Vittorio Cortellessa
Antinisca Di Marco
Paola Inverardi

Model-Based Software Performance Analysis

Springer

Model-Based Software Performance Analysis

Vittorio Cortellessa • Antinisca Di Marco •
Paola Inverardi

Model-Based Software Performance Analysis

Springer

Vittorio Cortellessa
Antinisca Di Marco
Paola Inverardi
Dipartimento di Informatica
Università di L'Aquila
Via Vetoio 1
Coppito, 67010 L'Aquila
Italy
vittorio.cortellessa@univaq.it
antinisca.dimarco@univaq.it
paola.inverardi@univaq.it

ISBN 978-3-642-13620-7 e-ISBN 978-3-642-13621-4
DOI 10.1007/978-3-642-13621-4
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011927935

ACM Computing Classification (1998): D.2, C.4, D.4.8, I.6

Cover design: KünkelLopka GmbH

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

This book is dedicated to the town of L'Aquila, destroyed by an earthquake on April 6th, 2009. L'Aquila is the locus of our past births in life, in work, in love. As for the future we wish to L'Aquila its own re-birth to host again many lives, many works, many loves.

Vittorio, Antinisca and Paola

Preface

Goal of the Book

In the last decade there has been a growing interest in the research and software industry communities toward techniques, methods and tools that allow one to manage system performance concerns in the software developer domain. Poor performance can often be the cause of software project failure, and the need to address performance concerns during software development is fully acknowledged. One of the main impediments to progress in this field lies in the different cultures in software and performance and in the lack of standard practices and tool support. One promising direction to bridge this gap is the one described in this book. Model-based Software Performance Analysis is the research domain that introduces performance concerns into the scope of software models, thus allowing the developer to carry out performance analysis all along the software lifecycle.

The goal of the book is to provide the cross-knowledge that allows developers to face software performance issues since the very early phases of the software development. On one hand, we provide the basic concepts of performance analysis. On the other hand, we introduce the readers to the problem of making a performance analysis a common practice in the software development process and describe the most representative methodologies proposed to annotate and translate software models into performance models.

For the sake of book uniformity, we do not report here applications of approaches to case studies from the real world (except for a few cases), because the intent of the book is to provide an introduction to this domain. We leave to possible future editions the treatment of real case studies and the lessons learned from them.

A Bird's-Eye View of Chapters Contents

Chapter 1 embeds the software performance analysis activity in the wider framework of non-functional validation. It explains how different non-functional attributes contribute to the quality of a software product. Thereafter we clarify the

difference between software performance analysis (where static/dynamic software models are explicitly built and play a crucial role in the analysis process) and system performance analysis (where the software is only meant as a system synthetic workload).

Chapter 2 gives an overview of the most widely adopted software notations, including the techniques for extending UML. Note that this chapter does not aim at presenting notational details, as most of the readers should be familiar with the majority of this chapter concepts.

Chapter 3 gives an overview of the most widely adopted performance notations. The notations that will be used in the following chapters are treated to a larger extent. The others are only briefly surveyed and mentioned for completeness purposes.

Chapter 4 focuses on the software performance process. It introduces the software lifecycle phases outlining the distance between software artifacts and performance models.

Chapter 5 introduces the reader to the transformation-based approaches that support the generation of performance models from software models. In particular, it describes how software models can be annotated with performance parameters. Then an overview of the major supporting tools for model transformations in this domain is presented. The chapter ends with a classification and taxonomy of the existing approaches.

Chapter 6 provides techniques and tools to solve the performance models obtained through model transformations. Analytical, approximate and simulation-based techniques are described.

In Chap. 7 the major trends that may affect the future evolution of software performance analysis are presented. They address very different areas as they span from the relation of performance and Model Driven Architecture to the need of a software performance ontology.

In terms of novelty of contents: Chaps. 1–3 and Chap. 6 contain well assessed notions in the field; Chap. 4 introduces a new view in the integration of performance into a software lifecycle; Chap. 5 describes techniques and tools that have influenced the discipline in the last decade; finally, Chap. 7 provides a glimpse to the current research trends, thus sketching the possible evolutions of the discipline in the next decade.

Figure 0.1 summarizes this bird's-eye view of the chapters and their relationships.

Using the Book as Teaching Text

The book is not primarily intended as a teaching text. However, it can be used to complement more traditional Software Engineering teaching texts to provide a deeper insight of model software performance analysis. To this extent it provides chapters on the basic software and performance modeling notions and describes in detail some transformational techniques. Instructors may choose their favorite transformational approach and follow the detailed case study presentation as a guideline to present the major concepts underlying this discipline.

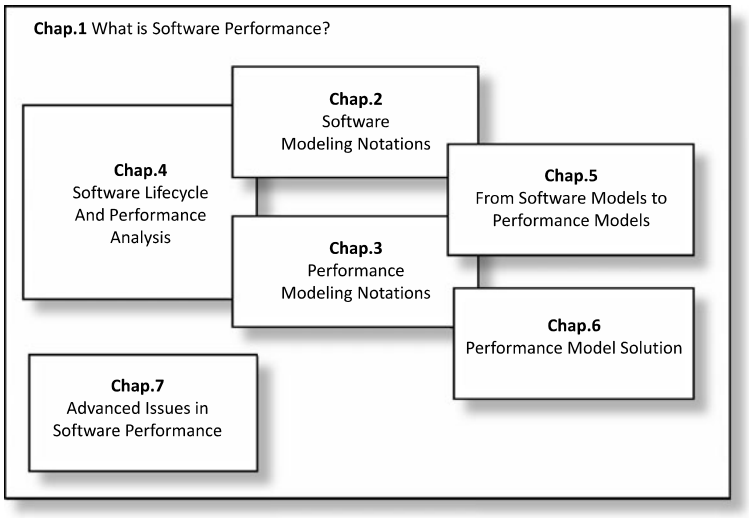


Fig. 0.1 Book chapters organization

Acknowledgments

Quite a large number of people have contributed over the years to this book contents, so it is difficult to mention all of them in a few lines.

First, thanks go to the "mother" and "father" of the software performance discipline, namely Connie Smith and Murray Woodside. Without their pioneering work, the whole community would likely not exist today.

Our earlier co-authors in this discipline have influenced, in different ways, the content of this book and have contributed to our passion in this field. Many thanks to Simonetta Balsamo, Giuseppe Iazeolla, Raffaela Mirandola and Vincenzo Grassi.

Special thanks go to Jane Hillston, who has allowed us to use her excellent lecture notes, and to Ed Lazowska and his co-authors, whose book is still today a reference point for us and our students. Also thanks to Dorina Petriu, for her extensive work in the area of Software Performance Engineering. Part of their work has been reported in this book.

Thanks are due to our Ph.D. students Pasqualina Potena, Catia Trubiani and Luca Berardinelli, who, besides sustaining us with their enthusiasm and criticism, have also concretely helped us with figures and tables. And thanks go to all our former, current and future students that, with their daily work, represent the lymphatic system of our working life, and contribute to our human and scientific growth.

Last but not least, we would like to thank the reviewers of this book, Andrea Polini and Jose Merseguer, who have helped us to largely improve the book's organization and contents. Many thanks for your deep and very useful reviews! Special thanks go to Ralf Gerstner, our editor at Springer, for his great patience and dedication.

Finally we would like to thank our families, because from the very beginning they have sustained our dreams (all of a different nature) and have allowed such dreams to become true. Every day it is clear to us that the existence of our families is per se an immense value that cannot be replaced by any other thing.

L'Aquila, Italy

Vittorio Cortellessa
Antinisca Di Marco
Paola Inverardi

Contents

Chapter 1
What Is Software Performance?

The increasing complexity of software and its pervasiveness in everyday life has in the last years motivated growing interest for software analysis. This has mainly been directed to assess functional properties of the software systems (related to their structure and their behavior) and, in the case of safety critical systems, dependability properties. The quantitative behavior of a software system has gained relevance only recently with the advent of software performance analysis. This kind of analysis aims at assessing the quantitative behavior of a software system by comprehensively analyzing its structure and its behavior, from design to code.

From a historical perspective software performance was initially addressed trying to export performance modeling and measurements from the hardware domain to the software domain. This was rather straightforward when considering the operating system layer, but it has assumed a new dimension when the focus was directed toward 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 [79, 106].

Earlier in the software development the information is more limited about the system to be developed. Therefore performance analysis has to target different goals, depending on the phase of development where it is applied. In any case the development artifacts should be adequately enriched to apply performance analysis techniques (e.g. stochastic data about the software behavior should be provided as part of the operational profile). With the advent of Model Driven Engineering in the software domain, these artifacts became models and assumed more active roles than simple design abstractions [102]. This evolution obviously helps to make (model-based) performance analysis a daily practice in the software development process, as will be shown throughout the whole book.

In this chapter we introduce the concepts and definitions that will be used throughout the entire book.

V. Cortellessa et al., *Model-Based Software Performance Analysis*,
DOI 10.1007/978-3-642-13621-4_1, © Springer-Verlag Berlin Heidelberg 2011

1.1 Non-functional Validation of Software Systems

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. Non-functional problems may be so severe that fixing them can require considerable changes at any stage of software lifecycle, notably 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 non-functional issues later in the development process. But this development style, called the "fix-it-later" approach, ever more frequently brings large-sized projects to failure [65].

On the contrary, non-functional validation consists of checking at any stage of the software lifecycle, through the analysis of the produced artifacts, whether the system under development meets non-functional requirements. Requirements play a key role in software validation, as they represent the target of comparison for the work in progress. Non-functional requirements have the special features of being expressed through software metrics that quantify certain system attributes that can be collected in two multi-attributes: *dependability* and *performance*.

Non-functional attributes related to dependability have been defined and extensively classified in [16]. The original definition of dependability is *the ability to deliver service that can justifiably be trusted*. This definition stresses the need for justification of trust. An alternate definition that provides the criterion for deciding if the service is dependable is *the dependability of a system is the ability to avoid service failures that are more frequent and more severe than is acceptable*.

Dependability is evidently made of a set of attributes that do not necessarily have to be quantified for each software system. They can contribute to this software property with different weights depending on the specific application domain, on the software requirements as well as other factors that may affect the definition of dependability for a specific software system. The same consideration holds for performance.

In [16] the dependability is composed by[1]:

- availability: readiness for correct service,
- reliability: continuity of correct service,
- safety: absence of catastrophic consequences on the user(s) and the environment,

[1] We do not enter in specific aspects of dependability, because it is not the focus of this book, however, we provide this definition to better place performance in the non-functional domain of software.

- integrity: absence of improper system alterations,
- maintainability: ability to undergo modifications and repairs.[2]

Performance is not included in the definition of dependability, although in some domains performance failures are as critical as other types of failures. Performance will be precisely defined in Sect. 1.2.

The non-functional requirements capture the quantitative characteristics of a software system. Different software architectures/designs that are equivalent with respect to functional requirements can substantially differ from a non-functional viewpoint. Software development teams often have to decide among different functionally equivalent design alternatives relying only on their own skills and experience. This choice should be driven by non-functional attributes 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. So, if these requirements are so important for the success of a software project, why are they usually not considered during the software lifecycle?

In software practice, it is generally acknowledged that the lack of non-functional requirement validation during the software development process is mostly due to the knowledge gap between software engineers/architects and quality experts that makes difficult the usage of non-functional analysis as an integrated activity in the software lifecycle. In addition, short time to market constraints make this situation even more critical.

This knowledge gap is typical of non-functional requirements, because functional ones refer to software properties and characteristics (e.g. architectural structure or scenario behavior) that software engineers usually deal with in their daily practice. Instead, first-class entities of non-functional validation refer to a completely different domain, and include terms like operational profile, throughput, failure rate, etc.

For example, a functional requirement for an e-commerce software system could claim: "The purchase operation must be executable only after the user has been authenticated through the login operation". For the same system a non-functional requirement could instead claim: "The average execution time of the purchase operation must be lower than 2 seconds". But what is an execution time? What are the parameters that contribute to it? How can it be analyzed?

As mentioned, non-functional attributes can be quantified through metrics that represent the indices of quality of a software product. In order to express these indices through their primary parameters, specific notations have been introduced, such as Bayesian Belief Networks, Fault Trees, Queueing Networks, etc. Such notations allow one to build non-functional models that (in more or less complex ways) express the relationships between software model characteristics and non-functional indices.

[2]When addressing security, an additional attribute has great prominence, confidentiality, that is, the absence of unauthorized disclosure of information. Security is a composite of the attributes of confidentiality, integrity, and availability, requiring the concurrent existence of: (i) availability for authorized actions only, (ii) confidentiality, and (iii) integrity with improper meaning unauthorized.

As outlined above, however, the software model characteristics that contribute to the construction of non-functional models are not only represented by the software structure and behavior, but also by quantified parameters (such as the operational profile) that usually cannot be found in software models because they are not necessary for the functional validation of software. This represents the gap introduced above between software development practices and non-functional validation.

This perspective has significantly changed since the end of the 1990s, when it has been perceived that pushing toward automation in the whole non-functional domain would have helped to overcome the main resistances from the software development world: (i) automation in generating and solving non-functional models from software artifacts eliminates the need of having special skilled people working on the hand-made construction of non-functional models, (ii) automation also allows one to cut effort and time of non-functional validation that usually represents an obstacle to the daily practice of this (only apparently optional) task.

As illustrated in this book, since then many approaches have been introduced with the aim of contributing to the automation of the non-functional validation activities. Two attributes appear crucial to make any approach acceptable by the software community: transparency, i.e. minimal impact on the software notation and the software process adopted, and effectiveness, i.e. minimal complexity of techniques to annotate and transform software models into performance models.

The price to pay to automation is that software artifacts have to be annotated with additional information that represents the parameters necessary to produce non-functional models. Such parameters have to be somehow estimated, with a certain level of accuracy depending on the lifecycle phase, and also for this task well-assessed techniques have been introduced in the last few years.

1.2 Performance as a Non-functional Attribute

As a general definition, performance measures how effective is a software system with respect to time constraints and allocation of resources. Performance is not composed by a set of attributes like dependability, although it can be expressed through multiple indices.

Traditional performance indices are *response time*, *throughput* and *utilization*. Response time is the end-to-end time that a task spends to traverse a certain path within the system. Throughput is the number of jobs that can be completed per unit of time by a certain part of the system. Utilization is the percentage of time that a certain part of the system is busy working.

As defined above, these indices do not refer to specific elements of a software system because, like many non-functional attributes, they can be applied at different levels of abstraction. For example, a response time may refer to the time that a resource spends to process a software component service, but it may also refer to the time that a whole platform spends to provide a response to the user invocation of a system functionality. Units of measure may change for different levels of abstractions to represent the indices at the appropriate level of granularity.

Beside these general purpose performance indices two categories of additional indices can be defined, which are: (i) notation-specific indices and (ii) domain-specific indices. The former ones are used when the notation adopted to build a performance model allows us to introduce additional indices; an example is the queue length in Queueing Network models (QN), because other models do not explicitly represent this attribute. The latter ones are used in specific application domains, where general purpose indices are not powerful enough to express specifics of the domain; an example is the battery consumption in applications that are engineered to run on devices with limited resources (e.g. palmtops).

Performance needs appropriate notations to be represented. These notations aim at capturing static and (mostly) dynamic characteristics of the software system, as well as the parameters that represent the stochastic behavior of the system (see Chap. 3 for a description of the main performance modeling notations).

The performance model parameters can be partitioned in the following main categories: (i) the operational profile, (ii) the workload, (iii) the resource demands.

The operational profile is a collection of data that stochastically represent the usage that users make of a system in a certain environment. The same system, in fact, in different environments can be used in different ways. As an example, a web site that allows one to organize trips can be frequently invoked for renting cars when used in countries without acceptable public transportation services, whereas it can be frequently invoked for booking flights in countries with largely distributed territories.

The typical parameters that belong to the operational profiles are the probability that a certain type of user invokes a certain software service, as well as the probabilities that the available alternatives are taken in branching points of a service behavior.

The workload represents the intensity of system invocations from users. It can be characterized over all types of users (i.e. without distinguishing among types of users) or it can be expressed for each type of user at a lower level of abstraction.

A workload can be open, in systems where the number of users is not pre-defined (such as the world wide web), and it can be closed, in systems with a fixed number of users (such as a LAN). An open workload is usually expressed as a number of requests per unit of time, whereas a closed workload is expressed as the number of users in the system and the average interval of time an user spends in between two requests.

The resource demands represent the amount of (software or hardware) resources that a piece of software (at any level of abstraction) requires to complete its task. This parameter points out that a certain amount of information about the target hardware platform is necessary in order to model and analyze software performance.

Time in software systems is determined by two contributions: the processing time usefully spent by software within the hardware platform (e.g. CPU processing time, disk access time, etc.), and the waiting time wasted by the software while waiting for the access to platform resources. It is the combination of these two factors that basically determines the software performance. Therefore the knowledge about the usage of resources is crucial in order to determine the values of the performance

model parameters. Each parameter is expressed in different units depending on the target resource. For example, the request of CPU by a software component can be expressed as the number of elementary operations that such a component has to execute on the CPU to be completed.

Once built, a performance model has to be solved in order to get values of performance indices. Analytical methods and simulation techniques are used to evaluate performance models and get values of performance indices. These can be expressed either as mean values or as probability distribution functions or as any probabilistic feature of the system component of interest. These values have to be compared to performance requirements in order to validate the current software design.

As for any non-functional attribute, performance does not have to be validated all over the system for all the system services, components, etc. Non-functional validation makes sense only when explicit non-functional requirements have been elicited and quantified for the system under development. Performance requirements usually concern only part of the system, such as certain software subsystems or certain behavioral scenarios, whereas other parts do not need to satisfy any performance requirements. Therefore detailed performance models should be built and solved only for critical parts of the developed system, whereas more abstract representations of non-critical parts can suffice to obtain useful results.

1.3 System vs. Software Performance Analysis

Originally in computing environments performance was associated to the elaboration capabilities of a certain hardware platform under a certain load given by the software system. The software was considered as a numerical (even complex) parameter of a performance model. The stochastic behavior of software was simply synthesized into a certain workload, so losing trace of its (static and dynamic) structure.

Performance validation conducted in this way was (and today still is) known as *system performance*, in that the obtained indices were referring to the system as a parameterized computing platform model. Without an explicit modeling of the software structure and logics, once solved such a performance model, the prevalent corrective actions that can be suggested mainly concern the hardware platform and its load. For example, to relieve an overloaded CPU it is usually suggested to increase the multiplicity of the CPU or, in the best case, to deviate part of its load (through a load balancing system) toward less stressed CPUs.

Software performance was born as a discipline at the beginning of the 1990s, when the complexity of software systems had dramatically increased and software could not be any more synthesized as a numerical parameter of a performance model. At that point software gained the dignity of a first-class entity in the field of performance modeling and analysis, and new notations to explicitly express this component started to be introduced.

The main effect of this explicit modeling of software structure and logics is that new types of solutions, which could not be expressed in a system performance set-

ting, become available in the hands of performance analysts. For example, an overloaded CPU can also be relieved by splitting a software component that runs on the CPU in two components and deploy one of the two components on another computing site. This is what we call a software performance solution, which differs from a system performance solution because it introduces a change in the software rather than the system hardware.

Nowadays a software performance approach is largely preferred to a system performance one, due to the complexity of software systems. Software solutions, in fact, contribute to effectively manage (and possibly decrease) such a complexity by searching for software alternatives that better exploit the underlying platform. Therefore, when a performance problem is evidenced, feasible software solutions should always be looked for first, because they are usually less expensive and contribute to act on software complexity. Moreover, such solutions survive in case of software porting, opposite to system solutions that are linked to a specific platform. Obviously there are situations in which no software solution effectively removes a performance problem. In these cases, as an ultimate possibility, a system solution must be adopted, if feasible.

At the beginning of the 1990s Connie U. Smith and other researchers introduced the term Software Performance Engineering (SPE) as a systematic approach to model and analyze software performance all along the software lifecycle.

In their seminal book [106], Smith and Williams have described their approach to SPE, which is based on an explicit software performance model notation to take into account the possibility of software solutions. Their methodology (further updated and refined in [110]) 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;[3] the second is based on QN models and represents the system platform, including hardware components and software deployment. The analysis of the software model gives information on 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. The solution of such a parameterized model provides insights about the performance of software and hardware together, and the explicit modeling of software (through Execution Graphs) helps one to find software solutions to performance problems.

The SPE definition has been a turning point in the management of performance in computing environments. Today it still remains a reference point for practitioners in the field. Together with the introduction, at the end of the 1990s, of approaches that automatically generate performance models, it represents the main breakthrough of the research in this field. Both have opened great perspectives for the software performance to become daily practice in the software engineers' activities.

[3] EGs were in fact created as a notation to model the software dynamics and the resource demands that the software makes to the underlying platform (see Chap. 3 for a description of Execution Graphs).

Chapter 2
Software Modeling Notations

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, see Fig. 2.1. The dynamics of a software system concerns its behavior at run time. There exist many notations to describe either the statics or the dynamics of a software system. This chapter focuses on notations that allow for the behavior description since performance is an attribute of the system dynamics. As discussed in Chap. 1, the system behavior is the necessary but not sufficient condition to carry out performance analysis of a system. In fact, the behavioral description of the system has to be enriched by additional information such as system operational profiles and service demands of the provided functionalities (see Chap. 3).

This chapter is divided into two parts: *(i)* basic notations historically introduced by computer scientists to model software systems, where Automata [73], Process Algebras [84], Petri Nets [98] and Message Sequence Charts [104] are briefly reviewed; *(ii)* Unified Modeling Language [87] that has become a de facto standard in modeling complex software systems.

Finally, examples of software system modeling by using the presented notations are presented along all the chapter.

2.1 Basic Notations

In this section classical notations to describe software system behavior are briefly reviewed. In particular, Automata, Process Algebras, Petri Nets and Message Sequence Charts are presented. All such notations are formally defined in terms of syntax and semantics and hence they are not ambiguous. To simplify the use of these notations in software modeling, for each of them the model of the same simple software system is presented. The chosen system is the XML Translator (XT) described below.

XML Translator—The XML Translator (XT) automatically builds an XML document from a text document with respect to a given XML schema [118]. The

V. Cortellessa et al., *Model-Based Software Performance Analysis*,
DOI 10.1007/978-3-642-13621-4_2, © Springer-Verlag Berlin Heidelberg 2011

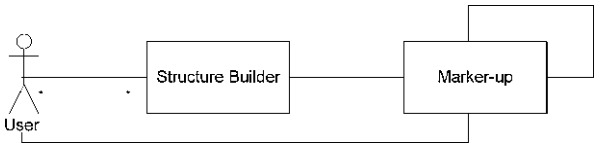


Fig. 2.1 Static description of XT system

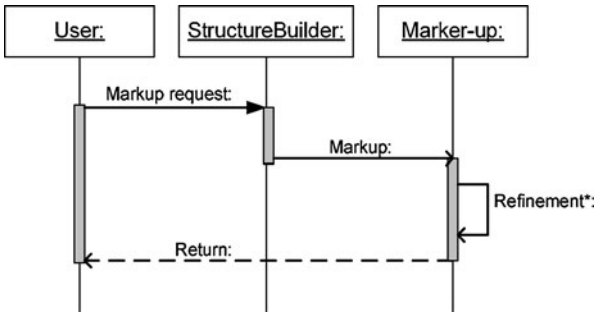


Fig. 2.2 Behavioral description of XT system

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 [117] 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.

From the previous description of the system two distinct software modules are identified:

- 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 it out 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 does emphasize most of the useful information under certain heuristic conditions.

A static description of XT system is shown in Fig. 2.1, whereas its behavior is defined by means of the UML sequence diagram [86] in Fig. 2.2, which shows that all the interactions among XT components are asynchronous.

2.1.1 Automata

Automata [73] is a simple mathematical and expressive formalism that allows one to model cooperation and synchronization between subsystems, concurrent and not. By means of this formalism a system is modeled as a set of states and its behavior is described by transitions between them, triggered by some input (named input symbol).

Automata is the first notation introduced to model software dynamics. All the other derived formalisms modify some of its characteristics depending on the particular modeling needs.

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 [73].

It is possible to associate a direct labeled graph to an automaton, called State Transition Graph (or State Transition Diagram), where nodes represent the states and labeled edges represent transitions of the automata triggered by the input symbols associated to the edges.

The automata formalism is compositional since automata can 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 have: *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; and *stochastic automata* which are non-deterministic automata where the next state of the system is determined by a probabilistic value associated to each possibility.

An Automaton for the XML Translator System

The state transition graph of the XML Translator automaton is shown in Fig. 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 $\langle q_0, q'_0 \rangle$ 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, their behavior is separately specified in Fig. 2.3(a) and (b), respectively.

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.

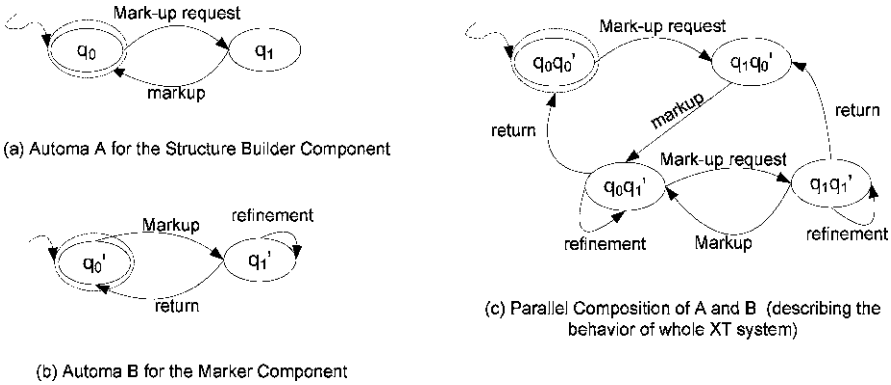


Fig. 2.3 State transition graph for the XML Translator automaton

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.

2.1.2 Process Algebras

Process Algebras, such as CCS [84] and CSP [72], 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 modeled in terms of the interactions of their subsystems). The semantics of these calculi is usually defined in terms of Labeled Transition Systems (LTS) following the structural operating semantics approach. Moreover, the Process Algebra formalism is used to detect undesirable properties and to derive desirable properties of a system specification. Notably, process algebra specifications can be used to verify that a system displays the desired external behavior, meaning that for each input the correct output is produced.[1]

Process Algebras can describe systems at different levels of abstraction. Many notations 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"

[1] More details of a specific class of Process Algebras, namely Stochastic Process Algebras, will be provided in Chap. 3.

```
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
```

Fig. 2.4 Process algebra model for the XML Translator

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 two further processes representing two queues are introduced. 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 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 these processes. Figure 2.4 shows the specification of the XML Translator system with three concurrent users and queues of capacity three, modeled by using the TIPP Process Algebra [68]. 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 [98] is defined by a set of *places*, a set of *transitions*, an *input function* relating places to 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.[2]

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 happening 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.

[2] More details of a specific class of Petri Nets, namely Stochastic Petri Nets, will be provided in Chap. 3.

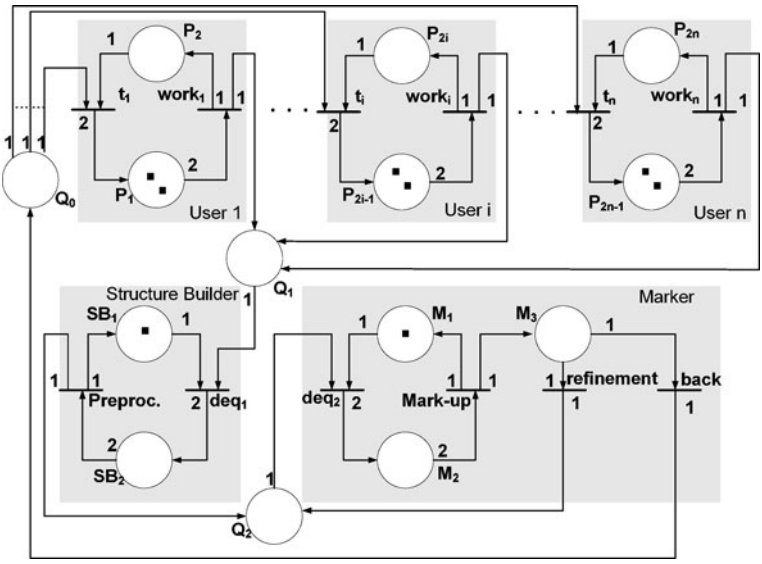


Fig. 2.5 Petri Net model of the XML Translator

Similarly, the two system components are modeled by the corresponding subnets 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 preprocessing 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. Similarly, the Marker component has one place M_3 and two transitions *refinement* and *back* needed to model the decision to refine or to send back to the users the result of the work. Places labeled Q_0, Q_1, Q_2 model the queues for asynchronous communication. Dynamically, the user requests to 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.

2.1.4 Message Sequence Charts

Message Sequence Charts (MSC) is a language specified by the International Telecommunication Union (ITU) in [104] to describe the interaction among a number of independent message-passing instances for example components, objects or processes, or between instances and the environment. It is a scenario language that describes the messages sent, messages received, and the local events, together with the ordering between them.

MSC supports complete and incomplete specifications and it can be used at different levels of abstraction. It allows for developing structured design since simple

scenarios described by basic MSC can be combined to form more complete specifications by means of high-level MSC.

MSC is also a graphical language which specifies 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. Different arrow heads are used to model synchronous and asynchronous interactions: full arrow heads model synchronous communications, whereas half arrow heads represent asynchronous ones.

MSC provides some other specific capabilities:

- It allows for expressing restrictions on transmitted data values and on the timing of events.
- It allows the modeling of conditions on the state of each component, before interactions occur, by means of the setting condition facility. A setting condition sets/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.
- MSC allows the representation of the interface of the modeled system with the environment, through its gate facilities. 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 into the MSC frame it will correspond to an output flow.

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

A Message Sequence Chart for the XML Translator

The dynamics of the XML Translator is very simple. Its description by MSC is given in Fig. 2.6. In the figure, the three vertical lines represent the lifetime of the User and of the *StructureBuilder* and *Marker-up* components. The User lifeline is outside of the MSC frame since she is external to the system.

The user asks to mark up a text to *StructureBuilder* that, in turn, during its execution, invokes *Marker-up*. When the original text is marked-up the control returns to the user. All the interactions in the MSC are synchronous as denoted by the full heads of the arrows modeling the interactions.

In the figure there are two gates, In_gate and Out_gate, that refer respectively to an input stream, the requests, and to an output stream, the elaboration results. They model inputs and outputs of the software system: XML Translator receives service requests from (external) users and it gives back the marked-up text.

No setting condition facilities appear in the figure since no state conditions on XML Translator components need to be considered.

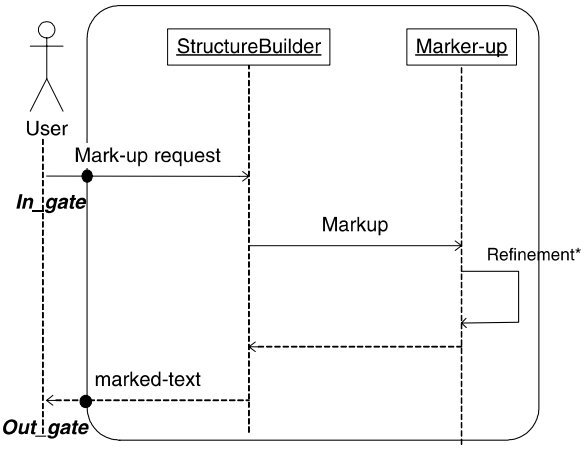


Fig. 2.6 Message sequence chart for the XML Translator

2.2 Unified Modeling Language

All the above discussed notations are formal specification languages so they have an exact semantics. Unfortunately, modeling real and complex software systems with such notations turns out to be very complex. This is the main reason for abandoning such notations in modern software systems modeling in favor of less precisely defined formalisms like the Unified Modeling Language [86].

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. Many of the basic notations describing software dynamics, such as Automata, Petri Nets, Message Sequence Charts, etc., have been the source of inspiration for UML diagrams.

UML diagrams allow the description of systems either statically or dynamically in an object-oriented style. 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.

The system structure, instead, can be described by component diagram and class diagram. The first one describes a more abstract view of the software system as an assembly of software components or subsystems; the second one provides a less abstract view that describes how the software system will be structured in an object-oriented paradigm identifying classes and relationships among them. The class diagram imposes a structure that should be respected in the subsequent object-oriented implementation of the system. Finally, UML allows the description of deployment

of software modules (components or subsystems) to hardware nodes by means of the deployment diagram.

Recently OMG has improved the UML notation releasing a new version of the language, UML 2.0 [87], which strengthens 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).

A complete treatment of UML is out of the scope of this book. For a detailed presentation of UML please refer to [51, 87]. In the following, a brief description of the diagrams commonly used in the software performance analysis field is reported. UML-based software performance analysis approaches use both diagrams describing the statics of the analyzed software system and the ones describing its behavior. For this reason, component and deployment diagrams are also surveyed hereafter.

To illustrate the UML 2.0 modeling diagrams, an e-commerce system is briefly introduced in the following paragraph. In particular, the description concerns how the system is decomposed into components and how they interact to accomplish the system functionalities. The purpose of this modeling is twofold: on one hand it aims at showing how to use UML 2.0 diagrams for software modeling, on the other hand it fully present the case study that will be used in Chap. 5 to show the selected model transformation methodologies at work.

2.2.1 E-commerce System

In the electronic commerce system [58], there is a supplier that publishes his catalog on the web. The catalog can be visioned by registered and unregistered users. Registered users become customers for the supplier. The supplier accepts customer orders and delivers the ordered items maintaining all the relevant data. He needs to maintain information on data, on the catalog and on the orders purchased by his customers. Each registered customer has a cart where 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.

Figure 2.7 shows the system structure at a high level of detail. In a first analysis, some databases, Customer DB, Cart DB, Order DB, etc., and four process type, CustomerProcess, SupplierProcess, etc., are identified. For each database involved, there is a server that permits one to communicate with it. The interactions with these servers are asynchronous. Each customer has associated an individual CustomerProcess. If the customer is not registered, the process allows the browsing of the catalog only, whereas if he is registered, the process activates all the functionalities the system provides for him, that is cart managing, order placement, etc. All the functionalities provided by the system to the supplier, instead, are coded in the SupplierProcess. The DeliveryOrderProcess and InvoiceProcess are processes that manage the activities to be executed to dispatch an order and to produce the corresponding invoice. In the system, one DeliveryOrderProcess instance and one InvoiceProcess instance are running, any time, for each order and for each invoice in process, respectively.

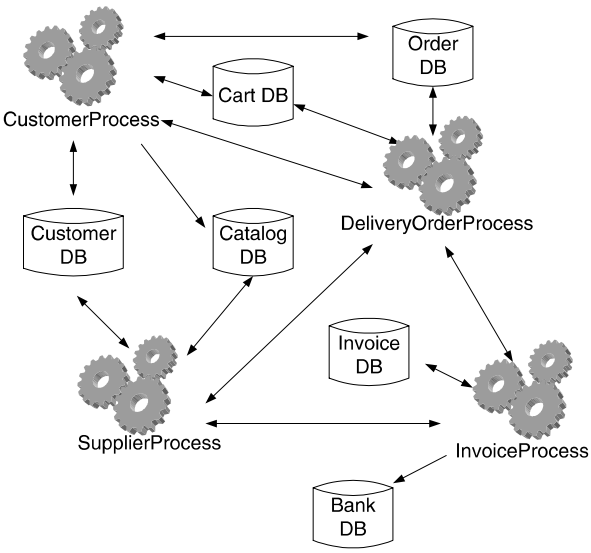


Fig. 2.7 SA components of the electronic commerce system

For the sake of presentation, in this book a simplified version of this system is considered. The focus considered is on a subset of customer functionalities. In particular, catalog browsing, cart browsing, item insertion and deletion to/from the cart, order placement are functionalities considered.

2.2.2 Use Case Diagram

Use case diagrams emphasize the interaction between users and the system. More precisely it identifies the system use cases, the actors (users) using the system and the associations between them. The system use cases is an high-level description of the functionalities provided by the system. Actors are entities external to the system that interacts with it; they can be either humans or other software systems. The associations describe which system functionalities an actor can invoke.

Figure 2.8 shows the use case diagram for the e-commerce system. There are three actors in the system: *Customer*, *Supplier* and *Bank*.

The e-commerce customer can browse both the cart (`BrowseCart`) and the catalog (`BrowseCatalog`), he can insert and delete items to/from the cart (`InsertItem` and `DeleteItem`, respectively). He can place an order and check its status (`PlaceOrder` and `BrowseOrderStatus`), and finally he can confirm the delivery (`ConfirmDelivery`) of the ordered items. The use case diagram also specifies that both `InsertItem` and `DeleteItem` utilize the `BrowseCart` in their execution.

The supplier maintains the catalog inserting new items from the catalog (`InsertNewItemInCatalogue`) and deleting from it the items no more sold

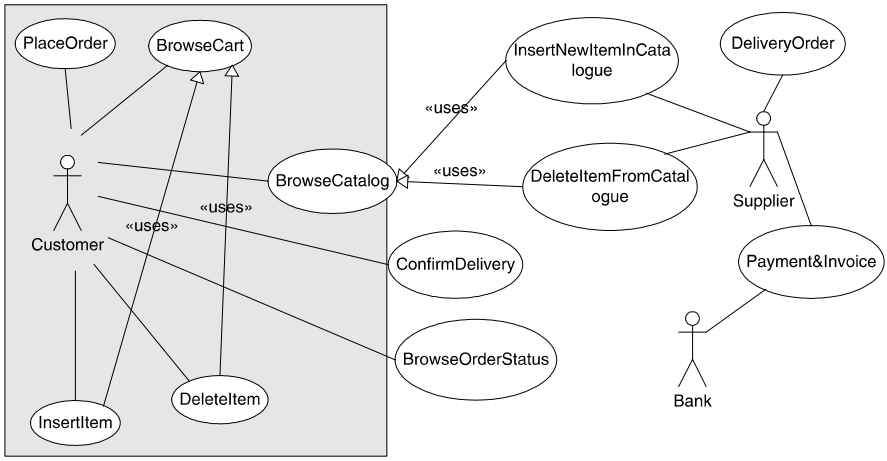


Fig. 2.8 E-commerce functionalities

(`DeleteItemFromCatalogue`). Moreover, he processes the orders by delivering the items sold (`DeliveryOrder`) and producing the invoice after the customer has payed (`Payment&Invoice`). In this last use case, the bank actor is in charge of the money transfer from the customer bank account to the supplier one. Again, the use case diagram specifies that both `InsertNewItemInCatalogue` and `DeleteItemFromCatalogue` use functionalities that use the `BrowseCatalogue` in their execution.

For the sake of presentation, in this book a simplified version of this system is considered. The focus considered is on the customer view by considering only the software system functionalities reported in the dashed box of the figure. The choice has been driven by the fact that the focus of this book is performance aspects; hence the customers are the system actors producing more workload to the system. Thus, the use cases critical from a performance perspective are those accessed from them.

2.2.3 Component Diagram

Component diagrams specify the decomposition of the system in software modules/components by highlighting their dependencies in terms of required and provided interfaces. A component is a modular unit with well-defined interfaces; it can be replaced by any module/component having compatible interfaces.

An interface defines a set of operations involving the component. Provided interfaces formally define services that the component provides to other components, while required interfaces define the services that the component requires from other system component to properly work.

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

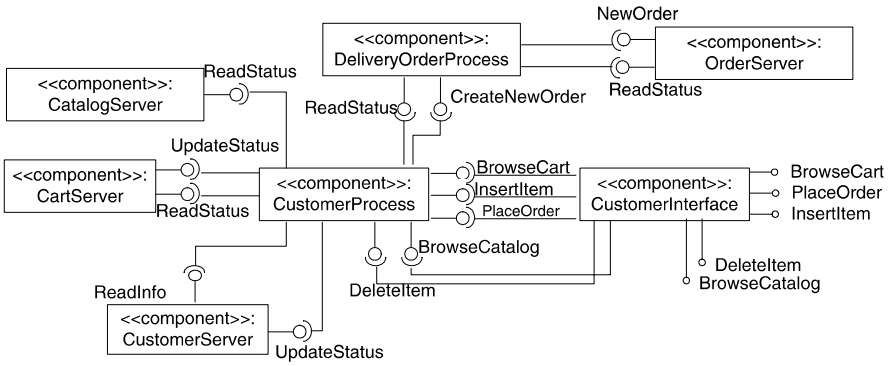


Fig. 2.9 Component diagram of the considered portion of the e-commerce system

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, or "black-box" view, exhibits only the public properties and operations which are encapsulated in the provided and required interfaces. The connections between components is specified by dependencies among their interfaces. The internal view, or the "white-box" view, shows how the components realize their functionalities.

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. Two components are connected together if the required interface(s) of the first one is connected to the provided interface(s) of the second one.

Figure 2.9 shows the UML 2.0 component diagram for the considered portion of the e-commerce system. This diagram highlights the software components and their required and provided interfaces by showing the external view of the system. In the diagram the component interconnections through required and provided interfaces are represented.

The (sub)system is composed by six components. Four of them, `OrderServer`, `CatalogServer`, `CartServer`, `CustomerServer`, interact with the corresponding databases by allowing for the insertion, deletion, reading and updating of data in the DBs. These components use asynchronous communications; hence all the interfaces they provide contain asynchronous operations. The remaining three components, namely `CustomerInterface`, `CustomerProcess` and `DeliveryOrderProcess`, provide the customer with the GUI to interact with the system, the back end process that manages all his requests made by the GUI and the business process to manage the order delivery, respectively. These components interact asynchronously with the DB servers and synchronously among them. Thus the interfaces they provide reflect such properties.

Finally, since the component diagram in the figure only describes a portion of the entire e-commerce system, it does not contain the components and the interfaces not

involved in the considered use cases. For example there is not the component that allows the supplier to delete or insert an item in the catalog. Similarly, there is no `CatalogServer` provided interfaces that manage the operations on the corresponding database.

2.2.4 Interaction Diagram

Interaction diagrams are a common mechanism for describing systems, at different levels of detail, in a way comprehensible to both software designers, potential end users, and stakeholders of the system. Typically interaction diagrams do not describe the whole system behavior but selected execution traces. There are normally other legal and possible interactions that are not contained within the drawn diagrams.

Originally, interaction diagrams represented system objects and how they interact. In UML 2.0, such diagrams describe the system by means of participants that can be system modules in a different level of abstraction such as for example subsystems, components, objects and so on, and by how they interact with each other to accomplish a task.

In UML terminology, interactions are units of behavior that focus on the observable exchange of information between elements (such as for example objects) in form of messages.

UML 2.0 defines four different interaction diagrams: sequence diagram, communication diagram, interaction overview diagram and timing diagram. Among these diagrams, only sequence diagrams are well known and widely used. For this reason in the following only sequence diagrams are detailed, while for more information on the other diagrams, interested readers can refer to the OMG specification [87].

Analogously to MSC, sequence diagrams emphasize the lifetime of each software modules and when interaction between them occurs. Sequence diagrams also allow the specification of conditions on message sending, the use of iteration marking that identifies multiple sending of a message to receiver modules, and the definition of the type of communication, synchronous or asynchronous.

The main evolutions in the interaction specification that 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.

An interaction fragment 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. Interacting elements are represented by means of lifelines and messages through arrows. All the elements in the rectangle are part of the modeled system. Entities external to the system can make requests to it. The request is an interaction among the actor using the system and the system itself, hence it is represented by an arrow. Same reasoning can be

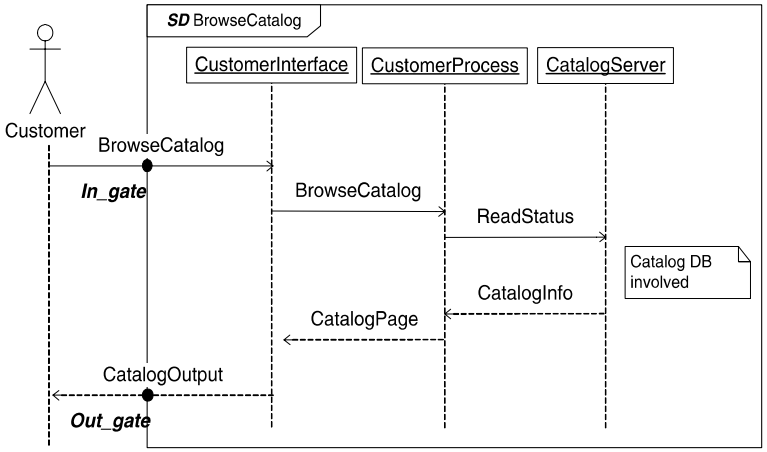


Fig. 2.10 Browse catalog interaction

made for the message returning to the actor. The point where these arrows cross the rectangle represents the gates that are connection points for relating a message outside an interaction fragment with a message inside the interaction fragment.

For an example of interaction fragment described by a sequence diagram please refer to Fig. 2.10 where is reported the sequence diagram for the `BrowseCatalogue` use case.

In Fig. 2.10, the interaction fragment the diagram represents is BrowseCatalog as indicated in the pentagon at the upper left corner of the rectangle. The sequence diagram represents how the system implements the browse catalog use case in terms of component interactions. The lifelines represent the software components identified in the component diagram of Fig. 2.9. The customer, that is external to the system, requests for a catalog browsing and waits for the required information. The points of the rectangle crossed by the arrows modeling the request and the message back to the customer defines two gates (in_gate and out_gate in the figure).

More interaction fragments can be combined by means of interaction operations. The obtained combined interaction fragment builds up a `CombinedFragment`. The semantics of a CombinedFragment depends on the semantics of its interaction operator. The interaction operators are:

- *Alternative*—the combined interaction fragments represent behavior alternatives;
- *Option*—the combined fragment represents a choice between the sole operand behavior or nothing;
- *Break*—the combined fragment designates a breaking scenario. The operand is a scenario performed instead of the remainder of the enclosing interaction fragment;
- *Parallel*—it represents a parallel merge between the behaviors of the combined interaction fragments;
- *Loop*—it designates the interaction fragment to iterate for a number of times;

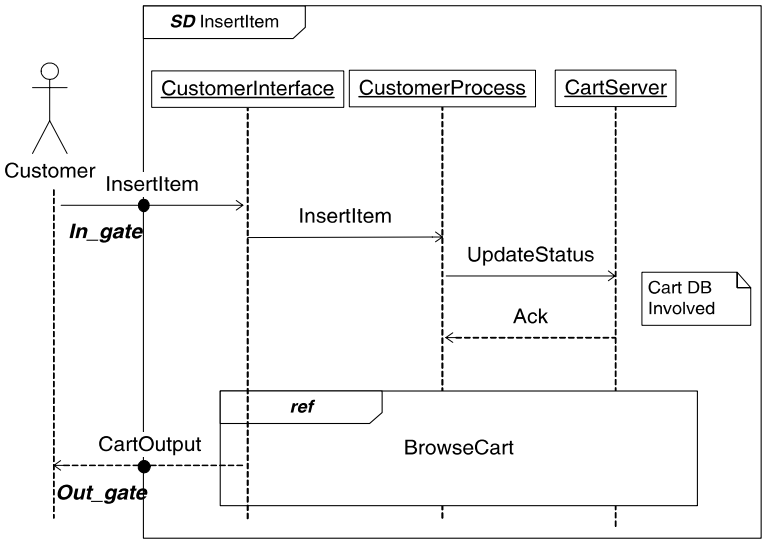


Fig. 2.11 Insert Item scenario

- *Weak/Strict Sequencing*—the combined fragment represents a weak/strict sequencing between the behaviors of the operands;
- *Negative*—it defines invalid traces;
- *Critical Region*—it models a critical region;
- *Assertion*—it represents an assertion;
- *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 ignored if they appear in a corresponding execution. An alternative interpretation of Ignore is 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, that is equivalent to specifying every other message as ignored.

How the system components interact to provide the use cases in the diagram is described by scenarios, one for each use case. From Fig. 2.10 to Fig. 2.13 the scenarios for the use case considered for the e-commerce system are reported. These scenarios are modeled by means of sequence diagrams, except for the BrowseCart use case that is modeled through an activity diagram in Fig. 2.14.

As for the sequence diagram for the BrowseCatalog use case in Fig. 2.10, in the sequence diagrams that follow the boxes at the top of the diagram represents component instances and the arrows among lifelines represent component interactions.

Figures 2.11 and 2.12 report the sequence diagrams modeling the system behavior for the InsertItem and DeleteItem use cases. In modeling these scenarios, the `uses` dependency in the use case diagram, see Fig. 2.8, must be considered. Such dependency indicates that the InsertItem and DeleteItem use cases make use of the

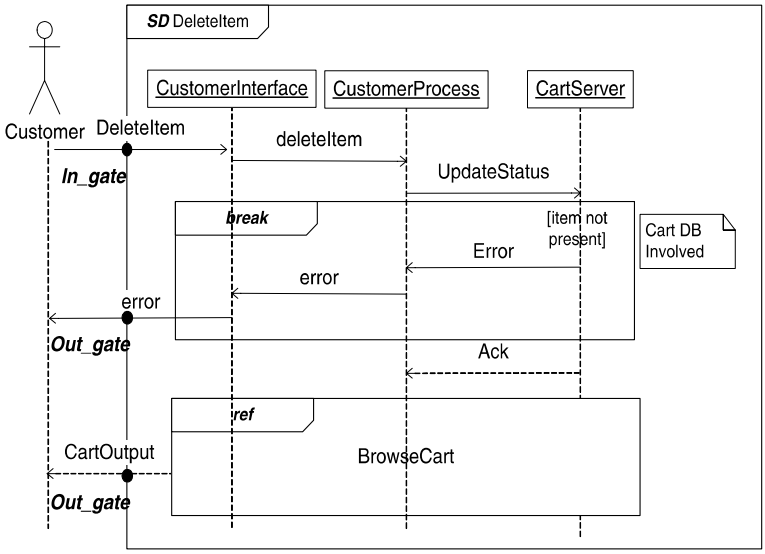


Fig. 2.12 Delete Item scenario

BrowseCatalog use case in their execution. In the sequence diagrams this leads to the use of the `ref` interaction operator referring to the BrowseCart scenario.

Moreover, an item can be removed from the cart only if it is in the cart, otherwise an error occurs. To model the error, the `break` operator is used in the DeleteItem sequence diagram (Fig. 2.12).

The final scenario is the PlaceOrder sequence diagram in Fig. 2.13. It is composed of several (nested) combined fragments. In particular, it shows an example of alternative behaviors by means of the `alt` operator. A customer asks for an order placing and, first of all, the CustomerProcess reads the Cart status. If the Cart is empty the order cannot be placed, see the second behavior alternative in the alt combined fragment. Otherwise the CustomerProcess proceeds in collecting the customer information, such as for example his mail address, and creates a new order in the Order DB. Finally, it empties the customer cart. In this scenario it is assumed that all the information the customer must provide to place an order are collected before the PlaceOrder request that corresponds to the finally submission from the customer. The payment procedure is encompassed in a different use case. For example, if the customer decides to pay by a credit card, the Payment&Invoice use case, see Fig. 2.8, will manage the bank transfer.

2.2.5 Activity Diagram

Activity modeling emphasizes the sequence and conditions for coordinating (lower-level) behaviors. The behaviors can be an activity or an action. The diagram that details an activity is called activity diagram and it can describe a control flow, an object flow or both.

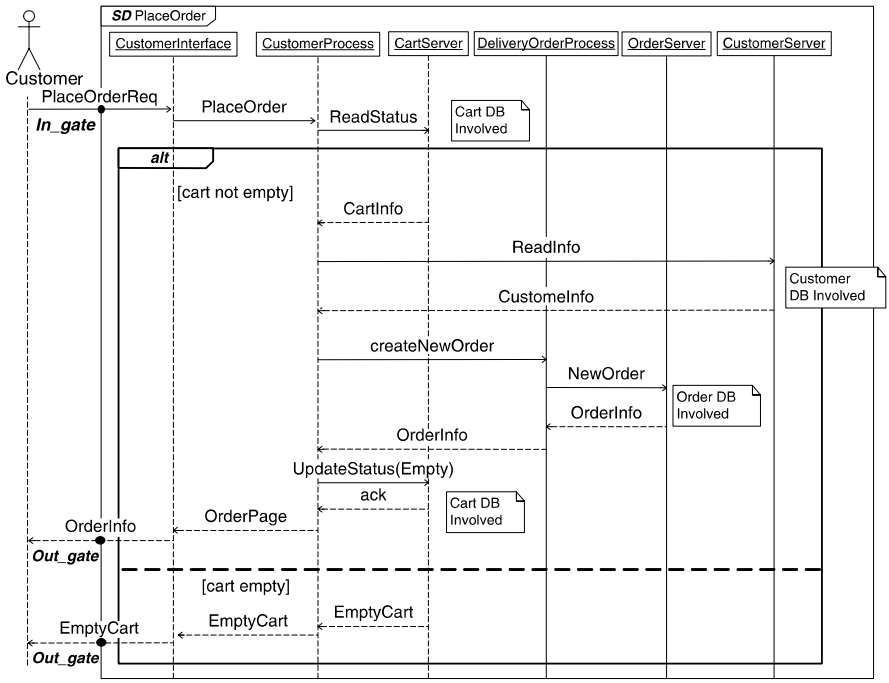


Fig. 2.13 Place Order scenario

Activities may be applied in several contexts, for example, in organizational modeling for business process, workflow and in information system modeling to specify system level processes.

In UML 2, Activities are redesigned with a Petri Net-like semantics instead of state machines'. The semantics of activities is based on token flow. A token contains an object, datum, or locus of control, and is present in the activity diagram at a particular node. A node may begin execution when specified conditions on its input tokens are satisfied; the conditions depend on the kind of node. When a node begins execution, tokens are accepted from some or all of its input edges and a token is placed on the node. When a node completes execution, a token is removed from the node and tokens are offered to some or all of its output edges.

An activity specifies the coordination of executions of subordinate behaviors, using a control and data flow model. An activity execution may be initiated because other activities in the model finish executing, because objects and data become available, or because events occur external to the flow. The flow of execution is modeled as activity nodes connected by activity edges. A node can be the execution of a subordinate behavior or manipulation of object contents. Activity nodes also include flow-of control constructs, such as synchronization, decision, and concurrency control. Activities may form invocation hierarchies invoking other activities, ultimately resolving to individual actions.

Actions have no further decomposition in the activity containing them, since they represent a single step within an activity. However, the execution of a single action may induce the execution of many other actions. In UML 2, it is possible to specify local pre- and post-conditions on actions that are constraints that should hold when the execution starts and completes, respectively. They hold only at the point in the flow where they are specified.

An activity edge is denoted by an open arrowhead line connecting two activity nodes. If the edge has a name, it is annotated near the arrow. An object flow is an activity edge that can have objects or data passing along it. Object flows are introduced to model the flow of data and objects in an activity. Object flows have been introduced in UML 2.

There are three kinds of activity node: action nodes, object nodes, and control nodes.

An action node denotes an action or an activity in case the behavior it represents is complex. Action nodes are represented by round-cornered rectangles. The name of the action or other description of it may appear in the rectangle.

An object node is an abstract activity node that is part of an object flow definition in an activity. An object node indicates an instance of a specific classifier, possibly in a particular state, may be available at a certain point in the activity. Object nodes can be used in a variety of ways, depending on where objects are flowing from and to. Object nodes may only contain at run-time values that conform to the type of the object node, in the state or states specified, if any. Object nodes are denoted by rectangles. A name labeling the node is placed inside the symbol, where the name indicates the type of the node, or the name and type of the node.

A control node is an activity node used to coordinate the flows between other nodes. It can be an initial node, two types of final node, fork node, join node, decision node, and merge node.

Initial node—An initial node is a control node at which the flow starts when the activity is invoked. An activity may have more than one initial node. An initial node is a starting point for executing an activity. If an activity has more than one initial node, then invoking the activity starts multiple flows, one at each initial node. Initial nodes are represented by a solid circle.

Final node—A final node is an abstract control node at which a flow in an activity stops. There are two kinds of final node: activity final and flow final. An activity final node stops all flows in the activity and is denoted by a solid circle with a hollow circle. A flow final node instead terminates a particular flow. It is represented by a circle with a cross inside.

Decision node—A decision node is a control node that chooses one among the outgoing flows. A decision node has one incoming edge and multiple outgoing activity edges. Most commonly, guards of the outgoing edges are evaluated to determine which edge should be traversed. The order in which guards are evaluated is not defined. A decision node is graphically represented by a diamond-shaped symbol.

Fork node—A fork node is a control node that splits a flow into multiple concurrent flows. A fork node has one incoming edge and multiple outgoing edges. The notation for a fork node is simply a line segment. In usage, however, the fork node must have a single activity edge entering it, and two or more edges leaving it.

Join node—A join node is a control node that synchronizes multiple flows. The notation for a join node is a line segment. The join node must have one or more activity edges entering it, and only one edge leaving it.

Loop node—A loop node is a structured activity node that represents a loop with setup, test, and body sections. The setup section is executed once on entry into the loop, and the test and body sections are executed repeatedly until the test produces a false value. The results of the final execution of the test or body are available after completion of execution of the loop. Loop nodes are introduced in UML 2 to provide a structured way to represent iteration. They do not have a particular graphical representation.

Merge node—A merge node is a control node that brings together multiple alternative flows. It is not used to synchronize concurrent flows but to accept one among several alternate flows. A merge node has multiple incoming edges and a single outgoing edge. The notation for a merge node is a diamond-shaped symbol. The functionalities of a merge node and a decision node can be combined. Merge nodes are introduced to support bringing multiple flows together in activities. For example, if a decision is used after a fork, the two flows coming out of the decision need to be merged into one before going to a join; otherwise, the join will wait for both flows, only one of which will arrive.

As already said at the beginning of this section, activity modeling emphasizes the sequence and conditions for coordinating lower-level behaviors, rather than which entities own those behaviors.

However, UML 2 provides mechanisms called activity partitions, that are a kind of activity grouping for identifying actions that have some characteristic in common. Partitions often correspond to organizational units in a business model or to software modules in a software model. In the first case the partitions indicates which organizational unit is responsible for the enclosing activities. In the second case, instead, they indicates which software system module (component, object, procedure, etc.) has to implement the contained activities.

Activity partition are indicated with two, usually parallel lines called swim lane, either horizontal or vertical, and a name labeling the partition. Any activity nodes and edge placed between these lines is considered to be contained within the partition. Diagrams can also be partitioned multidimensionally, where, each swim cell is an intersection of multiple partitions. When activities are considered to occur outside the domain of a particular model (for example outside the software system under development in case of software development process), the partition can be labeled with the keyword external.

In Fig. 2.14 the scenario related to BrowseCart use case is modeled through an activity diagram. The diagram is divided into several vertical stripes one for each software module (component) involved in the scenario. The first swim lane is stereotyped[3] with «external» indicating that Customer is an entity external to

[3]For stereotypes see Sect. 2.2.8.

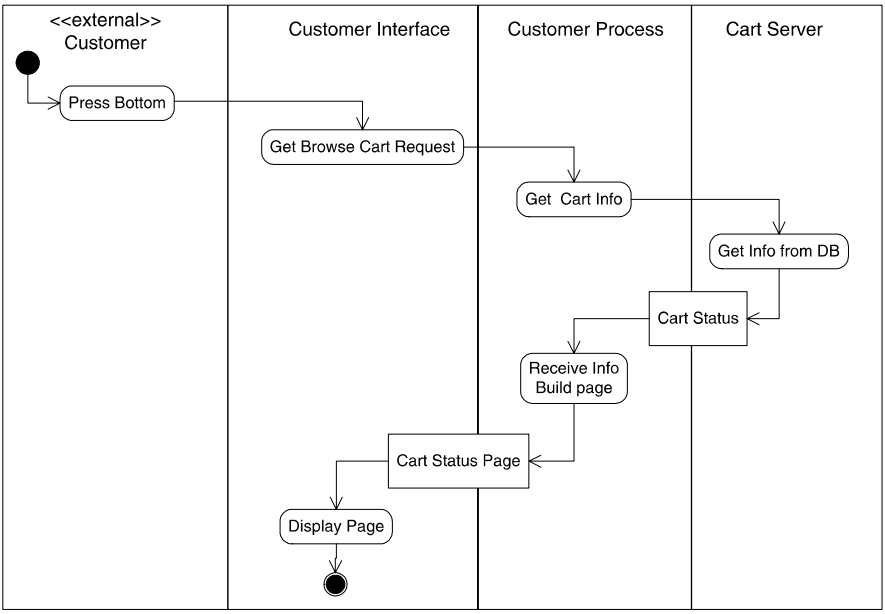


Fig. 2.14 Browse Cart scenario modeled by using UML 2.0 activity diagram

e-commerce system.[4] The ellipses in the stripes represent the activities performed by the software module indicated at the top. The flow of execution is represented by the arrows while the boxes on the stripes borderlines represent the data the modules pass to each other in their communications. The scenario starts when the customer asks for cart status by pressing the corresponding bottom, or surfing the link, in the web page.

Hence a request of cart browsing arrives to the system. The data are processed, the Cart status is retrieved from the DB and then formatted in the output page that is finally displayed to the customer. The components listed at the top of the diagram are responsible for the execution of such activities, for example the component CustomerInterface performs the action BrowseCartProcessing. Such responsibilities are represented in the mapping between the activities and the swim lanes as shown in the diagram.

In the diagram the activities model sub-behaviors implemented by some operations provided, in their public interface, by the software components responsible for such activities, as the swim lanes specify.

[4]This modeling respects the modeling in the use case diagram where it is specified that the customer is an actor of the e-commerce system.

2.2.6 State Machine Diagram

The state machine formalism described in UML 2 is an object-based variant of Harel statecharts [64]. They can be used for modeling discrete behavior through finite state transition systems, that is, a behavioral state machine, or to express the usage protocol of part of a system, that is, protocol state machines.

Behavioral state machines specify the behavior of various model elements at different level of detail. For example, they can model the dynamics of the whole system, part of it, i.e. subsystem, or individual entities like, component instances, class instances.

These diagrams show the state space of a given computational unit, the events that cause a transition from one state to another, and the resulting actions. The entity behavior is modeled as a graph of state nodes interconnected by one or more transition that are triggered by the dispatching of series of event occurrences. In each state, the state machine can execute a series of activities associated with elements of the state machine.

States and other types of vertices, called pseudostates, in the state machine graph are rendered by appropriate symbols, while transitions are generally rendered by directed arcs that connect them. A pseudostate is an abstraction that encompasses different types of transient vertices in the state machine graph. Pseudostates are typically used to connect multiple transitions into more complex state transitions paths. For example, by combining a transition entering a fork pseudostate with a set of transitions exiting the fork pseudostate, we get a compound transition that leads to a set of orthogonal target states. The pseudostate notation depends on its type.

The commonly used (pseudo)states in a state machine diagram are:

Final state—A special kind of state signifying that the enclosing region is completed. A final state is shown as a circle surrounding a small solid filled circle.

Initial pseudostate—This represents a default vertex that is the source for a single transition to the default state of a composite state. There can be at most one initial vertex. The outgoing transition from the initial vertex may have a behavior, but not a trigger or a guard. An initial pseudostate is shown as a small solid filled circle.

Join vertices serve to merge several transitions that cannot have guards or triggers.

Fork vertices serve to split an incoming transition into two or more transitions terminating on different target vertices. The transitions outgoing a fork must not have guards or triggers.

The notation for a fork and join is a short heavy bar. The bar has one or more arrows from source states to the bar when representing a joint. The bar has one or more arrows from the bar to states, when representing a fork. A transition string may be shown near the bar.

Junction vertices are semantic-free vertices that are used to chain together multiple transitions. They are used to construct compound transition paths between states. For example, a junction can be used to converge multiple incoming transitions into a single outgoing transition representing a shared transition path. This is known as

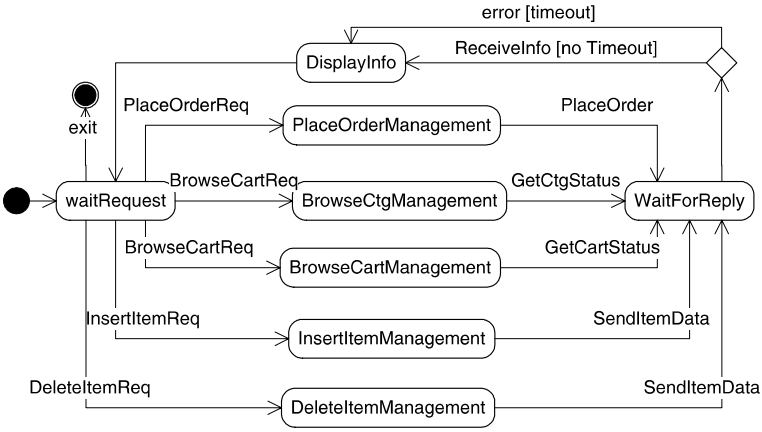


Fig. 2.15 State machine diagram for the CustomerInterface component

a merge. Conversely, they can be used to split an incoming transition into multiple outgoing transition segments with different guard conditions. This realizes a static conditional branch.

Choice vertices, when reached result in the dynamic evaluation of the guards of the triggers of their outgoing transitions. This realizes a dynamic conditional branch. It allows splitting transitions into multiple outgoing paths that the decision on which path to take may be a function of the results of prior performed actions. If more than one of the guards evaluates to true, an arbitrary one is selected. A choice pseudostate is shown as a diamond-shaped symbol.

In UML 2 we can distinguish three kinds of states: simple state, composite state, and submachine state. A simple state is a state that does not have substates. A composite state either contains one region or is decomposed into two or more orthogonal regions. Each region has a set of mutually exclusive disjoint subvertices and a set of transitions. Any state enclosed within a region of a composite state is called a substate of that composite state. A submachine state specifies the insertion of the specification of a submachine state machine. Submachine state is a decomposition mechanism that allows the factorization of common behaviors and their reuse.

For what concerns transitions, a transition can be external, internal or local: internal implies that the transition, if triggered, occurs without exiting or entering the source state. Thus, it does not cause a state change. Local implies that the transition, if triggered, will not exit the composite source state, but it will apply to any state within the composite state, and these will be exited and entered. External implies that the transition, if triggered, will exit the composite source state.

In Fig. 2.15 the state machine diagram for the CustomerInterface component is shown. CustomerInterface is the component at the customer side that allows the customer to access to e-commerce functionalities. When a CustomerInterface instance is running, it waits for requests to be forwarded toward the e-commerce provider side subsystem. Hence there is a transition, exiting the waitRequest state, for each

enabled functionality (e.g., insertItem) reaching an appropriate state (e.g., InsertItemManagement state) that models the work the component must do to properly format the data to be send (e.g., SendItemData). Then the component waits for reply from the provider. If the reply from remote system comes before the timeout finishes, the information received is displayed from the component otherwise an error page is showed to the customer. In both situations the component transits to the waitRequest state. The instance will stop to run when the customer exit the service.

2.2.7 Deployment Diagram

A UML 2 deployment diagram depicts a static view of the run-time system configuration, that is, the hardware used in system implementations, the components (and artifacts) deployed on the hardware, and the associations between them. Hence, the elements used in deployment diagrams are nodes, components/artifacts and associations among them. To be precise, a deployment is the allocation of an artifact or artifact instance to a deployment target. A component deployment is the deployment of one or more artifacts or artifact instances to a deployment target.

The hardware platform is modeled as a set of nodes and associations that represent (physical) communication paths between nodes. A node, depicted as a three-dimensional box, represents a computational unit, typically a single piece of hardware (a computer, network router, sensor, or PDA). Nodes are defined in a nested manner, and represent either hardware devices or software execution environments. The inner nodes indicate execution environments rather than hardware.

In UML 2.0, components are not placed in nodes. Instead artifacts and nodes are placed in nodes. An artifact may be a file, program, library, or data base. These artifacts implement collections of components.

An artifact is a product of the software development process. That may include, among others, process models (e.g. use case models, design models, etc.), source files, executables, design documents, prototypes. An artifact is denoted by a rectangle showing the artifact name, the artifact keyword and a document icon. A particular instance (or copy) of an artifact is deployed to a node instance. Artifacts may have composition associations to other artifacts that are nested within it.

Communication associations, often called connections, are depicted as lines connecting nodes. Dependencies between components are modeled as dashed arrows.

Deployment diagrams show the allocation of Artifacts to Nodes according to the Deployments defined between them. An alternative notation to containing the deployed artifacts within a deployment target symbol is to use a dependency labeled deploy that is drawn from the artifact to the deployment target.

In Fig. 2.16 we show the deployment diagram for the e-commerce application. There are a number of remote nodes, from *Remote Proc.1* to *Remote Proc.R* that represent the processors at the customer side. They are connected to the e-commerce web site through Internet that for modeling purposes is modeled as a separate deployment node. The access point for the customer requests at the server side is a

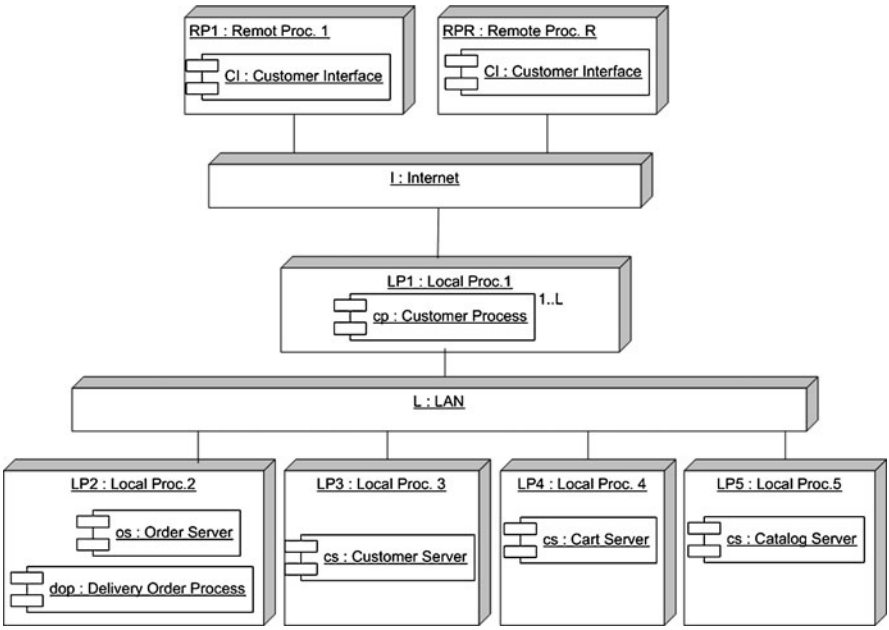


Fig. 2.16 Deployment diagram

CustomerProcess instance that is deployed on the *Local Proc.1* node. There will be one CustomerProcess instance for each connected costumer. The other software component instances are deployed on four different machines (from *Local Proc.2* to *Local Proc.5*) to distribute the load of the computation. The nodes the e-commerce component instances at the server side running on, are interconnected through a LAN network that is explicitly modeled with a deployment node.

2.2.8 Profiling UML

The UML is a general purpose, tool supported, and standardized modeling language. It is broadly applicable to different types of systems, domains, methods and processes since it does not contain domain-specific concepts. In case domain-specific concepts are necessary, it is possible to extend UML by introducing such concepts. There exist two different extension techniques, called heavyweight and lightweight, respectively.

The first one permits to add domain-specifics concepts by changing the UML definition. This can lead to the addition of new concepts that can be non-conforming to the standard UML or to incompatible changes to existing UML semantics/concepts. The resulting language is no more supported by standard UML tools hence one needs to re-implement them for the new language. For these reasons, this extension technique is inconvenient.

The lightweight extension, instead, specializes elements already present in UML in order to embed concepts of a specific domain. The obtained language is still conform to standard UML and the tool compatibility is maintained. The lightweight extension results in the definition of a UML Profile.

To specify a profile UML extensibility mechanisms are used. Such mechanisms are stereotypes, tagged values, constraints. Stereotypes create new elements from existing ones. A new element is an extension of an UML element having specific properties suitable for a particular domain. The specific properties are defined through tagged values that are attribute-value pairs. When a stereotype is used inside a model, the designer may specify a value either for all the tagged values the stereotype defines or for some of them. Finally, constraints are properties for specifying semantics and/or conditions that must be true at all time for the elements of a model.

At the end of the next chapter, in Sect. 3.8, some concepts of the UML Profile for Schedulability, Performance and Time (SPT) [85] are introduced. A more extensive description of SPT is provided in Sect. 7.4.1. SPT is used to annotate UML 1.x models with information allowing quantitative (i.e. performance), possibly predictive, analysis.

Chapter 3
Performance Modeling Notations

As outlined in Chap. 1, a major problem for stably embedding software performance modeling and analysis within the software lifecycle resides in the distance between notations for static and dynamic modeling of software (such as UML) and notations for modeling performance (such as Queueing Networks).

In Chap. 2 we have introduced the major notations for software modeling, in this chapter we introduce basic performance modeling notations, and in Chap. 5 we close the ideal path from software modeling to performance modeling by widely describing the existing approaches for automated generation of performance models from software models.

The performance indices that may be of interest for software systems mostly fall in the classical definitions of: end-to-end response time, throughput and utilization. However, in modern hardware/software systems new indices are emerging to model the critical performance aspects. For example, in a mobile network domain the consumption of a device battery is a major index of performance. Indeed many studies to deploy software applications that minimize this index are ongoing. Besides, as we will see in Chap. 5, the same performance indices may assume different meanings and can be expressed in different units, depending on the level of abstraction of the models.

A question may arise at this point from readers that are not familiar with performance analysis: "If all the performance notations are able to provide the desired indices, then why using different notations for performance modeling?". Somehow, a similar issue has been brought about some time ago: to conceive a unique language for software modeling, that is UML. The software performance community is still far from unifying languages and notations, although some recent efforts have been spent in the direction of building a performance ontology as a shared vocabulary of the domain (see Chap. 7).

Today the multiplicity of performance notations is, at the same time, a benefit and a detriment. On the positive side, multiple notations allow performance analysts to use the more appropriate one for each application domain and each specific model. On the negative side, different notations, which refer to different metamodels, may introduce different definitions (and semantics) for the same concept. Moreover, the

V. Cortellessa et al., *Model-Based Software Performance Analysis*,
DOI 10.1007/978-3-642-13621-4_3,

model parametrization step may also depend on the specific notation adopted. This implies that the results of performance analysis are often hard to compare if obtained with different model notations.

However, the co-existence of performance modeling notations is today a matter of fact; therefore, in this chapter we introduce the most common notations. Although it is out of the scope of this chapter to compare the different notations, their specific pros and cons will emerge within each description. In addition, note that we do not address here model solution issues that are the topic of Chap. 6.

The performance notations that we describe in this chapter are well described in the literature and many references can be found. In the following we provide a short and meaningful introduction to the notations, along with most relevant references. Although more sophisticated notations have been introduced, most of them build up over the basic notations that are described in this chapter.

The notations presented in this chapter are: Markov processes (Sect. 3.1), Queueing Networks (Sect. 3.2), Execution Graphs (Sect. 3.3), Layered Queueing Networks (Sect. 3.4), Stochastic Petri Nets (Sect. 3.5), Stochastic Process Algebras (Sect. 3.6), and Simulation Models (Sect. 3.7).

This chapter ends with an additional section on the UML Profile for Schedulability, Performance and Time (SPT). SPT is not an explicit performance modeling notation, but it is an extension of the UML notation to express concepts related to performance (and side domains). In Sect. 3.8 we only introduce the main stereotypes that are useful to understand the techniques introduced in Chap. 5. In Sect. 7.4.1 SPT is more extensively described (in its performance aspects) as a structured metamodel for software performance.

3.1 Markov Processes

Markov processes are stochastic processes for which the Markov property holds. The Markov property says that the conditional probability distribution of future states of the process only depends on the present state and not on any past state [113].

Markov processes play a central role in the quantitative analysis of systems. They can be used as a primary notation for the dynamics of a software system. However, even when they are not explicitly used as a primary notation, the analytical solutions of various classes of performance models (such as Queueing Networks and Stochastic Petri Nets) often rely on such types of stochastic process. This is the reason for introducing in this chapter Markov processes before the other performance notations.[1]

A stochastic process is a family of random variables $X = \{X(t):\ t \in T\}$, where $X(t) : T \times \Omega \to S$ is defined on a probability space Ω, an index set T (usually

[1] Several seminal books have been published on stochastic and Markov processes (see, for example, [49]). The description that follows, for sake of synthesis and readability, has been taken by Jane Hillston's class notes [3].

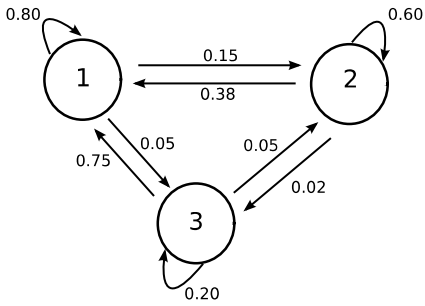


Fig. 3.1 Example of a Markov process transition graph

referred as time) with state space S. 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.

The state space S of the process is the set of all possible values that the random variables $X(t)$ can assume. Each of these values is called a state of the process. Any set of instances of $\{X(t)\colon\ t \in T\}$ can be regarded as a path of a particle moving randomly in the state space S, its position at time t being $X(t)$. These paths are called sample paths or realizations of the stochastic process.

A stochastic process is a Markov process if $\{X(t)\}$ has the *Markov* or *memoryless* property introduced above. More formally, the Markov property holds if and only if, given the value of $X(t)$ at some time $t \in T$, the future path $X(s)$ for $s > t$ does not depend on knowledge of the past history $X(u)$ for $u < t$, i.e. for $t_1 < \cdots < t_n < t_{n+1}$

$$\begin{aligned} &Pr\big(X(t_{n+1}) = x_{n+1} \mid X(t_n) = x_n, \ldots, X(t_1) = x_1\big) \\ &\quad = Pr\big(X(t_{n+1}) = x_{n+1} \mid X(t_n) = x_n\big) \end{aligned} \tag{3.1}$$

In other words, 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.

Various types of Markovian models have been introduced as special cases of Markov processes (e.g. Markov Chains [113], Hidden Markov Models [50]). A Markov process is usually represented either as a labeled graph (see Fig. 3.1) or as a transition matrix (see Fig. 3.2).

The graph of Fig. 3.1 represents a 3-state system. Labels on state transitions represent the probabilities that transitions can be fired if the system is in a certain state. The matrix of Fig. 3.2 represents the same system with the same transition probabilities, where each row i and each column j represent a state and $A(i, j)$ represent the probability for the transition from i to j to be fired. Obviously, to be consistent, in both cases the probabilities of all transitions leaving the same state must sum up to 1.

Fig. 3.2 Example of a Markov process transition matrix

$$A = \begin{bmatrix} 0.80 & 0.15 & 0.05 \\ 0.38 & 0.60 & 0.02 \\ 0.75 & 0.05 & 0.20 \end{bmatrix}$$

The primary objective of the analysis of a Markov process is to calculate the probability distribution of the random variable $X(t)$ over the state space S, as the system settles into a regular pattern of behavior. This is termed the steady state probability distribution. Performance measures based on subsets of states can be derived from this probability distribution.

Markov processes are widely used also because well-assessed theories to solve them have been built. The memoryless property means that once the process has arrived at a particular state its future behavior is always the same regardless of how it arrived in that state. The solution of Markov processes is closely related to their representation by matrices. Finding the average behavior of the model corresponds to solving a simple matrix equation.

The dynamic behavior of the system under modeling is represented by the transitions between states, and by the time spent in the states, i.e. *sojourn times*. In general, these times represent the duration of processing occurring in the system, the transitions represent event occurrences in the system.

If a state $i \in S$ is entered at time t and the next state transition takes place at time $t+T$, then T is the sojourn time in state i. By the Markov property, at any time point τ, the distribution of the time until the next change of state is independent of the time of the previous change of state. In other words, sojourn times are memoryless. Since the only probability distribution function which has this property is the exponential distribution, this means that the sojourn time in any state of a Markov process is an exponentially distributed random variable.[2]

Hence at time τ, the probability that there is a state transition in the interval $(\tau, \tau + dt)$ is $q_i dt + o(dt)$, where q_i is the parameter of the exponential distribution of the sojourn time in state i.

Suppose that when a transition out of state i occurs, the new state is j with probability p_{ij}. By the Markov property, this must only depend on i and j. Thus we have, for $i \neq j$, $i, j \in S$,

$$Pr\big(X(t+dt) = j \mid X(t) = i\big) = q_{ij} dt + o(dt) \tag{3.2}$$

where $q_{ij} = q_i p_{ij}$ by the decomposition property. The q_{ij}'s are called the instantaneous transition rates, and if the average time delay before a transition from i to j is exponentially distributed with parameter μ, then $q_{ij} = \mu$.

In practice, q_{ij} is usually given because the parameter of the distribution governing the time delay associated with the event which is represented by the state transition is known.

[2]This holds under the assumption of continuous time, whereas for discrete-time Markov processes the sojourn time is geometrically distributed.

From q_{ij} we can derive the *exit rate*, q_i, and the transition probabilities, p_{ij}, as follows. The exit rate is the rate at which the system leaves state i, i.e. it is the parameter of the exponential distribution governing the sojourn time. This will be the minimum of the delays until any of the possible transitions occurs. Thus, by the superposition property, it is the sum of the individual transition rates, i.e. $q_i = \sum_{j \in S} q_{ij}$. The transition probability p_{ij} is the probability, given that a transition out of state i occurs, that this is a transition to state j. By the definition of conditional probability, this is $p_{ij} = q_{ij}/q_i$.

As we show in Chap. 6, it is easy to derive the stationary state probability for a Markov process. The stationary solution of a Markov process has a time computational complexity of the order of the state space S cardinality.

3.2 Queueing Networks

Queueing Network (QN) models have been widely used as performance models to represent and analyze resource sharing systems [74, 76, 79, 114]. Their popularity in the performance evaluation domain is mostly due to the combination of a quite satisfying accuracy in performance results and the efficiency in model analysis and evaluation.

Informally a QN is a collection of interacting *service centers* representing system resources and a set of *customers* representing the users that share the resources. It can be represented as a direct graph whose nodes are service centers and edges represent the potential paths of customers' service requests. Several different classes of customers can circulate over the network at the same time, each class representing a set of customers with homogeneous behavior (i.e. paths and amounts of service requests).

The construction of a QN can be split in two steps: *definition*, which includes the representation of service centers, their number, and the interconnection topology of the network; *parameterization*, which aims at defining the input parameters of the network, that is: the arrival processes or the number of customers, the classes of customers, the service rates or latencies, scheduling policies and queue lengths at the service centers, and the routing probability over the network branches (given by the customers' behavior).

3.2.1 QN Definition

First of all, the number and types of QN service centers have to be defined. In Fig. 3.3 the vary basic types of service centers are illustrated.

A *Queued Center* is a node where jobs arrive and, if the server is busy, wait in a queue for their turn. Each service completion a new job is extracted from the queue, following a certain scheduling strategy, to be served. Parameters needed to define such a node are: queue length (that can also be infinite), service time (as a

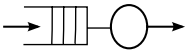
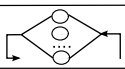
QN symbol	Graphical notation	Open QN	Closed QN
Queued center		×	×
Delay center		×	×
Source		×	
Sink		×	
Terminals			×

Fig. 3.3 Queueing Networks: basic elements

probability distribution function and related parameters like mean value), scheduling strategy and related parameters (if any).

A *Delay Center* is a node that makes each job traversing it to incur in a certain delay time. The parameter needed to define such a node is the delay time (as a probability distribution function and related parameters like mean value).

Queueing Networks can be *open* or *closed*, as defined in Sect. 3.2.2, thus *Source*, *Sink* and *Terminals* nodes will be discussed there.

Thereafter, the interconnection topology of the QN has to be built, based on the physical interconnections among service centers in the real system.

After the definition of classes of customers (in Sect. 3.2.2), the QN *chains* can also be identified. A chain represents the stochastic path of a class of jobs across the network (that has its own routing probabilities) as long as the amount of service requested to each service center.

3.2.2 QN Parameterization

The arrival processes and/or the number of customers represent the QN *workload*, that is the amount of requests that are addressed to the QN. Two main classes of QNs can be distinguished with respect to the type of workload: *open QN* and *closed QN*.

The workload of an open QN is completely determined by the stochastic processes that describe the arrivals of customer requests. One or more *sources* of requests (see Fig. 3.3) generate arrivals to (certain service centers of) the QN. The parameter needed to define such a node is the interarrival time (as a probability distribution function and related parameters like mean value).

One or more *sink* nodes (see Fig. 3.3) absorb the jobs corresponding to requests from (certain service centers of) the QN. No parameter is needed to define such a node.

Therefore, the number of customer requests that circulate in an open QN in any moment is not fixed.

The workload of a closed QN is instead completely determined by the fixed number of customers that circulate in the QN. The QN is meant to be closed because there are neither entry nor exit points. A special node that represents the customers, that is, the *Terminals* node of Fig. 3.3, generates requests to (certain service centers of) the QN and, some time after a request formulation (i.e. the time needed to the job for traversing the network and going back to *Terminals*), receives back responses to its requests. A certain amount of time after the reception of a response, which is called thinking time, a new request is generated and leaves the *Terminals* node.

Terminals node are said to have an Infinite Servers scheduling strategy in that, once the number of customers in the system is fixed, when a job arrives to this node it always will find an idle server ready to serve it. The service is here represented by the actions that a user makes before formulating a new request. The parameters needed to define such a node are: the number of users and the thinking time (as a probability distribution function and related parameters like mean value).

Therefore, the number of customer requests that circulate in a closed QN at any moment is fixed.

In both cases (i.e. open and closed QN) the classes of customers have to be defined. A customer class is a set of customers that formulate the same type of requests to the QN. In other words, a request originated by a certain class of customer obeys, within the network, the same service rate and routing probability distributions, no matter which specific customer of the class originates the request.

The parameterization task includes the specification of service rates, scheduling policies and queue lengths of all the service centers. Each service center may have different rates for different classes of jobs. Similarly, the waiting queue may be unique or one queue per class can be devised. As a consequence, one scheduling policy has to be associated to each queue.

To complete the QN parameterization, routing probabilities over the network branches must be specified. They represent a stochastic description of the customer request behavior (i.e. paths) over all the network.

In Fig. 3.4 an open QN with multiple classes of jobs is illustrated. Each class of job is represented by a different symbol and each service center is labeled with the following parameters: queue length, scheduling strategy, average service time for each class of customers

Two classes of jobs (A and B) originate from source nodes; both require service first to node S_1, which has a First-Come-First-Served scheduling strategy, an infinite queue, and two different service rates for classes of jobs. Thereafter each job can move to server S_2, S_3 or S_4, depending on the routing probabilities proper of the class of jobs it belongs to. After being served from one of the latter nodes, jobs exit the system through a sink node.

A parameterized QN can then be evaluated to compute the figures of merit (or performance indices) that characterize the quantitative behavior of the modeled system. Typical examples of performance indices are: utilization, throughput and response time. Each performance index can be evaluated at different levels of abstraction that span from *local* indices (referred to resources) to *global* indices (referred

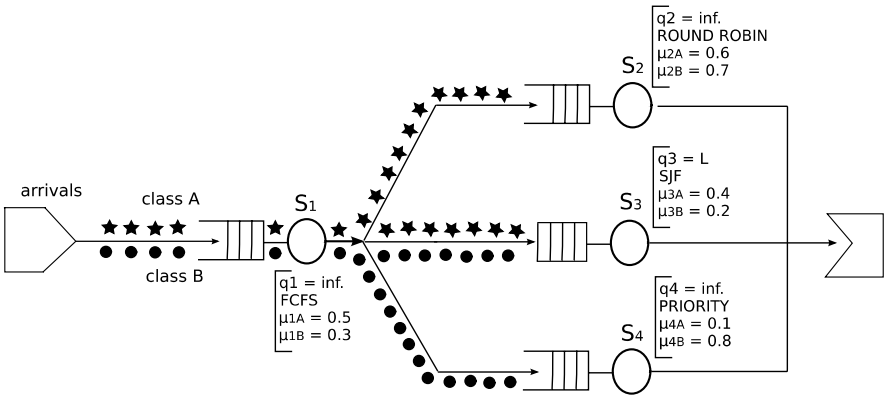


Fig. 3.4 Queueing Network example with multiple classes

to the whole system), passing through indices that refer to specific model sections (i.e. subsystems).

Product-Form QNs represent a very relevant specific class of QNs, because they can be easily solved by efficient algorithms to evaluate performance indices. Specifically, algorithms such as Convolution and Mean Value Analysis [79] have a polynomial computational complexity in the number of QN components. In order to belong to the Product Form class a QN must satisfy a set of properties on its types of nodes, scheduling strategies and classes of customers [79].

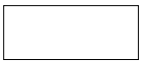
Extensions of classical QN models, namely Extended Queueing Network (EQN) models, have been introduced in order to represent several interesting features of real systems, such as synchronization and concurrency constraints, finite capacity queues, memory constraints and simultaneous resource possession [74, 79]. However, EQN cannot be solved with analytical approaches.

Examples of performance evaluation tools for QN and EQN are RESQ/IBM [100, 101], QNAP2 [116] and HIT [29]. In Sect. 6.1.2 a solution technique for QN is described.

3.3 Execution Graphs

Execution Graphs [110] have been basically introduced to represent, in the performance domain, complex behaviors of software. As mentioned in the previous section, Queueing Networks are suited to model computing platform characteristics and devices, where certain workloads run on. Traditionally, workload representations were synthesized on the basis of observed system behaviors or from the experience of system developers [79]. With the quick growth of software complexity, it has been necessary to explicitly represent the software logics with appropriate models. Nowadays Execution Graphs (EGs) appear to be the most successful formalism that addresses this problem.

Table 3.1 Basic Execution Graph notation

Types of nodes	Graphical notation	Description
Basic node		An operational step at current level of abstraction.
Expanded node		A macro-step: details are provided in an associated sub-graph.
Repetition node		The set of subsequent nodes is repeated *n* times; the last node in the loop has an arc to this node.
Case node		Attached nodes are conditionally executed with certain branching probabilities.
Pardo node		Attached nodes are executed in parallel; all must complete (join) before proceeding.
Split node		Attached nodes represent new processing threads; they do not need all to complete before proceeding.

An EG cannot be properly considered as a performance modeling notation by itself, because, as we will see later, no exhaustive performance analysis can be carried out only basing on EGs. However, in modern computing systems they play the crucial role of representing the software facet. As outlined in Chap. 1, software performance analysis is primarily aimed at evidencing potential performance problems and proposing solutions at the software level, that is: through software refactoring, re-designing, etc. Platform upgrades shall represent the uttermost solution to performance problems that has to be pursued only in cases where no action can be performed on software to overcome the problem. Thus, having an explicit representation of software dynamics greatly helps to work on software for addressing performance issues.

An EG is a graph having different types of nodes. Table 3.1 represents the main types of nodes of an Execution Graph, along with a description for each of them. Most types of nodes have an intuitive role, because they correspond to standard blocks used in flow graphs to describe the dynamics of software systems. The arcs between nodes determine the flow of software execution.

A peculiar characteristic that distinguishes EG from traditional flow graphs are the *resource demands*. Each basic and/or expanded node (i.e. the nodes that do not represent control structures) can be labeled with a vector that represents its demand of resources. Each cell of such vector represents a system resource and its value specifies the amount of that resource necessary to execute the task corresponding to the node. Algorithms have been introduced [110] to synthesize EG patterns and

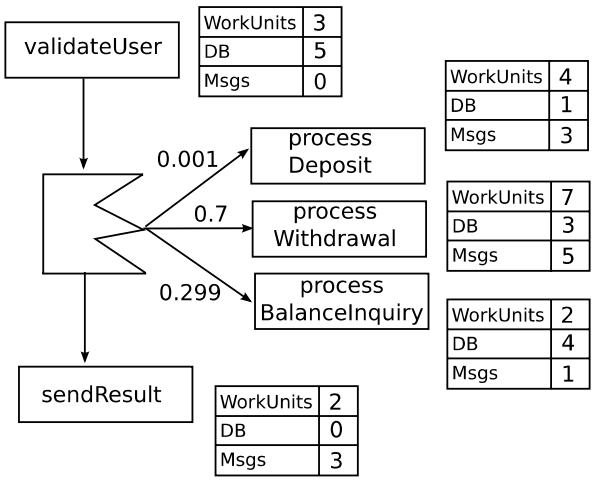


Fig. 3.5 Example of Execution Graph

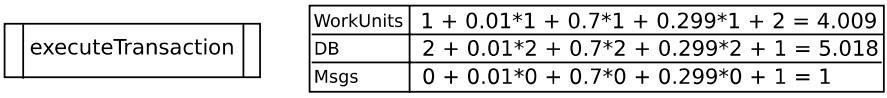


Fig. 3.6 Synthesis of the Execution Graph of Fig. 3.5

produce the (worst-case, best-case and average-case) resource demand of a set of EG nodes.

In Fig. 3.5 an excerpt of an Execution Graph is represented. A basic node is followed by a case node with three possible alternatives and finally another basic node is executed. A resource demand vector is associated to each basic node and branching probabilities are annotated on alternatives of the case node.

EGs can be used at any level of abstraction. The granularity of a basic block is not predetermined, thus it can represent from a single low-level instruction up to a whole software system. Software designers can arbitrarily decide the level of abstraction at which EGs are used, depending on their own needs and knowledge.

Being representative only of the software dynamics, no numerical performance analysis can be carried out on EGs as they are. As said above, the EG synthesis allows one to capture the demand of resources of more or less complex EG patterns. In Fig. 3.6 an example of EG synthesis is illustrated, where the EG of Fig. 3.5 is synthesized in an unique expanded node whose resource demands are obtained from the elaboration of labels of nodes in the original EG.

Moreover, by making hypotheses on physical resources of the hardware platform, a stand-alone performance analysis can be carried out under the assumption that no other software flow shares platform resources with the considered EG.

A refined version of EGs [110] allowed one to consider virtual resources instead of physical resources, through the introduction of an *overhead matrix*. An overhead matrix plays the role of mapping high-level virtual resources over low-level physical resources, as illustrated in Fig. 3.7.

Device	*CPU*	*Disk*	*Network*
Quantity	1	1	1
Service Unit	*KInstr.*	*Phys. I/O*	*Msgs*
WorkUnit	20	0	0
DB	500	2	1
Msgs	10	2	1
Service Time	0.00001	0.02	0.01

Fig. 3.7 Example of overhead matrix

Three virtual resources are represented in Fig. 3.7 in the rows and three types of physical resources are represented in the columns. Such a matrix allows one to raise the level of abstraction in the definition of resource demands. As shown in Fig. 3.7, a single request of the virtual *DB* resource corresponds, as defined in the matrix, to 500 service units of a CPU, two service units of a disk and one service units of a network.

This mechanism allows us to separately quantify and store, as values of the overhead matrix, the mapping of a software application on a hardware platform. The matrix can be modified without necessarily modifying the EG and the platform model when models have to be used in different contexts that require different definitions of virtual resources.[3] Resource demands can be expressed in units of virtual resources and, before mapping the EG onto a platform model, they can be elaborated through the overhead matrix to obtain demands referred to physical resources.

The major support that an EG provides to the performance analysis is twofold: (i) it explicitly represents the software dynamics and therefore any feedback that outcomes from an analysis step can be mapped onto the software structure; (ii) the synthesis of each EG can be aimed at identifying a class of jobs that runs over a Queueing Network.

With regard to issue (ii), usually an EG represents a single behavioral scenario of the software system. It can be synthesized to obtain the complete resource demands of that scenario that maps onto a single class of jobs running through the platform model (typically represented by a Queueing Network). However, analyzers can decide to synthesize EGs at different level of granularity, thus giving rise to lower or higher numbers of classes of jobs. In practice, the granularity of classes of jobs depends on the granularity of EG paths synthesis.

[3]Note that this is a prime example of structure that links (nowadays called) a Platform Independent Model to a Platform Specific Model.

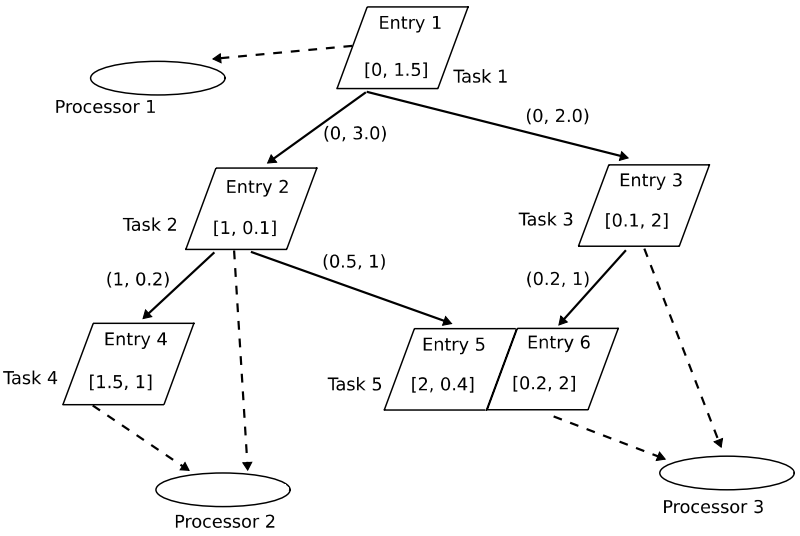


Fig. 3.8 An example of Layered Queueing Network

3.4 Layered Queueing Networks

A traditional Queueing Network that models a software/hardware system typically represents the hardware platform resources with a set of properly interconnected service centers and the software workload with a more or less complex workload made of a set of job classes sharing those resources.

This flat representation is not very suited to represent client/server systems, where nested interactions among software components take place and at the same time the software uses hardware resources. As shown in Sect. 3.3, Execution Graphs help to model such complex software behaviors and interactions. However, the process that synthesizes such a graph as a QN workload often leads to flattening of details in the software dynamics that can be relevant for performance analysis.

Layered Queueing Networks (LQN) [52, 99, 123] have been introduced to represent systems as layered models. With LQN, in the same model it is possible to represent the software components and their interconnections, as well as the platform resources utilized from, and possibly shared among, components. Tasks representing software components and resources representing platform devices are the basic elements of a LQN. An LQN is hierarchically structured in layers, as illustrated in Fig. 3.8, where square parallelograms represent tasks and circles represent resources.

The bottom-most layer of the hierarchy contains the platform resources, which are the ones where time is actually spent running jobs. Hence, resources are servers of the tasks directly connected to them in the hierarchy. In Fig. 3.8 Processors 1, 2 and 3 are the system resources, and their client tasks are connected to them through dashed arrows.

A task can act as a client of a resource or of another task and, in turn, it can act as a server of another task. For example, Task 2 in Fig. 3.8 is a server of Task 1 and a client of Task 4 and Task 5.

Each task and each resource has its own queue of requests to serve, and makes requests to lower layer servers, like another task or a server CPU. The constraint that no request can go towards higher layers tasks holds and represents, at the same time, the key for easy solution of LQN, as well as their major limitation since peer-to-peer communication patterns cannot be modeled.

A task has one or more *entries*, representing different operations it may perform and it is represented as a "slice" in the parallelogram representing the task. For example, Task 5 in Fig. 3.8 has two entries.

The topmost tasks represent the customers of the system, which are the generators of the LQN workload. They are also called *reference tasks* that endlessly cycle and create requests to other tasks. Separate classes of customers are modeled by separate reference tasks.

Calls are requests for service from one entry to an entry of another task. A call may be synchronous (represented by a solid arrowhead), asynchronous (an open arrowhead), or forwarding (a dashed arrow) [99], modeling a request forwarded to another entry for later reply. This gives a rich vocabulary of parallel and concurrent operations.

Demands are the total average amounts of host processing and average number of calls for service operations required to complete an entry. Demands for an entry can be specified using either *phases* or *activities*.

The phase-based method is simply a short hand notation for specifying a sequence of one to three activities. Phase one is a service phase and it ends after the server sends a reply. Subsequent phases are autonomous phases, launched at the end of phase one, that operate in parallel with the client (i.e., task or reference task) invoking the entry.

Phases consume time on processors and make requests to entries. Hence the parameters of an entry are:

- the mean number of requests for lower entries, shown as labels in parentheses on the request arcs, one for each modeled phase;
- the mean total host demand for the entry, in units of time, shown as a label on the entry in brackets, one for each modeled phase.

Figure 3.8 shows the phase method for the entry demand modeling. For example, Entry2 is modeled by two phases, requiring 1 and 0.1 time units, respectively, to the host. At the end of the first phase (i.e. when Processor2 has spent 1 time unit), a reply is sent to task Entry1 and, in average, 1 and 0.5 service requests are sent to Entry4 and Entry5, respectively. The second phase of Entry2 starts and, at its end, an average of 0.2 and 1 service requests are sent to Entry3 and to Entry5, respectively.

The activity-based method is typically used when a task has complex internal behavior such as forks and joins, or when its behavior is specified as an activity graph. Activities represent the lowest level of granularity in an LQN model and are linked together in a directed graph to indicate precedences. When a request arrives at

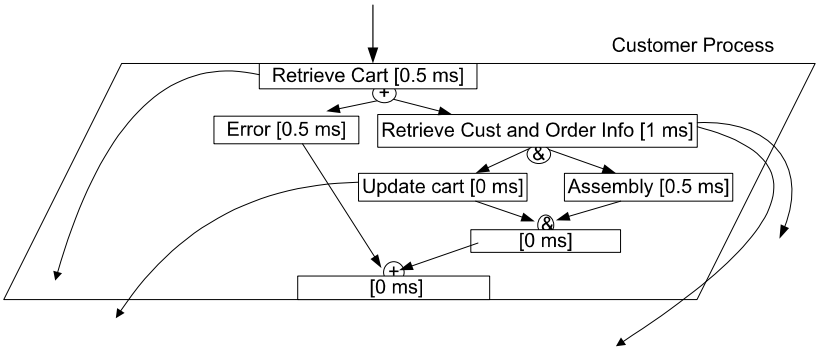


Fig. 3.9 Entry description by means of a graph of activities

an entry, it triggers the first activity of the activity graph. Subsequent activities may follow sequentially, or may fork into multiple paths which later join. Forks may take the form of an AND, which means that all the activities on the branch after the fork can run in parallel, or the form of an OR which chooses one of the branches with a specified probability.

The arcs in the graph model the calls among activities. Such calls, similar to the entries' ones, could be synchronous, asynchronous and forwarding. An activity has a single demand and a request number it makes, at the end of its execution, to the subsequent activity or entry linked by an arc.

Note that entries with different workload parameters are equivalent to separate classes in queueing networks. The workload parameters of an entry are: its host execution demand, its pure delay (or think time), and its calls to other entries.

In Fig. 3.9 the *retrieve Cart* entry of the *Customer Process* task is detailed through an activity graph. Each activity shows the execution time required to the processor the task is deployed on. As an example, *RetrieveCart* activity requires from its processing host 0.5 ms to accomplish its tasks. The graph proceeds with an "OR" fork (modeled by the "+" symbol) indicating an alternative behavior. `Retrieve Cust and Order Info` requires 1 ms, makes requests to two lower tasks (in the figure there are two out-coming arrows) and proceeds with two parallel behaviors (modeled by the "AND" fork syntactically indicated by the "&" symbol), that is: `Update Cart` requires no processing from its host but makes a request to a lower task, and `Assembly,` which does not make any request to other tasks but requires 0.5 ms to its processing host. The parallel and alternative paths terminate accordingly closing the activity graph.

LQN models can be solved by analytic approximation methods based on standard methods for QNs with simultaneous resource possession and Mean Value Analysis [79] or they can be simulated [54, 123]. Note that, if the entries are detailed by activities, the LQN model can be only simulated due to the presence of fork and join in the model. An experiment controller is also available that can execute parameterized experiments over parameter ranges.

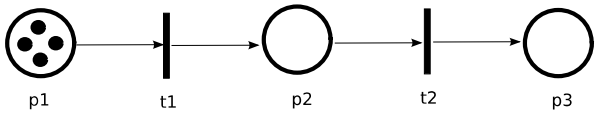


Fig. 3.10 An simple example of Petri Net

Even though LQNs apparently do not look QN-like at all, beyond the graphical representation of tasks and resources each layer corresponds to a canonical QN. Each QN has a symbolic workload that comes from the upper layer and utilizes resources that are, in practice, the lower layer tasks. The LQN solution process starts from the bottom-most layer QN and solves it with the workload represented by the upper layer. Then, it iteratively goes up one layer at a time and brings to the upper layer the resulting indices of the lower layer solution. Due to the approximation introduced by lumping the above layers as a workload for the currently processed layer, this process brings some error on the top of the model, hence it has to be iterated several times until the results obtained in two adjacent iterations are close enough to claim that stability of numerical results is achieved.

3.5 Stochastic Petri Nets

Stochastic Petri Nets (SPNs) are extensions of Petri Nets (PNs) [98]. Petri Nets are a notation suited to formally verify the correctness of synchronization between various activities of concurrent systems. A Petri Net is mainly made of: *places* that represent (possibly partial) states of a system, and *transitions* that are fired when a certain event occurs and make the system changing state. A certain number of *tokens*, representing jobs to be processed, circulate within the net driven by transition firing.

In Fig. 3.10 a PN with three *places*, two immediate *transitions* and four *tokens* is represented.

The transition firing in a PN is ruled as follows:

- A place may contain several tokens, which may be interpreted as resources.
- There may be several input and output arcs between a place and a transition.
- The number of these arcs is represented as the weight of a single arc.
- A transition is enabled if its each input place contains at least as many tokens as the corresponding input arc weight indicates.
- When an enabled transition is fired, its input arc weights are subtracted from the input place markings and its output arc weights are added to the output place markings.

The underlying assumption in PN is that each transition takes zero time, that is: once a transition is enabled it fires instantaneously. In order to answer performance-related questions beside the pure behavioral ones, PNs have been extended by associating a finite time duration with transitions and/or places, although the usual assumption is that only transitions are timed [11, 17, 74]. Stochastic Timed Petri Nets

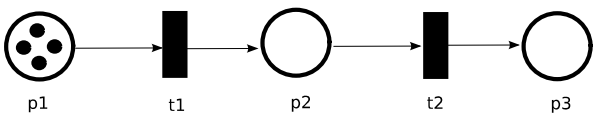


Fig. 3.11 An simple example of Stochastic Timed Petri Net

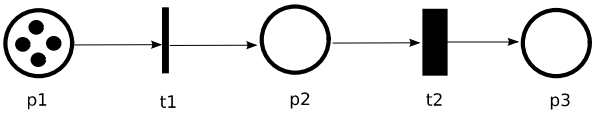


Fig. 3.12 A simple example of Generalized Stochastic Petri Net

(STPNs) are PNs where transitions have firing delays. From a graphical point of view the difference is on the representation of transitions, as we can see in Fig. 3.11 in which a STPN with three *places*, two timed *transitions* and four *tokens* is shown. In practice, timed transitions are ticker than immediate transitions, as will become evident in the next two figures.

In the last few decades STPNs have been receiving increasing interest in the modeling and performance analysis of discrete event systems. They are particularly useful for modeling systems which exhibit concurrent, asynchronous or nondeterministic behaviors, such as parallel and distributed systems, communication networks and flexible manufacturing systems.

The *firing time* of a transition is defined as 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.

Most literature of STPN is on Stochastic Petri Nets (SPN) and on their extensions, Generalized Stochastic Petri Nets (GSPN) [12]. In SPN transition firing times are mutually independent exponentially distributed random variables, whereas in GSPN immediate transitions (i.e. those without firing delay) are allowed besides timed ones. In Fig. 3.12 a GSPN with three *places*, one immediate *transition*, one timed *transition* and four *tokens* is represented.

Immediate transition fires immediately after enabling and has strict priority over timed transitions. The former 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.

In Fig. 3.13 we propose an example of modeling with GSPN for the e-commerce case study.

We assume that the system workload is modeled with a specific pattern. The *arrivals* transition is a timed one whose rate regulates the flow of requests for the ATM central server; the assumption is that there are a number of requests which need to be processed. These requests await to be served in the *requests* place. Between arrivals and waiting requests there is a special type of arc, namely an inhibitor arc. Such arc works with an opposite logic to a standard arc, that is: the transition will be fired

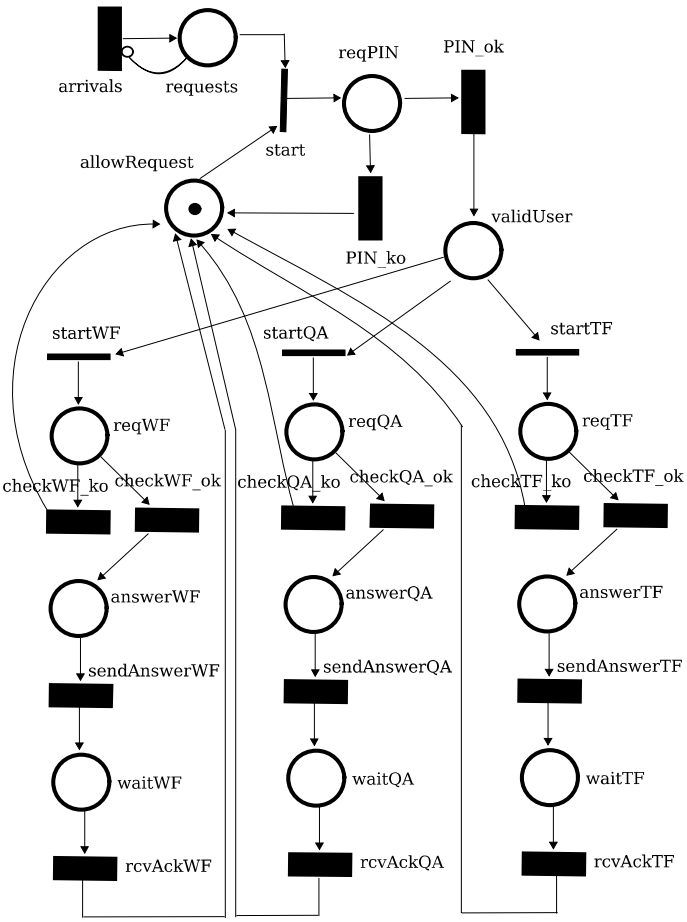


Fig. 3.13 E-commerce example modeled with Generalized Stochastic Petri Net

only if no tokens appear in the connected place. The maximum number of tokens in the latter place is defined by the multiplicity of the inhibitor arc, thus requests can be accumulated until the buffer size is achieved. Requests are admitted to the system through the *allowRequest* place, which represents the scheduling of a new request, if any.

The firing of the *start* transaction involves first of all the PIN validation represented by the *reqPIN* place. If this check is unsuccessful, i.e. *PIN_ko*, the entire transaction ends. Otherwise a successful PIN validation, i.e. *PIN_ok*, enables the system to recognize the ATM customer as a *validUser*. Thereafter, the transaction proceeds with one of these operations: Withdraw Funds (*startWF* transition); Query Account (*startQA* transition); Transfer Funds (*startTF* transition).

All these operations starts with the *reqOP* place, where OP stands for WF, QA, TF, depending on the customer request.[4] In this place the type of operation required is checked. Once again, we have two alternatives: if the check is unsuccessful, i.e. *checkOP_ko*, the transaction ends. Otherwise a successful result, i.e. *checkOP_ok*, enables the user to specify some parameters. The place *answerOP* gives the possibility to insert appropriate values for the request. The parameter values are sent to the ATM central server through the timed transition *send_answerOP*. Finally, the *waitOP* place represents the waiting step for an acknowledgment from the central server that denotes the success of the entire operation, i.e. the *rcvACK_OP* transition.

GSPN admit specific solution techniques [17]. In Sect. 6.1.2 we mention some approaches for solving SPNs when they are used for performance modeling.

3.6 Stochastic Process Algebras

Stochastic Process Algebras (SPAs) represent the class of performance modeling notations, which includes all the Process Algebras (PAs) [18] that have the additional capability of explicitly representing stochastic and time characteristics of a system dynamics. Hence SPAs are extensions of PAs, aiming at the integration of qualitative—functional and quantitative—temporal aspects into a single modeling notation.

The development of SPA has been very similar to that of STPN: in both cases an untimed formalism, used for studying the correct functional behavior of systems, is extended by associating exponential delays with actions, and reachability analysis is used to construct a corresponding Markov process. The advantages of SPAs are that they incorporate the attractive features of PAs and thus bring to the area of performance modeling several attributes which are not offered by the existing formalisms. Perhaps the most important of such features is the inherent composability of models that can be exploited for their analysis [3].

In practice, a SPA is obtained starting from a PA and adding in its grammar the constructs necessary to model timed actions and stochastic behaviors. A semantics for the additional constructs shall also be provided. It is therefore a too abstract task to introduce SPAs in general, without reference to any specific existing algebra. Therefore we introduce in this section some general concepts that can refer to any SPA, whereas details of major notations can be found, among others, in [33, 67, 69].

The basic elements of a SPA are: (basic and composed) (possibly timed) *actions* that model system behaviors, *operators* that are used to build up complex behaviors starting from simple ones.

Temporal information is typically added to actions by means of continuous random variables, representing activity durations. Such information makes it possible

[4] We describe the remainder of this model by using the OP pattern, as the internal dynamics of the three use cases is structurally the same.

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.

Almost all existing SPAs share the following basic operators [37]:

- *Sequence*—A behavior can be purely sequential, repeatedly undertaking one activity after another and eventually returning to the beginning of the sequence. A simple example is a web server, which allows one data item to transfer at a time. Each browser requiring web pages will need to acquire access to the server and only when the transfer is complete will the server be released and be available again for acquisition.
- *Choice*—A choice between two possible behaviors is represented as the sum of the possibilities. For example, if we consider a browser in an information system, displaying the current data may have two possible outcomes: demand for access to data available in the local cache (with a certain probability) or demand for access to data stored at the remote server (with complementary probability). A race condition is assumed to govern the behavior of simultaneously enabled actions and the continuous nature of the probability distributions ensures that the actions cannot occur simultaneously. Thus a sum will behave as either one summand or the other. When an action has more than one possible outcome it is represented by a choice of separate actions, one for each possible outcome. The rates of these actions are chosen to reflect their relative probabilities (decomposition principle).
- *Concurrency*—Two behaviors can occur at the same time in a system and this may require cooperation/synchronization among them. Therefore a specific operator is usually provided to make the concurrency of these behaviors. When needed, the operator can be labeled with a set of synchronizing actions. Actions in this set require the simultaneous involvement of both behaviors. The semantics of this operator is that both behaviors can concurrently run as long as no synchronizing action is encountered in any of them. Before executing any synchronizing action, a behavior has to wait until the other behavior (operand of the concurrency operator) reaches the same action. Those actions thus represent synchronization points of parallel executions.
- *Hiding*—It is often convenient to hide some actions, making them private to the behavior. For example, if a behavior is related to a certain software component, one can make visible only component actions that allow one to interact with other components, while hiding all the other (internal) actions. The type of the latter action is usually said to be *hidden*, and they are characterized by a special symbol. Components cannot synchronize on hidden actions. Use of the hiding operator has two implications. Firstly, it ensures that no components added to the model at a later stage can interact, or interfere, with the specified action. Secondly, private actions are deemed to have no contribution to the performance measures being calculated and this might subsequently suggest simplifications to the model.

Figure 3.14 shows a SPA model defined by using TIPP process algebra [67]. This model has been obtained from the model in Fig. 2.4, presented in Sect. 2.1.2, by adding performance-related information to some actions. Therefore, we have

```
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
```

Fig. 3.14 TIPP process algebra model for the XML Translator

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)).

In Sect. 6.1.2 we mention some approaches for solving SPAs.

3.7 Simulation Models

Even though simulation cannot properly be considered as a performance notation, but rather as a solution technique, it is a matter of fact that it has been widely used as an instrument to study software performance.

As it will be illustrated in Chap. 5, several approaches to automatically generate simulation models from software artifacts have been recently introduced and successfully experimented. Therefore this section is part of the chapter of performance notations due to the fact that efficient simulation models can be built to analyze software performance [28].

Beside this, simulation techniques can be applied to solve performance models in canonical notations, such as Queueing Networks or Petri Nets (see Chap. 6).

A simulation model is a computer program (written either with a general purpose language like C or with a simulation language like Simula [97]) that describes, on the basis of a discrete or continuous time progression, the dynamic behavior of a system. When executed, the program can produce outputs such as behavioral traces or amounts of simulated time to perform certain tasks.

If instrumented with appropriate probes, a simulation model can be useful to analyze different performance indices with a very high accuracy. Therefore a simulation model can be actually considered to be the most flexible and general performance model, since any specified behavior can be simulated. The limitations and assumptions that must apply to models based on other performance notations in order to be efficiently solved (such as Product Form Queueing Networks) do not hold for simulation models. In simulation models any detailed mechanism of a system can be described at the sole cost of correctly specifying its behavior. However, a key issue in simulation concerns the construction of the simulation model at the appropriate level of abstraction.

The simulation of a complex system includes the following phases [24]:

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

In order to obtain accurate and trustfulness results, all the above steps have to be carefully carried out. Hence, the main drawback of simulation is its development and execution costs. Existing simulation tools provide suitable specification languages for the definition of simulation models, and simulation environments to conduct system performance evaluation, such as CSIM [2], C++Sim [1] and JavaSim [4].

3.8 UML Profile for Schedulability, Performance and Time

The Schedulability, Performance and Time UML Profile [85] is adopted as an official OMG standard. The main aims of this profile 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. It provides the ability to specify quantitative information directly in UML models allowing quantitative analysis and predictive modeling. It is founded on a domain model that defines the main entities for the considered analysis.[5] The analysis methods considered in the profile are scheduling analysis and performance analysis based on queueing theory.

[5] For more details on the domain model please refer to [85].

In the following, we present the performance part of the profile by giving details on the PAprofile package that contains the stereotypes and tags used to annotate the UML diagrams. PAprofile is fully based on the General Resource Modeling (GRM).

3.8.1 PAprofile: Stereotypes and Tagged Values

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.

«PAcontext» Stereotype

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 performance values considered here are load intensity and various measures of response delay.

«PAcontext» stereotype models a performance analysis context. The base classes it can extend are Collaboration, CollaborationInstanceSet and ActivityGraph.

This stereotype does not present any tags, while the constraints to be satisfied are:

- 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.

Figure 3.15 shows a top-level context where two software resources, namely the Client and Server components, interact to provide a Browse Cart functionality when requested by the Customers.

«PAclosedLoad» and *«PAopenLoad»* Stereotypes

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. The stereotypes modeling open and closed workload can only be applied to be the first step in a performance context.

We recall that 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. *«PAopenLoad»* stereotype models an open workload.

In the following two tables (Tables 3.2 and 3.3) are reported the base classes the stereotype can extend and the definition of the tags it could have.

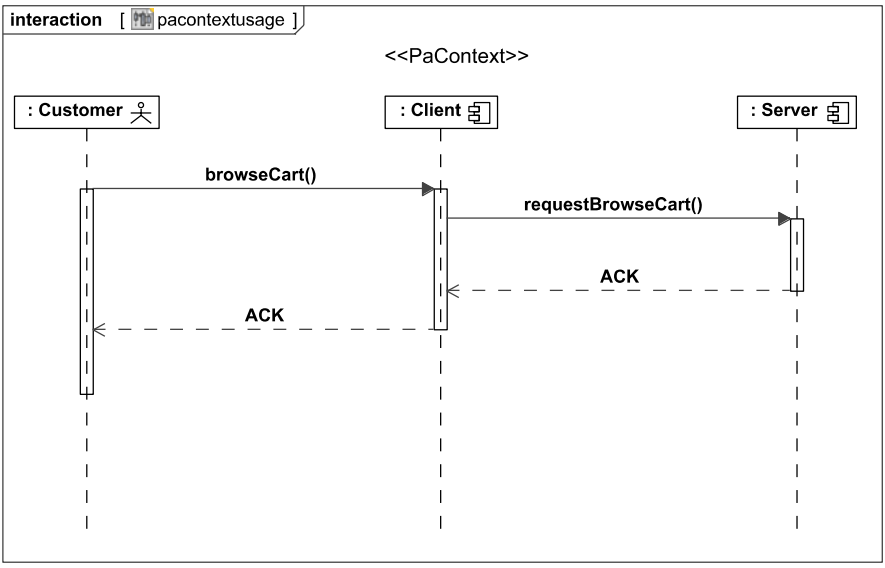


Fig. 3.15 Example of *«PAcontext»* annotation

Table 3.2 *«PAopenLoad»* stereotype

Stereotype	Base class	Tags
«PAopenLoad»	Message	PArespTime
	Stimulus	PApriority
	Action State	PAoccurrence
	SubactivityState	
	Action	
	ActionExecution	
	Operation	
	Method	
	Reception	

Table 3.3 *«PAopenLoad»* tags definition

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

In Fig. 3.16 an *open workload* is annotated on the first message of the Browse Cart functionality using the *«PAopenLoad»* stereotype. The *unbounded* string refers

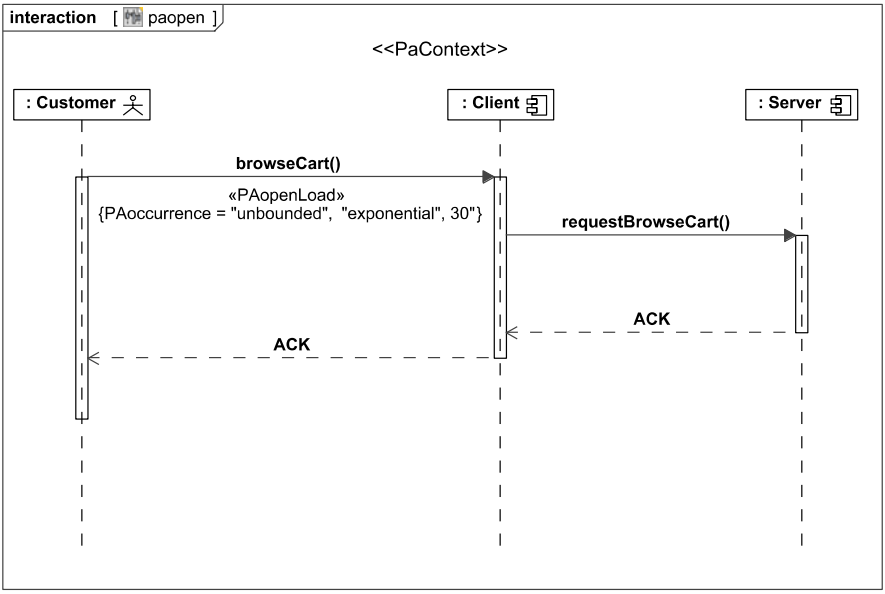


Fig. 3.16 Example of *«PAopenLoad»* annotation

Table 3.4 *«PAclosedLoad»* stereotype

Stereotype	Base class	Tags
«PAclosedLoad»	Message	PArespTime
	Stimulus	PApriority
	Action State	PApopulation
	SubactivityState	PAextDelay
	Action	
	ActionExecution	
	Operation	
	Method	
	Reception	

Table 3.5 *«PAclosedLoad»* tags definition

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

to a pattern whose interarrival time between two succeeding requests is specified by a probability distribution function (e.g. *exponential*, with an expected value of *30*).

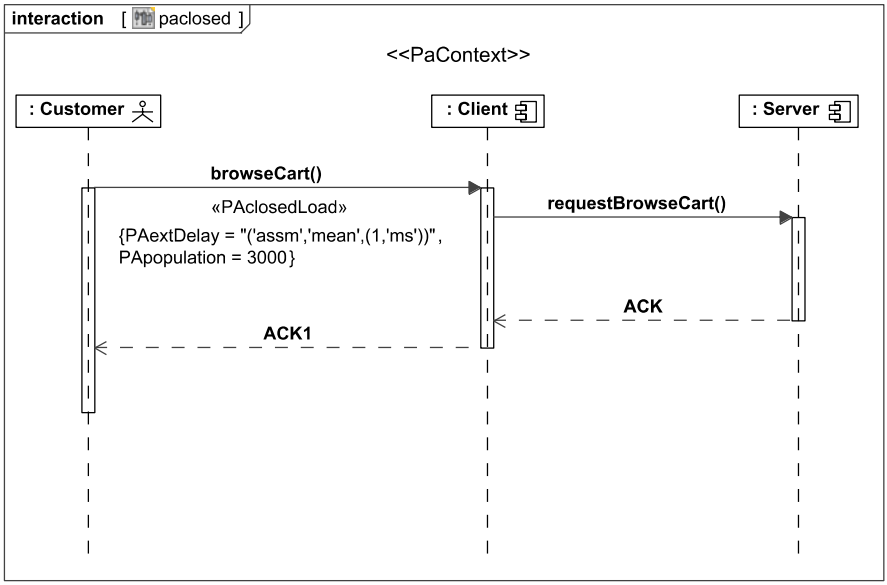


Fig. 3.17 Example of *«PAclosedLoad»* annotation

Instead, a *closed workload* has a fixed number of jobs (`population`) which cycle between 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`).

The *«PAclosedLoad»* stereotype models a closed workload. In Tables 3.4 and 3.5 the base classes that the stereotype can extend are reported, and the definition of the tags it could have, respectively.

In Fig. 3.17 a *closed workload* is annotated on the first message of the Browse Cart functionality using the *«PAclosedLoad»* stereotype. It defines a fixed number of 3000 jobs (PApopulation tag), each spending an assumed mean external delay period of 1 millisecond.

«PAstep» Stereotype

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 represents an increment in the execution of a scenario and it may use resources to perform its function. In general, a step takes a 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 performance properties and may specify the resource demands to the resources involved in the step achievement (characteristics inherited from the scenario entity).

Table 3.6 *«PAstep»* stereotype

Stereotype	Base class	Tags
«PAstep»	Message	PAdemand
	Stimulus	PArespTime
	Action State	PAprob
	SubActivityState	PArep
		PAdelay
		PAexOp
		PAinterval

Table 3.7 *«PAstep»* tags definition

Tag	Type	Multiplicity	Domain attribute name
PAdemand	PAperfValue	[0..*]	Step::hostExecutionDemand
PArespTime	PAperfValue	[0..*]	Step::responseTime
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

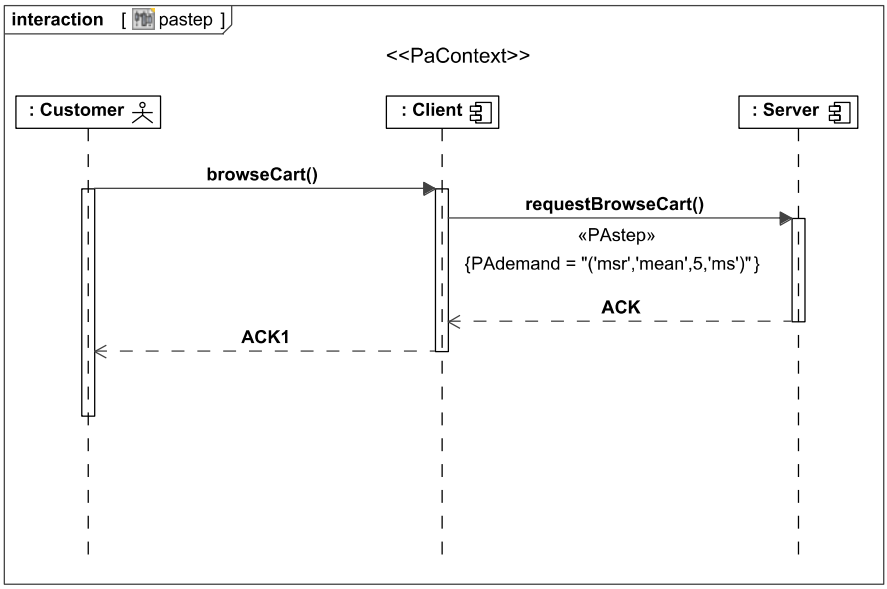


Fig. 3.18 Example of *«PAstep»* annotation

Table 3.8 *«PAhost»* stereotype

Stereotype	Base class	Tags
«PAhost»	Classifier	PAutilization
	Node	PAschdPolicy
	ClassifierRole	PArate
	Instance	PActxtSwT
	Partition	PAprioRange
		PApreemptable
		PAthroughput

Table 3.9 *«PAhost»* tags definition

Tag	Type	Multiplicity	Domain attribute name
PAutilization	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

«PAstep» stereotype models a step in a performance analysis scenario.

In Tables 3.6 and 3.7 are reported the base classes the stereotype can extend and the definition of the tags it could have, respectively.

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

In Fig. 3.18 the requestBrowseCart message is annotated using the *«PAstep»* stereotype. The value expression assigned to the PAdemand tag represents a measured mean service time demand of 5 milliseconds.

«PAhost» and *«PAresource»* Stereotypes

A `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

Table 3.10 *«PAresource»* stereotype

Stereotype	Base class	Tags
«PAresource»	Classifier	PAutilization
	Node	PAschdPolicy
	ClassifierRole	PAcapacity
	Instance	PAaxTime
	Partition	PArespTime
		PAwaitTime
		PAthroughput

Table 3.11 *«PAresource»* tags definition

Tag	Type	Multiplicity	Domain attribute name
PAutilization	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::responseTime
PAwaitTime	PAperfValue	[0..n]	PassiveResource::waitTime
PAthroughput	Real	[0..1]	Resource::throughput

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.

«PAhost» stereotype models a processing resource.

In Tables 3.8 and 3.9 are reported the base classes the stereotype can extend and the definition of the tags it could have, respectively.

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.

Based on the modeling of the performance analysis domain identified before, we here describe how the domain concepts can be represented in UML by introduc-

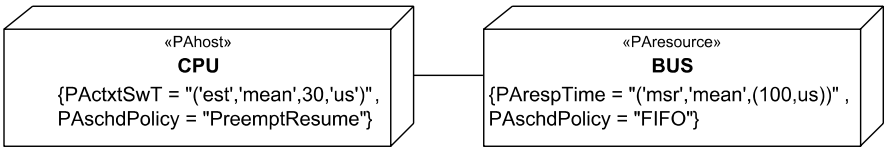


Fig. 3.19 Example of *«PAhost»* and *«PAresource»* annotation

ing the UML extensions defined for this purpose. These extensions 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.

«PAresource» stereotype models a passive resource.

In Tables 3.10 and 3.11 are reported the base classes the stereotype can extend and the definition of the tags it could have, respectively.

In Fig. 3.19 a CPU node is an active resource annotated with the *«PAhost»* stereotype. It applies a Preemption-Resume scheduling policy to each step executed (i.e. PAschdPolicy tag). The estimated mean context switching time required by such an active resource is 30 microseconds (i.e. PActxtSwT tag). The CPU communicates through a BUS that represents a passive resource that dispatches messages in 100 microseconds (i.e. PArespTime tag), while applying a FIFO scheduling policy (i.e. PAschdPolicy tag).

Chapter 4
Software Lifecycle and Performance Analysis

This chapter is aimed at illustrating performance modeling and analysis issues within the software lifecycle. After having introduced software and performance modeling notations, here the goal is to illustrate their role within the software development process. In Chap. 5 we will describe in more details several approaches that, based on model transformations, can be used to implement the integration of software performance analysis and software development process.

After briefly introducing the most common software lifecycle stages, we present our unifying view of software performance analysis as integrated within a software development process (i.e. the Q-Model). Without losing generality we consider the traditional waterfall process as a reference software process. However, many considerations introduced in this chapter can be exported to other process models.

4.1 Software Lifecycle

A software process is a set of activities that are suitably combined in order to produce a software system. Different set of activities and different ways of combining such activities lead to different software processes. However, there are some fundamental common stages that can be identified in every software process, where each stage includes a set of well-defined activities. In practice such stages identify different abstractions or maturity levels of the software under development.

Requirement specification focuses on the functionalities of the system and on its operational constraints. At the end of this stage, all the functionalities of the software system and the constraints on its operation are identified and specified. In this stage customers and software engineers collaborate to produce a document collecting all the requirements of the system. Such a document can be the basis of a contract among customers and developers since it defines the software application the developers have to produce for the customers.

Software design and implementation deals with the production of the software system according to its specifications. During this stage several models (or, more

V. Cortellessa et al., *Model-Based Software Performance Analysis*,
DOI 10.1007/978-3-642-13621-4_4, © Springer-Verlag Berlin Heidelberg 2011

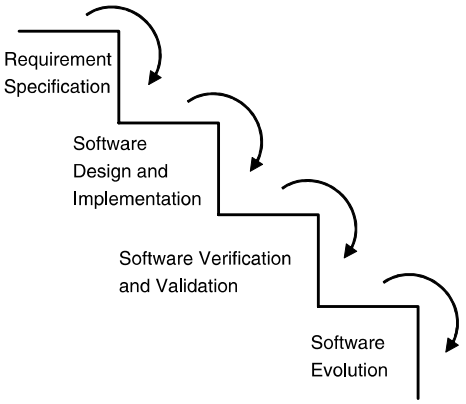


Fig. 4.1 The *waterfall* process model

generally, artifacts), describing the system at different levels of details, are produced. Typically they are architectural models and low level design models. The implementation can be obtained through a refinement process of such models.

Software verification and validation is a stage aimed at (more or less formally) proving that the software system conforms to the requirements and constraints identified in the specification stage, and at demonstrating that the system meets the customer expectations.

Each of these stages produces one or more software artifacts that represent the software system. Also, in each stage a set of activities that operate on the artifacts can be devised, in order to achieve the expected development process results. In the next section we detail these stages within the framework of a software process model.

With the recent progresses in the software development processes, the lifecycle time after the software has been deployed is becoming ever more crucial. Software evolution is the stage that manages the changes to the software product. It starts after the delivery of the software system since software is ever more subject to changes required, for example, by evolving needs of the customers or by changes in the running context/environment. However, for the sake of readability, we do not deal with software evolution in this chapter.

Many software processes exist that combine such stages in different ways. Different types of software system may need different software processes. Moreover, each industrial organization might have its own software process that it follows during software development. In order to describe the software lifecycle a software process model is used. In the following we briefly mention two of them and we refer to classical software engineering books for a comprehensive presentation of the topic [112, 56].

The *waterfall* process model organizes and details the lifecycle stages sequentially, as shown in Fig. 4.1, where we limit the illustration to the common stages described above.

Another popular software process model category is the *iterative* one. They carry on the specification, implementation and validation activities concurrently in order

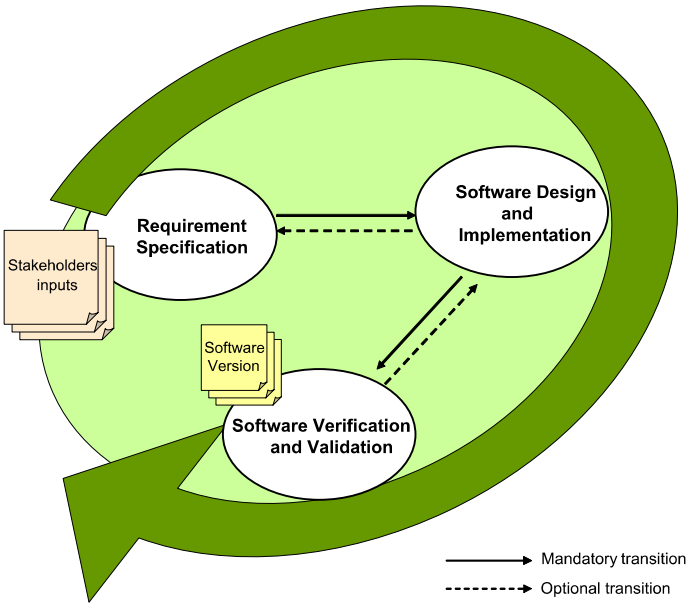


Fig. 4.2 An *iterative* process model

to quickly produce an initial version of the software system that can then be refined through iterations, as illustrated in Fig. 4.2. This kind of development process model has been recently promoted by the so-called agile development community [78].

For our purposes it is important to note that, independently of the considered software process, for each stage there can be one or more analysis tasks that concern the software artifacts involved in the stage. These analysis tasks are either specific tasks as part of the main stage or part of the overall software validation process. Let us, for example, consider the requirement stage. During the requirement stage there can be an analysis task that allows for improving the elicitation and the understanding of requirements as well as their correctness and completeness. Once the requirements are specified there can also be an analysis task that aims at validating the set of specified requirements with respect to the customer expectations.

4.2 Performance Analysis Within the Lifecycle

The aim of this section is to couple performance analysis with the development lifecycle, thus sharing the classical view of Software Performance Engineering of addressing performance concerns while developing the software system.

The starting point is the existence of a set of non-functional requirements, specifically performance ones. The goal of any development process that intends to satisfy such requirements is to start performance analysis as early as possible on the available software artifacts, possibly supported by suitable models. However, the use of

these models for performance analysis is analogous to the use of behavioral models for functional analysis. Namely they serve the purpose of the analysis at the concerned abstraction level with no intent to be considered *predictive* with respect to the performance of the final system.

Let us consider a conventional waterfall software development process. The first stage deals with requirements specification. At this stage the non-functional requirements are specified together with any operational constraints. During this stage, performance models can be built as any other model, mainly during the requirements engineering stage and in order to elicit and better understand the performance requirements. The same kind of reasoning applies to the architecture design and further down to the implementation and deployment, which can represent the last step where performance analysis reduces to simulate and/or monitor the actual behavior of the implemented system.

In this section we tackle a more detailed level of abstraction with respect to the description provided in the previous section. Therefore, we refine the concept of stages introduced before as phases of the lifecycle. The refinement logic is illustrated here below.

Taking inspiration from the familiar V-model for software validation, we customize this view toward performance analysis, thus obtaining what we will call the Q-model in the following. Figure 4.3 illustrates our view.

The left-hand side represents common development phases, that is: requirements elicitation and analysis, architectural design, detailed design and implementation. The right-hand side represents the performance analysis activities that can be carried on at each specific development phase.

With respect to the common stages described in the previous section, here we can consider the following mapping: (i) requirement specification stage has simply been rephrased as the requirement elicitation and analysis phase, (ii) software design and implementation stage has been partitioned in architectural design, low-level design and implementation phases, (iii) software verification and validation stage is represented by the middle and right-hand side of the figure, as will be illustrated here below.

In the middle, performance model generation activities connect each development phase with the corresponding performance analysis activity. Basically such intermediate activities derive from specific software artifacts the corresponding performance model. For example, the architectural design phase is connected to the performance analysis of software architecture through a performance model generation step that, starting from a software architecture specification, produces the corresponding performance model. Feedback arrows complement each horizontal connection and denote the feedback that the performance analysis can produce for the corresponding software development phase.

The connecting vertical arrows along the development path (i.e. the left-hand side) represent the successful completion of a phase and the transfer to the next development phase. In the Q-model a phase is complete only after appropriate performance analysis activities (i.e. the horizontal path for that phase). The connecting

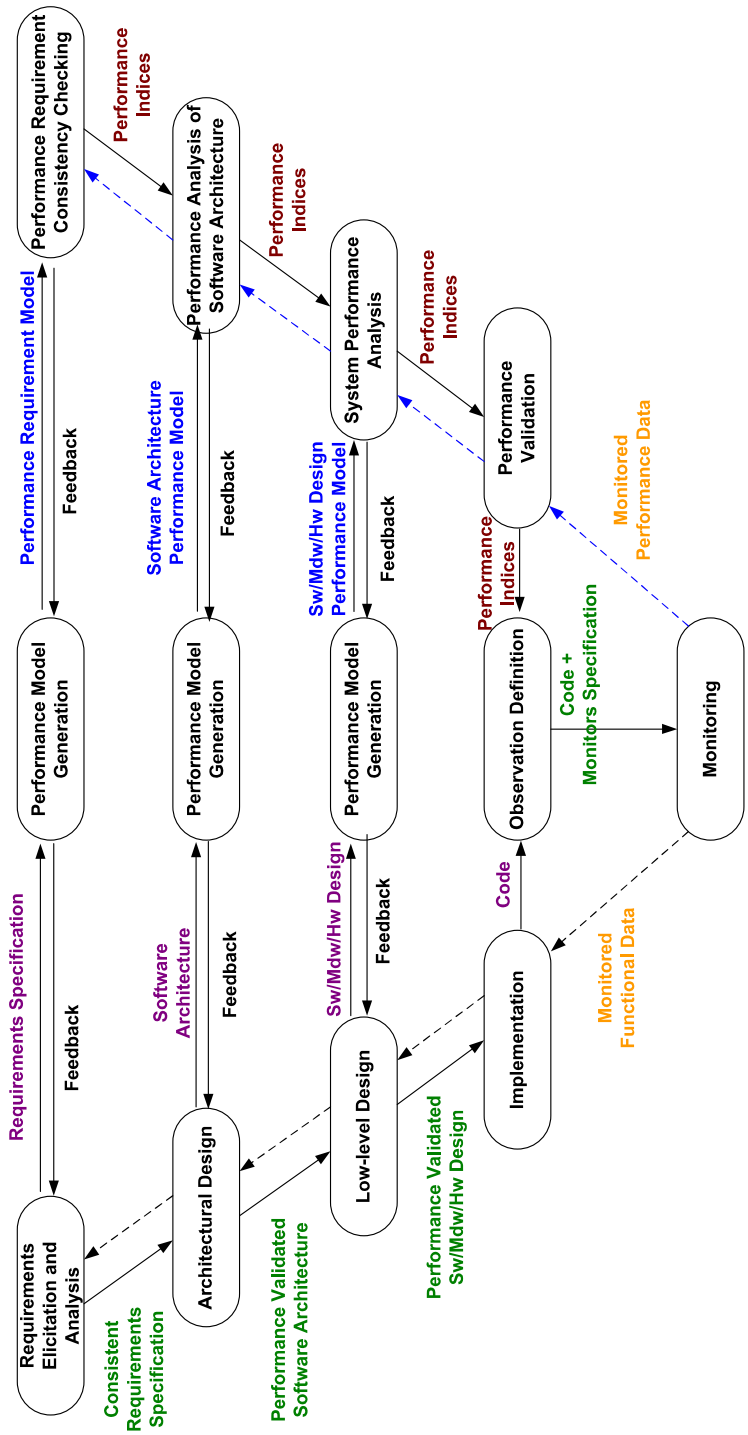


Fig. 4.3 Q-model for a waterfall process

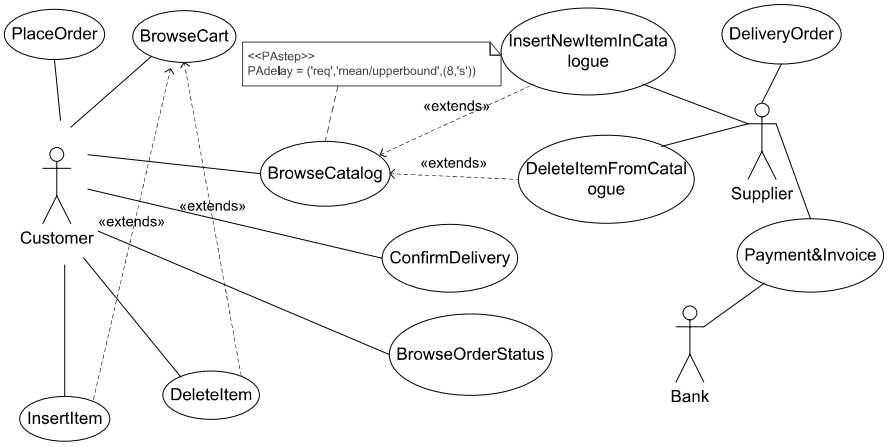


Fig. 4.4 Annotated use case diagram

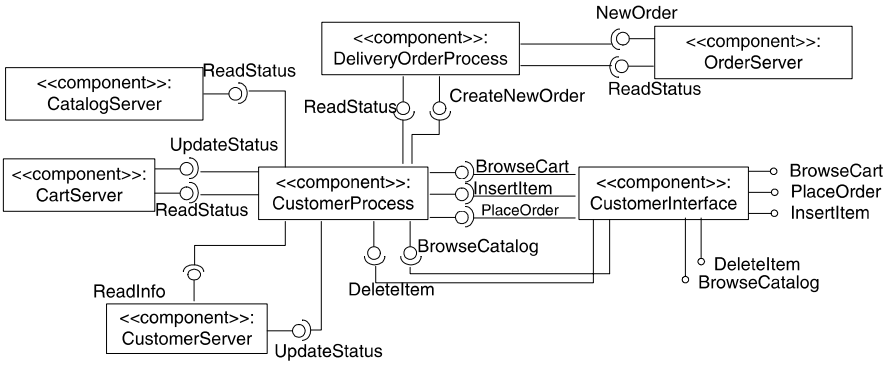


Fig. 4.5 Component diagram

vertical arrows along the performance analysis path (i.e. right-hand side) represents the information that analysis activities transfer to the next phase activities. For example, the performance bounds obtained at the architectural phase may represent reference values for the analysis at the low-level design phase. Like in any V-model, upstream vertical arrows appear in both paths, and they represent backward paths that might be traversed in case of problems at lower phases that cannot be fixed without re-executing the previous phases.

The lowest part of the Q-model deals with the implementation of the system and with the monitoring of its actual behavior. In this case the horizontal line denotes the process of defining suitable observation functions on the running code that may allow for performance indices validation. The bottom vertex is the monitoring activity that receives information on what to monitor, on the final executing code, from the observation definition process that also depends on the performance indices to validate. The monitoring phase provides feedback to both the implementation and the

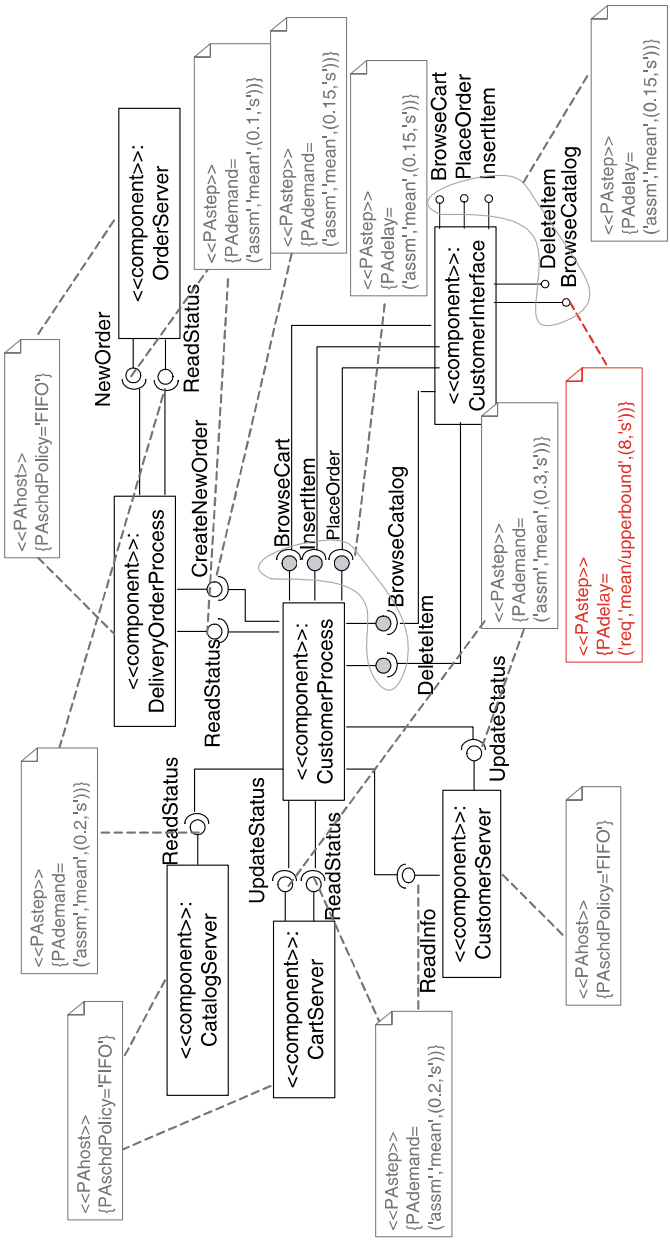


Fig. 4.6 Annotated component diagram

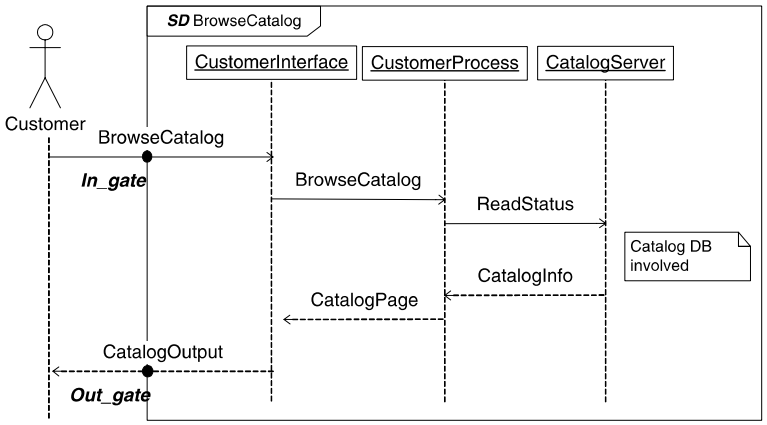


Fig. 4.7 Browse catalog sequence diagram

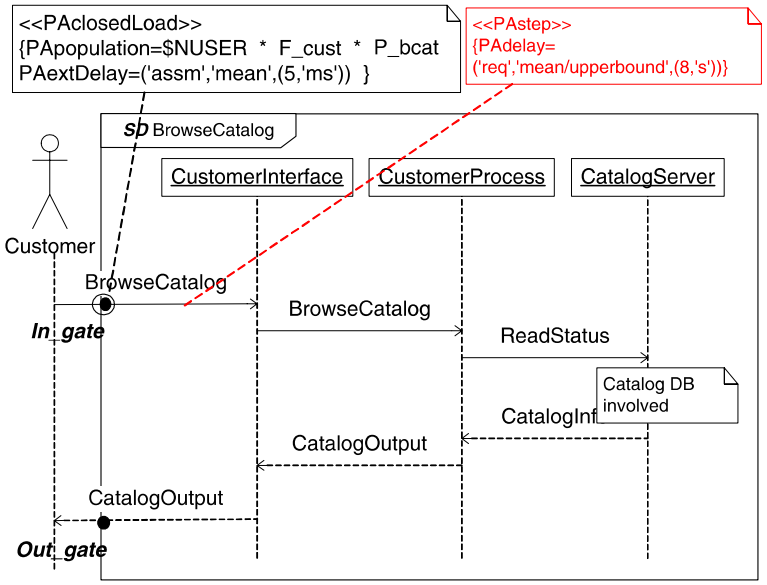


Fig. 4.8 Annotated browse catalog sequence diagram

performance validation analysis activities. The feedback can then vertically travel along both lateral sides, thus inducing changes backwards on the software artifacts and on the performance models, respectively. On the horizontal paths, it is worthwhile remarking that the feedback process starts from performance analysis activities; it can have effect on the generated model, and it can induce changes that must be reflected at the corresponding development level. We will see in Chap. 7 that this feedback process is not straightforward and still represents a challenging research issue.

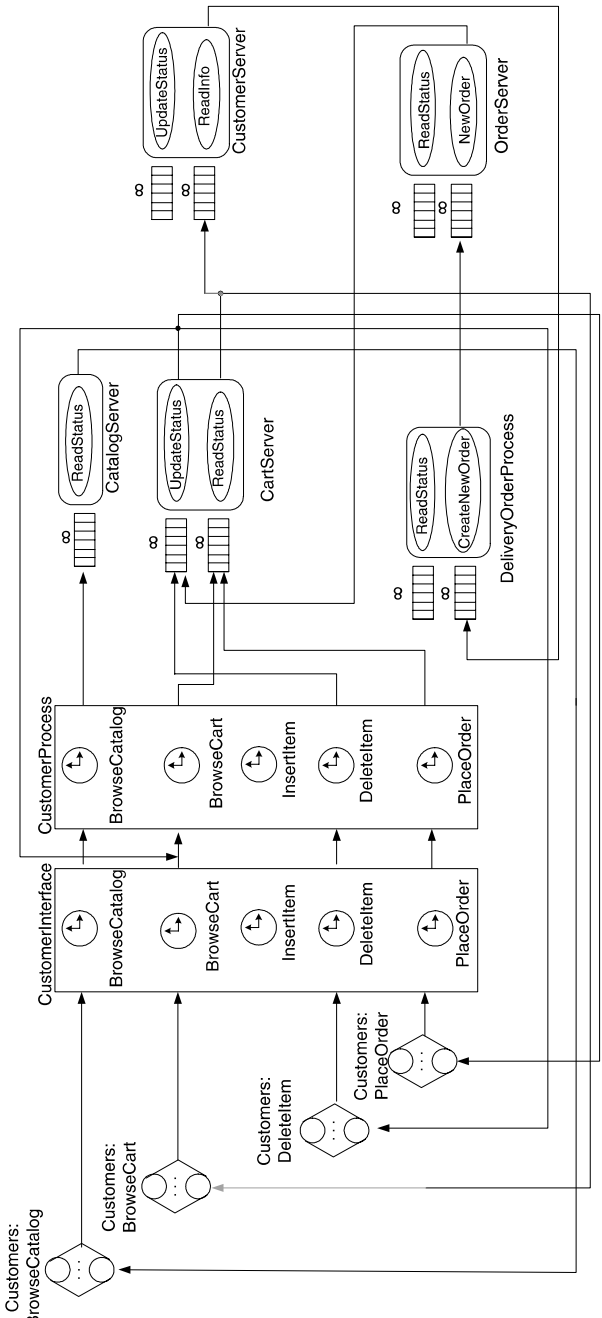


Fig. 4.9 Queueing Network model

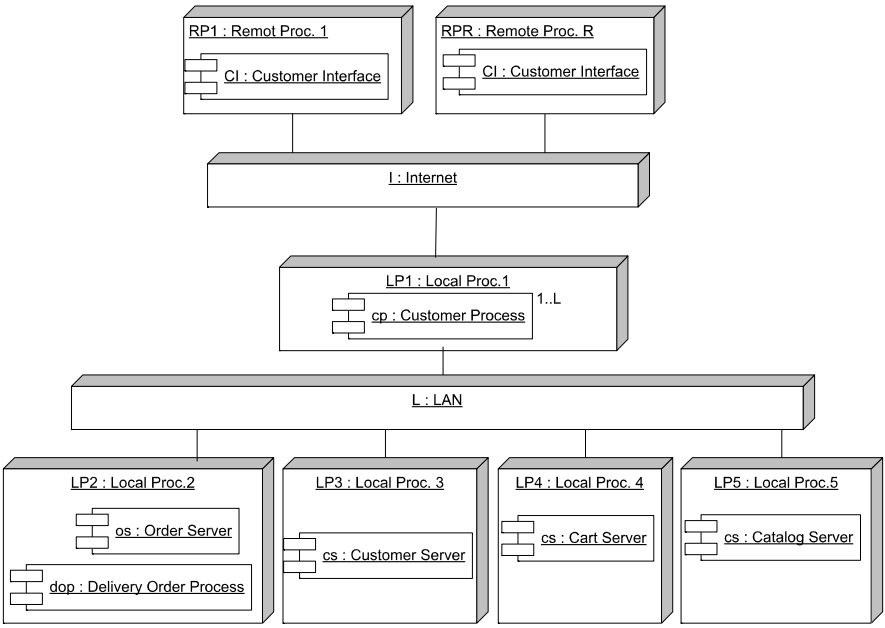


Fig. 4.10 Deployment diagram

4.3 A Simple Application Example

As an illustration of the Q-model let us consider the e-commerce example previously introduced in Chap. 2.

The use case diagram in Fig. 4.4 represents the artifact modeling the e-commerce system at the level of requirements specification. We have annotated the link connecting the Customer actor to the BrowseCatalog functionality to express a response time requirement, that is: a Customer should not wait more than 8 seconds to access the Catalog. From the analysis point of view we can interpret this limit either as an average or as an upper bound.[1]

In this case since we are dealing with just one non-functional requirement we are neither producing a performance model from the requirement specification nor performing any performance analysis for consistency checking.

While proceeding in the software development process, Figs. 4.5 and 4.6 show, respectively, a flat and an annotated UML component diagram of the example, namely a static view of the e-commerce example software architecture. The flat and annotated UML sequence diagrams of Figs. 4.7 and 4.8, respectively, represent the dynamic view of the same architecture.

[1]The capability of annotating UML diagrams with additional information (such as performance parameters and indices) is provided from the UML profiling technique that has been described in Chap. 2.

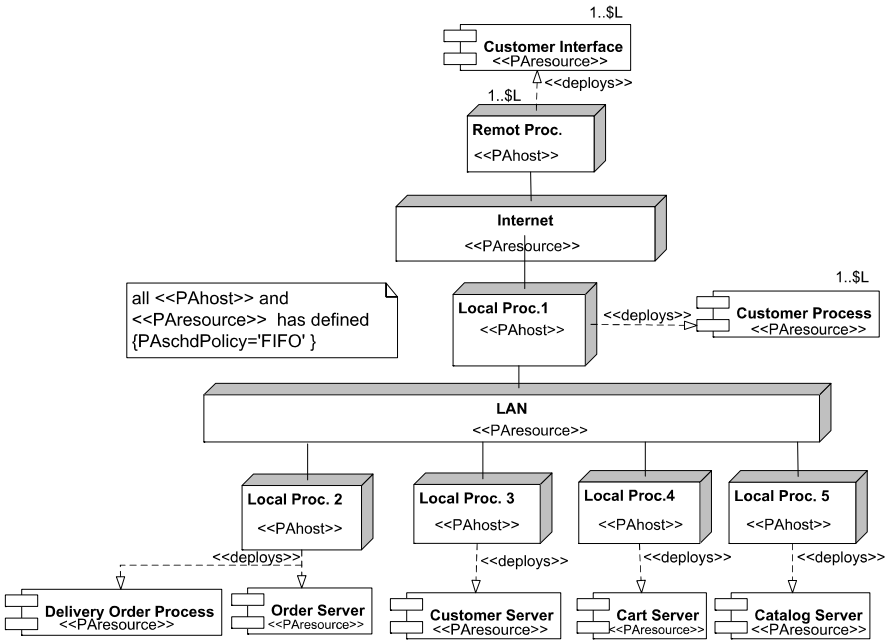


Fig. 4.11 Annotated deployment diagram

Annotations at the architectural phase can represent different performance-related data. For example, in Fig. 4.6 the resource demand of the service to read the status of a catalog is annotated on the corresponding component interface, and yet the same component CatalogServer is annotated with its policy of scheduling for pending requests. Similarly, in Fig. 4.8 the workload originated from triggering the service of browsing the catalog is annotated on the first message of the scenario represented by the UML sequence diagram.

Following the path on the right-hand side of Fig. 4.3, we notice that our initial performance requirement on the BrowseCatalog functionality in Fig. 4.4 is reflected in the annotation of the BrowseCatalog interface delay in Fig. 4.6. The latter annotation also refines the original requirement, in that the time limit is interpreted in the annotated component diagram as an average value.

If we focus on the architectural phase of the development process of Fig. 4.3, and we run the horizontal path leading from the Architectural Design to a Software Architecture Performance Model, we can generate a Queueing Network (QN) model from the previously introduced artifacts (i.e. the set of annotated UML models of the e-commerce example). The QN structure is shown in Fig. 4.9.

In order to perform analysis on this model, its parameterizations must be completed. As will be discussed in Chap. 6, depending on the available information the analysis can be totally or partially symbolic and it is usually oriented, at this development phase, at comparing alternative architectural designs.

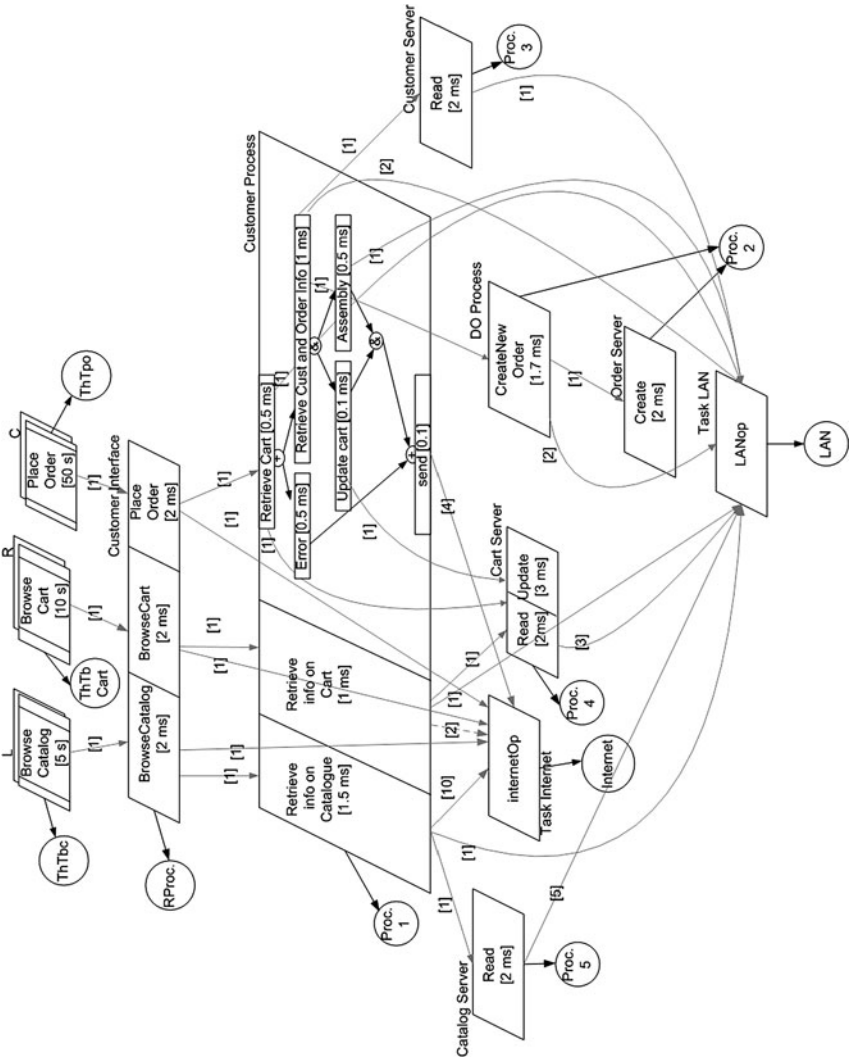


Fig. 4.12 Layered Queueing Network model

Following the development process (i.e. the left-hand side of Fig. 4.3), the detailed design of the e-commerce application is produced in terms of algorithms and data structures of each software component. This step may terminate with the construction of flat and annotated UML deployment diagrams of the software system, as shown, respectively, in Figs. 4.10 and 4.11. Annotations here may represent the low-level scheduling policy of a deployment host (e.g. the one of the operating system running on the specific host).

From these additional artifacts a further (more detailed) performance model can be generated, possibly using a different modeling notation, such as the Layered Queueing Network shown in Fig. 4.12. This model reflects the architectural decomposition of the system and its deployment structure. Hence it contains more information than the Queueing Network shown above, because it has been generated in a later development phase. The evaluation of this model produces performance indices that should be compared with the initial performance requirement.

As illustrated above, our focus in this book is on the process of producing performance models, by means of model transformations, from the software artifacts produced during the development process. In Chap. 5 we will describe in detail several approaches to build these model transformations.

In this chapter we have described the performance analysis in the context of a waterfall software development process. In order to address other software process models we need to provide an idea on how to generalize the Q-model previously described.

Let us recall that at each phase the software artifacts represent the system at different levels of abstraction, whereas the ultimate target of performance analysis is always to satisfy the initially formulated performance requirements. Therefore the performance models generated at each phase aim at the same kind of quantitative analysis, no matter what the name is assigned to the development phase.

Hence, the quantitative analysis can be always based on model transformation that, opportunely defined, generated the performance model at the same level of abstraction of the source development artifact. This amounts to saying that we need to attach at each software artifact the missing information (i.e. the model annotations illustrated above) and a corresponding model transformation that enables the performance model generation. Then the triple <*software artifact, missing information, model transformation*> can be freely embedded in different software processes. Thus, thanks to model transformation techniques, today we can concentrate on how to produce, for each significant software artifact of our development process, a model that allows the quantitative analysis to be made typical of the development activity the software artifact refers to.

Chapter 5
From Software Models to Performance Models

This chapter focuses on the transformational approaches from software system specifications to performance models. These transformations 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. Three approaches will be discussed in detail presenting their foundations and their application to the e-commerce case study. Section 5.3 briefly reviews representatives of other transformational approaches present in the literature. The last section discusses all the presented 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.

5.1 A General Framework for Model Transformation

The majority of the existing approaches for the automated generation of a performance model from a software model can be synthesized as in Fig. 5.1. A software model can be represented with different notations such as UML, Message Sequence Charts, Process Algebras, etc. Functional validation, such as deadlock detection, can be performed on the software model as it is, because only static and/or dynamic aspects of the system are needed for such type of validation. Differently, in order to validate the performance attributes, additional information shall necessarily be introduced in the model, possibly in the form of software annotations. An example of such a piece of information, typically missing in software models, is the operational profile, that is: a stochastic representation of the software system usage (e.g. probability of invocation of each system functionality). If the software model is represented in UML then UML profiles nicely support model annotation, as new stereotypes can be defined to represent the additional information. In the performance domain, the UML Profile for Schedulability, Performance and Time (SPT) [85] is the current standard that provides the necessary definitions for properly annotating an UML model with data related to performance. With the advent of UML 2,

V. Cortellessa et al., *Model-Based Software Performance Analysis*,
DOI 10.1007/978-3-642-13621-4_5, © Springer-Verlag Berlin Heidelberg 2011

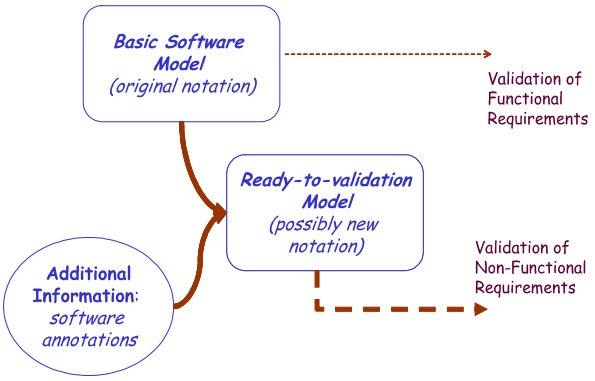


Fig. 5.1 General framework for model transformation

the new UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE) is going to replace SPT [88]. Model transformations take place, in the process of Fig. 5.1, between an annotated software model and a performance model.

Note, however, that this type of transformations are not aimed at refining an existing model like, for example, in classical Platform Independent Model-to-Platform Specific Model transformations in MDA. The transformations we focus on take place between a source and a target model that are at the same level of abstraction, but are represented in different notations, hence they comply with different metamodels [42].[1] The target is a performance model that is ready to be evaluated with existing solvers. In the following, we focus only on the model generation whereas the performance model solution is treated in the next chapter (Chap. 6).

5.2 Some Transformational Approaches at Work

In this section we review three transformational approaches by showing them at work on the e-commerce system we presented and modeled in Chap. 2. They are UML-ψ [26, 34, 82], the Petriu's approach that from UML diagrams obtains Layered Queueing Networks (LQN) [62, 63, 93, 94], and SAP•one [47, 48]. All of them start from UML diagrams, stereotyped with SPT profile (introduced in Sect. 3.8), and generate three different performance models: UML-ψ reaches simulation models (introduced in Sect. 3.7); the Petriu's approach generates LQN (defined in Sect. 3.4), and SAP•one obtains Queueing Networks (already presented in Sect. 3.2).

The choice fell on them for several reasons:

- **All Get in Input UML Diagrams**—the use of the same notation should prevent the reader to get lost in the software specification while studying the approaches.

[1] Readers that are not familiar with the basic concepts of Model-Driven Engineering, such as model/metamodel hierarchy, can refer to [36] for an introductory reading on this topic.

Moreover, she should get through the diagrams quickly since they are already discussed in Sect. 2.2. Finally, a reader can easily understand the different, software and performance-related, modeling information the three approaches require in input and capture the different flavors of the methodologies.

- **Different Target Models at Different Level of Details**—the section reports a significant view of performance models by providing insights on their potentialities. The performance models are used from the methodologies at different level of details and this permits to show the application of performance models, at different abstraction level, since the earliest phases of the software lifecycle.
- **Tool Support**—the selected approaches have been implemented fully or partially in (academic) tools to permit a mostly automated process of generation and analysis of the performance models. This aspect is very important in model-based software performance analysis since it reduces the time and the effort required to carry on the validation of software performance requirements.

The subsequent three sections are each one structured as follows: after a brief introduction of the approach it is divided into five parts. *(i)* Software Specification where the required software specification is described in terms of which UML diagrams and additional information are needed; *(ii)* Performance Model that specifies in details the generated performance model giving insights on its usage and on the meaning the methodology assigns to each performance model element; *(iii)* Software to Performance Model Mapping Rules that gives details on the transformation; *(iv)* E-commerce System Modeling where the methodology is applied on the e-commerce system; and *(v)* Tool Support that reports on the implementation state of the approach.

5.2.1 UML-ψ: From UML to a Simulation Model

The approach proposed by Balsamo and Marzolla in [26, 34, 82] generates a process-oriented discrete-event simulation model of software system at a high level of abstraction. The considered software model is a UML software specification describing the software architecture of the system. The used UML 1.x diagrams are use case, activity and deployment diagrams. The diagrams are annotated according to a subset of the UML SPT profile. Such annotations are used to parameterize the simulation model: the workload offered to the system, the resource consumption associated to each processing steps, the characteristics of each resources. The approach defines an (almost) one-to-one correspondence between the entities expressed in the UML model and the entities or processes in the simulation model. Thus, the obtained simulation model has the same structure of the software model. This correspondence allows easy visualization of the performance results back to the software specification. The approach implements this visualization by using suitably tag values of the SPT stereotypes. For instance, if the simulation reports that the utilization of a processing node (that is a deployment node annotated with «PAhost»

stereotype) is 90%, UML-Ψ is able to insert in the «PAhost» stereotype the PAutilizzation tag value containing the calculated value, that is 90%.

The performance figures derivable by UML-Ψ are: mean execution time of actions and use cases, utilization and throughput of resources. The system behavior is described as a workflow of activities/actions since each use case is described as a one or more activity diagrams. This modeling allows the designer to study via UML-ψ the execution time of both use cases and single activities in the corresponding activity diagram(s). As will be described in detail later, each activity is associated to physical, passive and active, resources whose contribution is needed to accomplish the activity. Thus, the approach permits one to measure the utilization and the throughput of such resources.

With respect to the Q-model presented in Chap. 4, this approach can be basically applied to all development phases, because there are no actual limits to the level of detail that a simulation model can describe in a software system. Therefore, we envisage this approach as suitable for all the development phases illustrated in Fig. 4.3.

Software Specification

UML-Ψ requires that the software system is modeled by UML 1.x use case, deployment and activity diagrams annotated with the quantitative information needed to carry on the performance analysis.

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 the use cases associated to the same actor must be 1, that is $\sum_{j=0}^{k} p_{i,j} = 1$, for each i.

Actors in the use case diagram are stereotyped with ≪PAOpenWorkload≫ or ≪PAClosedWorkload≫ denoting unlimited (open) and finite (closed) number of users accessing the system, respectively. For actors representing open workloads we must specify the interarrival pattern of users (RTarrivalPattern tagged value). For closed workloads we must specify *(i)* the number of system users (PApopulation tagged value) and *(ii)* the external delay experienced by users (PAextDelay tag value).

For an example of the use case diagram UML-Ψ requires, please refer to Fig. 5.6.

Deployment diagrams are used to describe the physical resources (processors) which are available. Differently to the standard usage of such diagram, the UML-Ψ approach does not use this diagram to represent the deployment of the software units over the hardware platform.

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”) indicating the scheduling policy of the processor; PActxSwT representing the context switch time; PArate that

	Process Type	Interactions with
Workloads	OpenWorkload	OpenWorkloadUser
	OpenWorkloadUser	CompositeAction
	ClosedWorkload	ClosedWorkloadUser
	ClosedWorkloadUser	CompositeAction
Activities	SimpleAction	Any Action, ActiveResource
	CompositeAction	Any Action
	ForkAction	Any Action
	JoinAction	Any Action
	AcquireAction	Any Action, PassiveResource
	ReleaseAction	Any Action, PassiveResource
Resources	ActiveResource	SimpleAction
	PassiveResource	AcquireAction, ReleaseAction

Fig. 5.2 UML-Ψ simulation process types

is 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.

See Fig. 5.7 for an example of the deployment diagram UML-Ψ requires.

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).

Each step of an activity diagram is stereotyped as ≪PAstep≫, and can be annotated with the following tagged values: `PArep` to express the number of times this step has to be repeated; `PAdelay` to indicate an additional delay in the execution of this step, for example to model a user interaction; `PAinterval` to annotate the time between repetitions of this step, if it has to be repeated multiple times; `PAdemand` for the processing demand of the step; and `PAhost` for the name of the host (deployment diagram node instance) to which the service is requested. In Fig. 5.9 we report an example of the UML-Ψ activity diagram.

Simulation Model

Simulation processes can be divided into three families corresponding to workloads, resources and activities. Several types of processes belong to each identified family as reported in the table on the left side of Fig. 5.2.

The workloads family contains four different process types used to simulate both open and closed workloads. *OpenWorkload* creates an infinite number of *OpenWorkloadUser* processes that represent the requests arriving to the software system. After each process creation, OpenWorkload pauses for a random amount of time to simulate the interarrival time between two different requests. OpenWorkloadUser is responsible for activating the system behavior that satisfies the request and then it terminates. Such system behavior is modeled as a *CompositeAction* process (see Activities family). OpenWorkload interacts with (infinite) OpenWorkloadUser, which in turn interacts with CompositeAction.

ClosedWorkload creates a fixed population of requests by instantiating a corresponding number of *ClosedWorkloadUser* processes. When all the requests are activated, the ClosedWorkload process terminates. Each ClosedWorkloadUser periodically triggers the (possibly different) system behavior satisfying the request it represents by waiting a random amount of time between two subsequent activations. As before, the activated system behavior is modeled as a CompositeAction process. ClosedWorkload interacts with ClosedWorkloadUser, which in turn interacts with CompositeAction.

The Activities family contains the processes that model the different types of actions the system must execute to satisfy the arriving requests. *CompositeAction* models a complex computation made of a number of sub-computations and it activates the first behavioral sub-step that can be any action listed in Fig. 5.2.

SimpleAction models a basic computational step executed by the system. This action can be repeated a number of time and may require service to an active resource. It interacts possibly with an *ActiveResource* and then with one of the subsequent actions.

ForkAction and *JoinAction* represent the beginning and the end of a parallel computation, respectively. ForkAction consists on activating all the parallel actions of the parallel computation. JoinAction, instead, waits for the completion of all its preceding actions and then activates one of its successors.

The last two actions the simulation model has are the *AcquireAction* and *ReleaseAction* actions. The former implements the acquisition of a passive resource and interacts with PassiveResource before activating one of the actions it precedes. The latter, instead, represents the action releasing a passive resource previously acquired. It interacts first with PassiveResource and then it activates one of the successor actions.

Finally, the resource process family is composed by two process types: *ActiveResource* modeling an active resource and *PassiveResource* representing a passive resource. An ActiveResource process waits for requests and satisfies them according to its scheduling policy. Also PassiveResource waits for requests, and satisfies them if the available amount of passive resource is sufficient for the request, otherwise it blocks until the necessary amount of passive resource becomes available. It interacts with AcquireAction and ReleaseAction processes.

A simplified representation of the simulation model is given in Fig. 5.3 that shows, through a diagram (similar to a collaboration diagram), an example of how processes of a simulation model are instantiated and activated. Nodes in the diagram represent the instances of the simulation process types described above, while

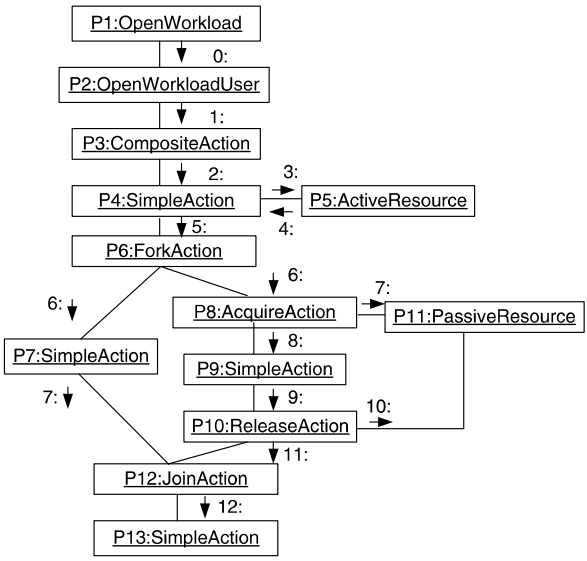


Fig. 5.3 UML-Ψ simulation processes instantiation

the arrows and respective labels model the order in which the processes are activated and the control flow among the processes. The aim of the figure is to show how the previous process types are combined in a model. In particular, the first instantiated process is the one for the workload (in this case the OpenWorkload) and the associated user (i.e., the OpenWorkloadUser). This last process activates the CompositeAction process modeling a system behavior (i.e., the selected one). The subsequent portion of the diagram describes the processes simulating the system behavior: a SimpleAction consuming some ActiveResource, a ForkAction modeling the starting point of two parallel execution flows. The right side flow consumes some PassiveResource by first acquiring it (AcquireAction process) and, when finished, releasing it (ReleaseAction process). The following JoinAction represents the end point of the parallel executions, ending with the conclusive SimpleAction.

Software to Performance Model Mapping Rules

UML elements are mapped directly into simulation processes as illustrated in Fig. 5.4. 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.

The translation algorithm is very simple and performs the following basic steps (refer to Fig. 5.5 for details on the mapping rules):

1. Each actor is translated into the corresponding workload and workload users process (open or closed ones) depending on its stereotype;

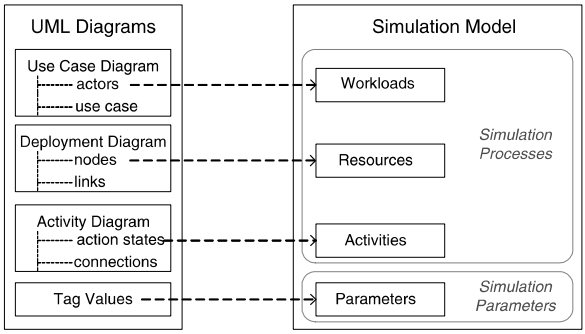


Fig. 5.4 UML-Ψ mapping rules

2. Each action in activity diagrams is translated into the related type of processing step (fork/join action, simple/composite action, require/release action) depending on its UML type and the associated stereotype;
3. Actions are linked together according to the predecessor-successor relation that relates them in the activity diagrams;
4. Finally, nodes in the deployment diagrams are mapped into processing or passive resources depending on their stereotype.

When a workload user is activated, it chooses the use case to execute among the ones it can trigger (i.e., the use cases associated to it in the use case diagram). The activity diagram associated with the selected use case is translated into a set

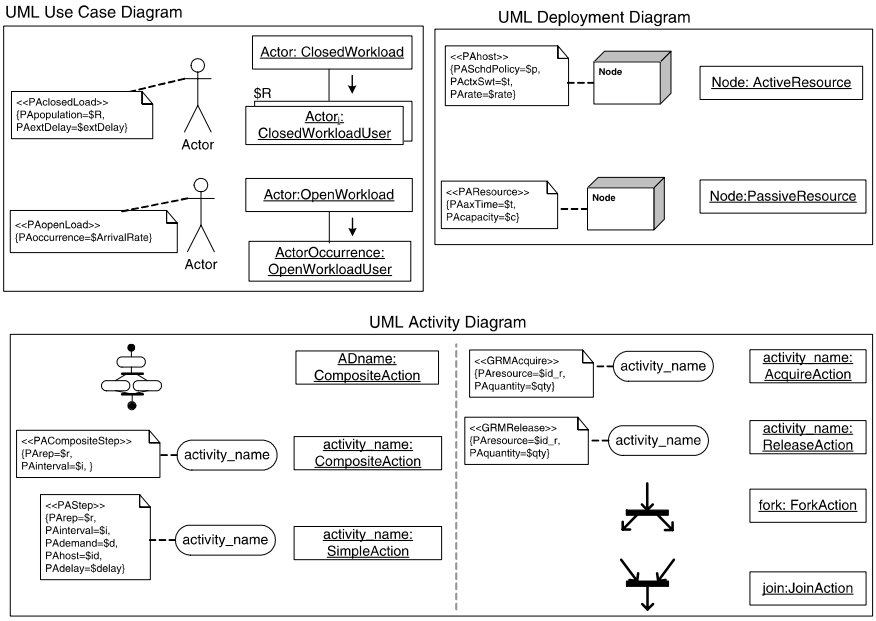


Fig. 5.5 Detailed UML-Ψ mapping rules

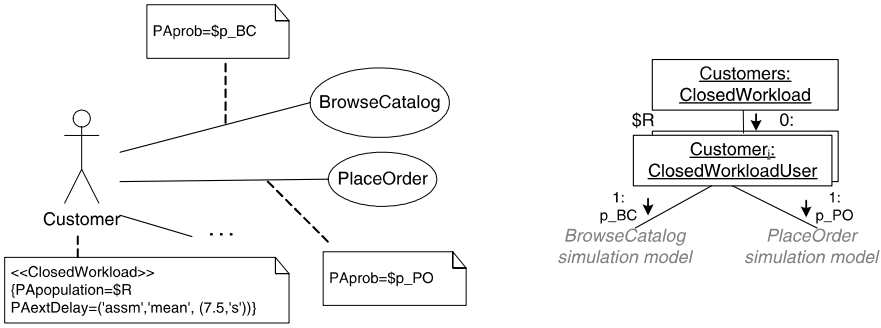


Fig. 5.6 Use case diagram and the generated simulation processes for the e-commerce system

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. During each step execution, the corresponding process consumes (active/passive) resources according to the annotation in the activity diagram. This means that the step process interacts with the involved resource process/es to simulate resources consumption. At the end of the scenario, the workload user is resumed.

UML-Ψ Approach on E-commerce System

This section shows the application of the UML-Ψ approach to the e-commerce system introduced in Sect. 2.2. In particular, the presentation is organized into three macro-steps, one for each type of the considered UML diagrams. The (incremental) simulation models resulting from the elaboration of each UML diagram are represented by means of the simplified representation of the simulation model introduced above.

Each macro-step description is supported by a figure, showing on the left side the UML diagram the step deals with and, on the right side, the resulting (partial) simulation model.

Use Case Diagram Processing—The left side of Fig. 5.6 shows the annotated use case diagram for the UML-Ψ approach. The diagram contains an actor, the Customer, and two use cases, BrowseCatalog and PlaceOrder. The performance related information is annotated in the diagram by means of the SPT Profile in a parametric way that is through variables identified by the $ symbol according to the UML syntax.

As marked by the ClosedWorkload stereotype, the Customer represents the source of the workload for the e-commerce system. The workload is closed, that is the number of requests that circulate in the system is fixed and such number is indicated by the PApopulation tagged value. In Fig. 5.6, the population is set to $R. When the customer receives back a response to its request, she generates a new re-

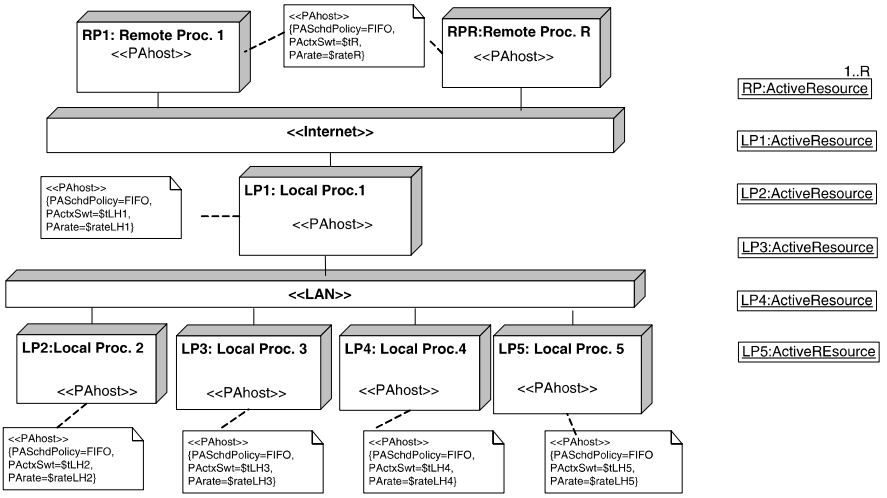


Fig. 5.7 UML-Ψ simulation processes originating from the e-commerce deployment diagram

quest. The time needed to formulate the new request is indicated via PAextDelay tagged value. In the figure this delay is assumed to be 7.5 seconds on average.

At each time, a Customer can generate either a BrowseCatalog or a PlaceOrder request with $\$p\_BC$ or $\$p\_PO$ probabilities, respectively, as indicated by the PAprob tagged value on the associations between the actor and the use cases.

On the right side of Fig. 5.6, the simulation processes generated from the use case diagram are shown.

The use case diagram processing generates $\$R + 1$ simulation processes: **Customers** of the *ClosedWorkload* type that, in turn, creates the fixed population of $R **Customer**, that are *ClosedWorkloadUser* processes. Each ClosedWorkloadUser periodically triggers the (possibly different) system behavior by waiting a random amount of time between two subsequent activations, as annotated in the PAextDelay tagged value. For the process communication, ClosedWorkload interacts with ClosedWorkloadUser, which in turn interacts with the processes modeling the use cases the Customer can activate. Of course the Customer process will embed the $\$p\_BC$ and $\$p\_PO$ probabilities to suitably model the activation of the system functionalities.

Deployment Diagram Processing—On the left side of Fig. 5.7 is shown the deployment diagram for the e-commerce system. In the diagram the nodes corresponding to processors are annotated with the PAhost stereotype. Such annotation indicates that the nodes are active resources and the tagged values show the scheduling policy (i.e., PASchdPolicy tagged value), the time spent from the processor to make a context switching (that is, the PActxSwt tagged value) and the processing rate (in the PArate tagged value). The scheduling policy for all the hosts in the figure is FIFO (First In First Out), while the other pieces of information are parametric, as indicated by the $ symbol.

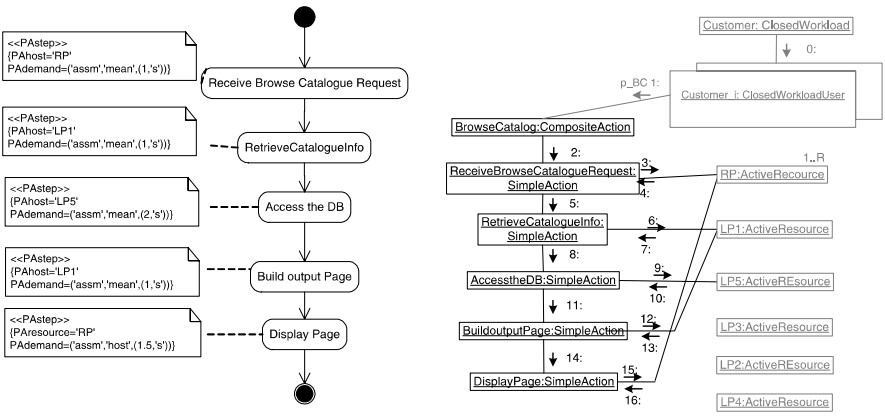


Fig. 5.8 BrowseCatalog activity diagrams and the corresponding UML-Ψ simulation processes

On the right side of Fig. 5.7, the corresponding simulation processes are shown. For each node stereotyped with PAhost, an ActiveResource process is generated whose behavior is set according to the annotation in the tagged values of stereotype. The Internet and LAN nodes are instead not stereotyped and hence they will not be part of the UML-Ψ simulation model. It is assumed that the time needed to access these networks is split between the caller and the provider of a remote processing step.

Activity Diagrams Processing—In Fig. 5.8, from the left side to the right side, the annotated activity diagram and the corresponding simulation processes for the BrowseCatalogue use case are shown. The simulation processes in gray are the ones previously generated by processing the use case and the deployment diagrams.

In the activity diagram, the actions are annotated via the *PAstep* stereotype. The used tagged values are the: *PAhost* indicating the active resource responsible for the computation and the *PAdemand* that indicates the processing time the action requires to the resource. In the figure, all the service demands are expressed in seconds and represent the assumed mean times required to the active resource in charge of the job.

UML-Ψ generates a SimpleAction simulation process for each PAstep stereotyped activity in the diagram. In addition it generates a CompositeAction process that represents the starting point of the BrowseCatalogue scenario. In the representation of the simulation model on the right side of Fig. 5.8, we also show a line for the communication channels among the processes. The numbered arrows associated with such lines represent the activation and execution of the processes when the BrowseCatalogue scenario is activated. By following the arrows a Customer asks the system to browse the catalogue then the CompositeAction starts to execute. When it finishes its work, it activates the ReceiveBrowseCatalogueRequest process that during its execution activates the RP ActiveResource asking to spend the time specified via PAdemand in the annotation of the Receive Browse Catalogue Request activity. When the ReceiveBrowseCatalogueRequest process terminates, the subsequent process starts. The process activation proceeds until the end of the scenario.

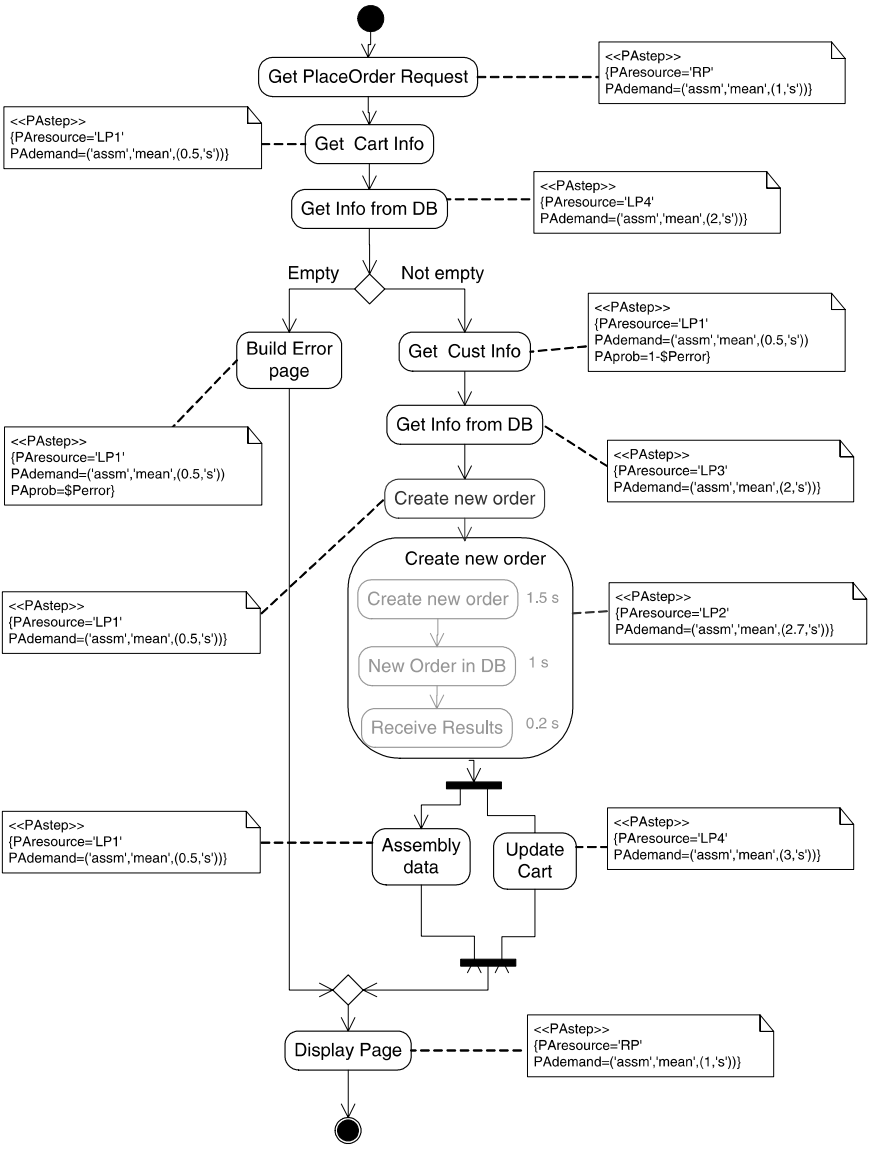


Fig. 5.9 PlaceOrder activity diagram

Figure 5.9 reports the UML-Ψ modeling of the Place Order scenario. The scenario provides two interesting behavioral patterns: the alternative and parallel executions. The former is related to the emptiness of the cart. If the cart is empty the scenario terminates showing an error page to the customer, otherwise it proceeds with the placement of the order. The probability of executing one of the two alternative behaviors is annotated by the PAprob tagged value. The error scenario can

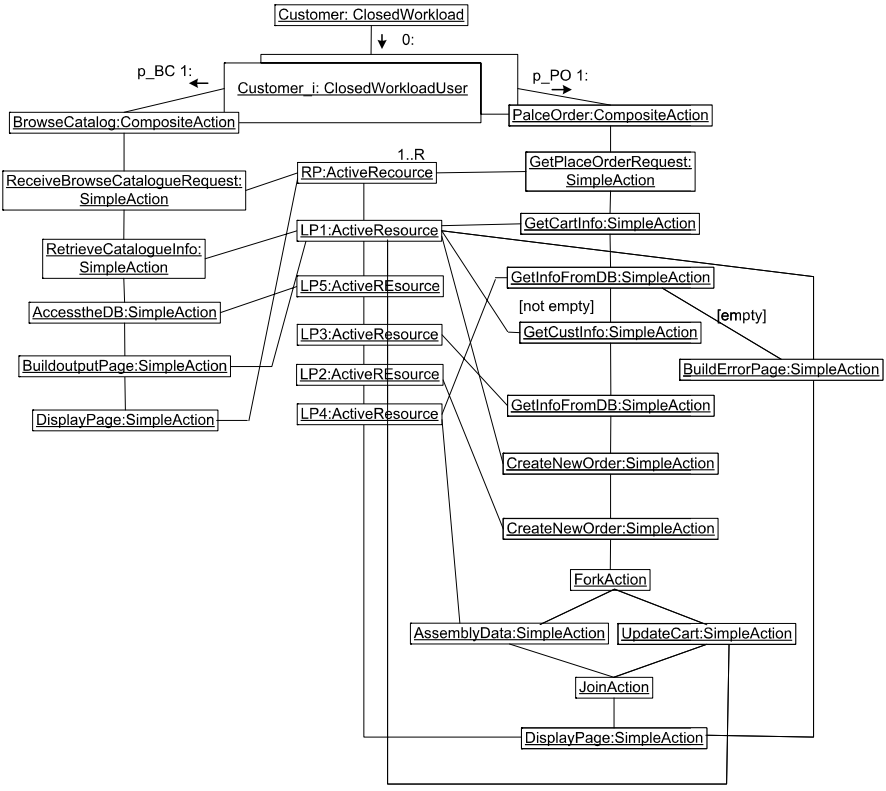


Fig. 5.10 UML-Ψ model for the e-commerce system

happen with $Perror probability, whereas the order placement completes with 1-$Perror probability. The latter corresponds to the last tasks before the order page is displayed. In fact, the "update cart" and the "assembly data" activities are executing in parallel to minimize the response time. Finally, note that in the activity diagram the *Create new order* is a macro-activity containing three basic actions. In the transformation toward the simulation model the basic actions are not considered, whereas the composite one will contribute to generate a *SimpleAction* process since it is the only one stereotyped by the *PAstep*.

Figure 5.10 shows the simulation model UML-Ψ generates for the e-commerce system. For the sake of presentation the target model is shown, considering only two use cases, the place order and the browse catalogue functionalities, and the deployment diagram of the whole system. In the center of the figure, from top to bottom, the processes simulating the closed workload and the physical resources are presented. The processes simulating the activities to be performed for the browse catalogue and for the place order scenarios appear on the left and right sides of the figure, respectively. The link among the processes represents the communication among them whereas the numbered arrows upon the links indicate the order in which

the process interactions occur, whenever a scenario is activated by the user. The probability the customer asks for a use case is reported on the link between the *ClosedWorkloadUser*, modeling a customer, and the *compositeAction* modeling the starting point of the use case.

Tool Support

The approach has been implemented in the prototype tool UML-Ψ (UML Performance SImulator).

The UML-Ψ tool parses the XMI representation [89] of the annotated UML model. Currently the XMI variant used by the ArgoUML [14] tool is supported.

While parsing the XMI representation, the UML-Ψ tool builds an internal representation of the UML model from which it derives a process-oriented simulation model. The obtained simulation model is executed by using both user-supplied parameters, given as tagged values in the UML diagrams, and the parameters included in a configuration file. Examples of simulation parameters are the repetition number an action is executed, the service demand of actions, expressed as random variables with a given distribution, the scheduling policies of active resources, and the confidence range.

The specification of tag values is given by the Tag Value Language, which is a subset of the Perl language [8]. This is motivated by the need to express such values in a complex way, for example by using expressions such as arithmetic or boolean ones. The configuration file is a Perl program which is executed before evaluating tag values. In this way it is possible to define variables in the configuration file and use them inside tag values.

The simulation model is implemented as a C++ program, using the facilities provided by the general-purpose simulation library. Upon execution of the simulation model the computed results are inserted into the XMI document as tagged values associated with the UML elements they refer to. In this way the results of performance analysis are available to the user which can access them by opening again the UML model by using the CASE tool. The software designer can then check whether the software architecture meets the performance goal and can possibly iterate the whole specification and analysis process. Simulation results are computed with steady state analysis and with confidence intervals.

5.2.2 From UML to a Layered Queueing Network

In [62, 63, 93–95] Petriu et al. incrementally propose a software performance engineering methodology that generates a Layered Queueing Network model of a software system at the software architecture level. The approach makes use of design patterns that introduce abstract design artifacts describing a specific type of collaboration between a set of prototypical components that play well-defined roles.

The approach assumes that the software architecture is modeled by UML 1.x diagrams. In particular, the software architecture structure is modeled by means of a UML collaboration diagram enriched by the indication of the design patterns the architecture conforms to, whereas the software architecture dynamics is modeled by means of a set of UML activity diagrams, one for each performance critical scenario. The UML modeling is completed by a deployment diagram embedding information about: (*i*) the deployment of software components over the hardware platform, and *(ii)* performance characteristics of devices composing the hardware platform. The diagrams are annotated according to the UML SPT profile. The following annotations are used to parameterize the LQN model: the workload offered to the system from the external users and to the lower tasks from the upper ones, the hardware/software resource consumption associated to each processing step (one or a set of actions in an activity diagram), the characteristics of each hardware resource (e.g., scheduling policy).

The transformation algorithm for the Petriu's approach is quite complex and is composed by two main steps:

1. *Derivation of the LQN structure*—In this step, only the structural aspect of the software system is taken into account. The inputs of this step are one or more UML collaborations representing the high-level architecture of the software system and its architectural patterns, and the deployment diagram. Each high-level software component instance is transformed into an LQN software task. The connections among the tasks come from the high-level architecture and the involved patterns, since they reflects the links among component instances in the architecture. The nodes in the UML deployment diagram become devices in the LQN. Finally, the connections between tasks and devices reflect the deployment choices described in the UML deployment diagram.
2. *Derivation of the LQN entries, phases, activities details*—The inputs of such step are represented by the LQN structure derived from the first step, the involved architectural patterns and the activity diagrams modeling the performance critical scenarios. For each UML activity a LQN sub-model is derived, and all the obtained sub-models are merged together at the end. Each swimlane in the activity diagram describes the actions the related component instance (LQN task) must execute to satisfy a service request. From the swimlanes the approach derives the *entry* for the corresponding *tasks* and its description either in terms of *phases* or *activity graph*. From the architectural patterns, the step extracts the ways different entries are connected, thus making consistent the inter-component communications with the pattern(s).

The obtained LQN model can be analytically solved or simulated by using the tool set presented in [52, 53], composed by the LQNSIM simulator, and the LQNS analytical solver. The LQNS solver makes use of specific solution techniques defined for LQN models that iteratively solve each layer of the LQN, starting from the bottom-most one. The performance figures calculated by the approach are all the ones allowed by the LQN model and by the tools used to evaluate it. Table 5.1 summarizes the main performance indices calculated from the LQN solution. The

Table 5.1 Main LQN model results

Figure name	Figure description
Mean Delay (Variance) for a Rendezvous	It is the (variance of) queueing time for a request from a client to a server. It does not include (the variance of) the time the customer spends at the server.
Mean Delay for a Send-No-Reply Request	It is the time the request spends in queue and in service in phase one at the destination.
Mean Delay for a Join	It is the maximum of the sum of the service times for each branch of a fork. The variance of the join time is also computed.
Service Time	The service time is the total time a phase or activity uses for processing a request.
Service Time Variance	It is the variance of the service time for the phases and activities in the model.
Throughputs and Utilizations per Phase	They are throughputs by entry and activity, and the utilization by phase and activity. The utilization is the task utilization, i.e., the reciprocal of the service time for the task.
Utilization and Waiting per Phase for Processor	They are the processor utilization and the queueing times for every entry and activity running on the processor.

table reports in the first column the name of the figure and in the second column its description.

With respect to the Q-model presented in Chap. 4, this approach can be applied to several development phases, mostly the architectural and the low-level design. In fact, LQN are a suitable notation for representing software components and platform devices where these components run. Their capability to detail the internal structure and behavior of a component enables the possibility of evolving the development artifacts as long as the process evolves.

Software Specification

The **high-level architecture** defines the structure of the system in terms of the concurrent (distributed) component instances composing it. Moreover, it shows the architectural patterns involved and the roles the components hold in each pattern.

A pattern introduces a higher level of abstraction design artifact as it describes a specific type of collaboration between a set of components playing well-defined roles (e.g., client–server, pipeline and filters, blackboard, etc.)

Each design pattern is described from two perspectives: structurally, by indicating the type of components involved and their interconnections; dynamically, by modeling the way those components/roles interact with each other. The components are usually concurrent entities that are executed in different threads of control, compete for resources, and their interaction may require some synchronization.

The patterns are represented as UML collaborations (not to be confused with UML collaboration diagrams that are interaction diagrams) [86]. A collaboration

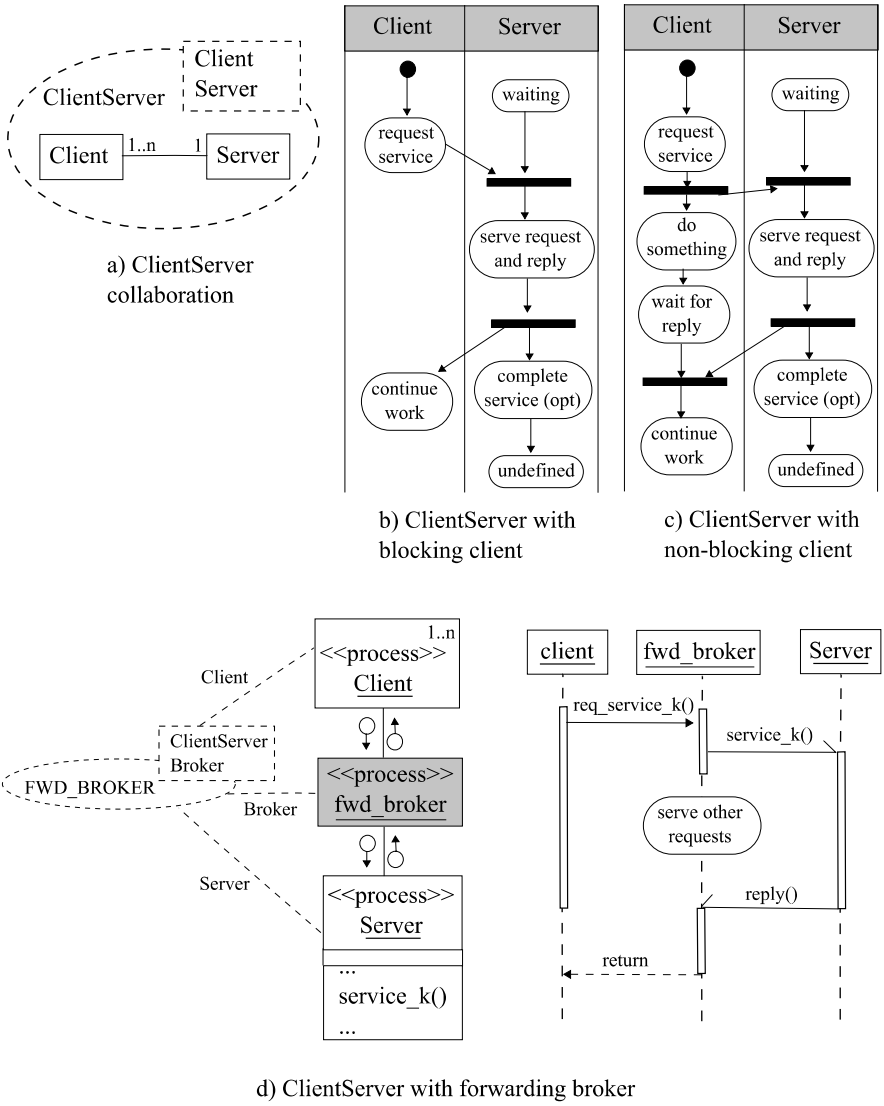


Fig. 5.11 Client–Server architectural patterns

is a notation for describing "a society of classes, interfaces, and other elements that work together to provide some cooperative behavior that is larger than the sum of all its parts" [55]. A collaboration has two aspects: structural (usually represented by a class/object diagram) and behavioral (represented by an interaction diagram). The symbol for a collaboration is an ellipse with dashed lines that may have "embedded" squares showing the roles played by different pattern participants.

An example of design pattern is the Client–Server architectural pattern shown in Fig. 5.11. Indeed, Fig. 5.11 shows three variants of the Client–Server architectural pattern [95]. Figure 5.11(a) shows the structural description of this pattern, identifying two roles, the Client and the Server. At least one (instance of) Client is connected to a single Server. Figure 5.11(b) shows a synchronous communication style among client and server (where the client sends the request and remains blocked until the sender replies), whereas the one in Fig. 5.11(c) describes an asynchronous communication style among them (where the client continues its work after sending the request and will accept the server's replay later). Finally, Fig. 5.11(d) shows the Client–Server pattern where the communication among the client and the server is implemented by involving a forwarding broker that acts as an intermediary between them in all their interactions. Please refer to [95, 96] for a wider example of architectural patterns definition.

The rationale behind the definition and usage of high-level architectural patterns is that, in general, developers of software systems are mostly interested in the components that are part of their application, and less in the details of the underlying middleware, operating system or networking software. The use of architectural patterns allows them to separate concerns in the modeling and to hide unnecessary details at the application level. For example, client–server applications using a CORBA interface do not have to show explicitly the "broker" component in their architecture (as it is not part of the software application). Instead, the apposite architectural pattern (that is the Client–Server with forwarding broker) can be used to indicate the type of desired client–server connection. As a result, Petriu's approach having in input this specific pattern generates a performance model explicitly representing the broker and its interaction with the client and server counterparts even if these concepts are not present in the application UML design.

Activity Diagrams are used to describe the system scenarios judged as critical for performance aspects. These diagrams are annotated by using the stereotypes of the Schedulability Performance and Time (SPT) Profile [85].

In particular, open and closed workloads are annotated in the first activity of the diagram by means of «PAopenLoad» and «PAclosedLoad» stereotypes, respectively.

All the activities in the diagram are annotated by «PAstep» stereotype to indicate: *(i)* the average service demand of the processing step on its processor (through the PAdemand tag value); *(ii)* the activity execution probability in case of optional/alternative execution flows (through the PAprob tag value); *(iii)* the name of the device and average number of visits in case the activity makes a request to a device (through the PAextOp tag value); and finally *(iv)* the average number of visits in case the activity requires service to another task entry (through the PArep tag value).

We recall that the authors enrich the activity diagrams with the concept of swimlanes in order to model the objects responsibility for the involved actions.

The **Deployment Diagram** provides information about the hardware platform and the deployment of the software system on it. The annotations added by using a SPT profile aim at identifying active and passive resources. In particular, nodes representing processors (that is active resources) are annotated by means of «PAhost»

stereotype to specify the scheduling policy (`PASchdPolicy` tag value), and nodes modeling the passive resources are annotated by means of the «`PAresource`» stereotype.

Layered Queueing Network

The approach uses the LQN models with the original semantics as introduced in Chap. 3. Indeed, it uses all the features of the LQN model since it defines transformation rules that detail the entry modeling by using activity graphs, if a finer grain of software behavior description is necessary. We recall here that if activity graphs are used, then the LQN model can be only simulated by using the LQNSIM simulator.

Software to Performance Model Mapping Rules

The transformation algorithm for the Petriu's approach is quite complex and applies many mapping rules that, for the sake of presentation, are partitioned into two distinct groups, one for each transformation core step described earlier at the beginning of Sect. 5.2.2.

Step 1—Derivation of the LQN Structure The LQN structure is extracted by the high-level architecture of the software system, the architectural patterns occurring in it, and by the deployment diagram. In particular, the LQN tasks and their interconnections are defined from the first two elements, whereas the definition of the LQN devices and the connections among the tasks and the devices come from the deployment diagram.

Figure 5.12 graphically illustrates the transformation rules devised for the LQN structure definition, by reporting on the left side a software model element and on the right side the corresponding LQN element:

- each Component Instance or Thread in the high-level architecture that is explicitly annotated as «`PAresource`» is translated into a LQN task, by preserving the multiplicity indicated in the high-level software architecture;
- the interconnections among tasks are created following the definition of architectural patterns in the software architecture (see below for specific transformation rules);
- each node in the deployment diagram annotated as «`PAHost`» or «`PAresource`» is translated into a LQN device that inherits all the node characteristics as indicated in the specific stereotype (e.g., the scheduling policy comes from the `PAschdPolicy` attribute of the stereotype);
- the «`deploys`» associations among the component instances and the nodes are translated into the links among the corresponding tasks and devices in the LQN model.

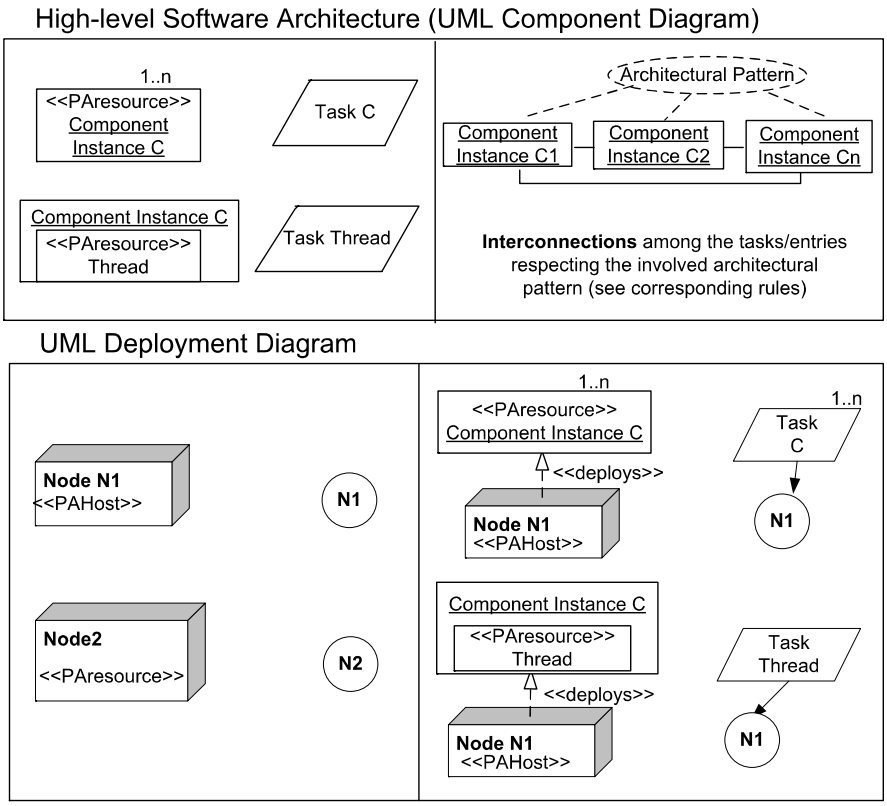


Fig. 5.12 Mapping rules for the LQN structure definition

The set of transformation rules for the LQN structure is completed by the ones that translate the architectural patterns into the interconnections among the LQN tasks. Frequently a software system contains many components involved in various architectural patterns, and a component may play different roles in different patterns. All the task interactions must respect the component communication types provided in the architectural patterns. Hence, the corresponding mapping rules must be applied in a systematic way, pattern by pattern.

Depending on the specific pattern, it can happen that some of such rules introduce additional LQN tasks to model software entities specific to the pattern and not explicitly modeled in the high-level software architecture, as it happens for the Client–Server with forwarding broker architectural pattern.

For the sake of presentation, only the transformation rules for three variants of the Client–Server architectural pattern are here below described. More rules for the transformation of the architectural patterns into LQN can be found in [94, 95].

Figure 5.13 graphically illustrates these rules by showing the architectural patterns on the left side, and the corresponding LQN general structure the pattern induces on the right side.

The first rule refers to the *Client–Server with blocking client* architectural pattern. In this pattern, a client communicates with the server through a synchronous communication, where the client sends a request to the server and blocks until the server reply comes back. A server may offer a wide range of services (represented as the server methods), each one with its own performance attributes (e.g., execution time). A client may invoke more than one of these services at different times. The rule specifies as synchronous the connections among the Client and the Server task entries.

The second rule refers to the *Client–Server with forwarding broker* pattern. In this variant the communication among the Client and the Server is not direct, but it occurs by means of a broker. The forwarding broker is modeled as a LQN multi-server with as many entries as server entries. Each task replication accepts a client's request, passes it to the server, then remains blocked until the server reply comes back and is delivered to the respective client. While a broker replication is blocked, other replications get to run on the processor on behalf of other requests.

The last rule refers to *Client–Server with handle-driven broker* pattern where a client sends two kinds of messages: one to the broker for getting the handle, and the other directly to the desired server entry. Since the broker does the same kind of work for all the requests, no matter what server entry they need, the broker is modeled with a single entry.

Step 2—Derivation of the LQN Entries, Phases, Activities Details The UML activity diagrams describe how the component instances (LQN tasks) collaborate to satisfy the requests incoming the system. From such diagrams Petriu's approach extracts tasks' entries details, defines the reference tasks, and identifies supplementary LQN tasks needed to model operations external to the system (as indicated in the UML activity diagrams through specific annotation).

We recall that the approach requires swimlanes in the UML activity diagrams. Swimlanes are used to model the component instance (or LQN task) responsible for the actions that must be performed to satisfy the incoming request.

As first action of Step 2, the approach parses the activity diagram to check whether it is correct w.r.t. the architectural patterns specified in the high-level architecture. To this goal, the approach overlaps the behavior of the architectural patterns extracted from the high-level architecture with the activity diagram, and verifies whether the communication between concurrent components is consistent with the pattern. Operatively, this is done by identifying the cross-transitions between swimlanes, and by checking if they follow the protocol defined by the pattern. Then, it applies the transformation rule corresponding to the respective pattern to properly define the entries and their interconnections among the LQN tasks.

The above action practically disconnects each swimlane from its neighbors (see Fig. 5.14 for a graphical representation of the result). Each swimlane defines an entry in details.

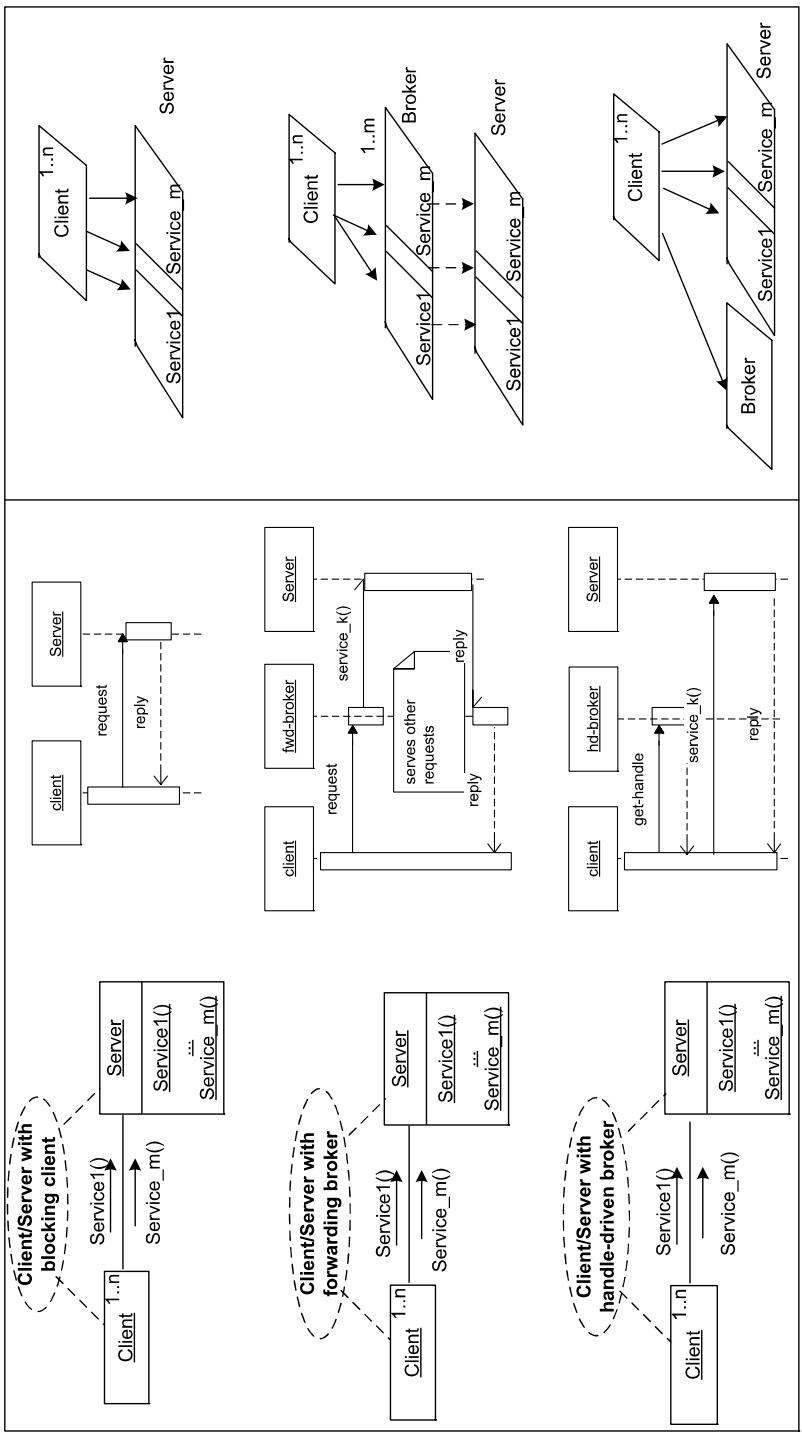


Fig. 5.13 Transformation rules of three variants of the Client–Server architectural pattern

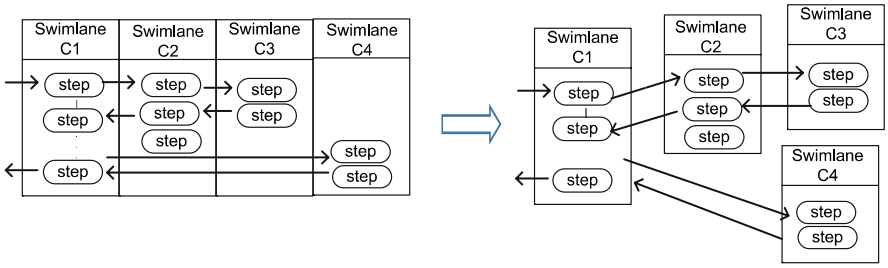


Fig. 5.14 Swimlane disconnection

Figure 5.15 graphically illustrates the rules to identify an entry (i.e., `rule1`), and to model it through phases (that is, `rule2` and `rule3`):

`rule1`: A task entry is generated for each service type offered by the corresponding software component instance. The services offered by each instance (LQN task) are identified by looking at the messages received by it (that is the arrows incoming the swimlane) in every activity diagram taken into account for performance analysis. A new entry is added to the task if a new type of request is received. If a request type is received more than once, its number of repetitions is increased by the contribution to it provided by the current activity diagram. In Fig. 5.15-rule1, from the four separated swimlane, the approach identifies four different entries (from e1 to e4), belonging to the involved tasks.

`rule2`: This rule is used to model an entry as a single phase. It is applied when the reply to the task asking for the service occurs at the end of the swimlane, when all its steps have been executed. In this case, the steps in the swimlane are all collapsed in a single LQN phase. This means that the approach defines a single service demand for the entry (in the figure, $[a]$ for *e1* entry) and the number of requests (in the figure, $[n1], \ldots, [nj]$ on the links) the entry generates to its server task at the end of its execution.

`rule3`: This rule is used when the reply to the client task occurs in the middle of a swimlane, before the end of the entry execution. In this case, the approach divides the entry modeling in two phases, the first one ending when the task replies to its client (*e2,ph1*, in Fig. 5.15-rule3) and the second one modeling the remaining execution (*e2,ph2*, in Fig. 5.15-rule3). The approach also defines two service demands $[a, b]$, a for the first phase and b for the second one, and the pair $[n_i, m_i]$ of mean numbers of service requests provided to the other tasks from each phase, respectively.

`rule4`: This applies if a conditional or non-conditional branching state is encountered while parsing a swimlane. In this case the entry is modeled by means of a LQN activity graph on the basis of the mappings illustrated in Fig. 5.16. The approach creates:

- an activity for a set of steps sequentially executed in a swimlane, whose service demand and service request definitions follow `rule5` and `rule6`, respectively;
- a LQN *&-fork* and an *&-join* node (represented by an & in a circle having, in the first case, one incoming arc and many outcoming arcs, while in the second case

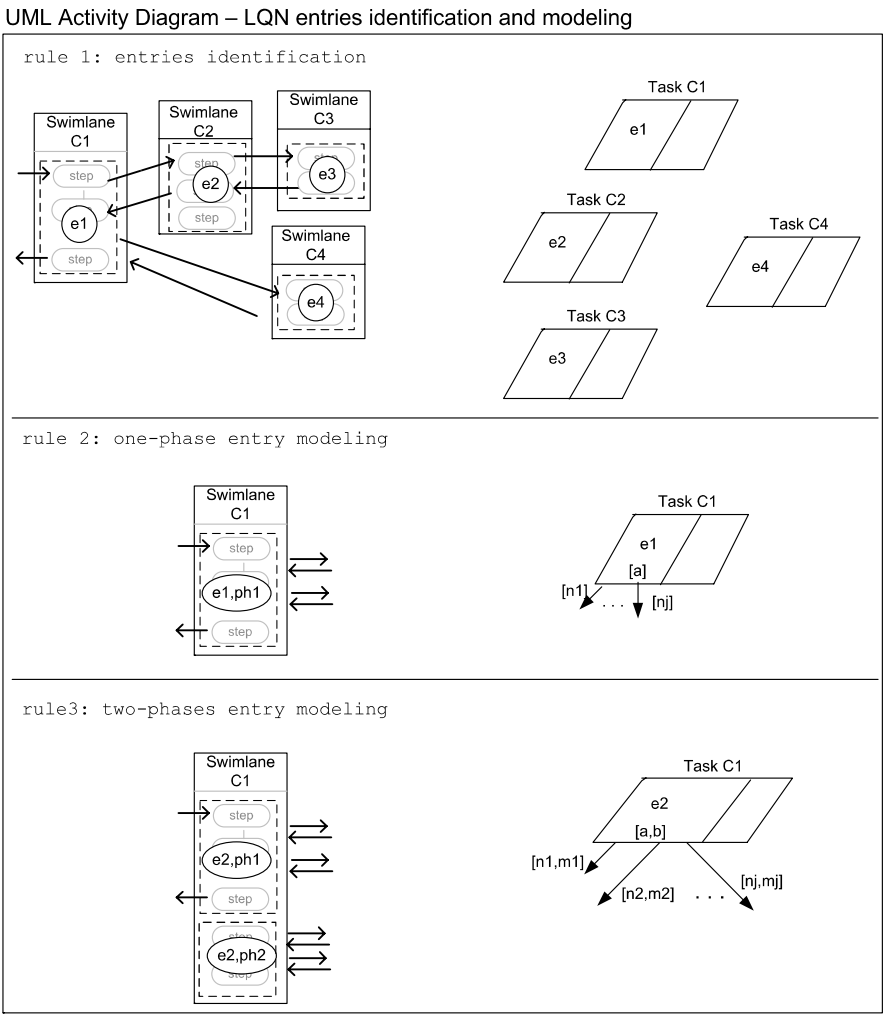


Fig. 5.15 Mapping rules for the LQN entries identification and modeling by using phases

many incoming arcs and only one outcoming arc) to the fork and the corresponding join node encountered in the UML activity diagram, respectively;

- an *or-fork* and an *or-join* node (represented by a + in a circle, having, in the first case, one incoming arc and many outcoming arcs, while in the second case many incoming arcs and only one outcoming arc) to each decision and the corresponding merge node encountered in the UML activity diagram, respectively. The probabilities annotated on the UML steps following the decision node are used to specify the probabilities of the alternative paths starting from the *or-fork* in the LQN model;

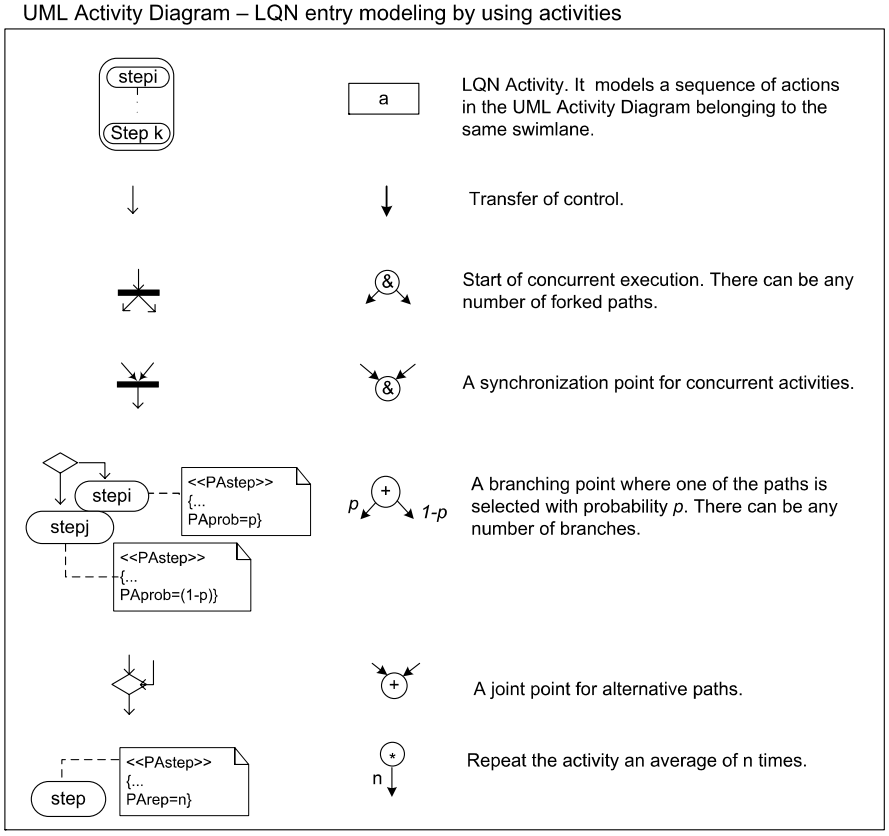


Fig. 5.16 Mapping rules for the LQN entries modeling with LQN activity graph activities

- a *loop* node (represented by a circle with * inside) is introduced whenever a step in the UML activity diagram must be executed several times (as indicated by the PArep tag value in the PAstep stereotype annotating the step). The loop node will precede the LQN activity representing the repeated step and the repetition number (i.e., the PArep tag value) is used to label the arc connecting the loop node with the activity.

Note that this rule does not apply whenever a fork node in the UML diagram is used by a server for sending a reply at the end of the first phase.

`rule5:` This rule defines the mean service demand of a LQN phase (or activity) obtained by using the additional information annotated by means of the «`PAstep`» stereotype. The service demand S is a function of the `PAdemand`, and `PArep` tag values of all the steps involved in the phase (or activity), defined as follows: $S = \sum_{j=1}^{k}(d_j * r_j)$ where k is the number of steps involved in the phase or activity, d_j and r_j represents the `PAdemand` and `PArep` tag values, respectively, of the step j, and r_j is equal to 1 if not specified in the stereotype.

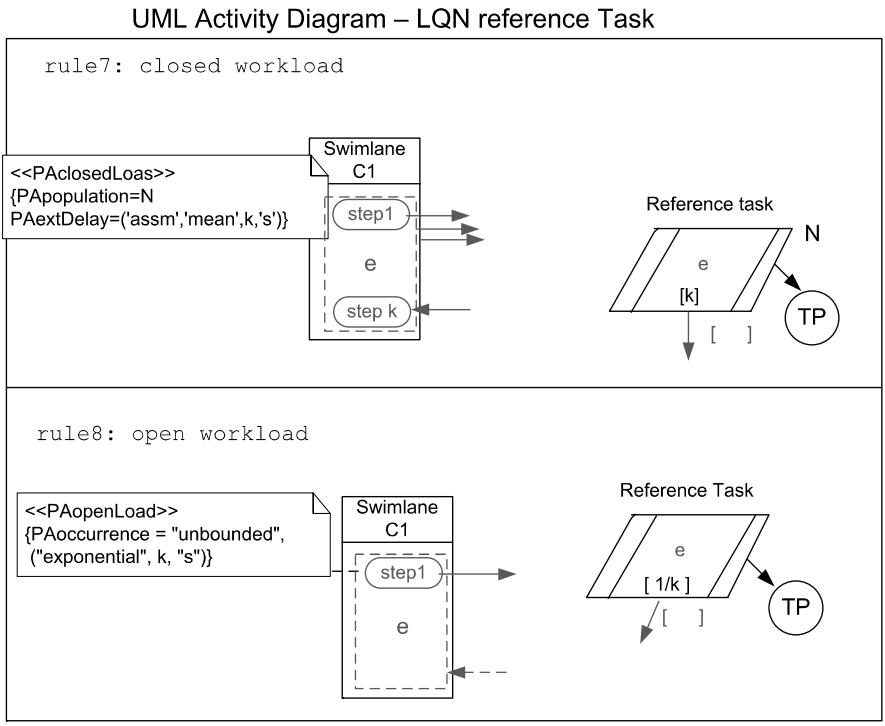


Fig. 5.17 Mapping rules for the LQN task reference modeling

`rule6`: An LQN request arc is generated when a communication is detected between a phase or activity of a component playing the role of client, and the entry of another server, according to the corresponding high-level pattern. The request number of the arc is given by the number of repetitions (annotated in the `PArep` tag value) of the scenario step originating the request multiplied, by the number of requests made in that step. If more steps contained in the same phase are sending a request to the same entry, then their call request numbers are added together.

Figure 5.17 shows the rules that define which identified tasks play the role of reference tasks. A reference task is a task that generates the workload to the LQN. Since Petriu's approach assumes that information on the workload is annotated on the first step of the activity diagram by means of «`PAclosedLoad`» or «`PAopenLoad`» stereotypes, these stereotypes are used to identify and specify the reference tasks. Note that, for each workload, the approach will generate a specific reference task that runs on a "virtual" device modeling an engine that spends the time the reference task specifies. The following rules describe the reference task parametrization in both closed and open workload:

`rule7`: This rule refers to a closed workload fully defined by the population number (indicated in the `PApopulation` tag value) and the think time (indicated by `PAextDelay`) each element in the population needs to formulate a request. The think time is used to define the entry service demand of the reference task; the

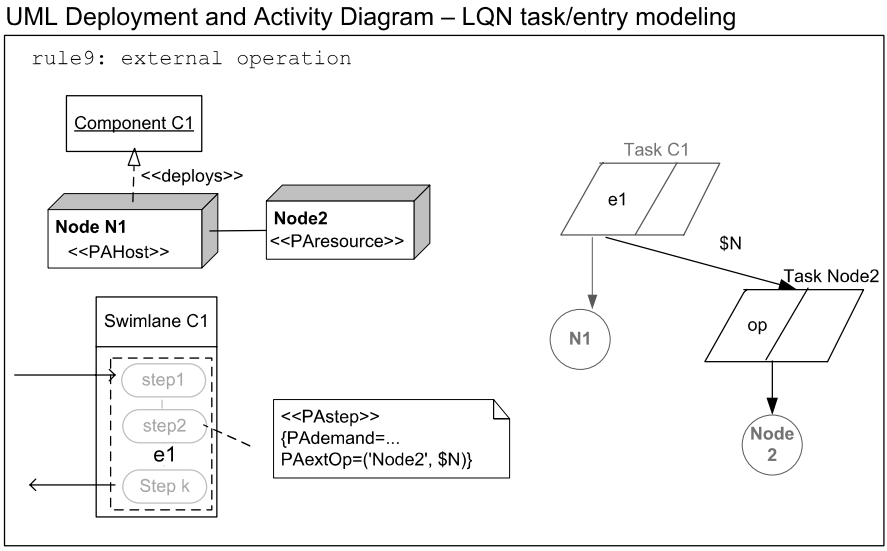


Fig. 5.18 Mapping rules for the LQN structure enhancement

population number is used as the reference task multiplicity, and the mean number of requests the reference task forwards to its server tasks is calculated as specified in `rule6`.

`rule8`: this rule refers to open workload that is fully described by the requests arrival rate (annotated in the activity diagram by means of the `PAoccurrence` tag value). In this case, the mean service demand of the reference task entries is not determined by using `rule5`, but it is the inverse of the rate annotated in the `PAoccurrence` tag value, while `rule6` still remain valid for the quantification of the mean number of requests the reference task entry produces to the other tasks.

Finally, Fig. 5.18 graphically illustrates `rule9` that deals with the `PAextOp` annotation. In fact, it can happen that some steps in the UML activity diagram requires some external operation to be completed as specified by the `PAextOP` tag value. To execute this operation $\$N$ service requests (or visits) are necessary on the device *node2* (possibly annotated as «`PAresource`» in the UML deployment diagram). In this situation, the approach enriches the LQN model with a new task (*Task Node2* in the figure) offering the external operation required as an entry (*op* in the figure), the first time it incurs in a new ID node. Since *Task Node2* is invoked from a step in the *C1* swimlane, the approach creates an arc linking the *C1* entry *e1* with it. The mean request number on this arc is exactly the number of visits $\$N$ annotated in the `PAextOp` tag value. The new task is linked to the LQN device node (*Node2*) as indicated in the `PAextOp` tag value. The *op* entry of the new task is modeled as a single phase whose mean service demand is the same of the processing capabilities of the *Node2*.

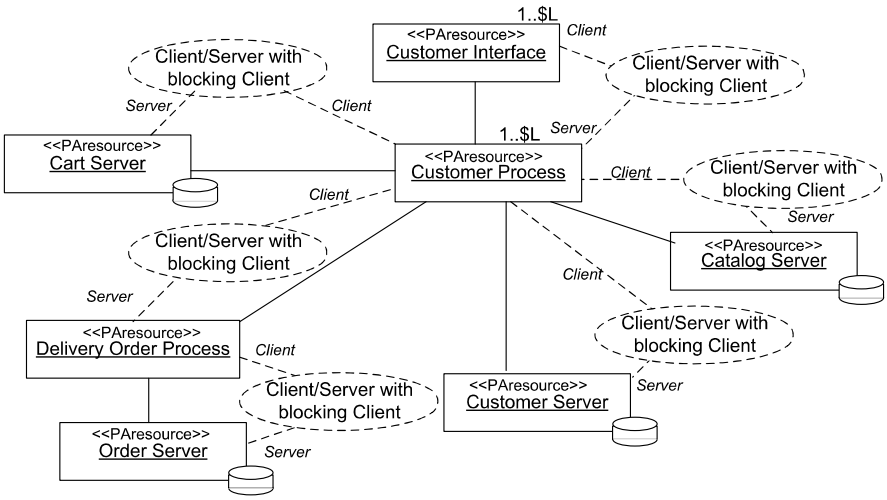


Fig. 5.19 High-level architecture of the e-commerce system

Petriu's Approach to E-commerce System

The Petriu's approach generates a LQN model by executing the two steps outlined above. In the following the approach is applied on the e-commerce case study. For sake of presentation, the scope of the modeling is reduced to the Browse Catalogue, Browse Cart, and Place Order use cases.

Figure 5.19 shows the high-level software architecture of the considered e-commerce system where architectural patterns and stereotypes for resources are represented. It is assumed that the system has $\$L$ different customers in the system. This means that $\$L$ different instances of *CustomerInterface* and *CustomerProcess* are running in the system every time. The high-level architecture further indicates the architectural patterns each component instance is involved in and the corresponding role the instance holds in it. In deed, in this example, only the *Client–Server with blocking client* pattern is used. We recall that this pattern has been introduced at the beginning of Sect. 5.2.2.[2]

Figure 5.20 shows the annotated UML deployment diagram for the e-commerce system. The system is composed by five local hosts (all annotated as «PAhost») hosting the component instances. They all communicate by means of a Local Area Network (LAN) annotated in the diagram as a «PAresource». Note that *Proc3*, *Proc4*, and *Proc5* are dedicated to *CustomerServer*, *CartServer* and *CatalogServer*, respectively. Instead, *Proc1* and *Proc2* host several component instances. In particular, on *Proc1* all the $\$L$ instances of *CustomerProcess* are deployed, while *Proc2* hosts *DeliveryOrderProcess* and *OrderServer*. Finally, each of the $\$L$ users runs the

[2]The *Client–Server with blocking client* is graphically described in Fig. 5.11 and the corresponding mapping rule is graphically illustrated in Fig. 5.13.

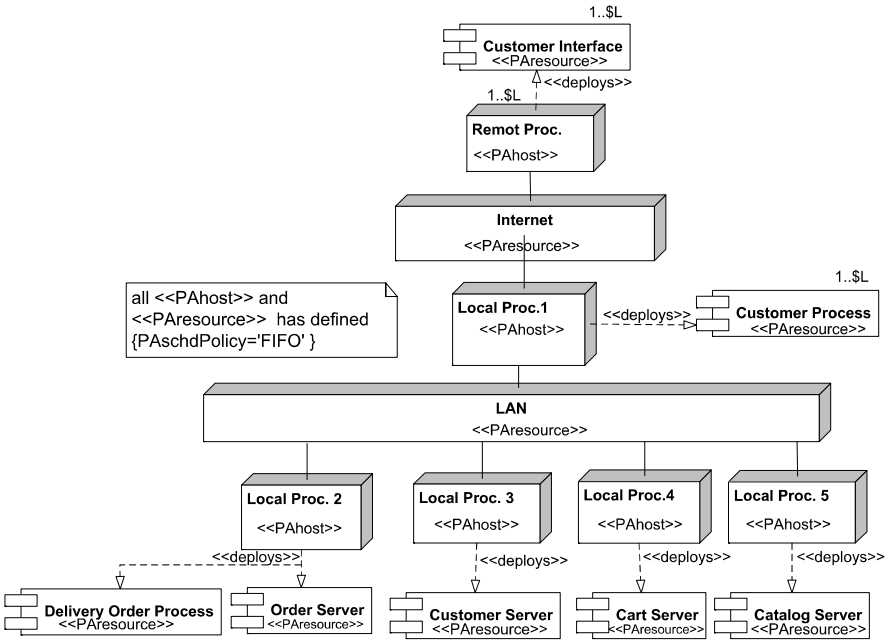


Fig. 5.20 Annotated deployment diagram

CustomerInterface component on their own computer (*Remote Proc.*) connected to the system through *Internet* (annotated as «`PAresource`»).

Figures 5.21, 5.22 and 5.23 show the annotated UML activity diagrams for the considered *BrowseCatalog*, *BrowseCart* and *PalaceOrder* use cases, respectively. All the activity diagrams defining the system behavior report annotations for the workload they impose to the system. The workload for each of them is closed and is defined by the number of users (`PApopulation` tag value) and its think time (`PAextDelay` tag value). Moreover, each step in the activity diagrams is annotated by the «`PAstep`» stereotype that indicates the mean assumed service demand the step requires to the processor running it (`PAdemand` tag value), possibly its probability execution (`PAprob` tag value in Fig. 5.23) and the additional external operation required to complete the step (`PAextOP`). Note that the `PAextOp` tag values are always used to model the usage of the network, since they refer to either *LAN* or *Internet* resources. This tag value appears everywhere, except for the interaction between *DeliveryOrderServer* and *OrderServer* components, since they are deployed on the same host and all the communications among them do not involve the network.

In the following, the Petriu's approach is executed to this e-commerce system. The presentation is divided in two parts, one for each transformation step.

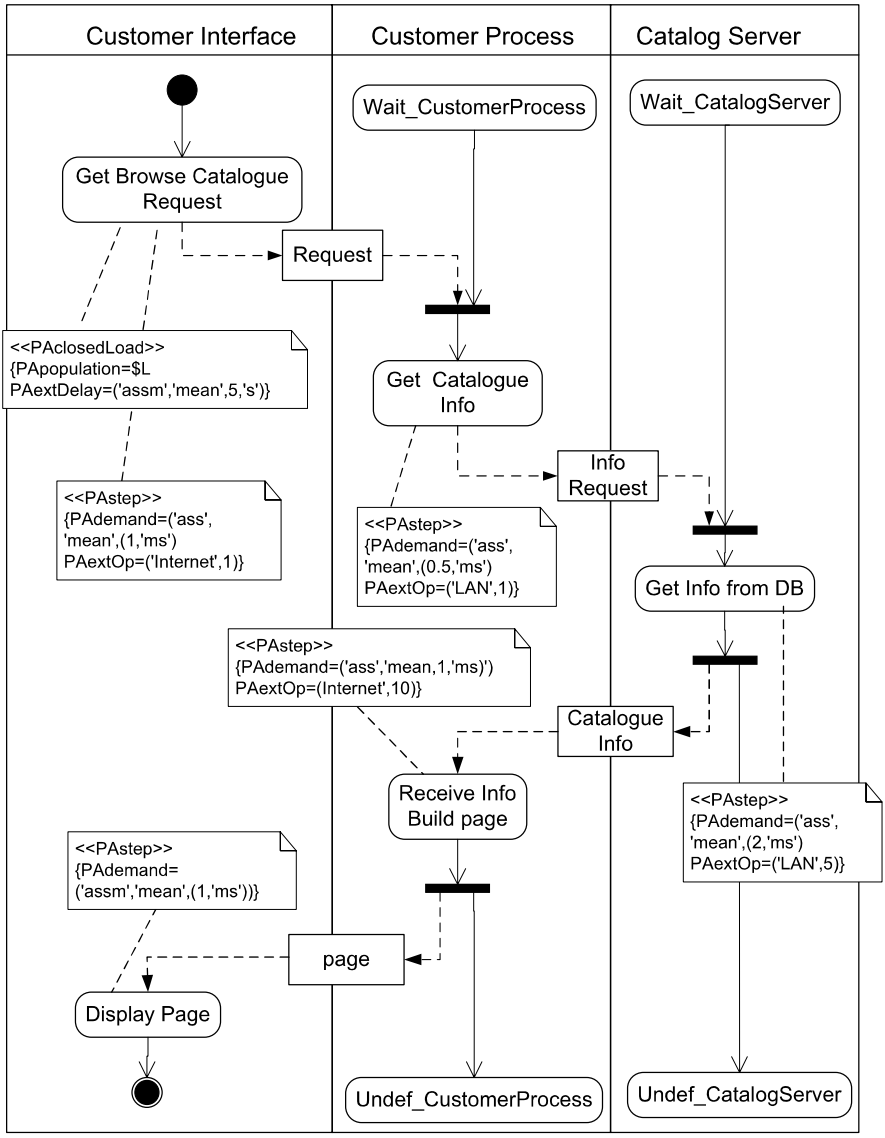


Fig. 5.21 Annotated BrowseCatalog activity diagrams

Step 1—Derivation of the LQN Structure From the high-level software architecture, the Client–Server architectural pattern and the deployment diagram, Petriu's approach derives the LQN structure of Fig. 5.24:

- a task for each component instance in the software architecture is created, each with the same multiplicity in the architecture. In particular, *CustomerInterface*

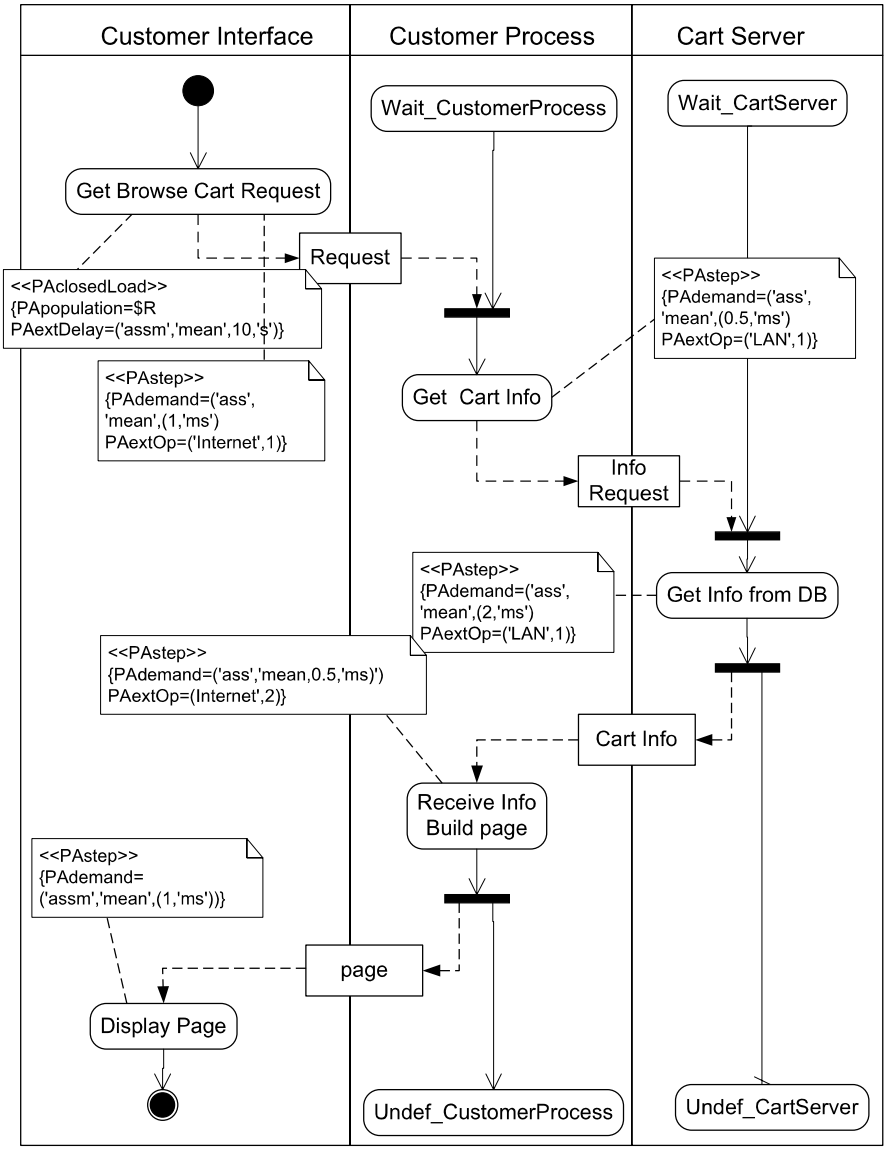


Fig. 5.22 Annotated BrowseCart activity diagrams

and *CustomerProcess* have the multiplicity equal to $\$L$, whereas the other tasks have single multiplicity;

- no additional task relative to architectural patterns is created since the *Client–Server with blocking client* pattern does not specify them;
- the connections among the tasks are defined on the basis of the links in the high-level software architecture, whereas the types of communications traversing such

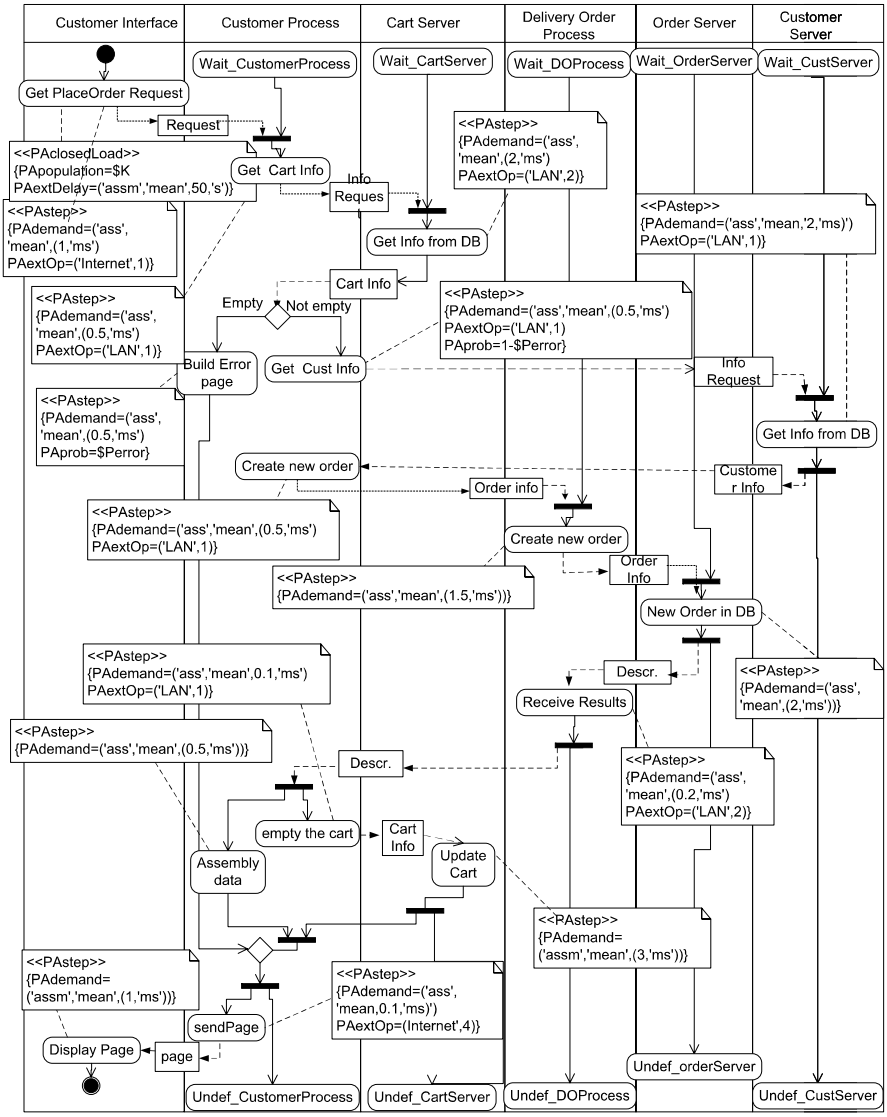


Fig. 5.23 Annotated PlaceOrder activity diagram

links are all synchronous as required by the *Client–Server with blocking client* pattern;

- a device is created for each node in the UML deployment diagram annotated with either «PAhost» or «PAresource» stereotype. As a result, the step creates a device with single multiplicity for each local processor node (from *Proc1* to

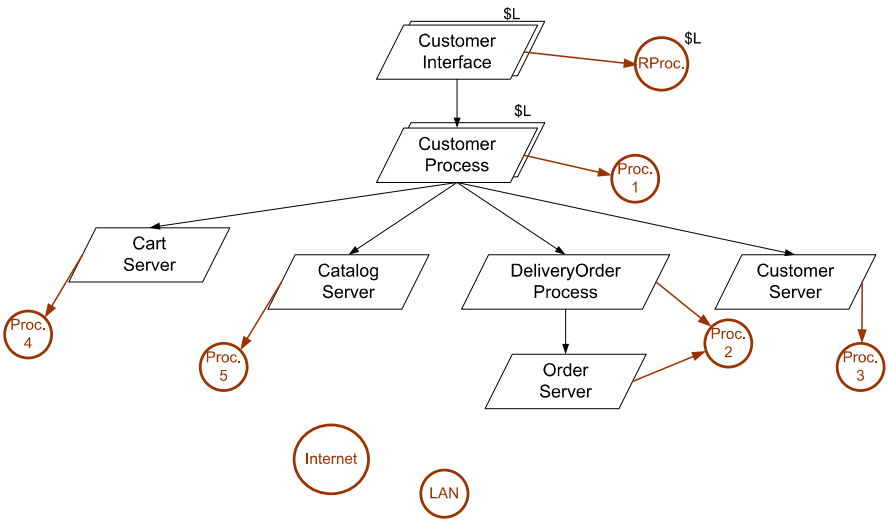


Fig. 5.24 LQN structure derived for the e-commerce system

Proc5), a device with $\$L$ multiplicity for the remote host and finally two more devices modeling the LAN and Internet.

- a link between a task and a device is created with respect to the «`deploy`» association among instances and nodes in the deployment diagram. At the moment, *LAN* and *Internet* devices are disconnected in the resulting LQN structure. This will be managed in the second step dealing with the `PAextOp` tag value.

Step 2—Derivation of the LQN Entries, Phases, Activities Details The approach proceeds with processing the activity diagrams to identify and detail the task entries and to generate the reference tasks.

Figure 5.25 shows the result (at the right side of the figure) of the application of the approach to the activity diagram describing the scenario of the *Browse Catalogue* use case (reported for convenience at the left side of the figure). First, the approach generates the *Browse Catalogue* reference task modeling the closed workload indicated in the corresponding stereotype (`rule7`). The `PApopulation` tag value is used to specify the task reference multiplicity, while the `PAextDelay` tag value specifies the thinking time used as the mean service demand of the *Browse Catalogue* reference task. This reference task uses the (virtual) *ThTbc* device to spend its thinking time.

Then the approach identifies and details the three entries of the involved tasks (i.e., *CustomerInterface*, *CustomerProcess*, and *CatalogServer*). All the entries have a single phase. The specification of the *Browse Catalog* entry of the *CustomerInterface* comes from the *Get BrowseCatalogue Request* and the *Display Page* steps in the *CustomerInterface* swimlane. The mean service demand of the phase is obtained by summing the `PAdemand` of the two steps (`rule5`). The entry makes a request

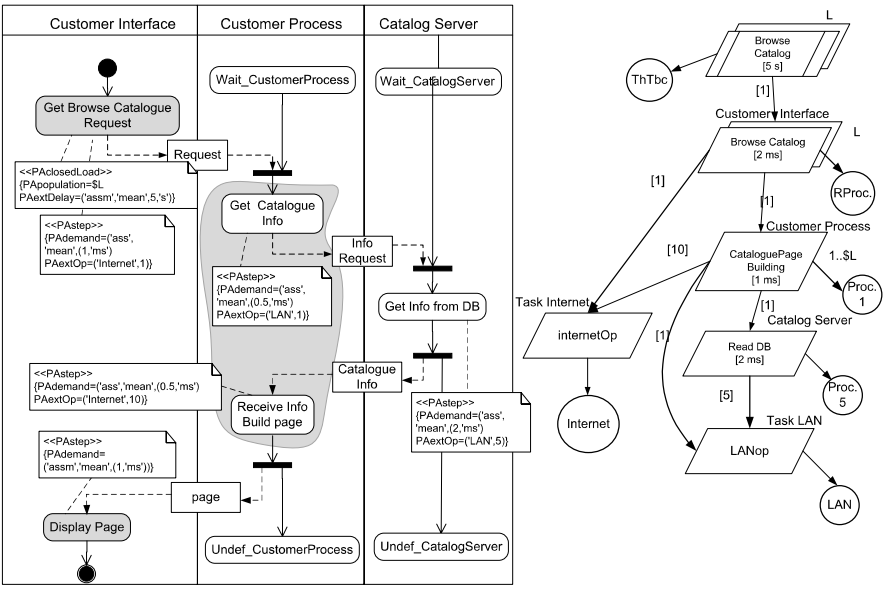


Fig. 5.25 LQN sub-model derived for the BrowseCatalog scenario

to both the *CustomerProcess* and the *Task Internet*, the former modeling the control flow in the diagram (`rule6`) and the latter introduced as required by the `PAextOp` tag value (`rule9`). Similarly, the *CataloguePageBuilding* entry is obtained by the *GetCatalogueInfo* and *ReceiveInfoBuildPage* steps of the *CustomerProcess* swimlane. The entry has 1 ms service demand and makes a request to the *CatalogServer* and to the *Task LAN*, and 10 requests to the *Task Internet*. Finally, the *ReadDB* entry details come only from the *Get Info from DB* action in the *CatalogServer* swimlane. It has 2 ms service demand and it makes five requests to the *Task LAN*.

Note that, even if the activity diagram of Fig. 5.25 has some fork and merge node, `rule4` does not apply. The reason is that `rule4` applies when a conditional branching or a fork node is found. By looking at all the swimlanes in the figure, the first non-action node is a merge node, hence this does not trigger the rule, while the last non-action node, i.e. a fork, matches with the rule exception described above.

Since the activity diagram of the BrowseCart use case has the same structure and complexity of the BrowseCatalog one, a detailed presentation of the approach application on it is here skipped.

Figure 5.26 graphically illustrates the application of the approach on the *CustomerProcess* swimlane of the activity diagram describing the PlaceOrder use case. Again, in the figure, an excerpt of the activity diagram is reported on the left side, while the approach output is on the right side. This figure is reported to show the application of `rule4` that models the entry as a LQN activity graph. In deed, the *CustomerProcess* swimlane is the only one in the presented e-commerce system that requires the `rule4` application. Since the swimlane presents a conditional branching (the decision node between *Get Cust Info* and *Build error Page* steps), `rule4`

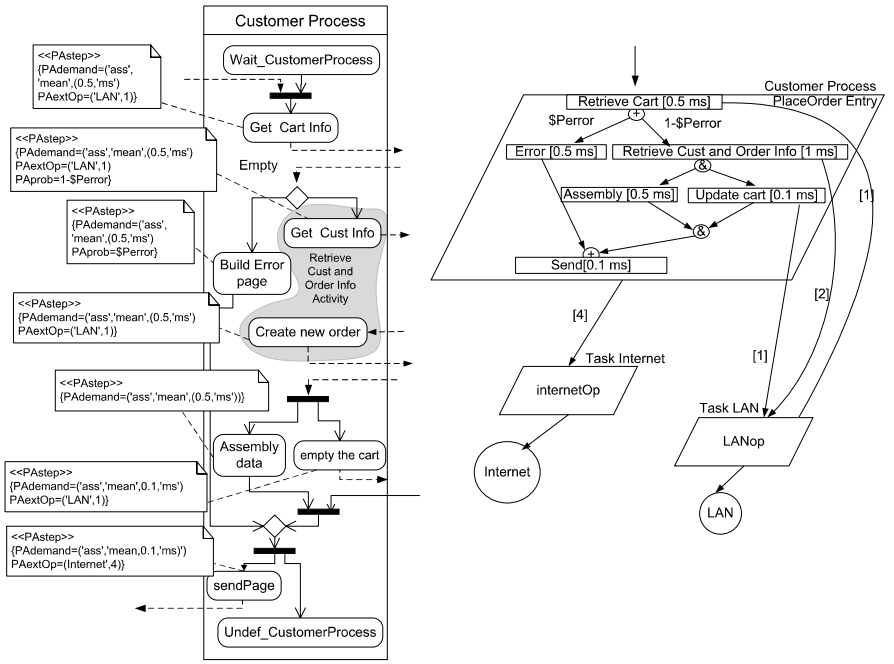


Fig. 5.26 CustomerProcess entry modeling in the PlaceOrder scenario

must be applied. Each step in the swimlane generates a LQN activity, unless *Get Cust Info* and *Create New Order* that together define *Retrieve Cust and Order Info* activity. In the first case, the service demand of the activity is exactly the same of the `PAdemand` tag value annotated on the step. In the second case, instead, the service demand is equal to the sum of the two `PAdemand`. For what concerns the number of requests each activity forwards to the other tasks, it is determined by the control flow in the diagram and by the `PAextOp`. The conditional node (merge node) in the swimlane is translated in an *or-fork* (*or-join*) whose branching probability is determined by the `PAprob` tag value of the alternative steps following the conditional node. The following *fork* (*join*) node is translated in an *&-fork* (*&-join*) node in the activity graph. Again the first join node and the last fork node in the swimlane are not considered by the approach that just ignores them.

Finally, Fig. 5.27 graphically illustrates the whole LQN model obtained executing the Petriu's approach on the e-commerce system.

Tool Support

Three different implementations have been devised for the Petriu's approach, which propose systematic methods of building LQN models of complex SA based on combinations of the considered patterns. Such implementations are based on graph transformation techniques.

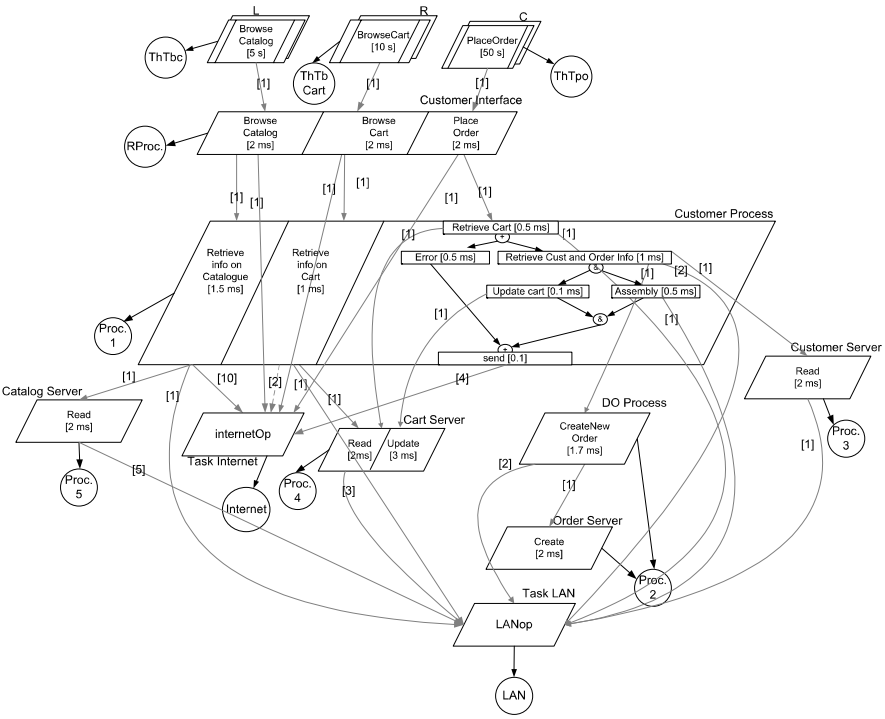


Fig. 5.27 LQN model for the e-commerce system

- The approach in [94] generates the software and system execution models by applying graph transformation techniques that are automatically executed by a general-purpose graph rewriting tool.
- In [93] 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 [62] uses the eXtensible Stylesheet Language Transformations (XSLT) to carry out the graph transformation step.[3] The input contains UML models in XML format, according to the standard XML Metadata Interchange (XMI) [89], 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. The last version of such approach [63] is implemented by an XML algebra-based model transformation.

[3] 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.

5.2.3 SAP•one: From UML to a Queueing Network

This section presents the Software Architecture Performance analysis approach, called SAP•one, that generates a multi-chain QN model from a software architecture description based on UML 2.0 [47, 48]. It deals with the QN topology definition and parametrization and it is defined for component-based software systems. The approach defines translation rules that map architectural interaction patterns into QN patterns. The target model is generated by composing the identified QN patterns suitably instantiated for the specific application.

The approach uses UML 2.0 as an Architecture Description Language (ADL) to describe the software system architecture. Software Architecture describes the system at a very high level of abstraction by specifying its structure and its behavior. The required UML 2.0 diagrams are: use case diagram, component diagram and sequence diagram. From the component diagram the approach identifies for each software component a service center that will form the target QN and extracts the relative parameterizations pieces of information. Moreover, from the linked interfaces it derives the connections among the service centers representing the software components. From the use cases in the use case diagram SAP•one takes information about the customer types that enter into the system. Finally, the sequence diagrams are used to defines the customer behaviors (or chain) inside the system and the workload provided to the system.

The UML diagrams are annotated by means of the UML Profile for the Schedulability, Performance and Time (SPT) [85] to provide additional information. Such data, used in the QN parametrization and in the workload definition, is: *(i)* the operational profile of the system, modeling the way the system will be used by the users; *(ii)* the workload entering the system; *(iii)* the service demand required by a request (job) to the system components it visits; and *(iv)* the performance characterization of the system components, such as service rate, scheduling policy and waiting queue capacity.

Differently from the existing QN-based approaches, the SAP•one methodology does not consider hardware platform information. Indeed the QN service centers do not represent hardware devices (such as disk and processor), rather they represent software components. The QN topology describes how the software components are combined to form the software system, while a chain represents how a system service request is accomplished through the software components interactions. The underlying assumption is that each software component is deployed on an hypothetical logic device. All the logical devices have the same processing power. This implies that the service requiring the minimum resource demand will always be faster. However the logical devices may have different waiting queue characteristics, in terms of capacity and scheduling policy. Thus, the time a request spends to be accomplished does not depend on the device speed factor but only on the queue characteristics. In this approach, hence, the service time is not different from the service demand for a class of requests (or jobs).

The performance figures derivable by SAP•one are: the throughput and utilization of service centers (i.e. software component), the waiting and response time of

each customer class in a single service center and/or for the global system (i.e. the time spent to accomplished the software system services).

The SAP•one analysis aims at identifying (potential) performance problems due to logical structure of the functionalities of the software system, and at identifying critical software components whose design needs special attention. The analysis is not absolute in terms of actual performance figures rather it is comparative in between alternative designs. The comparative analysis allows to identify portions of the software architecture that could raise performance problems. It is useful to *(i)* support the designers in the software architecture development when components are identified, in fact, in this step the analysis can suggest how to distribute the functionalities among the components in order to have good performance, and *(ii)* identify critical software components, which must be carefully developed in the subsequent lifecycle phases.

This comparative performance analysis fails when the software components are deployed on hardware devices showing high-variance speed factors. In this case the assumption about the uniform processing power of the logical devices is not valid any more and information about the hardware platform need to be taken into account.

With respect to the Q-model presented in Chap. 4, this approach can be easily applied to the initial development phases, such as architectural design. In fact, no information about the hosting platform is assumed here, therefore as long as the development process evolves, the semantics defined for the QN performance model is not rich enough to represent more details of the software system. However, this is a very powerful approach at the architectural level because, as shown above, it allows one to make a performance analysis independently of the platform characteristics.

Software Specification

SAP•one requires that the software system architecture is modeled by UML 2 use case, component and sequence diagrams. Only sequence and component diagrams are annotated with quantitative information through the UML SPT Profile. The SPT stereotypes are used as notes in the diagrams. The correctness of the numerical values annotated in the diagrams is *assumed* and it is based on the designers' experience. It is worth recalling that such kind of information is not available at the software architecture level.

Use Case Diagrams describe the software system at a very high level of abstraction by identifying its functionalities. These functionalities correspond to the requests (or customers) entering the system. From this type of diagram the approach extracts the customer types entering into the system. For each use case in the diagram the approach requires the definition of a set of sequence diagrams describing the corresponding behavior inside the system. For each customer type, hence, the approach knows which sequence diagrams describe the corresponding behavior. For the sake of simplicity, without losing generality, the approach assumes that the behavior of a customer is described by a single sequence diagram. Indeed, the new

interaction operators UML 2.0 allow the modeling of a complex behavior with several alternatives, in only one scenario. Figure 5.32 shows an example of use case diagram for the SAP•one approach.

The **Component Diagram** is used to model the static structure of the system in terms of the software component instances, that in the following are generally named components and connectors. The annotated component diagram provides information on the parameterizations of the service centers of the QN: the type of the service center (e.g. servers with waiting queues or a delay center), the rates of the services they provide and the scheduling policies they use to extract jobs from their waiting queues.

The approach uses the «PAhost» stereotype with the PAschdPolicy tag value to specify the scheduling policy of the component waiting queue. «PAhost» models an active resource. Since, the approach assumes that each component is deployed on its own logical active device, there is a 1-to-1 correspondence between the software components and the active (logical) resources processing them. This assumption supports the usage of the «PAhost» stereotype to annotate components. Therefore, since SAP•one assumes that all logical devices have the same processing power, no annotations about the processing power are required.

The interfaces in the component diagram are, instead, annotated with the execution time they require for the software component to be accomplished. This time represents the total execution time required locally to the component in order to satisfy the request. This information is specified by either the PAdemand or PAdelay tag values of the «PAstep» stereotype. The approach associates such a stereotype to each component interface by assuming 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. 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 parameterizations phase, we assume that an interface is composed of a single operation from with the interface derives the name. Figure 5.33 shows the component diagram of the e-commerce system annotated following the assumption of the SAP•one approach.

The description of the behavior of the software system is provided by the **sequence diagrams** where the lifelines represent the software component instances and the arrows model their interactions.

We recall that the approach assumes that a sequence diagram describes the system behavior for each use case in the use case diagram. The workload intensity, described by the system use case (i.e., a QN 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 (refer to Chap. 3 for more details on the workload definition). These stereotypes annotate the first message of the sequence diagram that describes the system behavior, with such workloads.

In the sequence diagrams the «PAstep» stereotype is also used to annotate the probability of execution of an alternative behavior when the sequence diagram presents either the alt, break or option fragment operators. It annotates the

first message of all the interaction fragments that are operands of the introduced operators. For an example of sequence diagram taken in input by the SAP•one approach please refer to Fig. 5.39.

Queueing Network

The generated QN model represents the software architecture of the system under analysis. The approach generates a QN model with the same topology of the software architecture, where each service center corresponds to a software component and the connections among the service centers represent the connectors among the software components.

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. Accordingly, the customers entering to the QN model are of different types. Moreover, the software system has different behaviors depending on the different types of request specified by a set of sequence diagrams. This implies that the QN customer types representing the users' requests should also have different behaviors 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 into them. Since each component may provide different services, each with different complexity and service time, different user's requests (QN customers) might require different works to a software component (QN service center). To cope with this situation the approach uses multi-chains QN models where the service center can be a queueing center or a delay center. Customers at a queueing center compete for the use of the services. Thus the time spent by a customer at a queueing center has two components: time spent waiting, and time spent receiving the service. There is no competition for service at a delay center. Thus the residence time of a customer at a delay center corresponds to the service demand. In the generated QN 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 to different jobs. When a software component does not represent a shared resource but it represents a logical resource dedicated to a system customer, the approach maps it into a delay center.

The resulting performance model can be very complex due to the wider variety of software behavior characteristics such as parallelism and synchronous communications. This complexity implies that more complex techniques to evaluate the new QN model are necessary. These techniques, quite often, are not analytic. In particular, whenever there is a parallel operator in a sequence diagram, there is a fork in the QN model. A QN with forks, namely Extended QN, can be only simulated.

There exist many algorithms and tools able to simulate and solve QN models [100, 116]. A QN model can be solved in an exact way if it is a product-form QN [76,

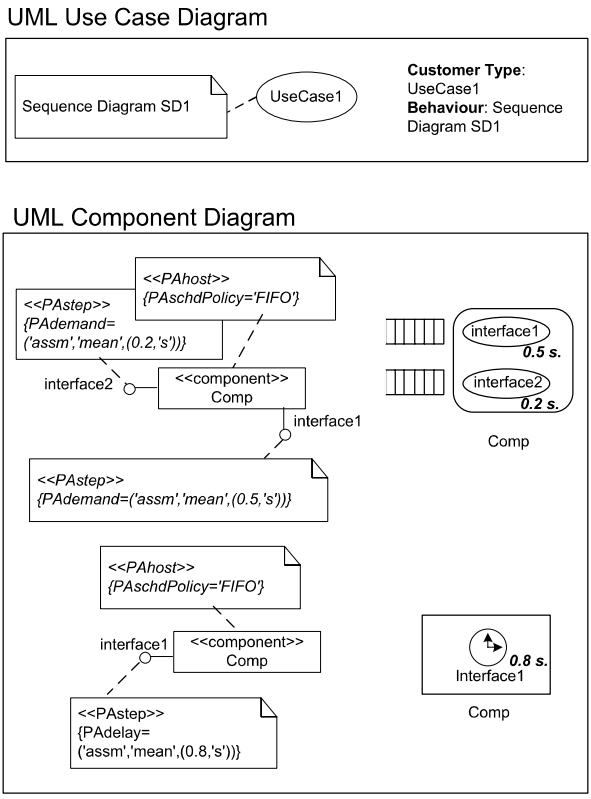


Fig. 5.28 SAP•one mapping rules (1/2)

79], otherwise several approximate solution techniques are available in the literature [23, 25, 77, 92, 115, 123].

Software to Performance Model Mapping Rules

The SAP•one approach extracts from the annotated UML diagrams all the information needed to generate and parameterize the QN model. The mapping rules it implements are summarized in Figs. 5.28, 5.29, 5.30, and 5.31. These figures have the same structure: in each box on the left-hand side there are the UML diagram elements the rule deals with, on the right hand side there is the corresponding target sub-model.

The approach is a three-step methodology:

1. *Identification of the QN customers*. Each use case in the use case diagram defines a QN customer type entering the system, while the annotation associated to the use case indicates the sequence diagram describing the QN chain (i.e., the QN customer behavior). This is represented in Fig. 5.28—Use Case Diagram box.
2. *Identification of the service centers and their characteristics*. This information is present in the component diagram: the approach maps each component into

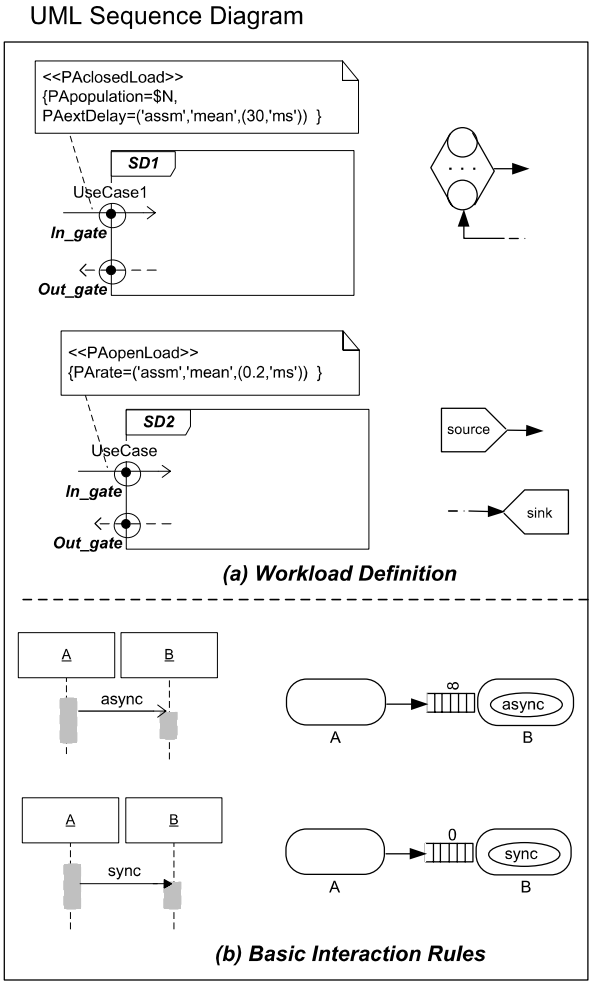


Fig. 5.29 SAP●one mapping rules (2/2)

a QN service center with either a non-preemptive server, or a delay center. The component interfaces define the classes of job processable by the corresponding service center. Since the software component can provide many interfaces (or services), the corresponding service center may have several classes, one for each interface in the component diagram. The service time of the identified class (or job class) is extracted from the `PAdemand` or `PAdelay` tag value of the «`PAstep`» stereotype that annotates the component interface. The center is a waiting center if the component interfaces are annotated with the `PAdemand` tag value, otherwise the center represents a delay center if the `PAdelay` tag value is used. 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

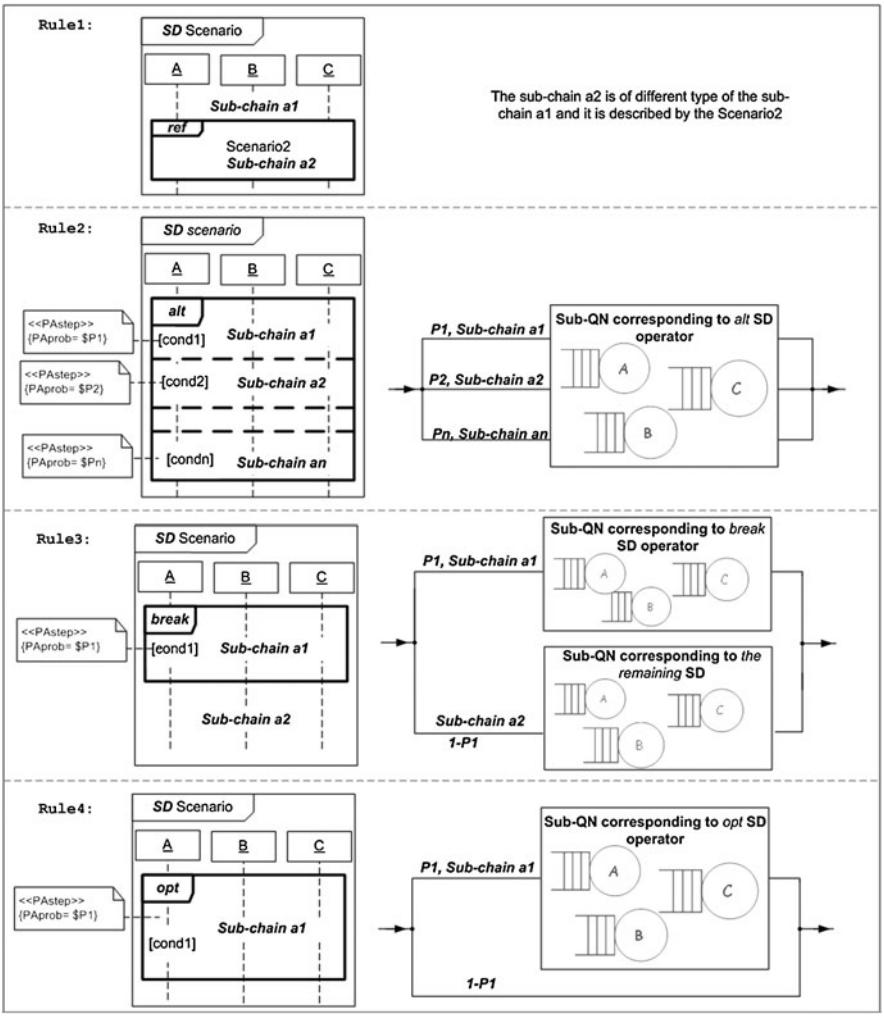


Fig. 5.30 Sequence Operators translation rules (1/2)

service centers. These mapping rules are summarized in Fig. 5.28—Component Diagram box.

3. *Definition of the QN chains and the relative workloads*. A sequence diagram shows how a customer of a given type moves among the service centers asking for services when it enters in the system. From each sequence diagram the approach generates a QN chain. The sequence diagrams indicate the workload, open or closed, that the corresponding chains impose to the system. The chain generation step consists of the application of three groups of mapping rules: the rules for workload definition (see Fig. 5.29(a)); the basic interaction rules (see Fig. 5.29(b)), and the operators rules that will be illustrated in Figs. 5.30 and 5.31.

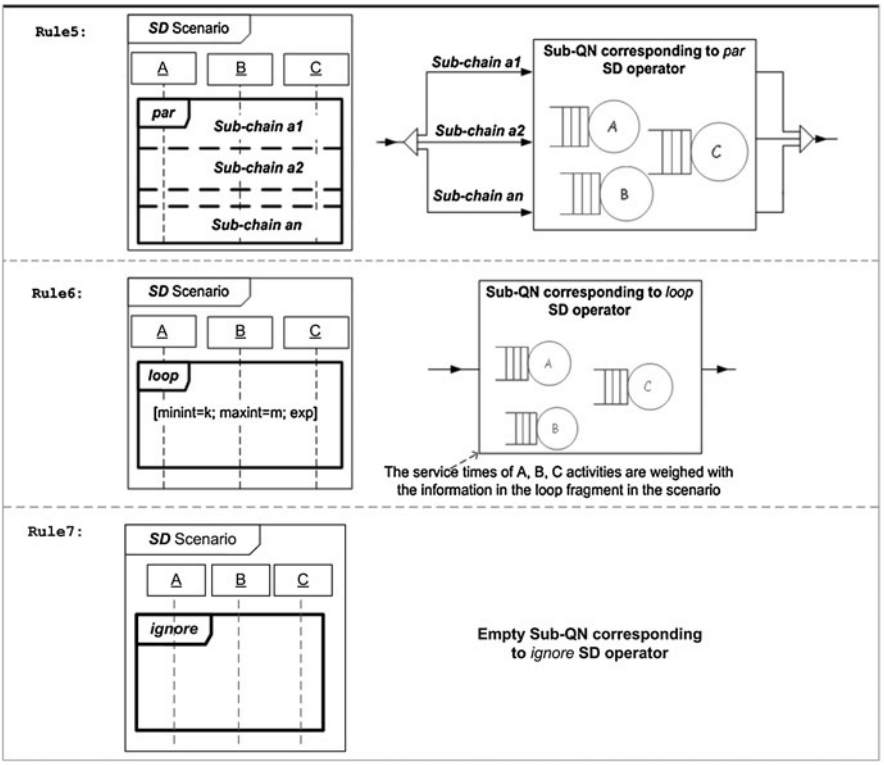


Fig. 5.31 Sequence Operators translation rules (2/2)

By referring to the *Workload Definition* rules, the approach uses the *gate* facility 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. It is an input gate if the interaction is entering into the frame. The input and output gates are translated as follows. If the first message in the sequence diagram is annotated by a «PAclodedLoad» stereotype, the input gate is translated into $N terminals by using a delay center, while the output gate is translated into a transit back to the terminals. Instead, if the first message in the sequence diagram is annotated by a «PAopenLoad» stereotype the input gate is translated into QN source node, while the output gate is translate into a QN sink node.

A *Basic interaction rule* in Fig. 5.29 translates a components interaction into a QN sub-model. More precisely, a software components interaction (an arrow in the sequence diagram) is modeled as a transit in the QN topology from the service center of the component sender to the one of the receiver. The SAP•one approach deals with synchronous and asynchronous interaction. 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 syn-

chronous. In the first case the request is buffered into the queue, leaving the sender free to continue its execution on the next request; in the second case, in order to simulate the blocking in the sender execution until the receiver becomes free, it is used a waiting queue with zero capacity and Blocking After Service (BAS) blocking protocol for the sender service center [13, 22].

Figures 5.30 and 5.31 summarizes the mapping rules for the sequence operators used to combine interaction fragments. In the following, such mapping rules are detailed:

`rule1`: A `reference` operator, identified by the *ref* keyword, is used to indicate that the behavior of the *Scenario* includes the behavior described in the sequence diagram *Scenario2*. 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 subchain corresponding to the one modeling the behavior of the *Scenario2* customer type.

`rule2`: 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 represent 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. The `alt` operator has n alternatives each guarded by a condition (`cond1`, `cond2`, ..., `condn`). Each alternative behavior has a probability to be executed (`P1`, `P2`, ..., `Pn`). The alternative behaviors are modeled as different routing in the QN model (or sub-chains) with the corresponding probabilities.

`rule3`: 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 [87]. To be properly processed, it should have annotated the probability that it will happen. To this aim the «`PAstep`» stereotype and the `PAprob` tag value are used. The stereotype annotates the first interaction inside the `opt` frame. The corresponding QN sub-model presents a branching point in correspondence of the *opt* operator. A job can enter in the QN sub-model with probability `P1` (that is the probability annotates in the sequence diagram) or it cannot enter with `1-P1` probability. Either it enters in the sub-model or it does not, it will perform according to the behavior described in the sequence diagram subsequent the `opt` fragment.

`rule4`: 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 probability that it happens must be annotated on the first message enclosed in the operator via the «`PAstep`» stereotype. Its similarities with the `alt` interaction operator are reflected on the QN sub-model. 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.

`rule5: parallel` operator, identified by the *par* keyword, defines n parallel behaviors. In a QN model, it represents a fork. A parallel operator with *n* concurrent behaviors is translated into a QN sub-model having a fork (the triangle at the left) with *n* different outcoming 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.

`rule6:` 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. To model the repetition of such a behavior, the SAP•one approach weights the service times of the component interactions involved in it by the repetition numbers (minint, maxint) in the guard of the `loop` operator. The approach can make a worst/best/mean case analysis by imposing a weight equal to the `maxint/minint/mean value` between `maxint` and `minint`, respectively.

`rule7:` 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. By means of the `ignore` operator the simplification of the target model can be done by reasoning at the software designer 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.

SAP•one Approach on E-commerce System

The SAP•one approach generates a QN model by executing the three steps outlined above. In the following the approach is applied on the e-commerce case study. The scope of the modeling is reduced to the Browse Catalogue, Browse Cart, Delete Item and Place Order use cases as indicated in the use case diagram of Fig. 5.32.

All the sequence diagrams defining the system behavior 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 like (`$NUSER` in Fig. 5.34b) and some other values suitable for the specific application like the (`F_cust`, `P_bc` values in Fig. 5.34b, which represent the

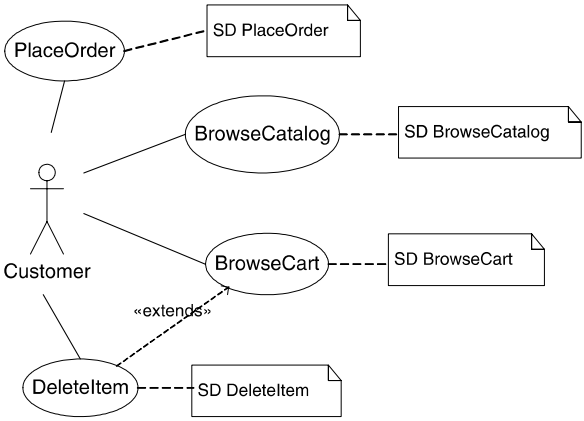


Fig. 5.32 UML use case diagram

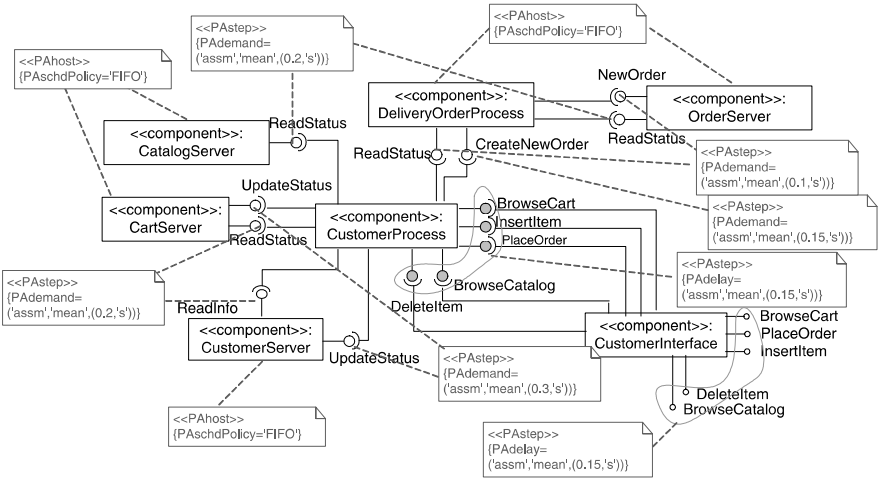


Fig. 5.33 Component diagram of the considered portion of the e-commerce system

probability that a customer logs in the system and the probability that a customer requests the BrowseCart service, respectively).

Identification of the QN Customers—The use case diagram in Fig. 5.32 has four use cases, hence the QN customers are of four different types. The behavior of such customers are described by the corresponding sequence diagrams annotated in the diagram. Such sequence diagrams define four different QN chains.

Identification of the Service Centers and of Their Characteristics—From the component diagram (see Fig. 5.33) seven QN centers are identified, five are queueing service centers while two, CustomerInterface and CustomerProcess, are delay centers. All the service centers have a service type for each interface of the corresponding component in the component diagram. The CustomerInterface and the CustomerProcess components, instead, are modeled by a set of delays, one for each

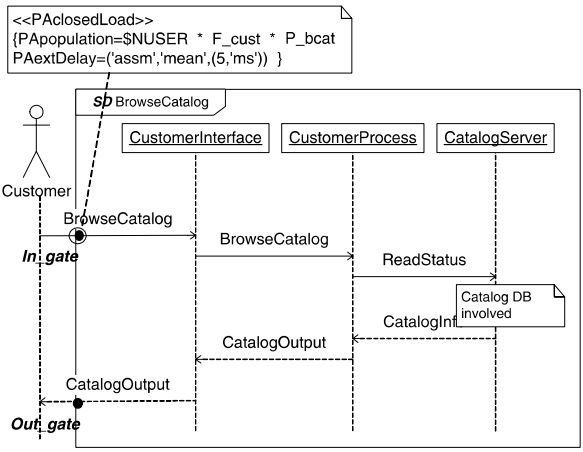


Fig. 5.34a Browse Catalogue scenario

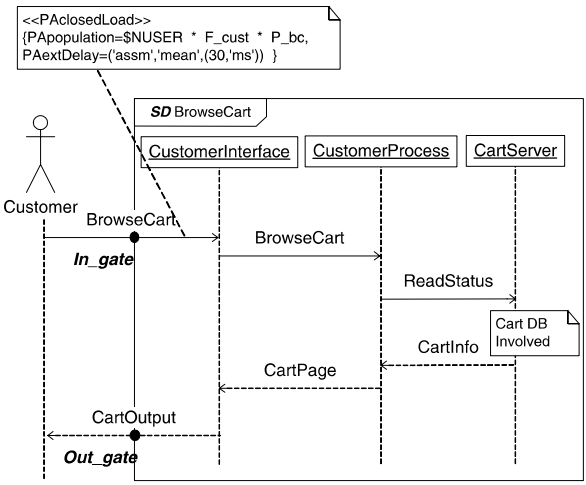
request type they are able to process. This choice conforms to the annotations in the component diagram, where it is specified that their provided interfaces are delays. Note that this modeling respects the architectural constraint discussed in Chap. 2 regarding the presence of a CustomerProcess and CustomerInterface instance for each e-commerce customer.

From the annotations on the provided component interfaces SAP●one extracts the service times or the delays of each service of the components, needed to parameterize the QN model. Finally, from the «PAhost» stereotypes the approach extracts the scheduling policy of each centers. In Fig. 5.35 the identified service centers are reported.

Definition of the QN Chains and the Relative Workloads—In this step, the workloads intensity entering the system is derived by the information annotated to the first message of the sequence diagrams. In the e-commerce system all the workloads are closed. By applying the rule for the closed workload definition, the approach generates a delay center for each input gate in the sequences. Such center will produce a job/request to the e-commerce system accordingly to the annotation in the «PAclosedWorkload» stereotypes.

The e-commerce system processes four types of requests, which are translated into four chains in the QN. To this aim, the basic operator rules in Figs. 5.28, 5.29, 5.30, and 5.31 are applied. Since the interactions among components are asynchronous, all the waiting queues of the service centers are infinite queues. 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. As an example, let us consider the chain for the *browseCart* (see Fig. 5.36), it involves only the CustomerInterface, the CustomerProcess and the CartServer, as specified in the corresponding sequence diagram of Fig. 5.34b.

In the following, the chains of the considered use cases are presented. A chain is defined as a graph-based object whose nodes are service centers and where arrows represent interactions among the centers.



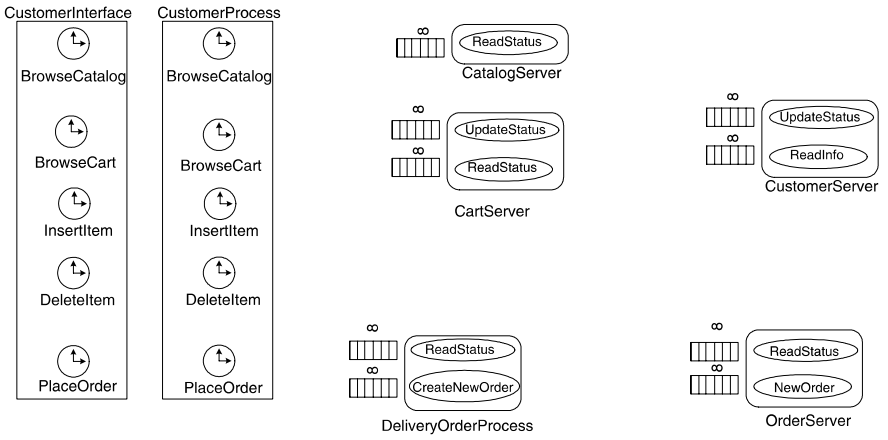


Fig. 5.34b Browse Cart scenario

Fig. 5.35 Identified QN service centers for the e-commerce system

Figures 5.34a and 5.34b show the sequence diagrams for the BrowseCatalog and BrowseCart use cases. Figure 5.36 shows the corresponding chains. These chains are the most simple, and both of them are generated by applying three times the mapping rule for asynchronous interactions. The chains terminate in the infinite servers (Customers) that generate the job types, according to the closed workload definition.

In Fig. 5.38 is reported the chain for the *DeleteItem* request type. To generate this chain the translation rules defined for the `reference` and `break` operators are applied since they appear in its sequence diagram (see Fig. 5.37). 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

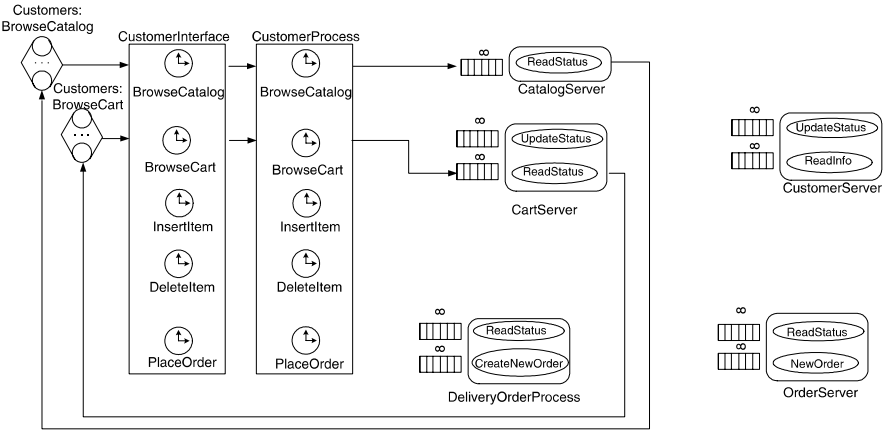


Fig. 5.36 Chains in the QN model corresponding to BrowseCatalog and BrowseCart scenario

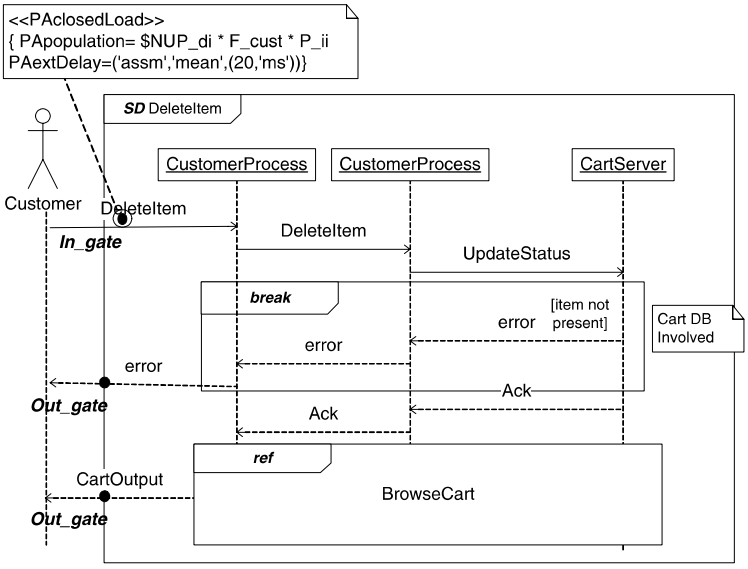


Fig. 5.37 Delete Item scenario

such scenario. As described in the previous section, 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 toward the PrepareOutput service of the CustomerProcess. The other one, with probability `1-Perror`, models the remainder of the scenario that routes the customer toward the BrowseCart service of the CustomerProcess, where the change of chain is executed. The probabilities of both routes are extracted from the sequence diagram.

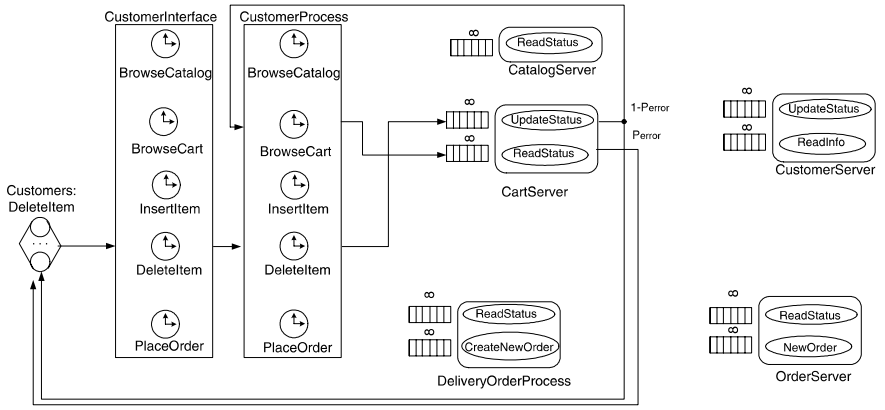


Fig. 5.38 Chain in the QN model corresponding to DeleteItem scenario

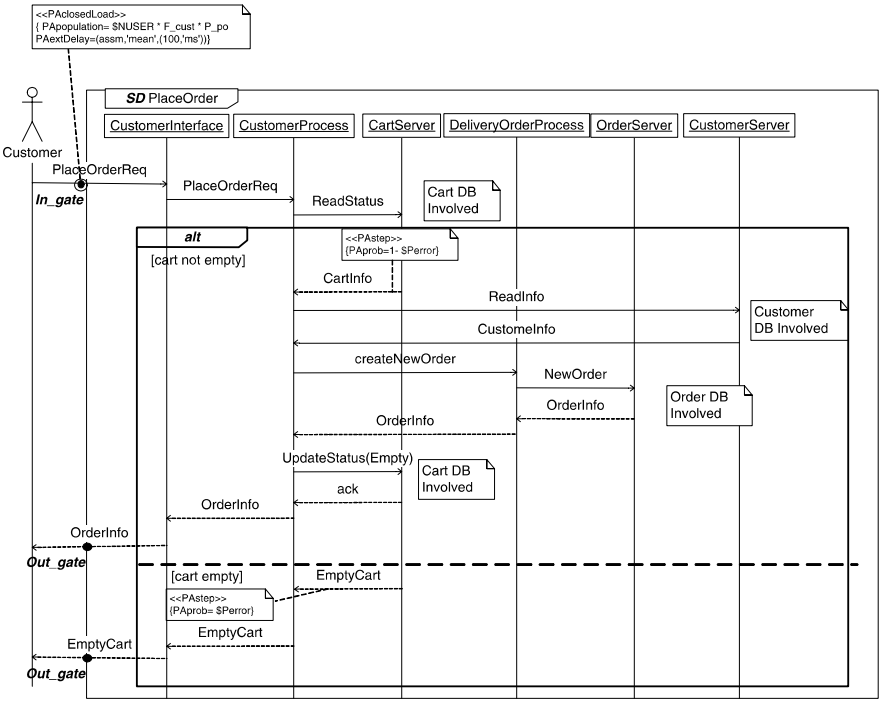


Fig. 5.39 Place Order scenario

Finally, Fig. 5.40 reports the chain for the *PlaceOrder* request type. Here the translation rule for the `alt` operator is used. As the sequence diagram in Fig. 5.39 shows, the `alt` operator has two alternative behaviors, one with a *not empty cart* guard expression and the other with an *empty cart* guard expression. The `alt` oper-

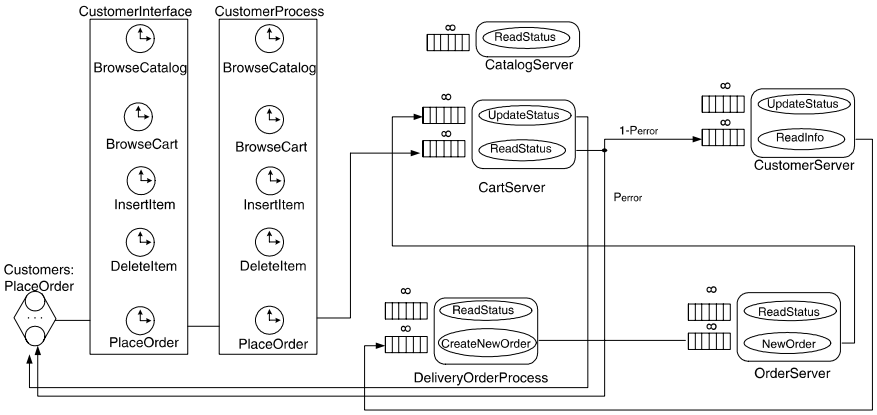


Fig. 5.40 Chain in the QN model corresponding to PlaceOrder scenario

ator 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 Fig. 5.40, the `alt` operator defines the branching point having two alternatives one with `1-$Perror` probability and the other with `$Perror` probability to be covered.

Tool Support

The SAP•one methodology is implemented in **MOSQUITO** (MOdel driven conSTruction of QUeuIng neTwOrks) [103]. It is a model transformation tool that generates Queueing Networks starting from a UML software model that has been suitably extended by means of a UML SPT profile [85].

MOSQUITO permits us to annotate UML software models with performance data and finally creates an XML representation of performance models. In its current release it can create three different models, which are: *(i)* Execution Graphs and *(ii)* Queueing Network models based on PRIMA-UML methodology [45] and *(iii)* Extended Queueing Network models based on the SAP•one methodology.

MOSQUITO has a client/server architecture. The plug-in implements the client side that provides the functionality to invoke the web services placed at the server side.

In particular MOSQUITO is exposed on the web as a web service. This will allow (in next MOSQUITO releases) any user to implement an alternative client, using the preferred languages and technologies, to use the services of MOSQUITO. The MOSQUITO client that we provide is a plug-in of Eclipse that allows an entirely automated process. The user must only properly create and annotate UML models according to the selected methodology (i.e. SAP•one or PRIMA-UML). After the creation of a model the user must only select the set of diagrams that compose the

model and activate the desired transformation. The result of the transformation will be shown in the xml editor of the plug-in.

5.3 Other Transformational Approaches

In this section three categories of transformation techniques are reviewed. Each category is characterized by the produced performance model. The most representative approaches in each category are briefly presented. Each approach is given a label, which will be used in the following section to discuss its merits.

5.3.1 Queueing Network Based Methodologies

Besides the ones presented in Sect. 5.2, many methodologies in the literature propose transformation techniques to derive Queueing Network (QN)-based models—possibly Extended QN (EQN) or Layered QN (LQN)—from Software Architecture (SA) specifications or SA patterns. Most of the proposed methods are based on the Software Performance Engineering (SPE) methodology introduced in [106] and briefly described in Chap. 1.

M1: This approach, presented in [110, 119], uses the SPE methodology to evaluate the performance characteristics of a software architecture specified by using the Unified Modeling Language (UML). The utilized diagrams are class and deployment diagrams and sequence diagrams enriched with ITU Message Sequence Chart (MSC) features. The emphasis of the approach 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 [108] to describe a case study that makes use of the tool SPE•ED for performance evaluation. In [121] 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.

Another approach in this class was developed in [45]. 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 approach is the evolution of a previous work from the same authors that was based on object-oriented software specifications in OMT [44].

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. In [60] the PRIMA-UML methodology was extended to cope with mobile SA by enhancing the UML description to model 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 related to the possible adoption of code mobility.

M2: The approaches in this group consider systems specified by architectural patterns. This simplifies the derivation of their corresponding performance models. Architectural patterns characterize frequently used architectural solutions. Each pattern is described by its structure (what the components are) and its behavior (how they interact). The approach described in Sect. 5.2.2 is the main representative of this group. It is worth to recall here that in this approach 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 activity (or sequence) diagrams. The approach proposes systematic methods of building LQN models of complex SA based on combinations of the considered patterns.

M3: The SAP•one approach presented in Sect. 5.2.3 falls in this class because it produces a QN model, but it differs from M1 and M2 since the approach does not rely on hardware platform information. Hence it can be applied earlier in the software lifecycle.

M4: In [122] a methodology to automatically derive a LQN model from a commercial software design environment called ObjecTime Developer [9] by means of an intermediate prototype tool called PAMB (Performance Analysis Model Builder) is described. 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. This approach 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 for building 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.

M5: An automated compositional approach for component-based performance engineering called CB-SPE is proposed in [34, 35]. This approach is applied at the *component layer* and at the *application layer*. At the component layer the goal is to obtain components (to be used later at the application layer) with predicted performance properties explicitly declared in the component interfaces. This implies that the component developer has to introduce and validate the performance requirements of each component in isolation. Such an analysis must be platform independent. Later, at the application level, it will be instantiated on a specific platform. The component developer is expected to fill a "component repository" with components whose interface explicitly declares the component predicted performance properties. The performance analysis of the assembled system is obtained by combining 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 the UML SPT profile. The approach, according to the SPE principles, provides two different models: a stand-alone performance model, namely an Execution Graph derived from the sequence diagrams, and a contention-based performance model, namely a QN model derived from the deployment diagram. This methodology is implemented in the CB-SPE Tool.

5.3.2 Petri Net-Based Approaches

M6: In [30] a systematic translation of state machine and sequence diagrams into GSPN is proposed. The approach consists of translating the two types of diagram into two separate labeled GSPN. The translation of a state machine 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 composition operators. In [81] the methodology is extended by using the UML activity diagrams to describe activities performed by the system that are usually expressed in a state machine as *doActivity*. The activity diagrams are then translated into labeled GSPN. Such target models are finally combined with the labeled GSPN modeling the state machines that use the *doActivity* modeled by the activity diagrams.

The GSPN generation is tool supported [59].

5.3.3 Methodologies Based on Simulation Methods

This category of approaches is 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.

M7: This approach, proposed in [46], 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 [10], 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.

The approach, proposed in [15], presents a simulation framework named Simulation Modeling Language (SimML) to automatically generate a simulation Java program (by means of the JavaSim tool [4]) 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.

The last approach of this category has been extensively presented in Sect. 5.2.1. It generates a process-oriented simulation model of a UML software specification describing the software architecture of the system. In [27] this approach has been extended to deal with mobile systems. The main contribution is modeling physical mobility of a user by means of UML activity diagrams called “high-level” activity diagrams. Each mobility user has associated a set of such “high-level” activity diagrams describing their physical mobility behavior.

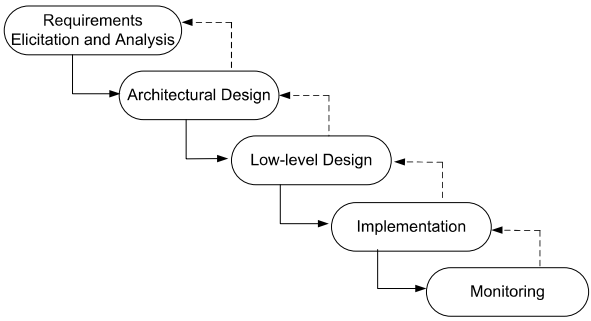


Fig. 5.41 Generic software lifecycle model

5.4 Discussion of the Approaches

In this section a discussion on the reviewed methodologies is presented. The various approaches differ with respect to several dimensions. In particular, the most important ones are 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.

In this chapter the focus has been restricted on the integration of performance analysis at the earliest stages of the software lifecycle, namely software design, software architecture and software specification.

The approaches are discussed by considering how each one integrates into the software lifecycle. The following discussion is highly based on [24, 47], where a more comprehensive comparison and classification of the approaches can be found.

To carry out the discussion and classification the generic model of the software lifecycle is considered that is presented in Chap. 4, which is re-called in Fig. 5.41.

The three indicators considered in the discussion are the *integration level of the software model with the performance model*, the *level of integration of performance analysis in the software lifecycle* and the *methodology automation degree* (Fig. 5.42). 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 indicator gives a measure of the distance the mapping that the methodologies define needs to cover to relate the software design artifacts with the performance model. A high level of integration corresponds to a strong semantic correspondence between the performance model and the software model. Syntactically related models permit the definition of a syntax driven translation from the syntactic specification of software artifacts to the performance model. The unique comprehensive model allows the integration of behavioral and performance analysis in the same conceptual framework. The level of integration of performance analysis in the software lifecycle identifies the development phase at which the analysis can be carried out. The earlier the prediction process can be applied, the better integration with the development process is obtained. This integration strongly depends on the additional information required by the approach to

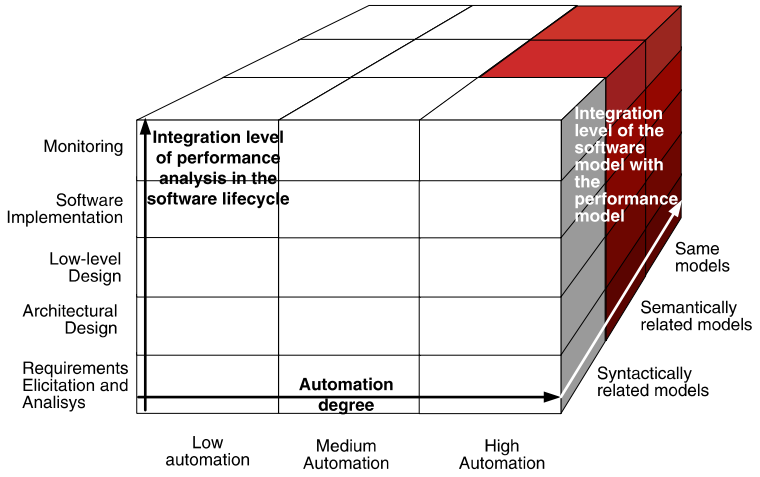


Fig. 5.42 Classification dimensions of software performance approaches

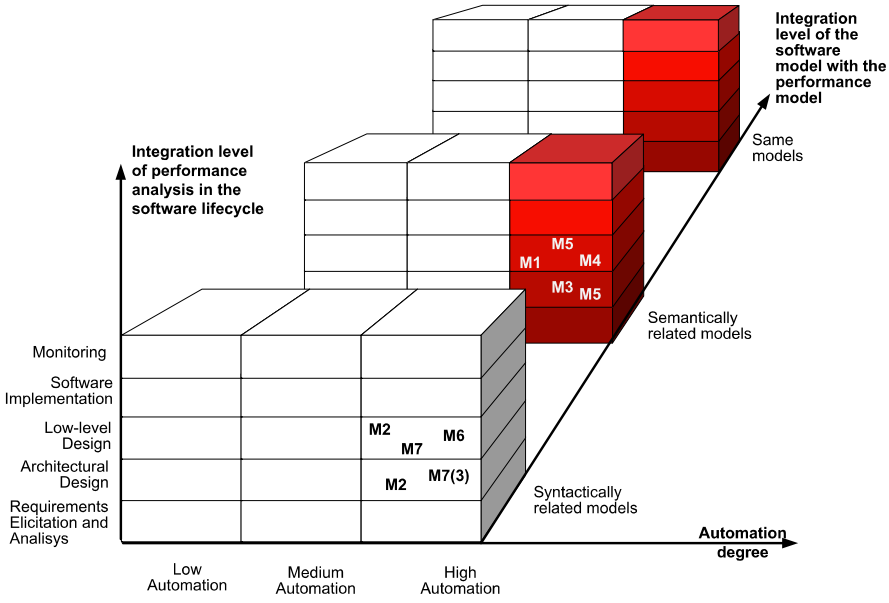


Fig. 5.43 Classification of considered methodologies

carry out performance analysis. Indeed these pieces of information are available in certain phases of the software lifecycle, typically toward the end. However, there exist methods that assume the availability of this information at the earlier phases of the software lifecycle to perform predictive analysis. In this case these methods assume that this information is available somehow, for example through an operational

profile extracted from similar systems or by assuming hypothetical implementation scenarios. 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 ideal methodologies are at the bottom of the two rightmost columns, since they show high integration of the software model with the performance model, high level of integration of performance analysis with the lifecycle, i.e., from the very beginning, and high degree of automation.

According to the above discussion, in Fig. 5.43 we classify the methodologies briefly reviewed in Sect. 5.3. In each internal small cube, label ordering has no meaning.

All the considered 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 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.

The group of semantically related models refers to methodologies whose mapping between design and performance models is based on the analysis of the dynamic behavior of the design artifacts. For example, in M4 the mapping is based on the analysis of the state machine model of the system design.

The last group singles out the methodologies that allow for the use of the same model both for design description and performance analysis. Here the mapping is the identity modulo timing information. This means that there is not a concrete transformation between software and performance models. This ideal integration can be obtained only if the software designer has the skill to work with sophisticated specification tools. These approaches are out of the scope of this chapter.

The integration level of performance analysis in the software lifecycle shows that the considered 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. Approaches operating at different abstraction levels can be seen as complementary since they address different needs. Performance prediction analysis at very early stages of design allows for choosing among different design alternatives. Later on, performance analysis serves the purpose of validating such choices with respect to non-functional requirements.

All the three dimensions concur to define the *complexity* of the methods as far as their ability to provide a performance model to be evaluated is concerned. To this extent the complexity of the translation algorithms to build the performance model

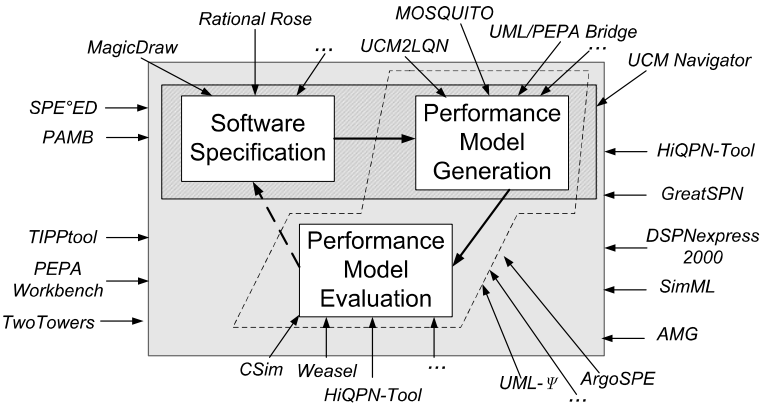


Fig. 5.44 Tools and performance process

and the information to carry out the analysis are considered. 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. 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. However, an accurate analysis of the complexity of the analysis process depends on a set of parameters that require extensive experimentation in the use of the methodologies; thus this kind of analysis is out of the scope of the present chapter.

With respect to the three dimensions of Fig. 5.43, the methods which appear at the bottom of the second group, on the left-most side of the automation dimension, exhibit a high complexity. This complexity decreases when moving along any of the three dimensions. This is due to different reasons. Toward syntactically related models or toward the same models, the mapping between behavioral and performance models simplifies or disappears. Moving up along the integration level of performance analysis into the software lifecycle dimension, the accuracy of the available information increases and simplifies the production of the performance model.

The last dimension to discuss concerns the automation of the methodologies.

Figure 5.44 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 Fig. 5.44, 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.

5.5 Desirable Attributes of Software Performance Analysis Techniques

Summing up the discussion carried out in the previous sections, a set of attributes that are desirable for software performance analysis techniques to be widely applied, even in a real context, is introduced in this section. In other words, they represent characteristics that a transformational approach should have to favor its applicability.

- **Transparency**—Transparency is the property of minimal influence of the performance analysis approach on the development process. It allows the software designer to not have specific skills in performance modeling. Transparency can be defined as a combination of the following aspects:
  - **Performance Model Derivation**—The performance model should be easily derived from the software specification. No additional efforts in the software modeling should be asked to the design team for sake of performance analysis.
  - **Software Model Annotation**—In order to obtain a performance model, it is necessary to provide quantitative, performance-oriented information that can be used to build the performance model. There are several ways to provide these pieces of information. The optimal solution is to annotate them directly on the software models by using the model extension features, if any, in order to require less efforts to the designers. The software model annotation task, if too complex, can delay the development process and prevent the usage of the performance analysis. The same conclusions can be made if the amount of the additional information needed to carry out the predictive analysis is too high or it is difficult to estimate.
  - **Performance Indices**—The performance indices of interest should be easily specified without asking for the knowledge of the underlying theory. Their specification 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 automation can 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. The successful integration of performance analysis in

the industrial software development process requires the use of an automated methodology that 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 eventually reports such analysis results on the software models by highlighting (potential) performance problems of the analyzed design.

- **Result Interpretation and Feedback Generation**—Upon performance indices calculation, from their interpretation the designer should assess the goodness of the design. However, the gap between the performance analysis results (i.e., mean values, probability distribution functions, etc.) and the suggestions that the designers expect (e.g., architectural alternatives that remove performance flaws) is evidently very large. Hence, automated techniques to fill such gap would be desirable in order to identify parts of the design that could lead to potential performance problems and suggest design alternatives to overcome the identified problems.
- **Generality**—This attribute gives information about the application domains and architectural styles (such as client/server, layered architectures and others) an approach applies to. The more the approach is general, the more it is powerful. This attribute indicates if the approach is able to deal with specific aspects such as modeling fork/join systems, simultaneous resource possession, general time distributions and arbitrary scheduling policies.
- **Scalability**—The selected approach should be scalable, meaning that the complexity of the performance model should ideally increase linearly with the software model size.

Chapter 6
Performance Model Solution

In the process of software performance modeling and analysis, although these two activities do not act in a strict pipeline, once generated/built (at whatever level of abstraction in the software lifecycle) a performance model has to be solved to get the values of performance indices of interest. The model solution actually represents the first step of the analysis, and its results produce a feedback on the performance model, and also propagate up to the software artifacts of the system under development.[1]

It is helpful to recall here that the main targets of a performance model solution are the values of performance indices. For several decades the metrics adopted in software performance have been based on three major indices: response time, throughput and utilization (as will be illustrated in Sect. 6.1.1). However, with the evolution of software and devices, performance analysis nowadays needs to address additional indices. A typical example is represented by the battery consumption rate of a mobile handheld device where the battery is a type of resource with limited capacity. It has been studied that the structure and the behavior of software on mobile devices may heavily affect this resource (e.g. repeated message sending rapidly consumes the device battery) [80], thus a software performance analysis in the mobile setting should consider this dimension. However, this dimension cannot be expressed by any of the major indices indicated above, therefore it represents in practice a new performance index.

The existing literature is rich of methodologies, techniques and tools for solving a wide variety of performance models. This is a very active research topic and, despite the complexity of problems encountered in this direction, in the last few decades very promising results have been obtained. Moreover, new tools have been developed to support this key step of software performance process (see, for example, [114]).

Therefore, the contents of this chapter are not limited to the basics of model solution techniques (Sect. 6.1). A short summary of the major tools for model solution is

[1] See Chap. 4 for relationships between performance modeling/analysis and software lifecycle.

V. Cortellessa et al., *Model-Based Software Performance Analysis*,
DOI 10.1007/978-3-642-13621-4_6, © Springer-Verlag Berlin Heidelberg 2011

provided in Sect. 6.2. Then, in the next chapter (Sect. 7.2) the problem of interpreting the performance analysis results and feeding back the performance model and the software artifacts is briefly discussed in light of new approaches introduced with this objective. Indeed a model solution is only the first step of performance analysis. The results have to be interpreted and, if unsatisfactory, finally translated into a (set of) model alternative(s) that may allow us to overcome the emerging weaknesses.

6.1 Model Solution: Foundations and Techniques

It would be a too ambitious task to confine in one book section all the existing approaches for performance model solution. The characteristics of the major categories of approaches are illustrated here, and the reader can refer to the cited bibliography for a deeper study of this vast topic.

In general, a performance model can be analyzed by using either exact mathematical methods or approximated ones or, ultimately, by simulation. Simulation is a widely used general technique whose main drawback is the high development and computational cost to obtain accurate results. A potential degree of inaccuracy of results is also a characteristic of approximate numerical solutions, although the development and computation time is reduced if compared to simulation. On the other hand, exact methods may have a much lower computational complexity if they rely on closed-form expressions, but this requires that the model satisfies a (sometime strict) set of assumptions and constraints. If instead the exact solution of the performance model relays on stochastic processes (e.g. discrete-space continuous-time homogeneous Markov chains), then the state explosion can be a problem [75].

The solution of models that correspond to complex systems can be based on decomposition and aggregation techniques [17, 70, 75]. Such techniques are also the basis of hierarchical modeling methodologies, which define structured and flexible approaches to the description and evaluation of complex systems, based on stepwise system modeling and analysis [40].

In line with *hierarchical modeling* in an 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 refined 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 from the most detailed model and requires the application of bottom–up aggregation techniques. Roughly speaking, at each level of abstraction the model can be analyzed by first solving all submodels, and then by aggregating those solutions following the constraints of the model at the next higher level.

Hierarchical performance models nicely allow the application of *hybrid approaches* to the solution of the various submodels, in that different modeling notations and analysis methodologies can be applied to different submodels before recombining the results. Some attempts exist in literature that propose the combined use of different performance models for describing submodels of a given system [20, 21].

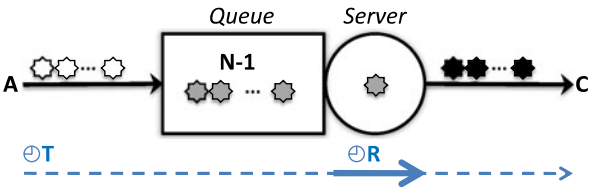


Fig. 6.1 A queued service center system

6.1.1 Operational Analysis

In this section the major operational laws of performance analysis are presented.[2]

Operational laws are simple equations which may be used as an abstract representation of the average performance behavior of almost any system. One of the advantages of the laws is that they are very general and make very few assumptions about the behavior of the random variables characterizing the system. Another advantage of the laws is their simplicity: this means that they can be applied quickly and easily by anyone. Based on a few simple observations of the system the performance analyst can, by applying these simple laws, derive more information. Using this information as input to further laws the performance analyst gradually builds up a more complete picture of the behavior of the system.

The foundation of the operational laws are observable variables. These are values that we could derive from watching a system over a finite period of time. We assume that the system receives requests from its environment. Each request generates a job or customer within the system. When the job has been processed the system responds to the environment with the completion of the corresponding request.

If we observe such an abstract system (see Fig. 6.1) we might measure the following quantities:

- T, the length of time we observe the system;
- A, the number of request arrivals we observe;
- C, the number of request completions we observe;
- B, the total amount of time during which the system is busy ($B \leq T$);
- N, the average number of jobs in the system.

From these observed values we can derive the following four important quantities:

- $\lambda = A/T$, the arrival rate;
- $X = C/T$, the throughput or completion rate;
- $U = B/T$, the utilization;
- $S = B/C$, the mean service time per completed job.

From the above definition, the simplest algebraic relationship that can be deducted is $U = X \cdot S$. It is known as

[2] Several seminal books have been published performance operational laws (see, for example, [79]). The description that follows has been taken from Jane Hillston's class notes [3].

Utilization law *The utilization is equal to the product of the throughput and the mean service time.*

If we apply this law to a single resource system like the one in Fig. 6.1 then its meaning is very relevant to figure out what can we expect from the analysis of such a system. Note that such a system is made of a server that represents the resource serving requests, and a queue that represents the structure where pending requests are stored while waiting for service.

By observing that the utilization ranges from 0 to 1, we see that the following relationship straightforwardly comes out: $X \leq 1/S$. This means that the service time S of a certain resource induces an upper bound on its throughput that is limited by the inverse of its response time. In other words, the maximum capability of a resource is obtained when such resource always have some job to be processed in its queue. Under this circumstance the resource throughput achieves its maximum that obviously is the inverse of its service time.

For example, a resource that serves every single job with an average service time of $S = 500$ m sec cannot produce a throughput higher than $X = 1/S = 2$ jobs/sec. Besides, if we know that this resource has a utilization $U = 0.6$, then its throughput is $X = 1.2$ jobs/sec.

Due to its intuitive aspect, the utilization law helps to understand a basic mechanism of system performance. It is wishful that the utilization of a certain resource is as high as possible, because this means that the resource has little idle time and it is being adequately exploited. However its throughput grows at the same pace as its utilization (because the service time is fixed). Hence, when the utilization achieves its maximum value of 1 the throughput achieves its maximum value of $1/S$. Is this situation desirable as well? What happens beyond this point? The utilization law helps to understand that a maximum utilization should never be achieved, because the system tends to be instable. In fact, the potential arrival of further jobs in its queue simply contributes to increase the queue length toward an unlimited value, whereas the system performance cannot further improve. On the opposite, the waiting time for jobs in the queue can indefinitely grow.

The best known and most commonly used operational law is Little's law. It is named after the researcher who published the first formal proof of the law in 1961, although it had been widely used before that time. Little's law is usually phrased in terms of the jobs in a system and relates the average number of jobs in the system N to the residence time R, that is the average time they spend in the system. Let X be the throughput, as above. Then Little's law states that $N = X \cdot R$.

Little's law *The average number of jobs in a system is equal to the product of the throughput of the system and the average time spent in that system by a job.*

Little's law can be easily deduced from the consideration that the response time R represents the total average time that a job has to spend within a certain system before completing its service. This time is simply given by the average time $1/X$ that the system spends to complete one job multiplied by the number N of jobs that are waiting to be served (including the job itself).

For example, if the average number of jobs in a resource is $N = 8$ and the resource is serving 32 jobs per second (i.e. $X = 32$ jobs/sec), then Little's law allows us to deduce that the average time spent within this resource by each single job is $R = N/X = 0.25$ sec. If we combine this result with the information about the service time of this resource (let us assume it is $S = 0.1$ sec), then we can obtain the average time that each job spends while waiting in the resource queue $W = R - S$, that in this case would be $W = (0.25 - 0.1)$ sec $= 0.15$ sec.

Given a computer system, Little's law can be applied at many different levels: to a single resource, to a subsystem or to the system as a whole. The definitions of the number of jobs, throughput and residence time used at the different levels must be coherent with each other, and this also holds for the definition of *request*. For example, when considering a disk, it is natural to define a request to be a disk access, and to measure throughput and residence time on this basis. When considering an entire transaction processing system, on the other hand, it is natural to define a request to be a user-level transaction, and to measure throughput and residence time on this basis. Each such transaction may generate several disk accesses.

It is often natural to regard a system as being made of a number of devices or resources. Each of these resources may be treated as a system in its own right as far as the operational laws are concerned, with the rest of the system forming the environment of that resource. A request from the environment generates a job within the system; this job may then circulate between the resources until all necessary processing has been done; as it arrives at each resource it is treated as a request, generating a job internal to that resource.

Suppose that during an observation interval we count not only completions of the whole system, but also the number of completions at each resource within the system. We define the visit count, V_i, of the ith resource to be the ratio of the number of completions at that resource and the number of system completions $V_i = C_i/C$.

More intuitively, we might think of V_i as the average number of visits that a system-level job makes to resource i. For example, if, during an observation interval, we measure 10 system completions and 150 completions at a specific disk, then on the average each system-level request requires 15 disk operations.

The forced flow law captures the relationship between the different components within a system. It states that the throughputs or flows in all parts of a system must be proportional to one another. In other words, it relates the throughput at the individual resources ($X_i = C_i/T$) to the throughput at the complete system ($X = C/T$). It comes straightforwardly from the above definition as follows: $X_i = X \cdot V_i$.

Forced flow law *The throughput at the ith resource is equal to the product of the throughput of the system and the visit count at that resource.*

An informal interpretation of this law is that, since the visit count defines the number of visits to a resource or device that each job needs in order to complete its processing, the resource must keep up a correspondingly scaled completion rate to ensure that the system completion rate is maintained.

For example, let us assume that in a system made of several resources a job visits a certain disk i in average 5 times before leaving the system (i.e. $V_i = 5$), and the

disk is serving 25 jobs per second (i.e. $X_i = 25$ job/sec). From these data we can deduce that the system throughput is $X = X_i / V_i = 25/5$ jobs/sec $= 5$ jobs/sec. On the basis of this result, if we know that another resource is serving 15 jobs per second (i.e. $X_j = 15$ jobs/sec), then we can assert that a job visits the latter resource in average $V_j = X_j / X = 15/5 = 3$ times before leaving the system.

If we know the amount of processing time that each job requires at a resource then we can calculate the utilization of the resource. Let us assume that each time a job visits the ith resource the amount of processing, or service, time it requires is S_i. Note that service time is not necessarily the same as the residence time of the job at that resource: in general a job might have to wait for some time before processing begins. The total amount of service time that a system job generates at the ith resource is called the service demand, D_i: $D_i = S_i \cdot V_i$.

The utilization of a resource, that is the percentage of time that the ith resource is in use processing a job, is denoted U_i. The utilization law applied to a specific resource in a system that is made of multiple resources states that: $U_i = X_i \cdot S_i = X \cdot D_i$.

Utilization law for the ith resource *The utilization of a resource is equal to the product of the throughput of that resource and the average service required at that resource.*

This re-formulation of the utilization law for a specific resource in a system made of multiple resources allows to introduce the concept of *bottleneck*. Informally, the bottleneck of a system is the resource that more adversely affects the performance of the system. In practice, it represents the most overloaded resource where jobs waste most of their time (as compared to service time).

One of the primary goals of performance analysis is to find system bottlenecks and remove them in order to improve the performance.[3] The utilization law for a resource, in practice, states a bound on the system throughput that depends on all the system resources, as follows.

The utilization of each resource ranges from 0 to 1, so the following relation holds: $(\forall i) X \cdot D_i \leq 1$, which can be re-written as $(\forall i) X \leq 1/D_i$. If we define $D_{max} = max_i\{D_i\}$ then we can write the latter set of inequality as this single relationship: $X \leq 1/D_{max}$. This means that the system throughput X is upper bounded from the inverse of the maximum service demand $1/D_{max}$ among all its resources. D_{max} being the largest time spent by a job in one of the system resources, $1/D_{max}$ is the minimum throughput among all system resources. It is obvious that the system throughput cannot be higher than the minimum throughput among all resources.[4]

The main side effect of the above consideration is that if all the service demands D_i are known in advance then it is possible to determine the system bottleneck as the

[3] Bottleneck analysis can be a quite complex process, based on a well-assessed theory to study system bottlenecks. We do not report the whole theory in this chapter, but we only provide a sketch of it. Readers interested to a simple and complete presentation of this theory can refer to [79].

[4] In a team race each team is as slow as its slowest runner.

one with maximum D_i. Hence, in case of performance problems, the analysts know where to take a corrective action on the system. But what are the potential corrective actions? From a numerical point of view, corrective actions are all those actions that decrease the value of D_{max}. Recall that $D_i = S_i \cdot V_i$, hence we can define two typical categories of actions: (i) *hardware actions* that decrease S_i, and (ii) *software actions* that decrease V_i.

In order to decrease the average service time S_i of a resource, the only type of effective action is to replace the resource with a quicker one, hence a hardware solution.[5] As opposite, in order to decrease the average number V_i of visits to the same resource the jobs should change their paths within the system. The latter improvement can be only achieved by changing the logics of the software that originates such amount of visits. For example, in order to decrease the number $V_i = 10$ of accesses to disk made by a certain routine, the latter can be replaced by another (optimized) routine that is able to provide the same functionality with a lower number of accesses to disk, e.g. $V_i = 8$.

This distinction on the type of actions that can be taken to overcome a performance problem directly relates to the distinction that we have made in Chap. 1 between system performance and software performance. In other words, the above formulas allows to quantitatively distinguish hardware actions form software actions. As discussed in Chap. 1, software actions should be preferred where feasible.

Along the same direction it is possible to reason on residence time and relate the one of the whole system to the ones of the single resources. One method of computing the mean residence or response time per job in a system is to apply Little's law to the system as a whole. However, if the mean number N of jobs in the system or the system level throughput X are not known then an alternative method can be used.

By applying Little's law to the ith resource we see that $N_i = X_i \cdot R_i$, where N_i is the mean number of jobs at the resource and R_i is the average response time of the resource. From the forced flow law we know that $X_i = X \cdot V_i$. Thus we can deduce that $N_i / X = V_i \cdot R_i$.

The mean number of jobs in the system is clearly the sum of the mean number of jobs at each resource, i.e. $N = N_1 + \cdots + N_M$ if there are M resources in the system. We know from Little's law that $R = N/X$ and from this we arrive at the general residence time, or general response time law: $R = \sum_{i=1}^{M} V_i \cdot R_i$.

General response time law *The average residence time of a job in the system will be the sum* (*overall resources*) *of the product of its average residence time at each resource by the number of visits it makes to each resource.*

In case of interactive systems the Little's law must be adapted to the case. Interactive systems are those in which jobs spend time in the system not engaged in

[5] Note, however, that modern definitions of resources widen their scope to complex combinations of hardware and software that provide certain services. In these cases the actions that decrease S_i may also concern the software component of a resource.

processing, or waiting for processing: this may be because of interaction with a human user, or may be for some other reason. In such systems the think time Z is defined as the average time that a user spends thinking at his terminal before submitting a new request after received a response to the previous request. In practice, Z is the average time between a response and the following request from the user's point of view. For example, if we are observing a cluster of workstations with a central file server to investigate the load on the file server, the think time might represent the average time that each workstation spends processing locally without access to the file server. At the end of this non-processing period the job generates a fresh request.

The key feature of such a system is that the residence time can no longer be taken as a true reflection of the response time of the system. The think time represents the time between processing being completed and the job becoming available as a request again. Thus the residence time of the job, as calculated by Little's law as the time from arrival to completion, is greater than the system's response time. The interactive response time law reflects this, as it calculates the response time R' as follows: $R' = N/X - Z$.

Interactive response time law *The response time in an interactive system is the response time minus the think time.*

Note that if $Z = 0$ then $R' = R$, and the interactive response time law simply becomes Little's law.

An opportunely combined usage of the laws introduced in this section allows to evaluate the performance indices of a system whose parameters are known.

6.1.2 Solution Techniques and Related Notations

In this section some existing solution techniques, based on the operational laws of Sect. 6.1.1, are introduced. They are all related to certain modeling notations, therefore the section is partitioned in subsections, one for each notation. The notations we deal with in this section are a subset of the ones described in Chap. 3, as from these techniques it is possible to build techniques to solve models represented in the notations excluded from this subset. Example of software tools that have been built to implement such techniques will be presented in Sect. 6.2.

Note, however, that all techniques included in this section follow certain algorithmic procedures to extract values of indices from performance models. A completely different approach, that can be considered as orthogonal to the modeling notations is the simulation and it will be separately discussed in Sect. 6.1.3.

Markov Models

The literature on solution techniques for Markov models is very rich, and beside classical approaches in the last few years sophisticated techniques have appeared to

tackle the problem of state explosion. In this section we only provide some basic notions on how to calculate the steady-state probabilities of a Markov model [19]. For all other techniques (e.g. transient-state probabilities) we direct interested readers to classical references such as [113].

First we recall that the steady-state probabilities of a Markov model (see Chap. 3) can be interpreted in two ways. One way is to see them as the long-run proportion of time the model spends in the respective states. The other way is to regard them as the probabilities that the model would be in a particular state if one would take a snapshot after a very long time. Let us see how to obtain these probabilities from the model parameters.

The probability of residence in state j after n steps is denoted by $\pi_j(n)$, and can be obtained as follows:

$$\pi_j(n) = \sum_i \pi_i(0)\, p_{i,j}(n) \tag{6.1}$$

where $p_{i,j}(n)$ are obtained from the state transition probability matrix $\mathbf{P} = [p_{ij}]$ as the probabilities to reach state j in n steps starting from state i. Equation (6.1) can be expressed in matrix–vector notation, with $\underline{\pi}(n) = (\pi_0(n), \pi_1(n), \ldots)$, as follows:

$$\underline{\pi}(n) = \underline{\pi}(0)\mathbf{P}^n \tag{6.2}$$

Under the hypothesis that a limit exists for all the rows of matrix $\mathbf{P}^n$, we define $\underline{v} = (\ldots, v_j, \ldots)$ as

$$v_j = \lim_{n\to\infty} \pi_j(n) = \lim_{n\to\infty} \sum_i \pi_i(0)\, p_{i,j}(n) \tag{6.3}$$

Equation (6.3) can be expressed in matrix–vector notation as follows:

$$\underline{v} = \lim_{n\to\infty} \pi(n) = \lim_{n\to\infty} \underline{\pi}(0)\mathbf{P}^n \tag{6.4}$$

However, we also have

$$\underline{v} = \lim_{n\to\infty} \pi(n+1) = \lim_{n\to\infty} \underline{\pi}(0)\mathbf{P}^{n+1} = \Big(\lim_{n\to\infty} \underline{\pi}(0)\mathbf{P}^n\Big)\mathbf{P} = \underline{v}\mathbf{P} \tag{6.5}$$

Hence, whenever the limit probabilities $\underline{v}$ exist, they can be obtained by solving the system of linear equations:

$$\underline{v} = \underline{v}\mathbf{P} \Rightarrow \underline{v}(\mathbf{I} - \mathbf{P}) = \underline{0} \tag{6.6}$$

where $\sum_i v_i = 1$ and $0 \le v_i \le 1$ because $\underline{v}$ is a probability vector, and $\mathbf{I}$ is the identity matrix. The vector $\underline{v}$ represents the stationary or steady-state probability vector of the Markov model [19].

The computation of steady-state probabilities is a core step in the solution of a Markov model. When such models are used to estimate performance, such probabilities have to be combined, for example, to state rewards in order to obtain values of performance indices. A reward can be associated to each state of a Markov model, and rewards are accumulated each time the model transits through the state. It is intuitive that in order to estimate the average reward of the model it is sufficient to make a weighted sum of state rewards on the basis of the steady-state probabilities.

Let us give an example. Assume that a Markov model represents a single queue system, where the semantics of each state is completely described by a single attribute that is the number of jobs in the queue. If a reward r_i is assigned to each state that simply corresponds to number of jobs in the queue when the system is in state i, then by combining these rewards with the steady-state probabilities v_i it is possible to compute the average queue length $\overline{ql}$ as follows:

$$\overline{ql} = \sum_i v_i \cdot r_i \tag{6.7}$$

Rewards can be assigned to states and transitions and, if properly processed, permit to express any performance index on a Markov model that represents the dynamics of a performance model.

Queueing Networks

Although they can be applied to multiple performance notations, the operational laws introduced in Sect. 6.1.1 were originated from the study and the observation of queueing models, where servers provide services and might have queues where jobs wait for their turn.

On the basis of those laws, several algorithms have been created to provide operational procedures for Queueing Networks solution. Some approaches provide exact solutions under certain assumptions on the model structure, whereas other approaches have an iterative behavior that, with a certain approximation, leads us to estimate the performance indices of interest after a certain number of iterations.

For the sake of completeness we describe here only the most common algorithm for closed queueing models, that is Mean Value Analysis (MVA), that we consider here only for models with single class of jobs (see Sect. 3.2). However details on other versions of this approach as well as different approaches to the algorithmic solution of QN models can be found in literature (see, for example, [79]).

MVA applies to Product Form Queueing Networks (PFQN), that is a class of QN that satisfies some assumptions on its topology (e.g. no fork and join nodes), on the queue disciplines (e.g. no load dependent scheduling strategies) and on the time distributions (e.g. only Poisson distribution for workload arrivals).

The MVA algorithm is based on three equations [79], which are

- *Little's law for interactive systems applied to the whole model.* Recall that Little's law for interactive systems is given by $R' = N/X - Z$, where all parameters are affected by the number N of jobs in the system. Another way of expressing this law is

$$X(N) = \frac{N}{Z + \sum_{k=1}^{K} R_k(N)} \tag{6.8}$$

  where K is the number of service centers and $R_k(N)$ is the response time of kth center when there are N jobs in the system.

for $k \leftarrow 1$ to K do $Q_k \leftarrow 0$
for $n \leftarrow 1$ to N do
begin
for $k \leftarrow 1$ to K do $R_k \leftarrow \begin{cases} D_k & \text{(delay centers)} \\ D_k(1+Q_k) & \text{(queueing centers)} \end{cases}$
$X \leftarrow \frac{n}{Z+\sum_{k=1}^{K} R_k}$
for $k \leftarrow 1$ to K do $Q_k \leftarrow X R_k$
end

Fig. 6.2 Exact MVA solution technique

- *Little's law applied to a single service center*. It can be expressed as $Q_k(N) = X(N) \cdot R_k(N)$, where $Q_k(N)$ represents the queue length of kth center when there are N jobs in the system.
- *Residence time equations for each service center*. It can be expressed as $R_k(N) = D_k \cdot (1 + A_k(N))$, where $A_k(N)$ is the average number of jobs at kth center when a new job arrives.

Note that the key entities to compute performance measures are the $A_k(N)$'s. If these are known, $R_k(N)$ can be computed, followed by $X(N)$ and $Q_k(N)$. In general, the arrival instant queue lengths $A_k(N)$ are computed at the instants that some job is arriving to the center (and so cannot itself be in the queue there), while the time averaged queue lengths $Q_k(N)$ are computed over randomly selected instants (so all jobs potentially could be in the queue). MVA introduces a technique to compute $A_k(N)$ by observing that $A_k(N) = Q_k(N-1)$.

In other words, the queue length seen at arrival to a queue when there are N jobs in the network is equal to the time averaged queue length with one less job in the network. This equation has an intuitive justification. At the moment a job arrives at a center, it is certain that this job itself is not already in that queue. Thus, there are only $N-1$ other jobs that could possibly interfere with the new arrival. The number of these jobs that are in queue, on average, is simply the average number when only those $N-1$ jobs are in the network.

The MVA solution technique, shown in Fig. 6.2, involves the iterative application of the equations above. These equations allow us to calculate the system throughput, device residence times, and average queue lengths when there are n jobs in the network, given the average queue lengths with $n-1$ jobs. The iteration begins with the observation that all queue lengths are zero with zero jobs in the network. From that trivial solution, the above equations can be used to compute the solution for one job in the network. Since the time averaged queue lengths with one job in the network are equal to the arrival instant queue lengths with two jobs in the network, the solution obtained for a population of one can be used to compute the solution with a population of two. Successive applications of the equations compute solutions for populations $3, 4, \ldots, N$ [79].

After N iterations, when MVA terminates, the values of R_k, X, and Q_k (all for population N) are available immediately. Other model outputs are obtained by using the operational laws.

The last aspect that we recall in this section concerns the model parameterization problem. As mentioned in Sect. 3.3, up to the end of the 1980s, QN were parameterized on the basis of observed system behaviors or the experience of system developers. Explicit models, oriented to performance, of software behavior (that represents the workload populating a QN) have started to appear about two decades ago. Execution Graphs were the most promising model and still today they represent the prevalent model notation used for this purpose [110]. In Sect. 6.2 we introduce a tool that allows us to synthesize EGs and to parameterize QN using the result of synthesis. However, it is out of the scope of this book to describe details of techniques to model and manage EG.[6]

Stochastic Petri Nets and Stochastic Process Algebras

The quantitative analysis of a SPN is based on the identification and solution of its associated Markov chain. 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 SPN. Non-polynomial algorithms exist for product-form SPN, under further structural constraints. Beside the product-form results, many approximation techniques have been defined [17]. There also exist analyses of STPNs without Markovian assumptions. Most of them provide performance bounds, others analyze stability conditions [17].

Although there exist various quantitative analysis techniques and some software tools (e.g. GreatSPN [39]) for SPN, the applications of SPN are often limited to small size problems. This is due essentially to the time and space complexity of the numerical analysis algorithms and of simulations.[7]

Similar solution techniques can be applied to SPAs. The quantitative analysis 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.

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 PA terms whose underlying Markov chain admits a product-form solution that could allow for more efficient solution algorithms [66, 71, 105].

Examples of performance evaluation tools for SPA are the TIPP tool [68], Two Towers [31, 32], and PEPA Workbench [57].

[6] Readers interested to EGs can refer to Smith's book [110].

[7] A database on (stochastic) Petri net tools can be found at [5].

6.1.3 Simulation

We recall that simulation can be viewed either as one of the available techniques to solve performance models built in one of the canonical performance notations illustrated in Chap. 3, or as a specific technique for models built for sake of simulation (i.e. simulation models) [28]. In this section we consider the latter case.[8]

We briefly recall the main steps, among the ones described in Chap. 3, that relate to the solution of a simulation model[9]:

1 building a simulation model/program;
2 planning the simulation experiments;
3 running the simulation program and analyzing the results.

Before becoming a program coded in some (general purpose or simulation) language, a simulation model is an abstract representation of the system that has to be simulated. Typical activities of Step 1 are: (i) the definition of types of events that may occur within the system; (ii) the construction of cause-effect relationships among types of events; (iii) the definition and modeling of simulation components, that are the subsystems acting as event producers/consumers in the simulation model.

Once completed these three activities, the complexity of the translation of the simulation model into an executable software piece of code mostly depends on the adopted language. The executable code is made of three parts: a simulation engine, that is the process that drives the execution of the simulation model; the data structures of the model that store information related to the system; the set of event routines, that are the procedure implementing the actions to be taken when a certain type of event must be simulated (obviously an event routine for each type of event must be implemented).

Quite sophisticated and integrated environments are available today to build, run and manage a simulation model, like Simula [97]. If such a framework is adopted then it usually provides a proprietary (graphical) language that helps designers to build the model without the need to address model details.

If, instead, a general purpose programming language, such as C++, is adopted to build a simulation program then all details must be explicitly implemented, in some cases with the help of appropriate libraries of functions, because the simulation engine and the model in this case are often embedded within the same code.

Two major types of simulation engines can be built: *event-driven* and *time-driven*.

An event-driven engine works as follows. The engine basically manages a list of events ordered on their timestamps, that are the virtual times at which the events

[8] Many excellent books have been published on (discrete event) simulation, whereas our intent here is only to mention it as a software performance modeling and analysis approach.

[9] In this section we implicitly refer to Discrete Event Simulation [28], as the most widely used in the software performance domain.

must occur.[10] The simulation time advances following the virtual time of the next event to be executed. Once extracted from the list, the event routine corresponding to this type of event is executed. The latter may read/write shared data structures, local variables, and finally it may originate one or more events to be executed at a later virtual time. These events are appropriately inserted in the event list. Hence, the event virtual times push ahead the simulation time.

A time-driven engine works differently from an event-driven one. The engine advances the simulation time at fixed steps, and after each increment verifies which events may happen at the current point of time and executes them. All the remaining mechanisms are similar to the ones of an event-driven engine.

Time-driven simulation is suitable for regular systems, where it is well-known that some events happen at each time step (e.g. the movement of an elementary particle). Event-driven simulation is suited, instead, for systems where events are sparse along the timeline, because the effort to jump from an event to the next one does not depend on their virtual time distance.

6.2 Model Solution: Tools

In this section we make a short overview of well-assessed tools that implement solution techniques that have been illustrated in Sect. 6.1.[11]

6.2.1 SHARPE—Multiple Performance Model Notations

SHARPE (Symbolic Hierarchical Automated Reliability and Performance Evaluator) is a tool for specifying and analyzing performance, reliability and performability models. It provides a specification language and solution methods for most of the commonly used model notations for performance, reliability and performability analysis. Notations include Fault Trees, Queueing Networks, Stochastic Petri Nets, as well as Markov models (with rewards). Steady-state, transient and interval measures can be computed. Output measures of a model can be used as parameters of other models. This facilitates the hierarchical combination of different model types.

SHARPE supports a variety of model notations and, for most of the model notations, provides more than one analysis algorithm from which the user can choose. It allows the user to choose when to combine different models, which ones to combine, and how to combine them. SHARPE models were designed to answer the question:

[10] We here only introduce the main mechanisms of simulation, and we assume that basic concepts (like virtual time) are known to the reader.

[11] Note that most of this section contents comes from the web pages of the tools and only partially has been re-arranged for use in this book. The URL of each tool has been reported in the respective section for retrieving further information.

given time-dependent functions that describe the behavior of the components of a system and a description of the structure of the system, what is the behavior of the whole system as a function of time? The functions might be cumulative distribution functions (CDFs) for component failure times, CDFs for task completion times, or the probabilities that components are available at a given time.

The system structure might be specified, for example, in the form of a fault tree, a queueing network or a Markov chain. The time-dependent functions describing the component behavior are restricted by SHARPE to be exponential polynomial in form. This is not a great restriction, because many of the most commonly used distribution functions have this form and much of the work that has been done in fitting distributions to the data has used exponential polynomials.

URL—http://people.ee.duke.edu/~kst/

6.2.2 SPE·ED—Execution Graphs and Queueing Networks

SPE·ED is a commercial tool designed specifically to support the Software Performance Engineering (SPE) methods and models defined by Connie U. Smith in [110]. The SPE techniques use performance models to provide data for the quantitative assessment of the performance characteristics of software systems as they are developed. Using a small amount of data about envisioned software processing, SPE·ED creates and solves performance models, and presents visual results. It provides performance data for requirements and design choices and facilitates comparison of software and hardware alternatives for solving performance problems.

With SPE·ED users can create graphical models of envisioned software processing and provide performance specifications by creating and specifying Execution Graphs. Queueing network models are automatically generated from the software model specifications. A combination of analytic and simulation model solutions identifies potential performance problems and software processing steps that may cause the problems.

The user's view of the model is a scenario, that is an Execution Graph of the software processing steps. Software scenarios are assigned to the facilities that execute the processing steps. Models of distributed processing systems may have many scenarios and many facilities. Users specify software resource requirements for each processing step. Software resources may be the number of messages transmitted, the number of SQL queries, the number of SQL updates, etc. depending on the type of system to be studied and the key performance drivers for that system.

A performance specialist provides overhead specifications that specify an estimate of the computer resource requirements for each software resource request. These are specified once and re-used for all software analysis that executes in that environment.

SPE·ED produces analytic results for the software models, and an approximate, analytic MVA solution of the generated QN model. A simulation solution is used for QN with multiple software scenarios executing on one or more computer system

facilities (i.e. multiple classes of jobs). It supports hybrid solutions, in that the user selects the type of solution appropriate for the development lifecycle stage and thus the precision of the data that feeds the model. There is no need for a detailed, lengthy simulation when only rough guesses of resource requirements are specified.

The results reported by SPE·ED are the end-to-end response time, the elapsed time for each processing step, the device utilization, and the amount of time spent at each computer device for each processing step. This identifies both the potential computer device bottlenecks, and the portions of the device usage by processing step (thus the potential software processing bottlenecks).

SPE·ED is intended to model software systems under development, although it may also be used for existing software systems. It may be any type of software: all types of software applications including web applications, operating systems, or database management systems.

URL—http://www.spe-ed.com/sped.htm

6.2.3 GreatSPN—Stochastic Petri Nets

GreatSPN is a software package for modeling, validation, and performance evaluation of distributed systems using Generalized Stochastic Petri Nets and their colored extension: Stochastic Well-formed Nets. The tool provides a friendly framework to experiment with timed Petri net based modeling techniques. It implements efficient analysis algorithms to allow its use on rather complex applications, not only toy examples.

GreatSPN is composed of many separate programs that cooperate in the construction and analysis of PN models by sharing files. Using network file system capabilities, different analysis modules can be executed on different machines in a distributed computing environment. The modular structure of GreatSPN makes it open to the addition of new analysis modules as new research results become available. All solution modules use special storage techniques to save memory both for intermediate result files and for program data structures.

Among the analysis procedures available within GreatSPN we mention Reachability Graph Generator equipped with a Reachability Graph Analyzer, Markov Chain Generator equipped with a steady-state solver and a transient solver, GSPN simulator.

Several state-of-the-art analysis prototypes have been recently added in the package: an algorithm for the fast computation of performance bounds based on linear programming techniques, working at a purely structural level (i.e. the computed bounds depend only on the average firing delay of the transitions while they do not depend on the Probability Distribution Functions of such delays); algorithms for the analysis of Stochastic Well-formed Nets providing the user with the possibility of designing models of complex systems in a more compact way, and allowing for a more efficient state space analysis that automatically exploits the model symmetries.

The graphical interface has recently achieved high portability under different hardware platforms.

URL—http://www.di.unito.it/~greatspn/index.html

6.2.4 TimeNET—Stochastic Petri Nets

The software package TimeNET is a graphical and interactive toolkit for modeling with Stochastic Petri Nets (SPNs) and Stochastic Colored Petri Nets (SCPNs). TimeNET has been developed at the Real-Time Systems and Robotics group of Technische Universität Berlin (Germany). TimeNET and its predecessor DSPNexpress were influenced by experiences with other well-known Petri Net tools, like GreatSPN and SPNP.

The project has been motivated by the need for powerful software for the efficient evaluation of Timed Petri Nets with arbitrary (non-exponential) firing delays.

The classical main model class of TimeNET are extended Deterministic and Stochastic Petri Nets (eDSPNs). Firing delays of transitions can either be zero (immediate), exponentially distributed, deterministic, or belong to a class of general distributions called *expolynomial* in an eDSPN. Such a distribution function can be piecewise defined by exponential polynomials and has finite support. It can contain even jumps, making it possible to mix discrete and continuous components. Many known distributions (uniform, triangular, truncated exponential, finite discrete) belong to this class.

Under the restriction that all transitions with non-exponentially distributed firing times are mutually exclusive, stationary numerical analysis is possible. If the non-exponentially timed transitions are restricted to have deterministic firing times, transient numerical analysis is also provided. For the case of concurrently enabled deterministically timed transitions, an approximation component based on a generalized phase type distribution has been implemented. If there are only immediate and exponentially timed transitions, the model is a GSPN and standard algorithms for steady-state and transient numerical evaluation based on an isomorphic Markov chain are applicable.

The tool also comprises a simulation component for eDSPN models, which is not subject to the restriction of only one enabled non-Markovian transition per marking. Steady-state and transient simulation algorithms are available. During the simulation run, intermediate results of the performance measures are displayed graphically together with the confidence intervals.

Simple stochastic Petri nets in TimeNET can either be interpreted in continuous time as an eDSPN, or as a Discrete Deterministic and Stochastic Petri Net (DDSPN). DDSPNs allow geometric distributions, deterministic times and discrete phase type distributions as delays. Steady-state and transient numerical analysis as well as efficient parallel simulation are available.

Variable-free Colored Petri Nets (VfCPNs) represent another model class of TimeNET. Firing delays of transitions have the same range as in eDSPNs. Numerical steady-state analysis, an iterative approximation method, and standard simulation are available for VfCPNs.

Stochastic Colored Petri Nets (SCPNs) have been added recently. Due to the inherent complexity of the considered models, a requirement of only one non-exponential transition per marking is too restrictive. Thus a standard discrete-event simulation has been implemented for the performance evaluation of SCPN models. A distributed simulation method has been developed and implemented for this model class in addition, which allows for the efficient simulation of complex models on a cluster of workstations.

URL—http://www.tu-ilmenau.de/TimeNET

6.2.5 TwoTowers—Stochastic Process Algebras

TwoTowers is a tool based on an Architectural Description Language called Æmilia that allows to describe static and dynamic models of software architectures. In addition, Æmilia models can be annotated with rates on actions. An annotated Æmilia model is automatically translated, through TwoTowers, in a Stochastic Process Algebra notation called Æmpa. Once translated, the model is ready to be solved with commonly adopted SPA solution techniques.

TwoTowers is able to evaluate the performance of correct Æmilia specifications in two different ways.

In the first case instant-of-time, stationary/transient performance measures, which are defined through state and transitions rewards, are computed by solving the Markov chain underlying the Æmilia specification. The value of each such performance measure is given by the sum of the stationary/transient state probabilities and transition frequencies of the Markov chain, each weighed by the corresponding state reward or transition reward, respectively. A state reward represents the rate at which gain is cumulated while staying in a certain state, whereas a transition reward represents the gain that is instantaneously earned when executing a certain transition. In TwoTowers three methods are available for solving Markov chains: Gaussian elimination, an adaptive variant of symmetric stochastic overrelaxation, and uniformization.

In the second case the method of independent replications, based on simulation experiments, is applied to estimate the mean, variance or distribution of performance measures, which are specified through an extension of state and transition rewards. The discrete event simulation can be trace driven, which means that certain values can be taken from a file instead of being sampled from pseudo-random number generators. Unlike the Markovian performance evaluation, the simulation-based performance evaluation can be applied to any (correct) Æmilia specification with no open and deadlock states, thus making the estimation of the performance measures of systems possible with generally distributed delays.

URL—http://www.sti.uniurb.it/bernardo/twotowers/

Chapter 7
Advanced Issues in Software Performance

In this chapter some advanced issues of software performance have been collected. They address different aspects of this discipline, not necessarily related to each other. The chapter is not meant to be a structured overview of the main open issues in the field, rather it is an anthology of issues that we have faced in the last few years.

Section 7.1 presents an approach to integrate performance modeling within the Model-Driven Architecture paradigm. It is a sort of instantiation of the general approach presented in Chap. 4. Section 7.2 illustrates our recent work on the interpretation of performance analysis results aimed at providing feedback to software engineers that allow one to overcome performance problems. Section 7.3 describes a methodology for run time using model-based performance analysis to re-configure software systems. Finally, in Sect. 7.4 the problem of unifying the existing approaches to the software performance is faced by defining the characteristics of a software performance ontology that may represent an unifying notation in this field.

7.1 Software Performance and Model-Driven Architecture

Model-Driven Architecture (MDA) [83] has given the opportunity to software scientists and developers to converge toward a common framework that defines the theory and practice of model-driven engineering. MDA is based on a 3-layered structure of models, suitably related through model transformations.

The Computation Independent Model (CIM), Platform Independent Model (PIM) and Platform Specific Model (PSM) provide good abstractions of many development practices in model-based software engineering. Indeed, independently of the development process, a model of requirements must be built at the beginning of the process (Computation Independent Model); then it inputs the architectural design phase that produces a model of the system logics (Platform Independent Model), and finally from the latter model the implementation phase produces the system code for a certain platform (Platform Specific Model).

V. Cortellessa et al., *Model-Based Software Performance Analysis*,
DOI 10.1007/978-3-642-13621-4_7, © Springer-Verlag Berlin Heidelberg 2011

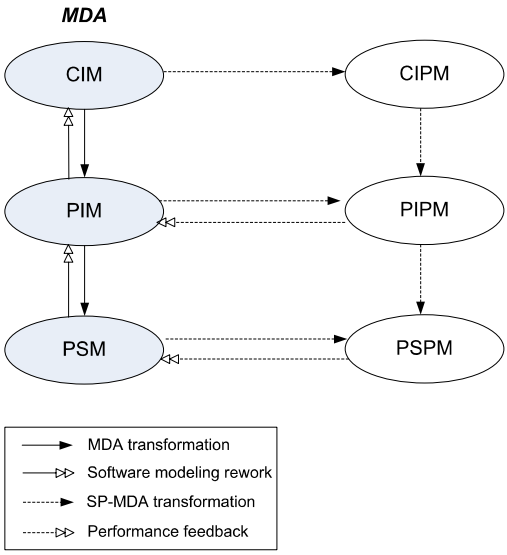


Fig. 7.1 The SPMDA framework

Although MDA nicely represents the functional aspects of a model-driven software development process, it falls short of representing the modeling and analysis necessary to guarantee non-functional properties of software products. Goal of this section is to present a general way to embed the performance validation activity in the MDA framework. This will take the form of an extension of the MDA framework, namely SPMDA (Software Performance Model-Driven Architecture) which, beside the typical MDA models and transformations, embeds new models and transformations that take into account the performance validation activities.

To this end, new types of model transformation are defined, which are different from those in MDA whose aim is to transform less refined models in more refined ones up to the automated code generation. In a software performance analysis context, instead, model transformation becomes an activity that does not necessarily aim at producing more refined models, but also allows a change of model notation to favor different types of analysis. Hence, in SPMDA is introduced the concept of horizontal transformation, that is: a transformation between models representing the system at the same level of detail from different viewpoints. For example, a horizontal transformation can be devised from a set of UML diagrams representing a software architecture annotated with performance data to a Queueing Network representing the same software architecture in a different formalism ready to be analyzed.

The extended performance view of the MDA approach is illustrated in Fig. 7.1. On the left side of the figure the canonical MDA models and transformations appear: a Computation Independent Model (CIM) becomes a Platform Independent Model (PIM) upon specifying business logics of the system; PIM becomes a Platform Specific Model (PSM) once the business logics is implemented on a particular platform. These models and transformations do not suffice to keep performance aspects under control during the software system development. In order to address

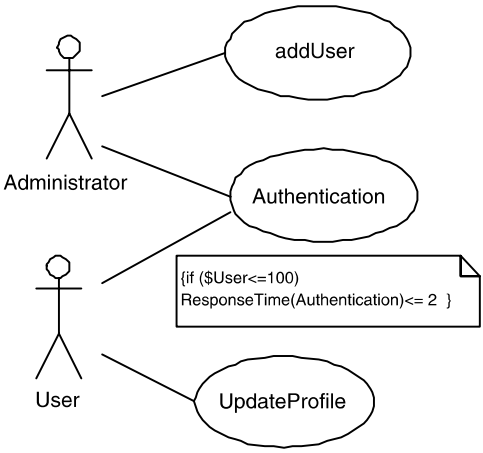


Fig. 7.2 An example of CIPM

this issue, in Fig. 7.1 three additional types of models and three additional types of transformations/relationships have been introduced. The new types of models are CIPM, PIPM, PSPM.

CIPM—A Computation Independent Performance Model represents the requirements and constraints related to the performance. Very often the performance requirements concern the response time of some system operations under certain workloads (e.g. the authentication service must be completed in average within 2 seconds under a workload of maximum 100 users). In the MDA framework the UML use case diagrams are the most widely used diagrams to model system requirements and their relationships. Therefore, a good example of CIPM may be a use case diagram annotated with performance requirements. Figure 7.2 shows the requirement formulated above.

PIPM, a Platform Independent Performance Model, is a representation of the business logics of the system along with an estimate of the amount of resources that such logics needs to be executed. The model must be computationally solvable, which means it should be expressed through a notation that allows performance analysis (e.g., Queueing Networks). In a PIPM platform characteristics (such as network latency) are not available. Therefore, the analysis of the performance models represented in a PIPM class can only permit to discover early design bugs. At this level of detail, given a set of resource types, any action of the business logics can be coupled with an estimated amount of each resource type needed to execute the action. For example, the amount of resources needed to execute the logics implementing an authentication service might be as follows: N high-level statements, K database accesses and M remote interactions. The estimates being expressed in non-canonical units of measure, the results of the analysis of a PIPM cannot be used as a target for comparison to the actual system performance. They are rather meant:

- to evaluate upper and lower bound in the system performance,
- to identify software performance bottleneck in the business logics (e.g. a software component that is overloaded),

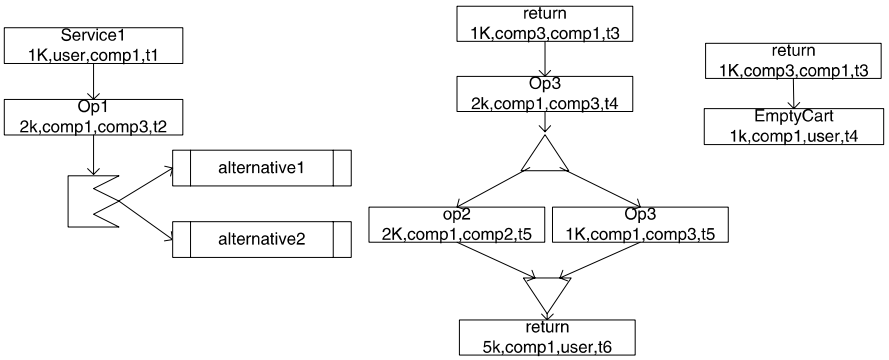


Fig. 7.3 An example of PIPM

- to compare the performance of different design alternatives of the business logics (e.g. two different software architectures).

In Fig. 7.3 a PIPM is represented as an Execution Graph. Notice that the model represents the system at the architectural level. Each block in the Execution Graph represents an action (*Service1*) that is quantitatively described by its annotation. For example, *Service1* in component *comp1* is invoked by a *user* that sends a message whose size is *1K* in *t1* units of time.

PSPM, a Platform Specific Performance Model, contains the merged representations of the business logics and of the platform adopted to run the logics. In a classical MDA approach a platform is represented by a set of subsystems and technologies that provide a coherent set of functionalities through interfaces and specified usage patterns (e.g. J2EE, CORBA, etc.). In a performance context a platform must also include the characteristics of the underlying hardware architecture, such as the CPU speed and the network latency. Being expressed in classical units of measures (e.g. execution time), the results of the analysis of a PSPM can be used as a target for comparison to the actual system performance in order to validate the model. Such a model can then be used to explore the system behavior in extreme real world scenarios (e.g. severely heavy workloads).

In Fig. 7.4 a PSPM model is represented as a Layered Queueing Network. In this model the platform characteristics are specified in terms of middleware (*CORBA*) and hardware characteristics (*Processor1*).

Behind the PIPM/PSPM duality there is the intuitive concept that the performance analysis results can be expressed with actual time-based metrics only after a PIM is bounded to its platform and becomes a PSM. Obviously the results coming out of a PIPM evaluation are not useful to validate the model against the system requirements, because they may take very different time values on different platforms. However, three types of actions can originate from this analysis:

- Lower and upper bounds on the system performance can be evaluated if some estimates of the performance of the possible target platforms are available. For example, if the lower bound on a system response time is larger than the corre-

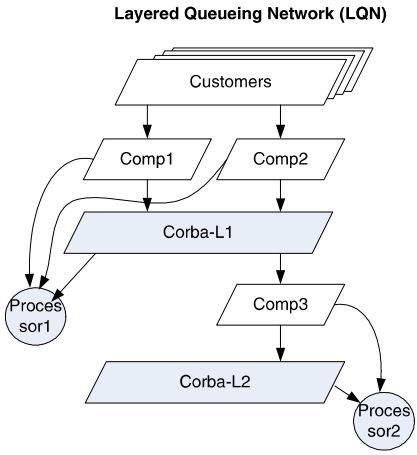


Fig. 7.4 An example of PSPM

sponding performance requirement, then it is useless to progress in the development process as performance problems are intrinsic in the software architecture. It is necessary to rework on the software models. However, even when the results are not so pessimistic, it is possible to take decisions that improve the software architecture.

- In order to identify the most overloaded components, the utilization and/or queue length of each service center in the PIPM must be computed versus the system population. An overloaded component has a very long waiting queue and represents a bottleneck in the software architecture. Some rework is necessary in the PIM to remove the bottleneck.
- Either as a consequence of the above decisions or as a planned performance test, different (functionally equivalent) alternative software designs can be modeled as PIMs, and then their performance can be compared through their PIPMs in order to select the optimal one.

The continuous arrows with single filled peak represent the typical MDA transformations that permit to obtain a PIM from a CIM, and a PSM from a PIM. The dashed arrows with single filled peak represent the additional transformations introduced to tie MDA and SPMDA models.

Horizontal and vertical SPMDA transformations can be characterized as follows:
SPMDA horizontal transformation—It transforms a software model into the corresponding performance model at any level in the MDA hierarchy. In Fig. 7.1 CIM → CIPM, PIM → PIPM and PSM → PSPM are horizontal transformations. The model transformations belonging to this class share a two-steps structure: the software model is first annotated with data related to the system performance (e.g. the operational profile), thereafter the annotated model is transformed into a performance model. Many examples of such transformation have been introduced in Chap. 5.
SPMDA vertical transformation—Even though it seems to play in the performance domain a similar role to the MDA ones, such transformation is instead intended to provide an input to the horizontal transformation. In other words, often the

horizontal transformations needs some input from the one-step-higher performance model in the hierarchy, hence: CIPM → PIPM is a contribution to a PIM → PIPM transformation and, likewise, PIPM → PSPM is a contribution to a PSM → PSPM transformation.

Arrows with double empty peak also appear in Fig. 7.1 that represent the feedback paths originating from the performance analysis. They give completeness to the software performance analysis process according to what described in Chap. 4.

For a more comprehensive presentation of the SPMDA framework, its extensions and related approaches, please refer to [41–43].

7.2 Interpretation of Performance Analysis Results

As we have seen throughout the whole book, there are quite well-assessed techniques to automatically generate performance models and solve them. As opposite, there is still a clear lack of automation in porting the analysis results back to the software model. In this section we briefly illustrate the open problems that are on the ground and a possible approach to tackle part of these issues.

The performance indices obtained from the model solution, which are typically represented by average values and/or distribution functions, have to be interpreted in order to search, if any, performance problems. This type of search may be quite complex and needs to be smartly driven toward the problematic areas of the model. The complexity of this step stems from several factors: (i) performance indices are basically numbers associated to model entities and often they have to be jointly examined to let problems emerge; (ii) performance indices can be estimated at different levels of granularity and, as it is unrealistic to keep under control all indices at all levels of abstraction, incomplete information often results from the model evaluation; (iii) software models can be quite complex, they involve different characteristics of a software system (such as static structure, dynamic behavior, etc.), and performance problems sometimes appear only if those characteristics are cross-checked.

Therefore the need of guidance in this searching process is clear enough. Strategies to drive the search can rely on quite different elements that may depend on the adopted model notation, on the application domain, on environmental constraints, etc.

Once performance problems have been detected (with a certain accuracy) somewhere in the model, solutions have to be applied to remove those problems.[1] Typical solutions consist in architectural alternatives, namely feedback, that modify the original software model to achieve better performance. It is obvious that if all performance requirements are satisfied then the feedback simply suggests that no changes have to be made on the software model.

[1] Note that this task very closely corresponds to the work of a physician: observing a sick patient (the model), studying the symptoms (some bad values of performance indices), making a diagnosis (performance problem), prescribing a treatment (performance solution).

Figure 7.5 schematically represents the process executed, at a generic point of the software lifecycle, to assess and (if needed) improve the performance of a software system under development. Rounded boxes in the figure represent operational steps, whereas square boxes represent input/output data. Vertical lines divide the process in three different phases: in the *modeling* phase a software model is built; in the *analysis* phase a performance model is obtained through model transformation, and such a model is solved to obtain the performance indices of interest; in the *refactoring* phase the performance indices are interpreted and, if necessary, feedback is generated as refactoring actions on the original software model.

The modeling and analysis phases represent the forward path from an (annotated) software model to performance indices. In this path several approaches have been introduced for model transformation (see, for example, [24]) and well-founded performance model solvers have been developed (see, for example, [39]). Instead there is a clear lack of automation in the backward path that elaborates the analysis results and brings back to the software model some form of feedback. The refactoring phase in Fig. 7.5, whose main task is the *result interpretation and feedback generation*, embraces the localization of performance flaws in the software model and their removal without violating design constraints.[2] Such activities are today exclusively based on the analysts' experience, and therefore their effectiveness often suffers the lack of automation.

Performance antipatterns represent a promising instrument to introduce automation in these activities. An antipattern is a well-known bad practice that should be avoided to achieve a better design. A performance antipattern identifies a practice that badly affects the software performance, and it may involve static and dynamic aspects of software as well as deployment features. A performance antipattern definition includes, beside the problem description, a possible solution of the problem. The main source of performance antipatterns is the work done over the last years by Smith and Williams [111] that have ultimately defined a number of 14 notation- and domain-independent antipatterns.

However, being this process unavoidably based on heuristic evaluations and decisions, there is no guarantee that the feedback actually solves the problems. Hence the analysis has to be performed again, starting from the updated software model and ending up with new performance indices that shall confirm whether the performance problems have been actually removed or not. Only after this validation the software lifecycle can proceed. If the validation is unsuccessful the refactoring has to be executed again and the whole process restarts.

7.3 Performance-Based Software Reconfiguration

Recently, growing attention focused on run-time management of Quality of Service (QoS) of complex software systems. For software systems whose performance

[2] It is obvious that if all performance requirements are satisfied, then the feedback simply suggests no change on the software model.

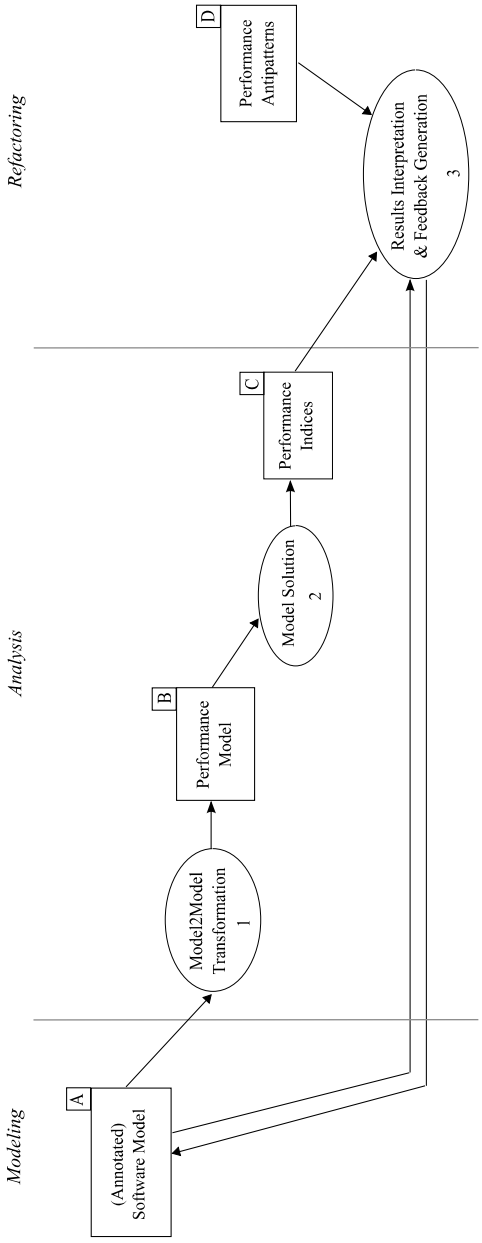


Fig. 7.5 Software performance modeling and analysis process

requirements are strict, in addition to performance validation at design time, performance attributes should be monitored during their execution, in order to react when performance degradations are experienced.

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 (see, for example, [90]). 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.

The automated and dynamic nature of the reconfiguration process imposes new challenges to the decision step that aims at choosing the next system configuration to overcome the observed problem. Most reconfiguration approaches use prefixed strategies that are in general coded in the application or in the reconfiguration framework. However, in QoS management a prefixed schema of decision can prevent the implementation of smart alternatives more suitable to effectively overcome the observed problems.

In this setting an interesting problem related to the approaches presented in the previous chapters concerns the use of predictive analysis based on performance models to support the decision step. These models may represent the application at the architecture level of details. Indeed, the use of a software architecture as abstraction of the system under analysis allows avoiding unnecessary details and simplifies the evaluation phase in terms of model complexity and resolution time.

The Performance Management Framework (PMF) is an example of system in this category [38]. It monitors the current performance of the application and, when some problem occurs, it chooses a new configuration based on the feedback provided by the on-line evaluation of the performance models corresponding to several reconfiguration alternatives. The main characteristic of PMF is the mechanism to generate such alternatives. Differently from other approaches it does not rely on a fixed repository of predefined configurations but on a reconfiguration policy defined as a suitable combination of basic reconfiguration rules. The reconfiguration policy is evaluated on the data retrieved by the on-line monitoring (that represents a snapshot of the system current state), thus generating a number of new configurations. Once such alternatives have been generated, the on-line evaluation is carried on to predict which one is most suitable to solve the observed problem.

The use of predictive system performance models improves the reconfiguration process. It allows the choice of the system reconfiguration alternative that guarantees the performance constrains and shows, in the predictive analysis, better performance. However, the run-time evaluation of predictive models representing the software systems poses strong requirements on the models themselves as later discussed.

The PMF approach is based on monitoring the running system to collect data, on dynamic reconfiguration to change the running configuration, and on model-based

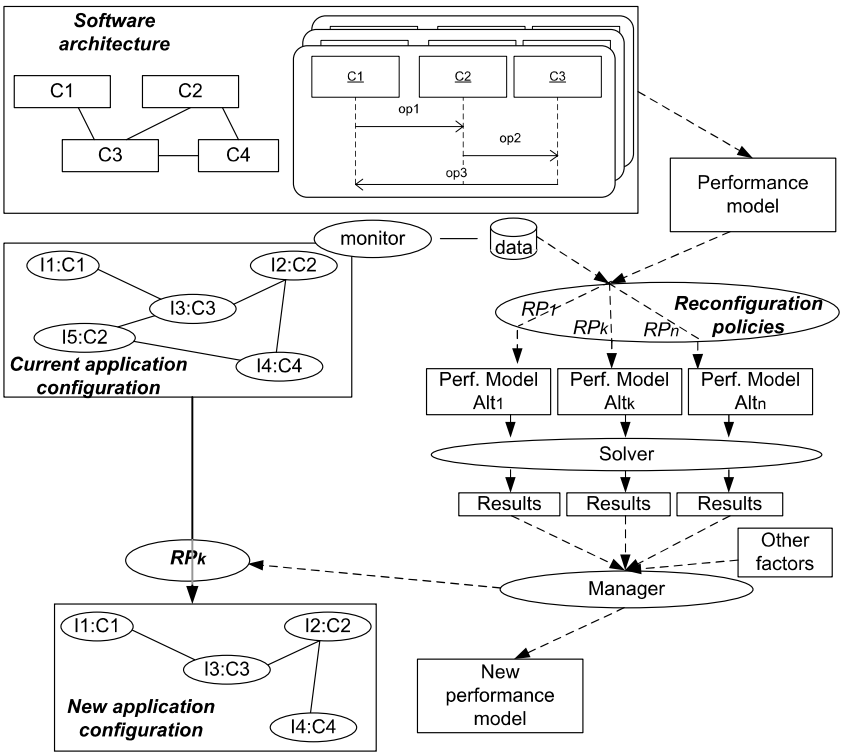


Fig. 7.6 The performance management framework

performance analysis to decide the next system configuration among the available ones.

Figure 7.6 outlines the PMF process and its flow of activities. PMF observes the software application during its execution to monitor performance attributes of the software application. Whenever the performance constraints are no longer satisfied, the adaptation management process will start. The monitored data are evaluated in order to identify the performance problem and the portion of the system affected by it. Such information is used to plan changes in the system configuration in order to overcome the observed problem. Whenever a new system configuration is determined, the changes are enacted and the system configuration is modified accordingly.

In PMF the configuration alternatives are built on-the-fly by applying reconfiguration policies suitable for the application. The initial performance model is, in general, specified by performance specialists. The next ones instead are generated on-the-fly by modifying the current performance model during the evaluation of the reconfiguration policies. Finally, the information collected during the monitoring phase is used to evaluate the predictive performance model(s).

The description of the alternatives need to be simple in order to reduce the overhead of the model generation and evaluation, and, hence, of the decision step. This

simplification may impact on the accuracy of the measured performance indices, but the evaluation should be accurate enough for the choice of the reconfiguration alternative. The intuition here is that all the alternatives are represented at the same level of abstraction and the actual data are observed through the same abstractions thus providing a uniform workbench to consistently evaluate different alternatives. In [38] this intuition is confirmed by an empirical experimentation that showed that the chosen alternative was indeed the best among the generated ones.

In the following some of the PMF assumptions on allowed reconfigurations and critical issues are discussed. However, for a more comprehensive presentation of PMF and related approaches, please refer to [38].

7.3.1 Allowed Reconfigurations

In PMF the allowed reconfigurations are of two kinds: they may change internal parameters of software components (such as the number of threads or other features preventively defined by the component developer); the application topology may change by adding/removing component and/or connector instances. In contrast a reconfiguration must not change the application functionalities (e.g. the substitution of a component with a new one providing different services). This restriction is necessary since a change in the application behavior would imply a re-design of the performance model, and not only a change in its topology or in some parameters. Consequently the reconfiguration process could not be automatically carried out.

To relax this restriction, the framework should rely on a database containing several different implementations of a component together with their performance models. When the reconfiguration policy requires the substitution of an implementation of a component, the adaptation of the performance model is done by replacing the sub-model of the first implementation with the one of the new implementation retrieved from the database.

7.3.2 Issues to Address

PMF requires several issues to be taken into account to guarantee the correctness of the process and its efficiency. These are related to the data to collect, the condition triggering the system reconfiguration and the reconfiguration alternatives to consider, the performance model to use, the technique for its evaluation and the mechanism to choose the next system configuration.

Relevant Data to Collect

Performance models are parameterized by means of monitored data. Such data are collected from the running system and are used to feed the performance model of

the system described at the software architecture level. This means that the collected data are more fine-grained than the performance model parameters, thus an abstraction step is needed. The system analyst, who decides which data to monitor, has to be able to identify the proper data abstractions to feed them back to the performance model.

When Should the Reconfiguration Be Performed?

The condition under which the adaptation process should start is a very critical issue in run-time management. It influences the execution frequency of the reconfiguration loop. Conditions that are verified too often lead to a high overhead: the management framework can consume more resources than the application itself. Conversely, conditions that rarely trigger the reconfiguration can prevent a timely management of performance problems.

The critical issue here is to determine the best tradeoff between computational overhead and timely resolution of performance problem.

Which Performance Model to Use?

Performance models have to be modified and evaluated on-line. These characteristics pose requirements on the models themselves. The choice of a suitable performance model of the system becomes one of the most important and critical step in the implementation of the adaptation management process. On one side (i) models must be as flexible as possible to be automatically changed according to the reconfiguration policy; on the other side (ii) models should allow for their analysis on-the-fly. These two characteristics might be incompatible. As a matter of fact, (i) requires detailed models which permit to apply changes, such as *re-parameterization* according to the measurements done on the system or *modification* in terms of their topology, in order to reflect the new system configuration. (ii) requires models which have a *short* time of execution. This implies that only models having analytical/numerical solution are admissible.

The challenge here is to design performance models expressive enough to describe sensible different alternatives with respect to performance behavior, but still having numerical/analytical solution.

On-Line Evaluation

In order to control the state space explosion of the analytical/numerical solution the model of the system should be as expressive as possible, by omitting useless details about components behavior. The choice of the software architectural abstraction for the application behavior description permits to address this problem. The software architecture is the minimal detailed description of the application having all the

behavioral information needed to carry on a predictive performance analysis, and it allows lightweight and fast model evaluation. Of course, there is a tradeoff between the simplicity of the model and the support of the feedback provided for on-line decision making.

How Is the Decision on the Next Configuration Taken?

In order to be effective, the reconfiguration process must actually improve the performance of the managed system. Indeed, complex systems must address several non-functional requirements. The risk of having a degradation of some other non-functional property (e.g., security) related to the reconfiguration is avoided by allowing only a controlled set of configuration alternatives, which are decided by the developer according to the risks associated with the reconfiguration. Moreover, at each reconfiguration step, the *costs* to place the system in the new selected configuration, should be considered during the selection. This can be achieved by combining the result of the model evaluation provided by the solver with a coefficient representing the cost of the reconfiguration process.

7.4 A Unifying Ontology for Software Performance

The approaches to software performance modeling and validation surveyed in Chap. 5 share the idea of annotating software models with information related to performance (e.g. the operational profile), and transforming the annotated model into a performance model (e.g. a Stochastic Petri Net). Up to date, no standard has been defined to represent the information related to performance in software artifacts, although clear advantages in tool interoperability and model transformations would stem from it.

This section discusses the possibility of determining a software performance ontology, that is: a common set of basic recurring concepts and relations, to relate the different vocabularies for performance data used in different approaches. The benefits of having a common ontology in this field would allow for tool interoperability in two directions: the same performance model could be much more easily solved from different solver tools (i.e. horizontal interoperability), and the tools transforming software models into performance models could rely on a common terminology (i.e. vertical interoperability). The software performance discipline includes additional concepts like performance measurement, monitoring, management, and workload generation. This section focuses on software performance modeling aspects and does not address these concepts, even though in a broader ontological study of performance they shall enter into the picture.

The key step to move toward the definition of an ontology is to devise the primary entities, and their relations, necessary to represent the software performance domain. To this end, we consider three relevant experiences aimed at standardizing the vocabulary of software performance experts, that is:

UML Profile for Schedulability, Performance and Time (SPT) [85], adopted from OMG to represent in UML, among other, performance data and partially introduced in Chap. 3[3];
Core Scenario Model (CSM) [91], that is: a meta-model developed from the Real-Time and Distributed Systems Group at Carleton University to integrate performance annotations into software models;
Software Performance Engineering meta-model (SPE-MM), due to the joint effort of C. Smith, L.Williams and C. Llado [107, 120], which defines entities and relationships useful to build Software Execution Models and System Execution Models [106].

It is interesting to observe that the SPT, CSM and SPE-MM experiences were developed in different environments, namely a company/university consortium, a university research group and a company. They all share the goal of defining performance data in a standard way.

Another recent experience in this direction has been introduced in [61], where KLAPER (Kernel LAnguage for PErformance and Reliability analysis) has been fully described. The scope of KLAPER is even larger than the ones considered here, but for the sake of readability we keep the discussions focused on the three above meta-models.

In this section we first describe the three meta-models, then we introduce a bottom–up approach to extract common knowledge from the meta-models, finally we propose questions that should drive a top–down approach to the ontology definition.

The three meta-models described in Sect. 7.4.1 have been originated by the need of unambiguously representing software performance concepts and relations. In this section we move a further step in this direction. We try to both distill their common knowledge and to identify different non-shared concepts, i.e. entities that do not appear in all the meta-models. No attempt to make a "qualitative" comparison among the three meta-models is intended. Based on the results of this bottom–up process, we devise some guidelines to drive a top–down process to the ontology definition that starts from the intrinsic perception that an expert has of the software performance domain.

7.4.1 Three Meta-models for Software Performance

SPT, CSM and SPE-MM are here described by partitioning their entities in three areas. The entities representing the software dynamics fall into the *sw-behavior* area, the ones representing resource aspects fall into the *resources* area, and workload entities into the *workload* area. Some entities cannot be uniquely classified. They are highlighted in the pictures in shaded areas and will be discussed in Sect. 7.4.2.

[3]Note that SPT profile has been replaced, in the UML 2 framework, by MARTE [88]. However, for the sake of this section SPT remains a valid example to discuss a performance ontology.

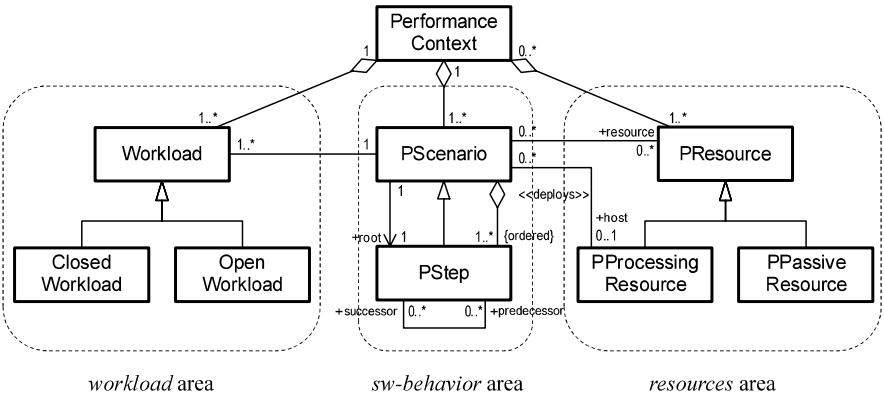


Fig. 7.7 The performance domain model in SPT

The choice of these basic ontological grouping relies on two considerations: (i) workload, software model and platform architecture have always been the founding element of a software performance model, (ii) the study of the three meta-models has confirmed such natural partition. The existence of entities in shaded areas evidences that these elements are not uniquely separable in the devised areas.

SPT Profile

The UML Profile for Schedulability, Performance and Time (namely SPT) has been adopted as a full specification in September 2003 [85].[4] It is an extensive document that introduces UML extensions for managing typical issues of real-time applications. The tags and stereotypes defined in SPT have been widely used, in the last few years, to annotate UML models and translate them into performance models [24].

SPT is partitioned in five main sections including a *Performance Modeling* one. In the latter all the entities related to performance issues are defined as stereotypes and tags. Some of them inherit from stereotypes belonging to the *General Resource Modeling* section.

Figure 7.7 shows the performance domain model of SPT, where the entities have been grouped following the three areas introduced above.

The software dynamics (i.e. the *sw-behavior* area) is based on a PScenario stereotype, which represents the system response to a, possibly external, event. A scenario can be described through an ordered sequence of steps. Each step is represented by an instance of the PStep stereotype. PScenario and PStep have attributes to model performance aspects of the software dynamics, such as step probability and scenario response time.

[4]For details about specific stereotypes of SPT, please refer to Chap. 3 of this book.

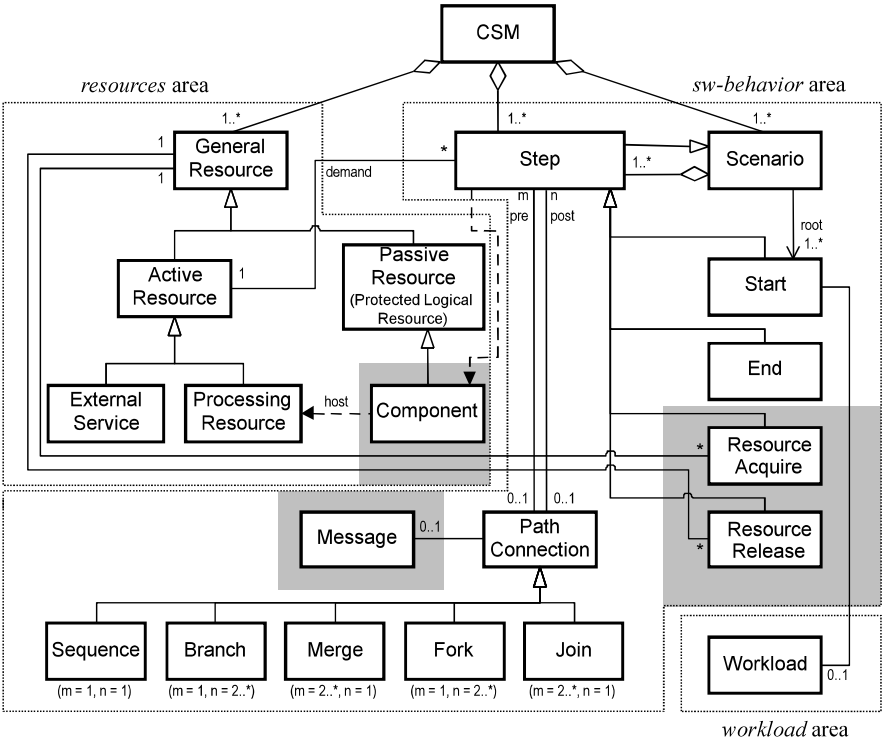


Fig. 7.8 The CSM meta-model

The resource definition (i.e. the *resources* area) is based on inheritance relations from entities in the *General Resource Modeling* section of the profile. All the resources share attributes like utilization and scheduling policy. Processing resources have attributes like processing rate and context switching time. Passive resources are accessed during the execution of an operation and are protected by an access mechanism. They have attributes like capacity and access time.

Workloads share the response time and the priority attributes, and they are split into closed and open workloads (see the *workload* area).

Core Scenario Model Meta-model

The Core Scenario Model (CSM) is an effort of the Real-Time and Distributed Systems Group at Carleton University to create a standard interface between different software specification tools and different performance models. This work is part of a wider project aimed at developing a unified approach to building performance models from design models [6]. CSM originates from the performance domain model of SPT. It makes explicit some information which has to be inferred from UML and SPT data, thus extending its scope to represent models beyond UML [6, 91].

Figure 7.8 shows the CSM meta-model, where the entities have been grouped in the three areas.

The roles of scenario and step within the *sw-behavior* area are very similar to the ones of SPT. Start and End steps are introduced to define the limits of a scenario. The sequencing of steps is made explicit by introducing five types of path connections with optional attributes like a condition and the reference to a message (i.e. the entities between Sequence and Join in the bottommost side of Fig. 7.8).

The shaded areas in *sw-behavior* emphasize three entities. Message, which models a potential interaction taking place between two contiguous steps, whose attributes are the interaction type (i.e. none, synchronous, asynchronous, reply), the size of message exchanged, and an optional multiplicity. Resource Acquire and Resource Release, which represent special types of steps modeling the acquisition and release of both passive and active resources (i.e. they are placeholders in the step sequence).

In the *resource* area, beside the classical distinction between active and passive resources, an External Service entity has been introduced. It models service operations executed by active resources (or subsystems) not included in the design document. The shaded area emphasizes the Component class. It models the operating system process that, on one side, is the dynamic image of a specific step and, on the other side, is hosted by a Processing Resource (see the dashed arrowed lines in Fig. 7.8).

All the information about the system workload are lumped into the Workload entity, which is the only one within the *workload* area.

Software Performance Engineering Meta-model

Software Performance Engineering has been shortly introduced in Chap. 1. The central idea is the separation of the Software Execution Model (SoEM, i.e. the software dynamics and load model, based on Execution Graphs) and the System Execution Model (SyEM, i.e. the platform architecture model, based on Queueing Networks). The population of a Queueing Network (i.e. classes and chains of jobs) is obtained from the processing of one or more Execution Graphs.

Since then, some effort has been spent to formalize meta-models on the top of SoEM and SyEM [107, 109, 120].[5]

Figures 7.9 and 7.10 represent the meta-models of SoEM and SyEM, respectively, where the entities have been grouped following the three areas. It is evident that Fig. 7.9 mostly represents the *sw-behavior* area, whereas Fig. 7.10 represents the other two areas.

Figure 7.9 is the definition of a scenario through an Execution Graph. Three types of nodes have been defined: Processing Node, which can be either a basic

[5]Work is still in progress in this direction, but for the sake of this section we take the Software Execution Model meta-model in [120] and the System Execution Model meta-model in [107] as references.

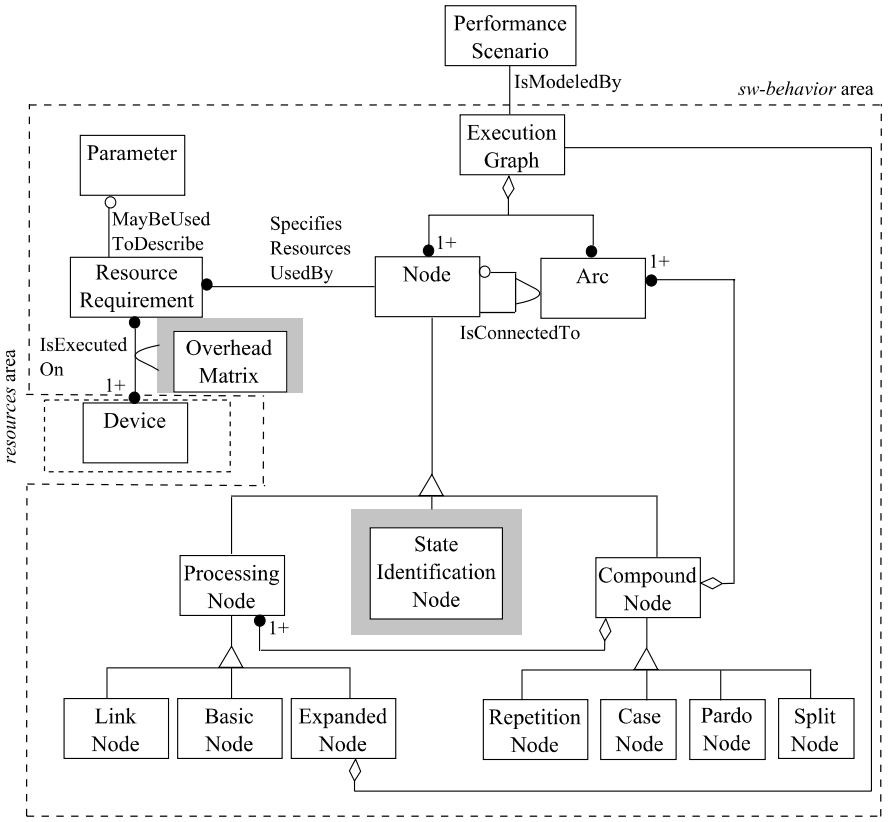


Fig. 7.9 SPE-MM: the software execution model definition

indivisible operation, or a sub-Execution Graph, or a link to an external scenario; State Identification Node (shaded in the figure, and discussed in Sect. 7.4.2), used for acquiring and releasing shared resources; Compound Node, which represents a node ruling special sequencing of nodes, such as loop, fork, etc. The left-hand side of the meta-model contains the entities that model the mapping of the software onto the hardware platform. Node resource requirements are defined and then translated (through an Overhead Matrix, shaded entity) into the actual load of the hardware platform. A reference to the latter is given by the Device entity, listing the types and multiplicities of devices in the hardware platform, and therefore placed in the *resources* area of SoEM in Fig. 7.9.

Figure 7.10 shows the meta-model that represents a Queueing Network (QN) and its load. The semantics given to a QN is such that a node of the network represents a device of the hardware platform where the software runs. Various types of (QN) nodes are defined in the *resources* area, and arcs between nodes are also introduced. In the *workload* area, besides the classical distinction between closed and open workload, entities, in the shaded zone, are introduced to refine the concept of service request, and they will be discussed in the next section.

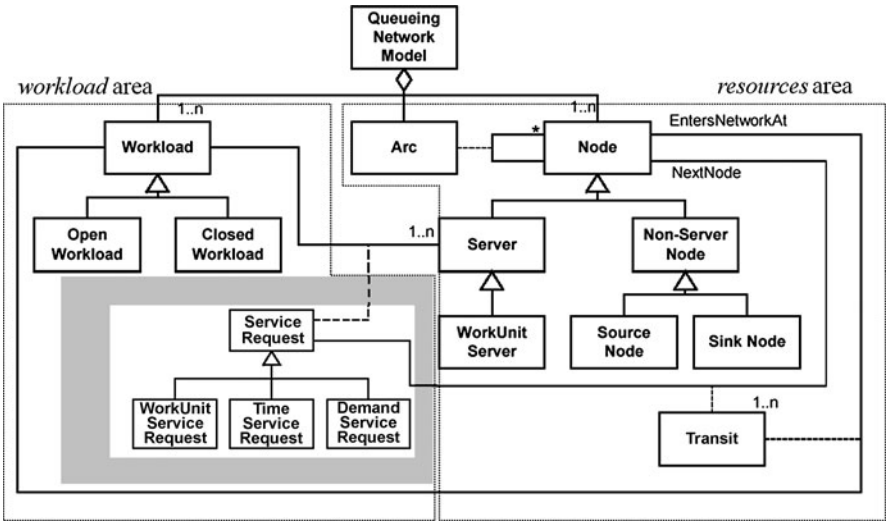


Fig. 7.10 SPE-MM: the system execution model definition

7.4.2 Building an Ontology from Common Entities: A Bottom–Up Approach

In this section the shared concepts among the three meta-models (i.e. SPT, CSM and SPE-MM) are analyzed by considering the *sw-behavior*, *resources*, and *workload* areas.

Sw-behavior

In this area, *scenario* and *step*, with a scenario made of a collection of steps are the shared core entities.

Even though in different ways, in all the meta-models a scenario relates to the *workload* area. In SPT multiple types of workload can be associated to a scenario, in CSM a scenario is associated to a workload through its starting step, in SPE-MM they are linked by a more complex mechanism.[6]

In SPT a scenario also relates to the resources through deployment and utilization associations. Differently, in the other two meta-models each single step (i.e. Step in CSM and Node in SPE-MM) relates to the resources: in CSM it is associated to Active Resource, whereas in SPE-MM to Device through the Resource Requirement

[6]The synthesis of an Execution Graph, or part of it (depending on the level of detail of the analysis), determines a workload that is assigned to the Queueing Network Model [106].

entity. In CSM a step is also associated to the Component entity, and this will be discussed in Sect. 7.4.2.

CSM and SPE-MM provide a much deeper detail on the step representation than SPT. Special types of step connections are introduced in CSM (other than Sequence) to represent complex software behaviors: Branch, Merge, Fork and Join. Similarly, in SPE-MM, even though each pair of nodes is connected through an arc, special types of nodes allow complex software behaviors: Repetition, Case, Pardo, Split. In SPE-MM two additional special types of nodes allow a node to refer to an external scenario (i.e. Link Node) or to an internal sub-scenario (i.e. Expanded Node).

The shaded areas introduced in Figs. 7.8 and 7.9, require further comments.

The State Identification Node entity in SPE-MM represents acquire and release operations on resources, which in CSM are represented by the pair of shaded entities on the right-hand side of Fig. 7.8. In SPE-MM the Overhead Matrix entity has also been shaded.

The Message entity in CSM represents the type, size and multiplicity of messages exchanged between system nodes. This concept was formerly introduced in [45], where arrows of UML sequence diagrams were annotated with size and type of messages exchanged during the interactions.

This analysis of the *sw-behavior* areas can be summarized as follows:

- A scenario is always a collection of steps.
- A scenario is always associated to workload, sometimes to multiple workload (i.e., SPT and SPE-MM) and sometimes to single workload (i.e., CSM).
- A step is often associated to resources (i.e., in CSM and SPE-MM), so the resource demand is defined at a scenario level only in SPT.
- Special steps are often defined (i.e., in CSM and SPE-MM), and the following correspondences hold: Case Node in SPE-MM $\leftrightarrow$ Branch and Merge in CSM, Pardo Node in SPE-MM $\leftrightarrow$ Fork and Join in CSM, Split Node in SPE-MM $\leftrightarrow$ Fork (alone) in CSM.
- Expanded and Link Nodes in SPE-MM allow a wider variety of scenario nesting with respect to SPT and CSM.
- A Message in CSM allows to describe in wider detail the characteristics of an interaction between system nodes.

Resources

First the SPT and CSM *resource* areas are analyzed, then the SPE-MM one is discussed. SPE-MM gives a special organization to the *resource* area (see Fig. 7.10) due to the strong relation of this meta-model with Queueing Networks.

The *resource* area of SPT corresponds to a subset of the CSM *resource* area, as follows: PResource in SPT $\leftrightarrow$ General Resource in CSM, PPassiveResource in SPT $\leftrightarrow$ Passive Resource in CSM, PProcessingResource in SPT $\leftrightarrow$ Processing Resource in CSM. The associations do not change, except for the introduction of a placeholder class in CSM, namely Active Resource, which allows one to distinguish between an internal processing resource and an external processing resource represented by the

External Service entity. The latter represents the possibility of requiring a resource outside the modeled system.

It has been already discussed in Sect. 7.4.2 the different relations between the *resource* area and the *sw-behavior* area in SPT with respect to CSM and SPE-MM. Indeed, the former meta-model acts at a scenario level, whereas the latter ones act at a step level.

The Component entity in CSM is shaded in Fig. 7.8. In its original definition it identifies an operating system process (or thread) that is assigned to a step to be executed on a processing resource. This entity can be very relevant in the performance validation of a software system, as often the number of threads on a certain host is one of the main parameters of the system scalability. Composition-related issues of a software performance ontology are discussed in Sect. 7.4.3.

The *resource* area of SPE-MM cannot be closely compared to the ones of the other two meta-models, as it is founded on Queueing Network principles. The entities in the right-hand side of Fig. 7.10 are used to represent the topology of a Queueing Network (i.e. through arc connections), different types of service centers (e.g. server nodes), and the routing matrix (i.e. the Transit entity over the node connections).

The analysis of the *resource* areas can be summarized as follows:

- Resources are partitioned in SPT and CSM in passive resources and active resources.
- External Service and Component entities in CSM are useful to represent, respectively, the usage of external resources and a sort of middle layer between a procedural step and the processing resource that executes it.
- The special organization of SPE-MM allows us to provide more details of the underlying platform, but it shall be generalized to represent other types of platform models.

Workload

The definition of workload is basically the same all over the three meta-models. A workload can be open or closed, even though this distinction is hidden in the Workload attributes in CSM. Common attributes to describe, for example, the arrival process and the population size of each workload type are introduced in all meta-models. Thus, except for the different association with the *sw-behavior* area (see Sect. 7.4.2), the only difference in the workload definition across the meta-models is in the Service Request entity of SPE-MM.

The analysis of the *workload* areas can be summarized as follows:

- The system workload is always characterized as either close workload or open workload (depending on whether the population size is fixed or not).
- The workload is always associated to a software scenario or sub-scenario.

7.4.3 Expressiveness Issues: A Top–Down Process

In this section a top–down approach to determine a "wish-list" of requirements for a common software performance ontology is presented. Each requirement is then considered in reference to the three considered meta-models.

A software performance ontology should be able to represent the results of the performance analysis.

The performance results obtained from the analysis should be unambiguously expressed, independently of the type of model and the model solution adopted to obtain them, so that they can be reported on the software model to identify bottlenecks and possible design alternatives. To (partially) address this requirement in SPT there are entities to explicitly represent the performance indices of interest (e.g. response time, utilization) as well as their mathematical representation (e.g. mean value, cumulative distribution, percentile). Both CSM and SPE-MM do not cope with the representation of performance indices, as they adopt the internal representations of their respective tools [7, 52] to handle the analysis results.

A crucial step in this direction is the "interpretation" of the performance analysis. This leads to two questions: (i) how to relate performance results (i.e. response time, utilization, etc.) to recommendations to provide software engineers with?; (ii) what are the concepts (and relations) needed to express this piece of information in the software performance ontology?

A software performance ontology should cope with as many performance formalisms as possible. The considered meta-models meet to a different extent this requirement. The translation of UML models annotated with SPT tags and stereotypes into models based on different performance notations has been extensively experimented and validated [24]; CSM-based models have appeared quite recently and some experience has been gained to translate them into Layered Queueing Networks and Stochastic Petri Nets [6]; SPE-MM-based models are quite strongly oriented to represent Queueing Networks.

A software performance ontology should allow easy integration of a software model with performance annotations. Despite the present predominance of the use of UML software models, as described in Chap. 2 many software modeling notations can be used. Among the three meta-models probably SPT shows the main drawbacks with regard to the ability to cope with various software notations, because it has been conceived as an UML profile. Also SPE-MM is fairly linked to the Execution Graphs representation, so preventing from easily annotating different types of software models. CSM is an attempt to abstract from a specific notation, even though the usage examples provided up to now are bounded to UML models.

A software performance ontology should be compatible with the internal representations of existing performance tools. This is a requirement aiming at reuse as performance experts are not prone to change their consolidated practice. Two out of three meta-models, namely CSM and SPE-MM, have been created in the same environments where performance solver tools had previously been created. Therefore the characteristics of these tools, i.e. LQNS [52] and SPEED [7], respectively, affect

the choices of the authors of these meta-models. The SPT authors, instead, worked in more freedom as no specific model solver was created for UML models.

A software performance ontology should deal with component-based software development. This is a very important requirement in the present, component-oriented, software world. Software composition is a well-studied aspect from a functional viewpoint, whereas composition of non-functional properties, such as performance, is still an open issue. Providing tools to represent this aspect is a subtle but crucial characteristic of an ontology in this area. SPT suffers the fact that components are not uniquely defined in UML 1.x. Similarly in SPE-MM Execution Graphs are better suited to represent the functionalities of a software system rather than of its software components. A great potential has been introduced in CSM thanks to the Component entity. Although its scope is quite limited, it seems to be a promising starting point to deal with questions like "Can the system performance be modeled starting from the performance of its components?".

References

1. C++SIM. http://cxxsim.ncl.ac.uk/
2. CSIM-performance simulator. http://www.atl.imco.com/proj/csim
3. Jane Hillston's notes of lectures: Private communication
4. JavaSim. http://javasim.ncl.ac.uk/
5. Petri Nets tools database. http://www.daimi.aau.dk/PetriNets
6. PUMA project. www.sce.carleton.ca/rads/puma/
7. SPEED performance modeling tool. http://www.perfeng.com/sped.htm
8. The Perl programming language. http://www.perl.org/
9. ObjecTime Ltd.: Developer 5.1 Reference Manual. ObjecTime Ltd. (1998)
10. OPNET Manuals: Mil 3, Inc. (1999)
11. Ajmone, M., Balbo, G., Conte, G.: A class of generalised stochastic petri nets for the performance evaluation of multiprocessor systems. ACM Transactions on Computer Systems **2**, 93–122 (1984)
12. Ajmone, M., Balbo, G., Conte, G.: Performance Models of Multiprocessor Performance. The MIT Press, Cambridge (1986)
13. Aquilani, F., Balsamo, S., Inverardi, P.: Performance analysis at the software architecture design level. Performance Evaluation **45**(4), 205–221 (2001)
14. ArgoUML: Object-oriented design tool with cognitive support
15. Arief, L.B., Speirs, N.A.: 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 (2000)
16. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing **1**(1), 11–33 (2004)
17. Baccelli, F., Balbo, G., Boucherie, R.J., Campos, J.J., Chiola, G.: Annotated bibliography on stochastic petri nets. In: Tract, C. (ed.) Performance Evaluation of Parallel and Distributed Systems-Solution Methods, Amsterdam, 1994, pp. 1–24 (1994)
18. Baeten, J.C.M., Weijland, W.P.: Process Algebra. Cambridge University Press, Cambridge (1990)
19. Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.-P., Siegle, M. (eds.): Validation of Stochastic Systems – A Guide to Current Research. Lecture Notes in Computer Science, vol. 2925. Springer, Berlin (2004)
20. Balbo, G., Bruell, S., Ghanta, S.: Combining queueing networks and generalized stochastic petri nets for the solution of complex models of system behavior. IEEE Transactions on Computers **37**, 1251–1268 (1988)
21. Balsamo, S., Bernardo, M., Simeoni, M.: Combining stochastic process algebras and queueing networks for software architecture analysis. In: Proceedings of the Third International Workshop on Software and Performance, WOSP02, pp. 190–202 (2002)

DOI 10.1007/978-3-642-13621-4,

22. Balsamo, S., De Nitto Persone, V., Inverardi, P.: A review on queueing network models with finite capacity queues for software architectures performance prediction. Perform. Eval. **51**(2/4), 269–288 (2003)
23. Balsamo, S., De Nitto Personé, V., Onvural, R.: Analysis of Queueing Networks with Blocking. Kluwer Academic Publishers, Dordrecht (2001)
24. Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development: A survey. IEEE Transactions of Software Engineering **30**(5), 295–310 (2004)
25. Balsamo, S., Donatiello, L., van Dijk, N.: Bounded performance analysis of parallel processing systems. IEEE Transactions on Parallel and Distributed Systems **9**, 1041–1056 (1998)
26. Balsamo, S., Marzolla, M.: A simulation-based approach to software performance modeling. In: Inverardi, P. (ed.) Proc. Joint 9th European Software Engineering Conference (ESEC) & 11th SIGSOFT Symposium on the Foundations of Software Engineering (FSE-11), Helsinki, FI, September 1–5, 2003. ACM Press, New York (2003)
27. Balsamo, S., Marzolla, M.: Towards performance evaluation of mobile systems in UML. In: di Martino, B., Yang, L.T., Bobenau, C. (eds.) Proc. of the European Simulation and Modelling Conference (ESMc'03), EUROSIS-ETI, Naples, Italy, October 27–29, 2003, pp. 61–68 (2003)
28. Banks, J., Carson, J.S., Nelson, B.L., Nicol, D.M.: Discrete-Event System Simulation. Pearson Prentice Hall, Upper Saddle River (2004)
29. Beilner, H., Matter, J., Wysocki, C.: The hierarchical evaluation tool HIT. In: Proceedings of the International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Wien, 1994
30. Bernardi, S., Donatelli, S., 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 (2002)
31. Bernardo, M.: TwoTowers 2.0 user manual. http://www.sti.uniurb.it/bernardo/twotowers (2002)
32. Bernardo, M.: Twotowers 3.0: Enhancing usability. In: Proc. of the 11th IEEE/ACM Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, Orlando, FL, pp. 188–193 (2003)
33. Bernardo, M., Bravetti, M.: Performance measurement sensitive congruences for Markovian process algebras. Theoretical Computer Science **290**, 117–160 (2003)
34. Bertolino, A., 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, pp. 1–6 (2003)
35. Bertolino, A., Mirandola, R.: CB-SPE tool: Putting component-based performance engineering into practice. In: CBSE, pp. 233–248 (2004)
36. Bézivin, J.: On the unification power of models. Software and System Modeling **4**(2), 171–188 (2005)
37. Brinksma, E., Hermanns, H., Katoen, J.-P. (eds.): Lectures on Formal Methods and Performance Analysis, First EEF/Euro Summer School on Trends in Computer Science, The Netherlands, 2001. Lecture Notes in Computer Science, vol. 2090. Springer, Berlin (2001)
38. Caporuscio, M., Di Marco, A., Inverardi, P.: Model-based system reconfiguration for dynamic performance management. Journal of Systems and Software **80**(4), 455–473 (2007)
39. Chiola, G., Franceschinis, G., Gaeta, R., Ribaudo, M.: Greatspn 1.7: Graphical editor and analyzer for timed and stochastic petri nets. Perform. Eval. **24**(1–2), 47–68 (1995)
40. Coe, P.S., Howell, F.W., Ibbett, R.N., Williams, L.M.: Technical note: a hierarchical computer architecture design and simulation environment. ACM Transactions on Modelling and Computer Simulation **8**(4), 431–446 (1998)
41. Cortellessa, V., Di Marco, A., Inverardi, P.: Software performance model-driven architecture. In: Proceedings of ACM Symposium on Applied Computing (SAC), pp. 1218–1223 (2006)

42. Cortellessa, V., Di Marco, A., Inverardi, P.: Integrating performance and reliability analysis in a non-functional mda framework. In: Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering (FASE 2007), pp. 57–71 (2007)
43. Cortellessa, V., Di Marco, A., Inverardi, P.: Non-functional modeling and validation in model-driven architecture. In: Proc. of 6th Working IEEE/IFIP Conf. on Software Architecture (WICSA 2007), p. 25 (2007)
44. Cortellessa, V., Iazeolla, G., Mirandola, R.: Early generation of performance models for object-oriented systems. IEE Proceedings-Software **147**(3), 61–72 (2000)
45. Cortellessa, V., Mirandola, R.: PRIMA-UML: a performance validation incremental methodology on early UML diagrams. Science of Computer Programming **44**(1) (2002)
46. De Miguel, M., Lambolais, T., Hannouz, M., Betgé-Brezetz, S., Piekarec, S.: 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 (2000)
47. Di Marco, A.: Model-based performance analysis of software architectures. PhD thesis, Dipartimento di Informatica, Università degli Studi de L'Aquila, L'Aquila, Italy (2005)
48. Di Marco, A., Inverardi, P.: Compositional generation of software architecture performance qn models. In: Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 37–46 (2004)
49. Doob, J.L.: Stochastic Processes. John Wiley and Sons, New York (1953)
50. Ephraim, Y., Merhav, N.: Hidden Markov processes. IEEE Transactions on Information Theory **48**(6), 1518–1569 (2002)
51. Eriksson, H.E., Penker, M., Lyons, B., Fado, D.: UML 2 Toolkit. Wiley, New York (2004)
52. Franks, G., Hubbard, A., Majumdar, S., Petriu, D.C., Rolia, J., Woodside, C.M.: A toolset for performance engineering and software design of client-server systems. Performance Evaluation **24**(1–2), 117–135 (1995)
53. Franks, G., Maly, P., Woodside, M., Petriu, D.C., Hubbard, A.: Layered queueing network solver and simulator user manual. http://www.sce.carleton.ca/rads/lqns/LQNSUserMan.pdf (2005)
54. Franks, R., Woodside, C.M.: Performance of multi-level client-server systems with parallel service operations. In: ACM Proceedings of the First Workshop on Software and Performance (WOSP98), Santa Fe, New Mexico, pp. 120–130 (1998)
55. Booch, G., Rumbaugh, I.J.: The Unified Modeling Language User Guide. Addison-Wesley, Reading (1999)
56. Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of Software Engineering. Prentice Hall PTR, Upper Saddle River (2002).
57. Gilmore, S., Hillston, J.: The PEPA workbench: 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, vol. 794, pp. 353–368. Springer, Berlin (1994)
58. Gomaa, H.: Designing Concurrent, Distributed, and Real-Time Applications with UML. Addison-Wesley, Reading (2000)
59. Gómez-Martínez, E., Merseguer, J.: ArgoSPE: model-based software performance engineering. In: Donatelli, S., Thiagarajan, P.S. (eds.) Petri Nets and Other Models of Concurrency – 27th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency (ICATPN 2006), Turku, Finland, 2006. Lecture Notes in Computer Science, vol. 4024, pp. 401–410. Springer, Berlin (2006)
60. Grassi, V., 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 (2002)
61. Grassi, V., Mirandola, R., Randazzo, E., Sabetta, A.: KLAPER: an intermediate language for model-driven predictive analysis of performance and reliability. In: CoCoME – The Common Component Modeling Example: Comparing Software Component Models, Dagstuhl Research Seminar for CoCoME, August 1–3, 2007. Lecture Notes in Computer Science, vol. 5153, pp. 327–356. Springer, Berlin (2008)

62. Gu, G., 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 (2002)
63. Gu, G.P., Petriu, D.C.: From UML to LQN by XML algebra-based model transformations. In: Proceedings of the Fifth ACM Workshop on Software and Performance (WOSP 2005), pp. 99–110 (2005)
64. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming **8**(3), 231–274 (1987)
65. Harreld, H.: NASA delays satellite launch after finding bugs in software program. In: Federal Computer Week, 1998
66. Harrison, P.G., Hillston, J.: Exploiting quasi-reversible structures in Markovian process algebra models. Computer Journal **38**(7), 510–520 (1995)
67. Hermanns, H., Herzog, U., Katoen, J.P.: Process algebra for performance evaluation. Theoretical Computer Science **274**(1–2), 43–87 (2002)
68. Herzog, U., Klehmet, U., Mertsiotakis, V., Siegle, M.: Compositional performance modelling with the `TIPPtool`. Performance Evaluation **39**(1–4), 5–35 (2000)
69. Hillston, J.: PEPA – performance enhanced process algebra. Technical report, Dept. of Computer Science, University of Edinburgh (1993)
70. Hillston, J., Pooley, R.: Stochastic process algebras and their application to performance modelling. In: Proc. of TOOLS'98 Tutorials, 1998
71. Hillston, J., Thomas, N.: Product form solution for a class of PEPA models. Performance Evaluation **35**(3), 171–192 (1999)
72. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall International, London (1985)
73. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computations. Addison-Wesley, Reading (1979)
74. Kant, K.: Introduction to Computer System Performance Evaluation. McGraw-Hill, New York (1992)
75. Kemeny, J.G., Snell, J.L.: Finite Markov Chains. Springer, New York (1976)
76. Kleinrock, L.: Queueing Systems, vol. 1: Theory. Wiley, New York (1975)
77. Kouvatsos, D.D., Balsamo, S.: Queueing networks with blocking. Perform. Eval. **51**(2/4), 79–81 (2003)
78. Larman, C.: Agile and Iterative Development: A Manager's Guide. Addison-Wesley, Reading (2004)
79. Lazowska, E.D., Kahorjan, J., Graham, G.S., Sevcik, K.C.: Quantitative System Performance: Computer System Analysis Using Queueing Network Models. Prentice-Hall, Englewood Cliffs (1984)
80. Liu, X., Shenoy, P.J., Corner, M.D.: Chameleon: Application-level power management. IEEE Trans. Mob. Comput. **7**(8), 995–1010 (2008)
81. López-Grao, J.P., Merseguer, J., 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, pp. 25–36. ACM Press, New York (2004). doi:10.1145/974044.974048
82. 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 (February 2004)
83. Miller, J. (ed.): Model-Driven Architecture Guide. omg/2003-06-01 (2003)
84. Milner, R.: Communication and Concurrency. International Series on Computer Science. Prentice-Hall International, Englewood Cliffs (1989).
85. Object Management Group: UML profile, for schedulability, performance, and time. OMG document ptc/2002-03-02. http://www.omg.org/cgi-bin/doc?ptc/2002-03-02
86. Object Management Group: Unified modeling language (UML), version 1.4, OMG documentation. http://www.omg.org/techonology/documents/formal/uml.htm
87. Object Management Group: Unified modeling language: superstructure – version 2.1.1, formal/2007-02-05. http://www.omg.org/docs/formal/07-11-04.pdf (2007)

88. Object Management Group: UML profile for MARTE, ptc/08-06-09. http://www.omgmarte.org/Documents/Specifications/08-06-09.pdf (2008)
89. Object Management Group (OMG): XML metadata interchange (XMI) specification, version 2.0, 2002
90. Perez-Palacin, D., Merseguer, J., Bernardi, S.: Performance aware open-world software in a 3-layer architecture. In: Adamson, A., Bondi, A.B., Juiz, C., Squillante, M.S. (eds.) Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering, San Jose, California, USA, January 28–30, 2010, pp. 49–56. ACM, New York (2010)
91. Petriu, D.B., Woodside, M.: A metamodel for generating performance models from UML designs. In: Proceedings of UML Conference. Lecture Notes in Computer Science, vol. 3273
92. Petriu, D.C.: Approximate mean value analysis of client-server systems with multi-class requests. In: Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Performance Evaluation Review, 1994
93. Petriu, D.C., 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. Lecture Notes in Computer Science, vol. 2324, pp. 159–177, (2002)
94. Petriu, D.C., Wang, X.: From UML descriptions of high-level software architectures to LQN performance models. In: Verlag, S. (ed.) Proceedings of AGTIVE'99, pp. 47–62 (1999)
95. Petriu, D.C., Wang, X.: Deriving software performance models from architectural patterns by graph transformations. In: TAGT'98: Selected Papers from the 6th International Workshop on Theory and Application of Graph Transformations, London, UK, 2000, pp. 475–488. Springer, Berlin (2000)
96. Petriu, D.C., Woodside, C.M.: 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. Lecture Notes in Computer Science, vol. 2324, pp. 141–158 (2002)
97. Pooley, R.J.: An Introduction to Programming in SIMULA. Blackwell Scientific Publications, Oxford (1987)
98. Reisig, W.: Petri Nets: An Introduction. EATCS Monographs on Theoretical Computer Science, vol. 4 (1985)
99. Rolia, J.A., Sevcik, K.C.: The method of layers. IEEE Transaction on Software Engineering **21**(8), 622–688 (1995)
100. Sauer, C.H., MacNair, E.A.: Queueing network software for systems modeling. In: Research Report RC-7143, IBM Thomas J. Watson Research Center, Yorktown Heights (1978)
101. Sauer, C.H., Reiser, M., MacNair, E.A.: RESQ – a package for solution of generalized queueing networks. In: Proceedings, National Computer Conference, Dallas, TX, 1977, pp. 977–986 (1977)
102. Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. IEEE Computer **39**(2), 25–31 (2006)
103. SEAlab – Software Quality Group, Dipartimento di Informatica, Università dell'Aquila: MOSQUITO 1.6 User Manual (2008).
104. Sector, I.-T.S.: Message sequence charts. ITU-T Recommendation Z. **120**(11/99) (1999)
105. Sereno, M.: Towards a product form solution for stochastic process algebras. Computer Journal **38**, 622–632 (1995)
106. Smith, C.U.: Performance Engineering of Software Systems. Addison-Wesley, Reading (1990)
107. Smith, C.U., 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)
108. Smith, C.U., Williams, L.G.: Performance engineering evaluation of object-oriented systems with SPE•ED^{TM}. In: Proceedings of Computer Performance Evaluation, Berlin, Germany, 1997. Lecture Notes in Computer Science, vol. 1245, pp. 135–154 (1997)

109. Smith, C.U., Williams, L.G.: A performance model interchange format. Journal of Systems and Software **49**(1) (1999)
110. Smith, C.U., Williams, L.G.: Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Addison-Wesley, Reading (2002)
111. Smith, C.U., Williams, L.G.: More new software performance antipatterns: Even more ways to shoot yourself in the foot. In: Computer Measurement Group Conference, 2003
112. Sommerville, I.: Software Engineering, 7th edn. Addison-Wesley, Reading (2004)
113. Tijms, H.C.: Stochastic Models, An Algorithmic Approach. John Wiley and Sons Ltd, New York (1994)
114. Trivedi, K.S.: Probability and Statistics with Reliability, Queueing and Computer Science Applications. John Wiley and Sons, New York (2001)
115. Varki, E., Dowdy, L.W.: Analysis of balanced fork-join queueing networks. In: Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (1996)
116. Veran, M., Potier, D.: QNAP 2: a portable environment for queueing systems modelling. In: Rapport de recherche de l'INRIA-Rocquencourt. http://www.inria.fr/rrrt/rr-0314.html (1984)
117. W3C: eXtensible Markup Language (XML) 1.0, 2nd edn. W3C recommendation 6 October 2000. http://www.w3.org/TR/2000/REC-xml-20001006
118. 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/, http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/
119. Williams, L.G., Smith, C.U.: Performance evaluation of software architectures. In: ACM Proceedings of the First Workshop on Software and Performance, pp. 164–177
120. Williams, L.G., Smith, C.U.: Information requirements for software performance engineering. In: Proceedings of International Conference on Modeling Techniques and Tools for Computer Performance Evaluation, Heidelberg, Germany, 1995. Springer, Berlin (1995)
121. Williams, L.G., Smith, C.U.: `PASA`: A method for the performance assessment of software architectures. In: Proceedings of the Third International Workshop on Software and Performance (WOSP02), Rome, Italy, July 2002, pp. 179–189 (2002)
122. Woodside, C.M., Hrischuk, C., Selic, B., Brayarov, S.: Automated performance modeling of software generated by a design environment. Performance Evaluation **45**, 107–123 (2001)
123. Woodside, C.M., Neilson, J., Petriu, S., Mjumdar, S.: The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. IEEE Transaction on Computer **44**, 20–34 (1995)

Index

V. Cortellessa et al., *Model-Based Software Performance Analysis*,
DOI 10.1007/978-3-642-13621-4, © Springer-Verlag Berlin Heidelberg 2011