CARZANIGA, A., AND WOLF, A. L. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. *IEEE Transactions on Software Engineering* (Dec 2003).

[51] CAPORUSCIO, M., DI MARCO, A., AND INVERARDI, P. Run-time performance management of the siena publish/subscribe middleware. In *Fifth International Workshop on Software and Performance WOSP 2005. To appear.* (Palma de Mallorca, Illes Balears,SPAIN., July 2005).

[52] CARZANIGA, A., ROSENBLUM, D., AND WOLF, A. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing* (Portland, Oregon, July 2000), pp. 219–227.

[53] CARZANIGA, A., ROSENBLUM, D., AND WOLF, A. Design and Evaluation of a Wide-Area Event Notification Service. 332–383.

[54] CASTALDI, M. *Dynamic Reconfiguration of Component Based Applications*. PhD thesis, University of L'Aquila, 2004.

[55] CASTALDI, M., DI MARCO, A., AND INVERARDI, P. Data driven reconfiguration for performance improvements: a model based approach. In *2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems, (RAMSS04)* (Edinburgh, Scotland, UK, May 2004).

[56] COE, P., HOWELL, F., IBBETT, R., AND WILLIAMS, L. Techical note:a hierarchical computer architecture design and simulation environment. *ACM Transactions on Modelling and Computer Simulation 8*, 4 (1998), 431–446.

[57] COMPARE, D., DI MARCO, A., D'ONOFRIO, A., AND INVERARDI, P. Our experience in the integration of process algebra based performance validation in an industrial context. Tech. rep.

[58] COMPARE, D., D'ONOFRIO, A., DI MARCO, A., AND INVERARDI, P. Automated performance validation of software design: An industrial experience. In *ASE* (2004), pp. 298–301.

[59] COMPARE, D., INVERARDI, P., PELLICCIONE, P., AND SEBASTIANI, A. Integrating model-checking architectural analysis and validation in a real software life-cycle. In *Proceedings of the 12th Formal Methods 2003, European Symposium (FM03)* (San Diego, California, September 2003), vol. 2805, Springer.

[60] CORTELLESSA, V., D'AMBROGIO, A., AND IAZEOLLA, G. Automatic derivation of software performance models from CASE documents. *Performance Evaluation 45* (2001), 81–105.

[61] CORTELLESSA, V., DI MARCO, A., AND INVERARDI, P. Three performance models at work: A software designer perspective. *Electr. Notes Theor. Comput. Sci. 97* (2004), 219–239.

[62] CORTELLESSA, V., DI MARCO, A., INVERARDI, P., MANCINELLI, F., AND PELLICCIONE, P. A framework for the integration of functional and non-functional analysis of software architectures. In *Int. Workshop on Test and Analysis of Component Based Systems (TACOS 2004)* (Barcellona,SPAIN., March 2004).

[63] CORTELLESSA, V., DI MARCO, A., INVERARDI, P., MUCCINI, H., AND PELLICCIONE, P. Using UML for SA-based modeling and analysis. In *Workshop on Software Architecture Description & UML (UML04 workshop)* (Lisbon, Portugal, October 2004).

[64] CORTELLESSA, V., GENTILE, M., AND PIZZUTI, M. `XPRIT`: An XML-based tool to translate UML diagrams into execution graphs and queueing networks (tool paper). In *Proc. of 1st Int. Conf. on the Quantitative Evaluation of Systems* (Enschede, NL, 2004), IEEE Computer Society.

[65] CORTELLESSA, V., IAZEOLLA, G., AND MIRANDOLA, R. Early generation of performance models for object-oriented systems. *IEE Proceedings-Software 147*, 3 (2000), 61–72.

[66] CORTELLESSA, V., AND MIRANDOLA, R. `PRIMA-UML`: a performance validation incremental methodology on early UML diagrams. *Science of Computer Programming*.

[67] CYSNEIROS, L., DE MELO SABAT NETO, J., , AND DO PRADO LEITE, G. S. A framework for integrating non-functional requirements into conceptual models. *Requirements Engineering 6*, 2 (April 2001), 97–115.

[68] DALRYMPLE, E., AND EDWARDS, M. Independent integrated verification and validation. In *Proceedings of the Acquisition Conference* (Arlington, Virginia, January 2003).

[69] DAS, O., AND WOODSIDE, C. M. Computing the performability of layered distributed systems with a management architecture. In *WOSP '04: Proceedings of the fourth international workshop on Software and performance* (New York, NY, USA, 2004), ACM Press, pp. 174–185.

[70] DELLA PENNA, G., DI MARCO, A., INTRIGILA, B., MELATTI, I., AND PIERANTONIO, A. Interoperability mapping from xml schemas to er diagrams. *Submitted to Data & Knowledge Engineering - Elsevier Science Journal.*.

[71] DELLA PENNA, G., DI MARCO, A., INTRIGILA, B., MELATTI, I., AND PIERANTONIO, A. Xere: Towards a natural interoperability between xml and er diagrams. In *FASE* (2003), pp. 356–371.

[72] DI MARCO, A., AND INVERARDI, P. Starting from message sequence chart for software architecture early performance analysis. In *2nd International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools* (Portland, Oregon, USA. http://www.di.univaq.it/di/pub.php?username=adimarco, May 2003).

[73] DI MARCO, A., AND INVERARDI, P. Compositional generation of software architecture performance qn models. In *Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA)*. (2004), pp. 37–46.

[74] DUTOIT, A., KERKOW, D., PAECH, B., AND VON KNETHEN, A. Functional requirements, non-functional requirements, and architecture should not be separated. In *Proceedings of International Workshop REFSQ '02* (Essen, September 2002).

[75] ENGINEERING DEPARTMENT, E. I. A. Cdif - case data interchange format overview. vol. EIA/IS-106.

[76] ENSLOW JR, P. H. What is a "distributed" data processing system? *Computer 11*, 1 (January 1978), 13–21.

[77] FRANKS, G., HUBBARD, A., MAJUMDAR, S., PETRIU, D., ROLIA, J., AND WOODSIDE, C. A toolset for performance engineering and software design of client-server systems. *Performance Evaluation 24*, 1-2 (1995), 117–135.

[78] FRANKS, R., AND WOODSIDE, C. Performance of multi-level client-server systems with parallel service operations. In *ACM Proceedings of the First Workshop on Software and Performance* (Santa Fe, New Mexico, 1998), pp. 120–130.

[79] GARLAN, D., KHERSONSKY, S., AND KIM, J. S. Model checking publish/subscribe systems. In *Proceedings of The 10th International SPIN Workshop on Model Checking of Software (SPIN 03)* (Portland, Oregon, May 2003).

[80] GARLAN, D., MONROE, R., AND WILE, D. Acme: Architectural description of component-based systems. In *Foundations of Component-Based Systems*, G. T. Leavens and M. Sitaraman, Eds. Cambridge University Press, 2000, pp. 47–68.

[81] GARLAN, D., SCHMERL, B., AND CHANG, J. Using gauges for architecture-based monitoring and adaptation. In *Proceedings of Working Conference on Complex and Dynamic Systems Architecture* (Brisbane, Australia, December 2001).

[82] GILMORE, S., AND HILLSTON, J. The pepa wokbench: A tool to support a process algebra-based approach to performance modelling. In *Proceedings of the 7th International Conference on Modelling Techniques and Tools for Performance Evaluation* (1994), Springer, Ed., vol. 794, pp. 353–368.

[83] GOMAA, H. *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Addison-Wesley, 2000.

[84] GOMAA, H., AND MENASCÉ, D. Design and performance modeling of component interconnection patterns for distributed software architectures. In *Proceedings of the Second International Workshop on Software and Performance (WOSP00)* (Ottawa, Canada, September 2000), pp. 117–126.

[85] GOMAA, H., AND MENASCÉ, D. Performance engineering of component-based distributed software system. In *Performance Engineering* (2001), vol. 2047, Springer, pp. 40–55.

[86] GRASSI, V., AND MIRANDOLA, R. `PRIMAmob-UML`: A methodology for performance analysis of mobile software architectures. In *Proceedings of the Third International Workshop on Software and Performance (WOSP02)* (Rome, Italy, July 2002), pp. 262–274.

[87] GROUP, O. M. *UML Profile, for Schedulability, Performance, and Time*. OMG document ptc/2002-03-02, http://www.omg.org/cgi-bin/doc?ptc/2002-03-02.

[88] GU, G., AND PETRIU, D. C. XSLT transformation from UML models to LQN performance models. In *Proceedings of the Third International Workshop on Software and Performance (WOSP02)* (Rome, Italy, July 2002), pp. 227–234.

[89] HARRELD, H. Nasa delays satellite launch after finding bugs in software program. In *Federal Computer Week* (1998).

[90] HARRISON, P., AND HILLSTON, J. Exploiting quasi-reversible structures in markovian process algebra models. *Computer Journal 38*, 7 (1995), 510–520.

[91] HERMANNS, H., HERZOG, U., AND KATOEN, J. P. Process algebra for performance evaluation. *Theoretical Computer Science 274*, 1–2 (Mar. 2002), 43–87.

[92] HERMANNS, H., MERTSIOTAKIS, V., AND RETTELBACH, M. A construction and analysis tool based on the stochastic process algebra TIPP. Springer, Ed., no. LNCS 1055, pp. 427–430.

[93] HERZOG, U., KLEHMET, U., MERTSIOTAKIS, V., AND SIEGLE, M. Compositional performance modelling with the `TIPPtool`. *Performance Evaluation 39*, 1-4 (2000), 5–35.

[94] HILLSTON, J. Pepa-performance enhanced process algebra. Tech. rep., Dept. of Computer Science, University of Edimburgh, 1993.

[95] HILLSTON, J., AND THOMAS, N. Product form solution for a class of pepa models. *Performance Evaluation 35*, 3 (1999), 171–192.

[96] HOARE, C. *Communicating Sequential Processes*. Prentice-Hall International, London, 1985.

[97] HOEBEN, F. Using UML models for performance calculation. In *Proceedings of the Second International Workshop on Software and Performance (WOSP00)* (Ottawa, Canada, September 2000), pp. 77–82.

[98] HOLZMANN, G. J. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, September 2003.

[99] HOPCROFT, J., AND ULLMAN, J. *Introduction to automata theory, languages and computations*. Addison-Wesley, 1979.

[100] HOWARD, R. *Dynamic pobabilistic Systems*. John Wiley and Sons, 1971.

[101] INVERARDI, P., MUCCINI, H., AND PELLICCIONE, P. Automated check of architectural models consistency using SPIN. In *Proceedings of Automated Software Engineering Conference Proceedings (ASE 2001)* (San Diego, California, November 2001).

[102] KÄHKIPURO, P. Uml-based performance modeling framework for component-based distributed systems. In *Proceedings of Performance Engineering* (2001), Springer, Ed., no. 2047, pp. 167–184.

[103] KANT, K. *Introduction to Computer System Performance Evaluation*. McGraw-Hill, 1992.

[104] KEMENY, J., AND SNELL, J. *Finite Markov Chains*. Springer, New York, 1976.

[105] KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. An overview of AspectJ. *Lecture Notes in Computer Science 2072* (2001), 327–355.

[106] KICZALES, G., LAMPING, J., MENHDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. Aspect-Oriented Programming. In *Proceedings European Conference on Object-Oriented Programming*, M. Akşit and S. Matsuoka, Eds., vol. 1241. Springer-Verlag, Berlin, Heidelberg, and New York, 1997, pp. 220–242.

[107] KING, P., AND POOLEY, R. Derivation of petri net performance models from UML specifications of communication software. In *Proceedings of XV UK Performance Engineering Workshop* (1999).

[108] KLEHMET, U., AND MERTSIOTAKIS, V. `TIPPtool`: Timed processes and performability evaluation - user's guide. Tech. rep., University of Erlangen, 1998.

[109] KLEINROCK, L. *Queueing Systems Vol. 1:Theory*. Wiley, 1975.

[110] LAW, A. M., AND KELTON, W. D. *Simulation Modeling and Analysis*, 3rd ed. McGraw–Hill, 2000.

[111] LAZOWSKA, E., KAHORJAN, J., GRAHAM, G. S., AND SEVCIK, K. C. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., Englewood Cliffs, 1984.

[112] LINDEMANN, C., THÜMMLER, A., KLEMM, A., LOHMANN, M., AND WALDHORST, O. P. Performance analysis of time-enhanced UML diagrams based on stochastic processes. In *Proceedings of the Third International Workshop on Software and Performance (WOSP02)* (Rome, Italy, July 2002), pp. 25–34.

[113] LÓPEZ-GRAO, J. P., MERSEGUER, J., AND CAMPOS, J. From uml activity diagrams to stochastic petri nets: application to software performance engineering. In *WOSP '04: Proceedings of the fourth international workshop on Software and performance* (New York, NY, USA, 2004), ACM Press, pp. 25–36.

[114] MARZOLLA, M. *Simulation-Based Performance Modeling of UML Software Architectures*. PhD Thesis TD-2004-1, Dipartimento di Informatica, Università Ca' Foscari di Venezia, Mestre, Italy, Feb. 2004.

[115] MENASCÉ, D. A. A framework for software performance engineering of client/server systems. In *Proceedings of the 1997 Computer Measurement Group Conference* (Orlando, Florida, 1997).

[116] MENASCÉ, D. A., AND GOMAA, H. On a language based method for software performance engineering of client/server systems. In *in ACM Proceedings of the First Workshop on Software and Performance*, pp. 63–69.

[117] MENASCÉ, D. A., AND GOMAA, H. A method for design and performance amodeling of client/server systems. *IEEE Transacion on Software Engineering 26*, 11 (2000), 1066–1085.

[118] MENASCÉ, D. A., RUAN, H., AND GOMAA, H. A framework for QoS-aware software components. In *Proceedings of the fourth international workshop on Software and performance* (Redwood Shores, California), pp. 186–196.

[119] MIGUEL, M. D., LAMBOLAIS, T., AND M. HANNOUZ, S. BETGÉ-BREZETZ, S. P. Uml extensions for the specification and evaluation of latency constraints in architectural models. In *Proceedings of the Second International Workshop on Software and Performance (WOSP00)* (September 2000), pp. 83–880.

[120] MILNER, R. *Communication and Concurrency*. Prentice-Hall International, International Series on Conputer Science, 1989. International Series on Computer Science.

[121] OBJECT MANAGEMENT GROUP (OMG). XML Metadata Interchange (XMI) specification, version 2.0, 2002.

[122] OMONDO. *Eclipseuml*. http://www.omondo.com.

[123] PETRIU, D., AMYOT, D., AND WOODSIDE., M. Scenario-based performance engineering with `UCMNav`.

[124] PETRIU, D. C., AND SHEN, H. Applying the UML performance profile: Graph grammar-based derivation of LQN models from UML specifications. In *Proceedings of Computer Performance Evaluation, Modelling Techniques and Tools 12th International Conference, TOOLS 2002* (London, UK, 2002), vol. 2324 of *LNCS*, pp. 159–177.

[125] PETRIU, D. C., AND WANG, X. From uml descriptions of high-level software architectures to lqn performance models. In *Proceedings of AGTIVE'99* (1999), S. Verlag, Ed., pp. 47–62.

[126] PETRIU, D. C., AND WOODSIDE, C. Software performance models from system scenarios in use case maps. In *Proceedings of Computer Performance Evaluation, Modelling Techniques and Tools 12th International Conference, TOOLS 2002* (2002), vol. 2324 of *LNCS*, pp. 141–158.

[127] PNUELI, A. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundation of Computer Science* (1977), pp. 46–57.

[128] POOLEY, R. Using UML to derive stochastic process algebra models. In *Proceedings of XV UK Performance Engineering Workshop* (1999), pp. 23–34.

[129] POOLEY, R., AND KING, P. The unified modeling language and performance engineering. In *Proceedings of IEE Software* (1999), pp. 2–10.

[130] PORCARELLI, S., CASTALDI, M., GIANDOMENICO, F. D., BONDAVALLI, A., AND INVERARDI, P. A framework for reconfiguration-based fault tolerance in distributed systems. In *Architecting Dependable Systems II. LNCS 3069* (March 2004), C. G. R. De Lemos, A. Romanovsky, Ed., pp. 167–190.

[131] PROJECT., T. E. Eclipse platform technical overview. Tech. rep., The Eclipse project, 2001.

[132] REISIG, W. Petri nets: an introduction. *EATCS Monographs on Theoretical Computer Science Vol.4* (1985).

[133] ROLIA, J., AND SEVCIK, K. The method of layers. *IEEE Transaction on Software Engineering 21/8* (1995), 622–688.

[134] SAUER, C. H., AND MACNAIR, E. A. Queueing network software for systems modeling. In *Research Report RC-7143, IBM Thomas J. Watson Research Center* (Yorktown,Heights, NY, 1978).

[135] SAUER, C. H., REISER, M., AND MACNAIR, E. A. RESQ - a package for solution of generalized queueing networks. In *Proceedings, National Computer Conference* (Dallas, TX, 1977), pp. 977–986.

[136] S.BALSAMO, L.DONATIELLO, AND VAN DIJK, N. Bounded performance analysis of parallel processing systems. *IEEE Transactions on Parallel and Distributed Systems 9* (October 1998), 1041–1056.

[137] SECTOR, I. T. S. *Message Sequence Charts, ITU-T Recommentation Z.120(11/99)*. 1999.

[138] SERENO, M. Towards a product form solution for stochastic process algebras. *Computer Journal 38* (1995), 622–632.

[139] SMITH, C. *Performance Engineering of Software Systems*. Addison-Wesley, 1990.

[140] SMITH, C. Definition of a performance model interchange format. In *Performance Engineering Services* (October 1994), vol. PES-1001-94.

[141] SMITH, C., AND LLADÓ, C. M. Performance model interchange format (`PMIF 2.0`): XML definition and implementation. In *Proc. 1st Int. Conf. on Quantitative Evaluation of Systems (QEST)* (Enschede, NL, 2004).

[142] SMITH, C., AND WILLIAMS, L. Performance engineering evaluation of object-oriented systems with SPE•ED^{TM}. In *Proceedings of Computer Performance Evaluation* (Berlin, Germany, 1997), vol. 1245 of *LNCS*, pp. 135–154.

[143] SMITH, C., AND WILLIAMS, L. Performance engineering evaluation of CORBA-based distributed systems with SPE•ED. In *10th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation* (Palma de Mallorca, Spain, 1998), Springer Verlag.

[144] SMITH, C., AND WILLIAMS, L. A performance model interchange format. *Journal of Systems and Software 49*, 1 (1999).

[145] SMITH, C., AND WILLIAMS, L. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, Reading, MA, 2002.

[146] SMITH, C. U., LLADÓ, C. M., CORTELLESSA, V., DI MARCO, A., AND WILLIAMS, L. G. From UML models to software performance results: An SPE process based on XML interchange formats. In *Fifth International Workshop on Software and Performance WOSP 2005. To appear.* (Palma de Mallorca, Illes Balears,SPAIN., July 2005).

[147] SMITH, C. U., WILLIAMS, L. G., LLADÓ, C. M., CORTELLESSA, V., AND DI MARCO, A. Software performance engineering model interchange format (`S-PMIF 2.0`): XML definition and implementation technical report. Tech. rep., L&S Computer Technology, Inc., Dec. 2004.

[148] OBJECT MANAGEMENT GROUP. *Unified Modeling Language 2.0*. http://www.omg.org/uml/, 2003.

[149] SEA GROUP, UNIVERSITY OF L'AQUILA. *Tool-One project*. http://www.di.univaq.it/di/project.php?id=8, 2003.

[150] TIJMS, H. C. *Stochastic Models, An Algorithmic Approach*. John Wiley and Sons Ltd, 1994.

[151] TRIVEDI, K. S. *Probability and Statistics with Reliability, Queueing and Computer Science Applications*. John Wiley and Sons, 2001.

[152] UCHITEL, S., KRAMER, J., AND MAGEE, J. Synthesis of behavorial models from scenarios. *IEEE Trans. on Software Engineering 29-2* (2003).

[153] VERAN, M., AND POTIER, D. QNAP 2:a portable environment for queueing systems modelling. In *Rapport de recherche de l'INRIA-Rocquencourt* (At http://www.inria.fr/rrrt/rr-0314.html, 1984).

[154] W3C. eXtensible Markup Language (XML) 1.0 (second edition) W3C recommendation 6 october 2000.

[155] W3C. World Wide Web Consortium. http://www.w3.org.

[156] W3C. *XML Schema Part 1: Structures and XML Schema Part 2: Datatypes, W3C Recommendation 2 May 2001*. http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/ and http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/.

[157] W3C. XML Path Language, 1999. http://www.w3.org/TR/xpath.

[158] W3C. eXtensible Stylesheet Language (XSL), 2001. http://www.w3.org/Style/XSL/.

[159] WILLIAMS, L., AND SMITH, C. Performance evaluation of software architectures. In *in ACM Proceedings of the First Workshop on Software and Performance*, pp. 164–177.

[160] WILLIAMS, L., AND SMITH, C. Information requirements for software performance engineering. In *Proceedings of International Conference on Modeling Techniques and Tools for Computer Performance Evaluation* (Heidelberg, Germany, 1995), Springer Verlag.

[161] WILLIAMS, L., AND SMITH, C. PASA: A method for the performance assesment of software architectures. In *Proceedings of the Third International Workshop on Software and Performance (WOSP02)* (Rome, Italy, July 2002), pp. 179–189.

[162] WOODSIDE, C., HRISCHUK, C., SELIC, B., AND S.BRAYAROV. Automated performance modeling of software generated by a design environment. *Performance Evaluation 45* (2001), 107–123.

[163] WOODSIDE, C., NEILSON, J., PETRIU, S., AND MJUMDAR, S. The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. *IEEE Transaction on Computer 44* (1995), 20–34.

[164] WU, X., AND DAVID MCMULLAN, M. W. Component-based performance prediction. In *Proc. 6th Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction, ACM/IEEE 25th International Conference on Software Engineering ICSE 2003* (Portland, Oregon, USA, 2003), pp. 13–18.