Università Ca' Foscari di Venezia

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis: TD-2004-1

Simulation-Based Performance Modeling of UML Software Architectures

Moreno Marzolla

Supervisor
Prof. Simonetta Balsamo

February, 2004

Author's Web Page: http://www.dsi.unive.it/~marzolla

Author's e-mail: marzolla@dsi.unive.it

Author's address:

Dipartimento di Informatica
Università Ca' Foscari di Venezia
Via Torino, 155
30172 Venezia Mestre – Italia
tel. +39 041 2348411
fax. +39 041 2348419
web: http://www.dsi.unive.it

Abstract

Quantitative analysis of software systems is being recognized as an important issue in the software development process. Performance analysis can help to address quantitative system analysis from the early stages of the software development life cycle, e.g, to compare design alternatives or to identify system bottlenecks. Early identification of performance problems is desirable as the cost of design change increases with the later phases of the software development cycle.

This thesis addresses the problem of performance analysis of software systems described at a high level of detail. We adopt a model-based approach: starting from a software model, we derive a performance model which is then evaluated. This kind of approach has the advantage of being applicable since the early software development phases; in contrast, a measurement-based approach consisting on identifying problems by direct measurements on a running system can not.

We consider software descriptions as a set of annotated Unified Modeling Language (UML) diagrams. UML is a widely used notation for describing and specifying software artifacts, and recently it is being also considered for performance evaluation purposes. We define the performance model as a process-oriented simulation model. Simulation is a powerful modeling technique which can represent general and unconstrained system models, so that the software model can be more accurately represented. An algorithm for translating UML software specifications into simulation models is described. The proposed technique defines a set of annotation of UML specifications to add quantitative, performance-oriented informations. The profile is based on the UML profile for Schedulability, Performance and Time specification. The system is described in term of Use Case, Activity and Deployment diagrams. Use Case diagrams correspond to workloads applied to the system. Activity diagrams provide a high-level description of the computation steps performed by the system, and Deployment diagrams describe the physical resources on which the computations take place. A process-oriented simulation model can then be automatically derived from the annotated specification. Execution of the simulation program provides performance results that can be directly interpreted at the UML software specification level. The described algorithm has been implemented in a prototype tool called UML-Ψ (UML Performance SImulator), which is demonstrated on a case study to show the validity of the approach. The UML-Ψ tool is written in C++ and is based on a general-purpose process-oriented simulation library. It

parses annotated UML models, automatically builds the corresponding simulation model and executes it. Performance results are inserted into the original UML model as tagged values, in order to give feedback to the user.

Sommario

L'analisi quantitativa di sistemi software viene riconosciuta come un aspetto importante nel processo di sviluppo del software. L'analisi quantitativa può aiutare a valutarne le prestazioni a partire dai primi passi del ciclo di sviluppo, ad esempio consentendo di confrontare differenti alternative o individuare colli di bottiglia nel sistema. L'individuazione precoce di problemi legati a scarse prestazioni è desiderabile, poiché il costo di cambiare il disegno del software cresce col progredire delle fasi di sviluppo.

Questa tesi affronta il problema dell'analisi delle prestazioni di sistemi software descritti ad alto livello. Viene adottato un approccio basato su modelli: a partire da un modello del sistema software, viene descritto un modello di prestazione che è poi valutato. Un approccio di questo tipo ha il vantaggio di poter essere applicato fin dalle prime fasi del ciclo di sviluppo del software; al contrario, non possono essere applicati durante le fasi iniziali gli approcci basati sulle misurazioni, perché consistono nell'identificare problemi tramite misura diretta su un sistema in esecuzione.

Vengono considerate descrizioni di sistemi software espresse come un insieme di diagrammi UML annotati. UML è una notazione ampiamente utilizzata per la descrizione e la specifica di artefatti software, e recentemente è stata considerata anche nella valutazione di prestazioni. Definiremo un modello di prestazioni in termini di un modello di simulazione orientato a processi. La simulazione è una tecnica molto espressiva con la quale è possibile rappresentare modelli generali e senza vincoli, tali da poter rappresentare più accuratamente il modello del software. Verrà descritto un algoritmo per la traduzione di specifiche UML annotate nel modello di simulazione. La tecnica proposta definisce un insieme di annotazioni di specifiche UML per aggiungere informazioni quantitative orientate alle prestazioni. Il profilo è basato sul " UML profile for Schedulability, Performance and Time Specification". Il sistema viene descritto in termini di diagrammi di caso d'uso, di attività e di deployment. I diagrammi di caso d'uso corrispondono ai carichi di lavoro applicati al sistema. I diagrammi di attività forniscono una descrizione ad alto livello dei passi di computazione effettuati dal sistema, e i diagrammi di deployment descrivono le risorse fisiche su cui le computazioni avvengono. A partire dalle specifiche annotate viene derivato un modello di simulazione orientato a processi. L'esecuzione del programma di simulazione fornisce risultati di prestazioni che possono essere direttamente interpretati a livello di specifica UML. L'algoritmo proposto è stato implementato in un

prototipo chiamato UML-Ψ(UML Performance SImulator), che verrà applicato ad un caso di studio per mostrare la validità dell'approccio. UML-Ψ è scritto in C++ ed è basato su una libreria di simulazione generica orientata a processi. UML-Ψ legge modelli UML annotati, costruisce automaticamente il corrispondente modello di simulazione e lo esegue. Le misure di prestazione sono inserite nel modello UML originale come tagged values, allo scopo di fornire un riscontro all'utente.

Acknowledgments

I would like to thank my advisor, Simonetta Balsamo, for her continuous support and supervision in developing this research and the related publications.

This work has been carried out at the Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italy. I would like to thank all the personnel for their support and for making the department a nice and friendly place for doing research. In particular, special thanks to my friends and colleagues: Chiara, Claudio, Damiano, Fabrizio, Marta, Matteo, Ombretta, Silvia (for many rotationally invariant ideas) and Valentino. Many thanks to all the people I met at schools and conferences, including Andrea (the ghost of unemployment is already knocking my door), Mimmo and Claudio. All my gratitude and friendship goes to Paolo who shared with me this three-years long adventure, and who hosted me many times in Florence and Pisa; I wish you a good luck for all your future activities.

During my PhD I was initially supported by INFN (Istituto Nazionale di Fisica Nucleare) Padova, Italy. Many thanks to my supervisors Roberto and Mauro, and the many friends there: Alvise, Andrea L., Andrea P., Enrico, Federico, Fulvio, Marcello and Monica.

The work described in this thesis has been partially supported by research funds provided by MURST Research Project *Sahara* and by MIUR Project FIRB "Performance Evaluation of Complex Systems: Techniques, Methodologies and Tools".

Finally, many thanks to my family for their enduring and constant support and encouragement: Gabriella, Gianni, Osvaldo, and also nonna Rina.

Contents

List of Figures

List of Tables

Acronyms

ADL Architectural Description Language

AQN Augmented Queuing Network

EQN Extended Queuing Network

FEL Future Event List

GSPN Generalized Stochastic Petri Nets

LQN Layered Queuing Network

NICE Naval Integrated Communication Environment

OMG Object Management Group

OS Operating System

QN Queuing Network

SA Software Architecture

SPA Stochastic Process Algebra

SPE Software Performance Engineering

SQS Sequencing Set

TVL Tag Value Language

UML Unified Modeling Language

XMI XML Metadata Interchange

XML Extensible Markup Language

XSLT Extensible Stylesheet Language Transformation

I

Setting the Context

1
Introduction

Developing a complex software system is a challenging task, requiring resources in term of time and manpower to be accomplished. It is therefore very important that the system, once built, satisfies its functional and non functional requirements.

In recent years it has been recognized that the software development processes should be supported by a suitable mechanism for early assessment of software performance. Early identification of unsatisfactory performance of Software Architecture (SA) can greatly reduce the cost of design change [91, 92]. The reason is that correcting a design flaw is more expensive the later the change is applied during the software development process. This is particularly true if a waterfall-style model [93] of software development is employed, as any change requires the development process to start back from the beginning. However, this is still a relevant issue whenever a different software development process is used.

Both quantitative and qualitative analysis can be performed at the software architectural design level. Qualitative analysis deals with functional properties of the software system such as deadlock-freedom or security. Qualitative analysis is carried out by measurement or by modeling the software system to derive quantitative figures of merit, such as, for example, the execution profile of the software, memory utilization or network utilization.

It should be noted that software performance modeling is challenging for different reasons. First, it is difficult to derive meaningful performance measures from a static analysis of the code. The reason is that software performances heavily depend on the hardware platform on which the software executes, and also on the usage pattern the software is subject to. Moreover, software performance modeling can not be performed on one component at a time, as critical issues may arise only when different components interact.

We focus on models of software systems at the SA level. Many performance models have been proposed in the literature; these models include Queuing Network [57] and their various extensions (as Layered Queuing Network, which have been specifically developed to model client-server communication patterns in distributed systems [38, 86, 103]), Stochastic Petri Nets [69] and Stochastic Process Algebra [50]. At the moment there is no clear consensus on which model should be preferred in practice. The general understanding is that different models are suit-

able for different architectural styles and/or application domains. It is interesting to observe that most of the software performance evaluation approaches proposed so far are based on analytical models. This is motivated by the fact that these models are well studied and understood. Also, these models can sometimes be solved analytically, providing performance results which are both exact and optionally can be expressed parametrically with respect to one or more variables. This, however, comes at a price.

Analytical models which can be solved exactly can be derived only by imposing some limitations on the SA from which they are derived. For example, Queuing Network (QN) models have a so-called product-form solution, which can be efficiently computed, only for special classes of networks. These produce-form models are constrained both on their topology and on the distributions of jobs interarrival and service times, which are often assumed to be exponentially distributed [28]. Hence, the hypothetical software system from which they are derived is constrained to being modeled only under the exponential distributions assumption. Even when solvable analytical models can be derived, their size can grow exponentially with the size of the corresponding SA, making the performances model impossible to handle [7].

Analytical performance models can be structurally very different from the SA from which they are derived. This makes it very difficult to report performance results from the performance model to the original SA. This is very limiting, as software performance evaluation is supposed to provide the modeler with an immediate feedback about possible performance problems.

There exist partial solutions to these problems. Approximate numerical techniques are available in many cases where product-form solutions do not exist [28]. Unfortunately, those techniques themselves are subject to constraints in order to be applied. They do not provide closed form solutions, which would be very useful in order to study figures of merit as a function of some unknown parameter. Finally, many approximate techniques do not provide any error estimation on their results, meaning that it is impossible to quantify whether the computed values are good approximations of the exact results or not.

We address these limitations by developing a simulation model of SA. Simulation is a powerful modeling technique that allows general system models, that is simulation models can represent arbitrarily complex real-world situations, which can be too complex or even impossible to represent by analytical models. Examples of complex systems are computer systems with asynchronous communications, fork and join and simultaneous resource possessions, for which analytical models often become difficult to analyze. We propose an approach for software performance analysis based on simulation to take advantage of this modeling technique.

Once the performance model has been chosen, one more problem remains: it is necessary to choose a notation to use for describing software systems. Existing approaches can be divided in two categories: the first deal with creating special-purpose notations for expressing both architectural features and performance informations; the second deals with building performance models from already existing design

formalisms.

Examples in the first category (special-purpose notations for both architectural and performance-oriented models) include the SPE approach by Smith, and Smith and Williams [91, 92, 102]. SPE is a comprehensive approach integrating performance analysis into the software development process. The software execution model is represented by Execution Graphs, while the system model is based on QN and represents the physical resources on which the software executes.

Examples in the second category (performance models derived by software specification notations already in use) include SDL [21] and LOTOS [29]. SDL is used for the specification of real-time systems; LOTOS is the CCITT recommended protocol specification language. Both have been used for deriving various kinds of performance models (see [79] for references).

The problem with all those approaches is that both domain-specific and completely new notations are unlikely to gain wide acceptance among the software engineering community. In recent years, the Unified Modeling Language [74, 87, 88] is emerging as a *de facto* standard for the high-level specification of software systems and business processes, with particular emphasis of Object-oriented features. UML is based on a graphical notation for specifying software artifacts; the notation is quite rich, including different kind of diagrams which can be used to model different point of views of the system. An introduction to UML will be given in Chapter 2.

It comes at no surprise that the software performance community is looking with great interest toward using UML as the preferred software specification language. Unfortunately, this choice has some drawbacks. The main problem is that UML is not a formal language. It lacks any formal semantics, and is only intuitively specified [85]; this makes reasoning on UML specifications troublesome. Efforts are ongoing to provide a precise specification of at least some subset of UML [81].

However, the fact that UML is informally specified does not mean that it has no semantics at all. It is still possible to reason on UML specifications by taking into account their intuitive (and widely agreed on) meaning. We propose a mapping between annotated UML specifications into process-oriented simulation models. The mapping will be described in detail in Part II, and is consistent with the intuitive semantics of the UML diagrams considered. Moreover, the UML profile we define in Chapter 5 enforces an additional, informally specified semantics of UML use case, activity and deployment diagrams corresponding to that of the performance model generated from them. This is exactly what a UML profile is supposed to be used for [74]. We are then considering UML specifications representing a dynamic behavior described more precisely by their corresponding simulation models.

We consider quantitative evaluation of the performances of SA at the design level based on simulation models. We use SA specifications expressed in terms of UML [74] diagrams. We propose to annotate the UML diagrams using a subset of annotations defined in the *UML Profile for Schedulability, Performance and Time Specification* [75] (hereafter referred as *UML performance profile*). We define a simulation model of an UML software architecture specification introducing an almost

one-to-one correspondence between behaviors expressed in the UML model and the entities or processes in the simulation model.

The advantage of the proposed approach is twofold. First, this correspondence between the system and the model helps the feedback process to report performance results obtained by simulation back into the original SA. Second, this allow to define a simple translation algorithm from the SA to the simulation performance model that can be fully automated.

1.1 Software Performance Evaluation

Performance analysis of software systems can be carried out by measurement or by modeling techniques. Direct measurement of an actual implementation provides an accurate assessment of the performance of a software system. This is relatively easy to do, but requires to build a system implementation before the measurement can take place. Implementing a complex system is usually a time-consuming, error-prone and expensive task; mastering this complexity is the goal of all the software development processes which have been proposed in the literature. We focus on software system at the SA design level. When SA exhibits performance-related problems, it is unlikely that such problems will be fixed once the architecture has been deployed, given the high costs associated with changing the design. Hence it is useful or even necessary to evaluate early performance measures at the architectural design level, by developing and evaluating a model of SA. Performance model evaluation can is obtained by analytical, simulative or hybrid techniques.

Most of the research in the area of Software Performance Engineering (SPE) is based on developing analytical models of SA [9, 15]. However, such models can usually be built only by imposing some structural restrictions on the original system model, depending on the specific modeling formalism which has been chosen; the reason is that analytical models have often a limited expressiveness. While it is sometimes possible to simplify the model of the system in order to make it analytically tractable, there are many cases in which the significant aspects of the system can not be effectively represented into the performance model. Examples are concurrency, simultaneous resource possession and fork and join systems.

For these reasons we propose a software performance evaluation approach based on simulation models. Simulation models can be arbitrarily detailed, in that, informally, they impose no restrictions on what they can model. The analyst has the maximum degree of freedom in selecting the aspects of the real system, that is the SA in our context, to model, and at which level of detail. This freedom comes at some cost: the drawback of simulation is that very complex models may require a lot of time and computational resources in order to be executed. The results also require sophisticated statistical techniques in order to be correctly understood [55]. While it is true that any given system can be represented at an arbitrarily high level of detail by a simulation model, the analyst often ignores the exact inner working of

the system being simulated. This is certainly the case with SA, since they are defined at a high level of abstraction, and many details are postponed until the implementation phase. However, while the software architect may ignore the inner details of the system being designed, he could have some more or less detailed knowledge of part of the architecture (for example if some pieces are taken from an existing, already implemented system). Whenever additional informations are available, they should be used to obtain better and more realistic performance measures.

In order to easily apply SPE techniques it is convenient to refer to a common notation for software specification. Many notations are design-oriented and do not provide built-in facilities to express informations necessary to derive a performance model. Two main approaches have been employed to overcome this limitation. The first is the introduction of performance-oriented formalisms applied as informations specifically added to software models [91]. The second is the extension of modeling languages with additional notations [22, 52, 67]. While both approaches produced encouraging results, they are not widespread since they require software engineers to learn new and non-standard modeling formalisms and notations.

After its adoption in 1997 as an official Object Management Group (OMG) standard, UML gained wide acceptance among software engineers thanks to its flexibility and ease of use. UML is a language for specifying, visualizing, constructing, and documenting software and non-software systems. It is particularly useful for, although not limited to, designing Object-Oriented systems. UML provides users with a visual modeling notation to develop and exchange models, and it defines extensibility and specialization mechanisms. It supports specifications which are independent from the particular programming language and development process. Moreover, it supports higher-level development concepts such as components, collaborations, frameworks and patterns.

Given its wide acceptance, many software engineers are already acquainted with at least the basics of the UML notation. For this reason several recent SPE approaches consider UML as a starting notation on which existing and new performance evaluation techniques can be applied. This can be done because UML provides extension mechanisms to add new concepts and notations for those situations which are not covered by the core language, and to specialize the concepts, notation, and constraints for particular application domains.

We consider the performance-oriented modeling process illustrated in Fig. 1.1. The starting point is a description of the SA. We consider a description as a set of UML diagrams annotated with quantitative informations in order to derive a simulation-based performance model. The model is obtained using a suitable *Modeling Algorithm* and then implemented in a simulation program, which is eventually executed. Simulation results are a set of performance measures that can be used to provide a feedback at the original SA design level. The feedback should pinpoint performance problems on the SA, and possibly provide suggestions to the software designer about how the problem can be solved. The modeling cycle shown in Fig. 1.1 can be iterated until a SA with satisfactory performance is developed.

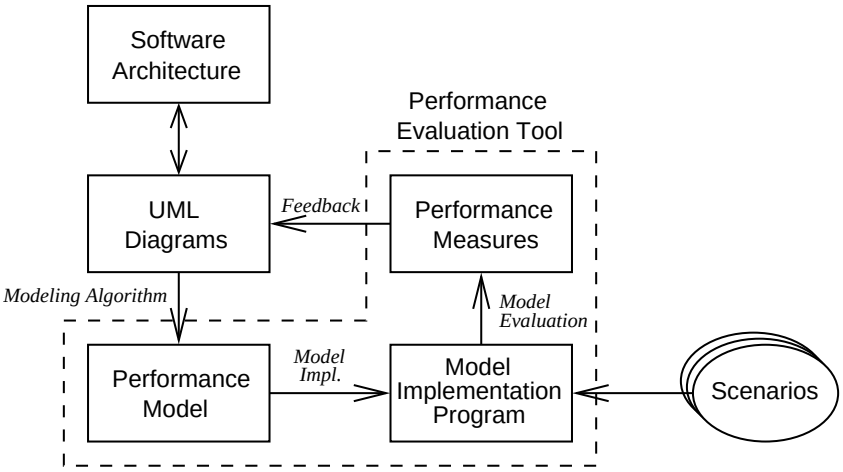


Figure 1.1: Phases of the modeling and performance evaluation process

1.2 Previous work

Several approaches have been proposed on deriving performance models from architectural specifications. Since recent approaches (including our proposal) use UML as the notation for describing SA, we now briefly review some relevant works dealing with performance evaluation of UML SA. As we will see shortly, most of them propose the derivation of analytical performance models, while simulation models have been only recently considered in this field. Detailed considerations about analytical performance models for SA are outside the scope of this work. The interested reader is referred to [9, 15] for additional references and a thorough discussion covering a wider spectrum of approaches. More informations can also be found in [53].

Approaches deriving analytic models

King and Pooley [56] propose a method for deriving performance models based on Generalized Stochastic Petri Nets (GSPN) [69] from UML collaboration and statechart diagrams. They propose to use a combination of UML diagrams – state diagrams embedded into collaboration diagrams – to better express the global state of a system.

In [79], Pooley and King describe how the various kinds of UML diagrams can be used for performance evaluation purposes. Namely, they identify Actors in Use Case diagrams with workloads applied to the system. Implementation diagrams (Deployment diagrams) are mapped on a queuing network model, representing the computational resources (service centers in the QN) and communication links (queues in the QN). Sequence diagrams can be used as traces to drive a simulation program. Finally, they suggest modeling of UML State diagrams using Markovian models. To demonstrate their approach, a queuing network model is derived from a UML

description of an ATM system. The approach adds textual notations to UML diagrams to include useful information for performance evaluation (e.g., time labels in sequence diagrams). Such annotations are used to produce more complete models of software systems. It should be noted that their approach, in general, can be applied regardless of the particular performance model derived.

In [23], Bernardi et al. derive a GSPN model from UML state and sequence diagrams. They define two levels of modeling: a class level and an instance level. The class level is represented by state diagrams, and is used to describe the behavior of single entities of a system. The instance level uses sequence diagrams to show patterns of interaction among objects (instances of classes). The GSPN model is then created merging together the information provided by the two kinds of diagrams.

Cortellessa and Mirandola [33] present an approach for translating UML sequence diagrams, use case diagrams and deployment diagrams into performance model based on Extended Queuing Network (EQN) [59], using an intermediate transformation into *Execution Graphs* [91]. System performance evaluation is presented as an incremental process integrated in the software development life cycle, by using information of different kinds of UML diagrams from the early stages of the development process. The level of detail of the model is extended as the software development proceeds. This allows incremental building of a performance model of the system, which can be used to improve or modify the SA. The UML diagrams are annotated with quantitative informations, which are necessary to set the parameters of the model. Actors are annotated with the frequency they may appear in the system. Associations between actors and use cases are annotated with the probabilities that each actor executes each use case. Sequence diagrams are annotated with timing informations attached to events, and messages sent among objects are tagged with their sizes. Deployment diagrams represent various kinds of resources, and are annotated with suitable parameters such as bandwidth for network links, or speed of computational resources. Finally the three types of UML diagrams (use case, sequence and deployment diagrams) are used together to build a EQN model.

In [43], Gomaa and Menascé use UML diagrams to represent the interconnection pattern of a distributed software architecture. class diagrams are used to illustrate the static view of a system, while collaboration diagrams show the dynamic behavior. collaboration diagrams are extended with new elements showing interconnections and communication ports, and are added with performance annotations written in Extensible Markup Language (XML). Such annotations refer to routing probability between objects, average message processing time, average message size and average arrival rate of requests. Then they derive a performance model based on QN.

Gu and Petriu [45] and Petriu and Shen [77] derive performance models based on Layered Queuing Network (LQN) models [86] from a description of SA. They use UML activity diagrams, annotated as defined in the UML Performance Profile. Diagrams and annotations are saved in XML files in the XML Metadata Interchange

(XMI) format [76] and then translated in LQN models through Extensible Stylesheet Language Transformation (XSLT) [104].

Lindemann et al. [63] develop an algorithm for deriving performance models based on *Generalized Semi-Markov Processes* from UML state and activity diagrams. These diagrams are annotated with exponentially distributed or deterministic delays applied to events, and timed events trigger a state transition. Annotations are based on an extension the UML Performance Profile.

Kähkipuro [54] proposes a framework on UML notation for describing performance models of component-based distributed systems. The performance model is based on Augmented Queuing Network (AQN). The approach works as follows. UML diagrams are first converted into a textual notation called Performance Modeling Language (PML). Then the PML performance model is translated into an AQN, which is then solved with approximate techniques. The results obtained from the AQN model are subsequently propagated to the PML model and finally to the software architectural model. The entire SA performance evaluation cycle is represented in Fig. 1.2.

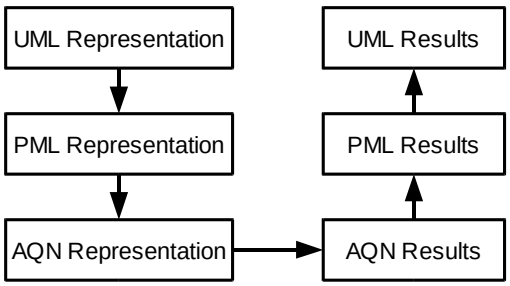


Figure 1.2: The modeling process proposed by Kähkipuro [54]

The PERformance Modelling for Atm Based Applications and SErvices (PERMABASE) project [98] deals with the automatic generation of performance models from software systems specified in UML. The software system is described in term of the following specifications:

- Workload specification: a description of the workloads driving the system;
- Application specification: a description of the behavior of the software system;
- Execution environment specification: this includes the description of physical environment, including processors, network links and other resources;
- System scenario specification: the specification of a particular system instance, describing the configuration of the system (which components are presents and how they are connected).

UML is used to specify the workloads, the application behavior and the execution environment. The descriptions above are translated into a Composite Model Data Structure (CMDS) which can be checked for inconsistencies and refined if necessary. The CMDS is then translated into a performance model, whose execution provides feedback which is reported back into the CMDS. The cycle is represented in Fig. 1.3.

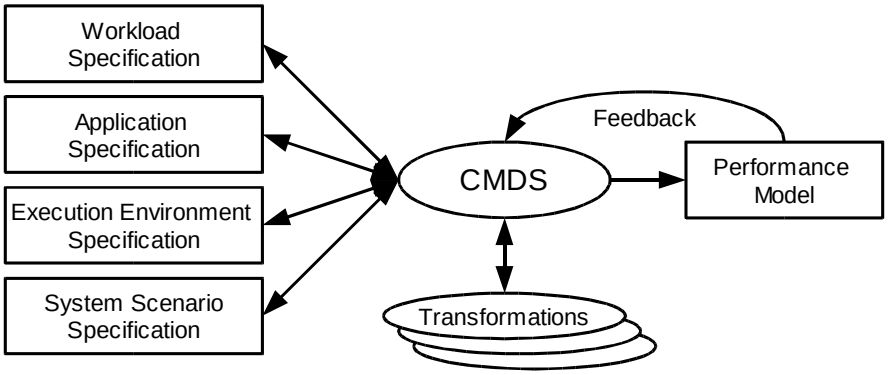


Figure 1.3: The PERMABASE modeling and performance evaluation process [98]

Hoeben [51] describes how UML diagrams can be used for performance evaluation. The approach uses UML class and component diagrams to represent informations used for modeling the system dynamics. Sequence and collaboration diagrams are used to model the behavior of the system, and deployment diagrams represent processors and network connections in the system. The performance model derived from the combination of the diagrams is based on QN.

Approaches deriving simulation models

There currently are only a few works which derive simulation-based performance models from software systems. They are briefly described in this section.

Arief and Speirs [3, 4, 5, 6] develop an automatic tool for deriving simulation models from UML class and sequence diagrams. Their approach (see Fig. 1.4) consists on transforming the UML diagrams into a simulation model described as an XML document. The XML notation used to describe the simulation model has been called SimML (Simulation Modeling Language). This model is then translated into a simulation program, which can be executed and provides performance results. What makes this approach particularly interesting is that the simulation model is decoupled from its implementation. This makes it possible to implement the simulation model using different languages. The authors developed two different back-ends for translating the SimML model into simulation programs written in C++Sim [66] and JavaSim [65].

De Miguel et al. [36] introduce UML extensions for the representation and automatic evaluation of temporal requirements and resource usage, particularly targeted

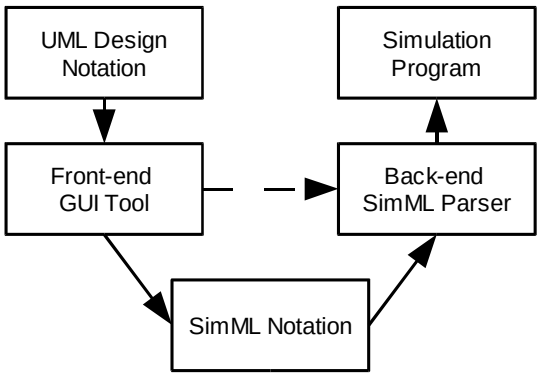


Figure 1.4: The modeling process proposed by Arief and Speirs [6]

at real-time systems. Their proposed performance evaluation sequence is depicted in Fig. 1.5.

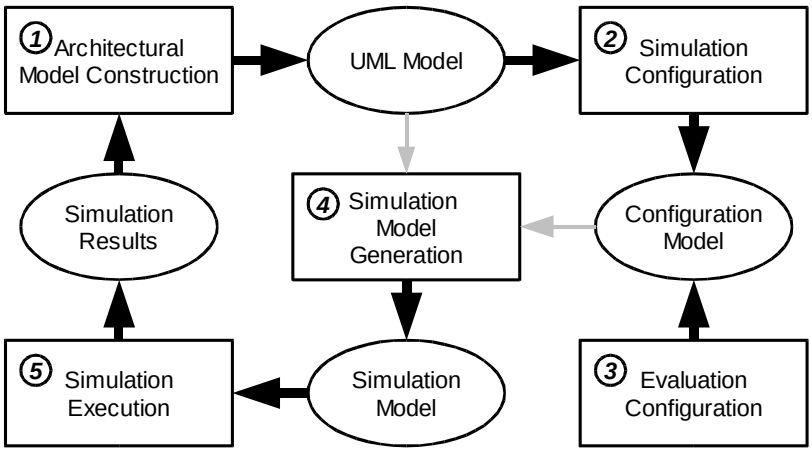


Figure 1.5: The modeling process proposed by De Miguel et al. [36]

Step 1 deals with the construction of the architectural model of the system to be analyzed; this is done by a UML CASE tool. UML elements are annotated with stereotypes, tagged values and stereotyped constraints in order to provide parameters to the simulation step. An XMI representation of the UML model is exported and used in the next phases. Step 2 deals with the configuration of the simulation; given that the UML model might contain different alternative scenarios and alternative behavioral specifications for the same elements, during this phase it is possible to choose which scenarios and which behaviors to execute. Step 3 deals with the configuration of the simulation parameters, such as the statistics to collect, which filters to apply to simulation results, and the length of the simulation period. Step 4 is the generation of the OPNET simulation model, which is executed during step 5. Simulation results are finally displayed using OPNET facilities, and may pro-

vide hints to reconfigure the UML architecture or used as criteria for architecture selection.

Hennig et al. [47, 49] describe a UML-based simulation framework for early performance assessment of software/hardware systems described as UML Sequence and Deployment diagrams. The framework follows the design-evaluation-feedback cycle depicted in Fig. 1.6.

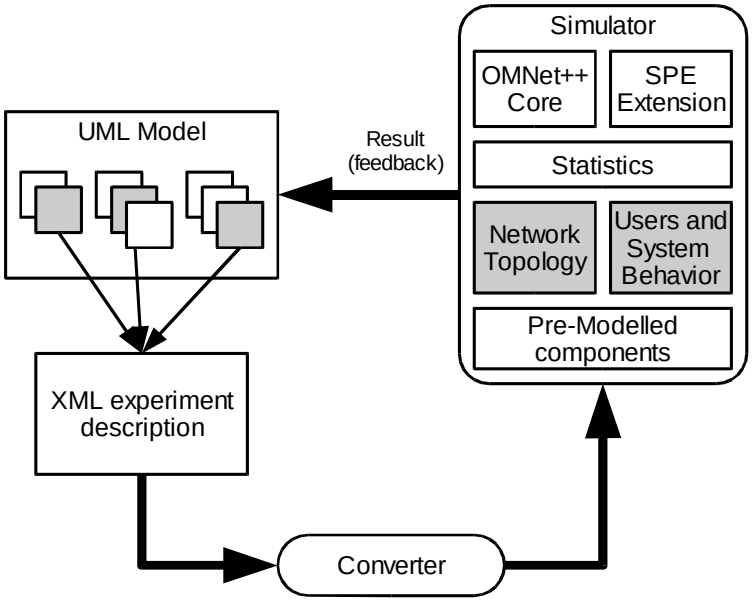


Figure 1.6: The performance simulation cycle proposed by Hennig et al. [49]

The simulation cycle starts from a collection of UML diagrams, from which a subset (the gray boxes in the picture) is extracted and compiled into an XML document describing the simulation experiment. The UML diagrams which are extracted are Deployment diagrams, used to describe the physical environment on which the software system executes, and Collaboration diagrams used to model the workload applied to the system and the internal behavior of the application being modeled. The converter module generates code for the network and behavior components of the simulator. The simulator itself is based on the discrete event simulation package OMNet++ [96]. It contains core modules and specific SPE extensions (scheduler, workflow execution engine) as well as pre-modelled components. The statistics of performance observations collected during the execution of the simulator can be fed back into the original UML model as tagged values. The same approach is used in [48] for the automatic generation of small components emulating the behavior of real one.

1.3 Contribution and outline of this thesis

This work deals with automatic generation of simulation performance models from high-level UML descriptions of SA. To the best of our knowledge, this is one of the few approaches considering UML diagrams annotated with (a subset of) the UML Performance Profile [75]; the other we are aware of is a work by Petriu and Shen [77] in which the authors describe a graph-grammar derivation of performance models based on LQN specifications from annotated UML models. The reason why the UML Performance Profile has not been widely considered yet is probably that it has been released only very recently; we expect many other approaches along this direction in the future.

Motivating the use of UML We have already seen the advantages of using UML as a notation for description of software systems. It is widely used and accepted in the Software Engineering community, provides a set of diagrams allowing users to represent a system from different point of views and at different level of details, and there are many tools able to create and manipulate UML models. The drawback of the approach is that UML is only informally defined, so that software modelers may use different diagrams for the same purpose, or use the same notation with different implicit meaning. While addressing this limitation is outside the scope of this work (among others, the Precise UML group [81] is specifically working on this issue), the performance-oriented UML profile we will define in Chapter 5 actually enforces an (operational) semantics of UML in terms of the derived performance model. Intuitively, we define the dynamic behavior of a set of annotated UML diagrams in terms of the behavior of the implementation of the automatically derived simulation model.

Motivating the use of simulation model We chose to derive a simulation performance model for different reasons, which have in part already been introduced in the previous discussion.

The main reason is that simulation models allow for unconstrained representations of software models. This means that the performance model captures, and is able to analyze, all behaviors of the software under study. This is not always easy to do with analytical models. For example, in QN models it is difficult to handle situations arising from finite capacity of queues (and subsequent blocking behavior); only approximate techniques can be used in some cases, and simulation is the only approach in general [14]. Other difficulties arise when analyzing simultaneous resource possession, fork and join systems, synchronous vs asynchronous communications, and many queuing disciplines (especially with complex priorities between requests).

A direct consequence of using simulation is that deriving the performance model from the software specification is very easy, as we will see in detail in Chapter 6.

We define an almost one-to-one mapping between UML elements and simulation processes. The result is that the structure of the simulation model is very similar to the structure of the software model; this can be seen by the class diagram of the UML-Ψ tool described in Chapter 9. Direct translation of software model into performance model not only makes the modeling process easier and less error-prone, but allows an easier interpretation of the performance results back at the SA level. This is very important, as it is obvious that all indices computed by the performance model must be reported back at the design level, where the software modeler can decide whether the proposed SA fulfills its performance goals. This step, labeled as *feedback* in the schema of Fig. 1.1, p. 8, is not trivial when the performance model has been obtained by applying a complex transformation to the software model. Reversing this transformation may be feasible only in special circumstances, meaning that the software architect discovers a performance problem but is unable to identify the specific component (or set of components) responsible for that misbehavior.

Motivating the use of process-oriented simulation As we will see in Chapter 3, there are different types of simulation models. The relevant ones for this application are *event-oriented* and *process-oriented* simulation models. In event-oriented simulations the user must specify the different types of events which may happen, an event being defined as an instantaneous action taking no (simulation) time to execute, and causing a change in the system's status. In a process-oriented simulation, the user defines a set of simulation processes, each containing a set of actions to be performed. Some of these actions may require some (simulated) time to execute, while others may be instantaneous. We decided to implement the software performance model as a process-oriented simulation. In general, event-oriented simulations are preferred when the number of event types to be modeled is relatively low [17]. In contrast, the performance model derived from SA can be more easily described as a set of interacting processes, each one with an internal behavior which can be procedurally described. Process-oriented simulations are well suited for this purpose.

Contribution The contribution of this work is threefold. First we define a process-oriented simulation model for evaluating performances of SA at early development stage. Then, we propose a UML profile, based on the UML Performance Profile, providing facilities for

- Associating performance-oriented, quantitative annotations to UML elements;
- Specifying parameters to be used for the execution of the simulation performance model;
- Providing feedback to the software architect about the performance results found by running the simulation.

Finally, we show an actual software performance evaluation tool called called UML-Ψ (UML Performance SImulator). UML-Ψ automatically derives a simulation model from annotated UML specifications, executes the model and provides results at the SA design level.

This thesis is divided in four main parts:

- In Part I we introduce the problem of software performance evaluation at the architectural level. In Chapter 2 we given an introduction to the UML notation, and in Chapter 3 we introduce the basic concepts of discrete-event simulation modeling.

- In Part II we describe the structure of the simulation model and the profile for annotating UML elements. Chapter 4 introduces the performance model which will be used to represent the dynamic behavior of software systems. In Chapter 5 we show how concepts from the performance domain are implemented into UML models by means of an UML profile. In Chapter 6 we describe the simulation model in detail, including detailing the behavior of all simulation processes involved. The performance evaluation technique will be extended in Chapter 7 to an integrated UML-based performance and mobility modeling approach.

- Part III is devoted to describe implementation details of `libcppsim`, a general-purpose C++ process-oriented simulation library, and UML-Ψ, a prototype performance evaluation tool built to implement the proposed performance evaluation approach. The `libcppsim` library is described in Chapter 8, and the UML-Ψ tool is described in Chapter 9. We present in Chapter 10 a case study in which the proposed performance evaluation approach is applied to the architecture of a naval communication system.

- The final part presents the conclusions and future works.

2

Introduction to UML

This thesis assumes that the reader is familiar with UML. This chapter provides an overview of UML, with a particular emphasis on the parts of UML which are relevant for the proposed performance modeling algorithm. More informations can be found in [74, 87].

UML is a semi formal language developed by the OMG [73] for specifying, visualizing and documenting software artifacts; it can also be applied to non-software systems, such as business precesses. UML is widely used in the software engineering community to describe systems developed according to the object-oriented paradigm.

UML is a graphical notation which allows the user to describe an artifact using a suitable combination of diagrams, chosen among the available ones. UML defines several kind of diagrams:

- Use Case diagram;
- Class diagram;
- Behavior diagrams: Statechart diagram, Activity diagram;
- Interaction diagrams: Sequence diagram, Collaboration diagram;
- Implementation diagrams: Component diagram, Deployment diagram

The above diagrams can be partitioned in three categories:

- Static diagrams are used to model the logical or physical structure of the system. They include Class diagram, Component diagram and Deployment diagram.
- Dynamic diagrams are used to describe the behavior of the system. They include Use Case diagram, Statechart diagram, Activity diagram, Sequence diagram, Collaboration diagram.
- Model Management diagrams are used to group other model elements, and include Component diagram.

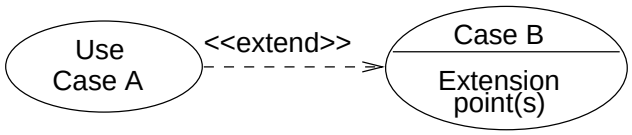
It should be observed that UML deliberately lacks a formal semantics, as it has not been developed to be a visual programming language. From one side this has the benefit that users can use and combine UML elements with few restrictions. Unfortunately this makes any formal reasoning on UML models an extremely difficult task. It is impossible to derive the correct meaning of a particular combination of diagrams, as the UML semantics is only informally specified. The Precise UML (PUML) group [81] is currently investigating the problems associated with the semi-formal specification of UML, and how they can be solved.

2.1 Use Case diagram

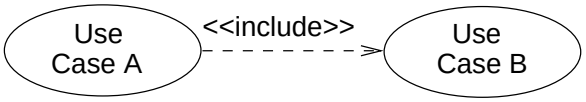
Use case diagrams describe at a high level the interaction between the system and *actors* requiring service. An actor is any entity (both physical or logical) which may interact with the system. Actors are graphically represented as "stick people"; they can interact with the system in possibly different ways, each being a different *use case*. Each use case represents one or more scenarios. A use case is graphically represented as an oval connected to an actor. This connection may represent the fact that the actor generates or takes part to the use case.

UML defines different kind of relations between use cases:

extend An extend relationship between use cases is shown by a dashed arrow with an open arrow-head from the use case providing the extension to the base use case. The arrow is labeled with the keyword ≪ extend ≫. An extend relationship from use case A to use case B indicates that B may be augmented by the additional behaviors specified by A. The user may indicate explicitly the extension points inside the use case B.



include An include relationship between use cases is shown by a dashed arrow with an open arrow-head from the base use case to the included use case. The arrow is labeled with the keyword ≪ include ≫. An include relationship from use case A to use case B indicates that A will also contain the behavior specified by B.



Generalization A generalization between use cases is shown by a generalization arrow, that is, a solid line with a closed, hollow arrow head pointing at the

parent use case. A generalization from use case A to use case B indicates that A is a specialization of B.

2.2 Class diagram

A class diagram represents the static structure of a system. It includes the static model elements such as classes, interfaces, objects and their relationships, connected as a graph to each other. A class is the description of a set of objects with similar structure, behavior, and relationships. A class has a name identifying it, a set of operations and a set of attributes. An operation is a service that an instance of the class may be requested to perform. It has a name and a list of arguments. An attribute is a piece of information describing part of the state of objects of that class. Both operations and attributes can have different visibility, denoted by prepending the symbol '+' (for public visibility), '#' (for protected visibility), '-' (for private visibility) and '~' (for package visibility). An object is a specific instance of a class; an object is characterized by specific values for the attributes of the corresponding class.

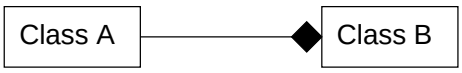
Different kinds of relations can be defined between classes or objects:

Association An association indicates that two classes or objects are related. There are some additional notations available for the association:

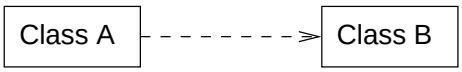
Multiplicity Indicates how many instances a class can have in the association. It can be indicated as a number (exact multiplicity), an asterisk (unbounded multiplicity) or an interval $m..n$;

Aggregation Denotes that one class is a collection of several;

Composition Denotes that one class is a part of the other class;



Dependency Denotes that one class depends on the other.



Generalization Denotes the relationship between a more general element A and a more specific element B. B inherits the properties (attributes and operations) of its supertype A; optionally, it may have some additional attributes and operations, or redefine the meaning of operations of its supertype.

2.3 Interaction diagrams

Interaction diagrams are used to describe the dynamic interaction of objects; an interaction is an exchange of messages or stimuli.

2.3.1 Sequence diagram

A UML sequence diagram can be used to specify the internal behavior of a use case in term of interactions between objects. An interaction is displayed as a set of partially ordered messages, each message being a communication between sender and receiver. Objects taking part to the interaction are displayed horizontally. Each object has an associated lifeline, which visually denotes when each interaction takes place in time. Time advances downward on the sequence diagram. It is possible to indicate different events along each lifeline. Periods of activity of each objects are denoted with rectangles; synchronous/asynchronous interactions are shown as arrowed lines going from the sender's lifeline to the receiver's one. Different types of arrowheads denote if the communication is synchronous or asynchronous.

The user may label the messages with a sequence number in order to show the order in which the interactions happen. Moreover, each message may be labeled with a condition which must hold in order for the communication to occur, and the number of times it should be sent.

Object creation and destruction can be displayed in UML sequence diagrams by means of special creation and destruction messages respectively. Object creation instantiates a new object, whose lifeline starts from the position of the creation message. Object destruction causes an existing object to be destroyed; this is reflected in the diagram by terminating the object's lifeline at the destruction message arrowhead.

2.3.2 Collaboration diagram

A UML collaboration diagram is used to display the interactions between objects, in a similar way as a sequence diagram. However, collaboration diagrams put more emphasis on the structure of the interaction (relation between components), while sequence diagrams put more emphasis on the flow of time. In collaboration diagrams

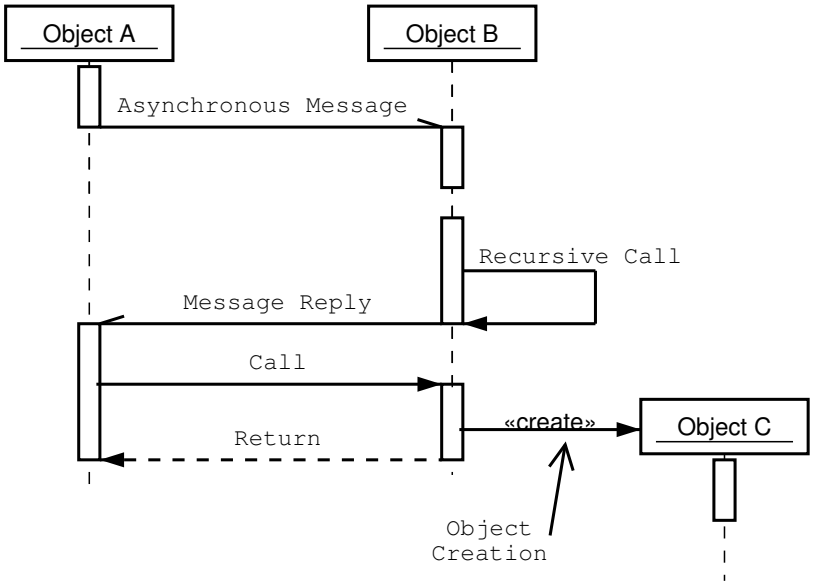


Figure 2.1: A UML Sequence diagram

the flowing of time is not shown explicitly as in sequence diagrams. A collaboration diagram is a graph whose nodes are objects, and edges show which node interact to which other. Messages can be indicated with additional arrows along the edges of the graph, in much the same way as in sequence diagrams. The partial ordering of messages can be deduced from their sequence numbers.

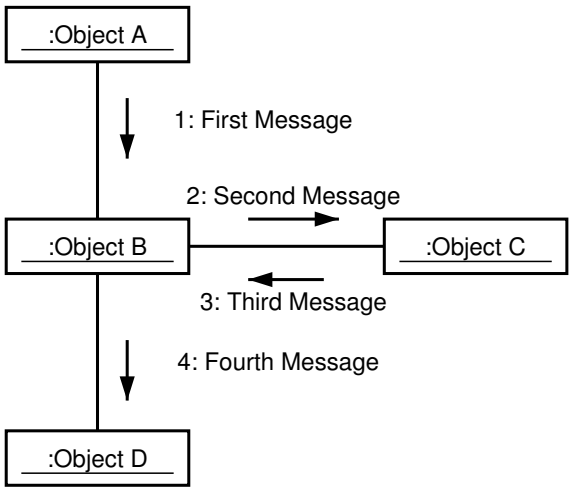


Figure 2.2: A UML Collaboration diagram

2.4 State diagram

State diagrams show how the internal state of an object evolves during the time. It is directed graph which represents a state machine, whose nodes represent all the possible states of the objects, and the edges represent the transitions between states. A state diagram, thus, shows how objects evolve from their initial state toward a final state. Each state may contain another state diagram, which allows to specify its behavior at a deeper level of detail.

Transitions may be triggered both by external or internal events. An example of an internal event is the completion of an internal activity. An example of external event is the reception of an interrupt signal from other parts of the system.

A state may have an optional name, and an *internal transitions compartment* containing a list of internal transitions or activities which are performed while the object is in that state. Actions have an optional label identifying the condition under which the action is performed. Several labels are predefined and have special meaning:

entry Identifies the first action which should be executed upon entering the state;

exit Identifies the last action which should be executed immediately before exiting the state;

do This label identifies an "ongoing activity", that is, an activity which is continuously executed as long as the object is in this state;

include This label identifies a submachine invocation.

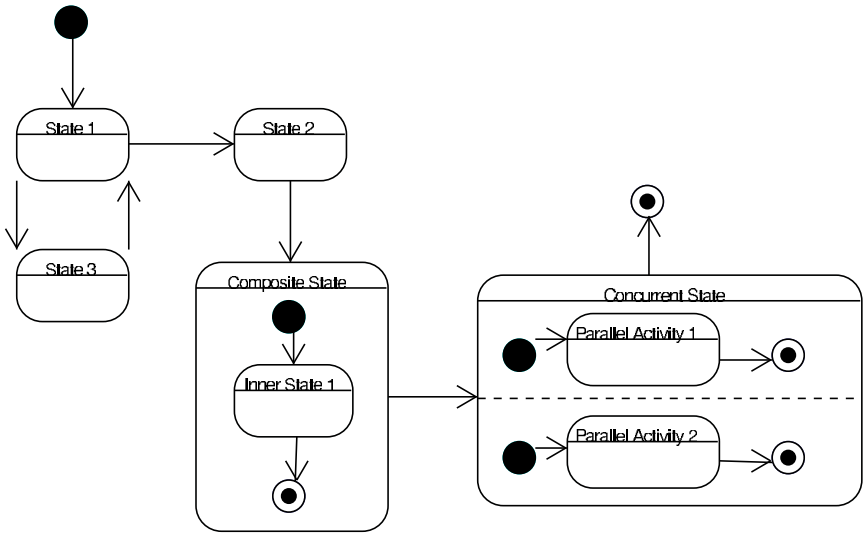


Figure 2.3: A UML State diagram

2.5 Activity diagram

An activity diagram is a special case of a state diagram in which most of the states are subactivity states or action states (states corresponding to the execution of an action). Moreover, transitions in an activity diagram are generally triggered by completion of the actions or subactivities in the source states. An activity diagram specifies the behavior of a use case, a package, or the implementation of an operation.

Activity diagrams are an evolution of *flow charts*, from which they inherit the ability to show execution flows depending on internal processing (as opposed to external events). State diagrams are preferred in situations where external, asynchronous events occur.

Activity diagrams include the possibility to represent concurrent execution of multiple computations through fork/join nodes. A *fork* node denotes the point where the computation splits in concurrent execution threads, each evolving independently from the others. A *join* node denotes a point where different execution threads synchronize.

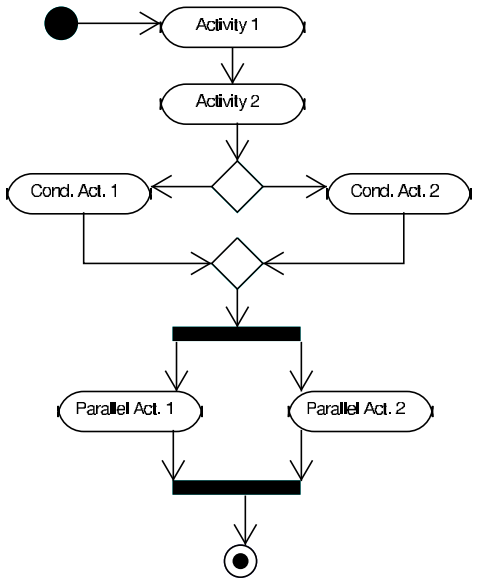


Figure 2.4: A UML Activity diagram

2.6 Deployment diagram

Deployment diagrams show the configuration of run-time processing elements and the software components, processes, and objects that execute on them. Software component instances represent run-time manifestations of software code units. Components that do not exist as run-time entities (because they have been compiled

away) do not appear on these diagrams, but should be shown on component diagrams.

A deployment diagram is a graph of nodes connected by communication associations. Nodes may contain component instances: this indicates that the component runs or executes on the node. Components may contain instances of classifiers, which indicates that the instance resides on the component. Components are connected to other components by dashed-arrow dependencies (possibly through interfaces). This indicates that one component uses the services of another component. A stereotype may be used to indicate the precise dependency, if needed. The deployment type diagram may also be used to show which components may reside on which nodes, by using dashed arrows with the stereotype ≪deploy≫ from the component symbol to the node symbol or by graphically nesting the component symbol within the node symbol.

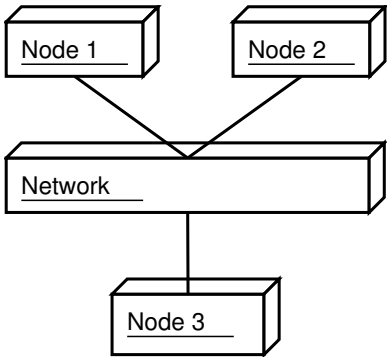


Figure 2.5: A UML Deployment diagram

2.7 Extension Mechanisms

UML provides some built-in functionalities for extending its metamodel. Such extension mechanisms can be used to add specific informations to existing UML elements, or to define new types of metamodel elements based on existing ones. The constraint on all extensions defined using the extension mechanism is that extensions must not contradict or conflict the standard semantics. Thus, extensions must be strictly additive to the standard UML semantics. The extension mechanisms are a means for refining the standard semantics of UML, but do not support arbitrary semantic extension [74]. UML extension mechanisms include Stereotypes and Tagged Values.

Stereotypes

Stereotypes are the main UML extension mechanism. They are used to define subclasses of existing metamodel elements. Such new metamodel elements have the

same structure as the original one, but may have additional constraints from the base metamodel class, or it may require tagged values (see below) to be included to the elements with that stereotype.

A stereotype is graphically represented as a textual label enclosed in double quotes (eg ≪ abstract ≫). The label is associated to the model element to be stereotyped.

Tagged Values

Tag definitions specify new kinds of properties that may be attached to model elements. The actual properties of individual model elements are specified using Tagged Values. These may either be simple datatype values or references to other model elements. Tagged values may be used to represent properties such as code generation information or quantitative informations for performance evaluation purpose (as seen in more detail in the next part).

UML Profiles

A profile is a stereotyped package that contains model elements that have been customized for a specific domain or purpose by extending the metamodel using stereotypes, tagged definitions, and constraints. A profile may specify model libraries on which it depends and the metamodel subset that it extends.

Examples of UML profiles include the UML profile for Schedulability, Performance and Time specification [75].

3

Simulation

3.1 Introduction to Simulation

A *model* is defined as an abstract representation of a real system. A model can be used to investigate the effects of proposed system changes, without modifying the real system. Also, a model can be used to study systems in the early design stage, before they are even built. Thus, system modeling can be used to study an existing system under various scenarios without modifying it, or for planning the construction of a new system which does not exist yet. System models are developed in order to evaluate some functional or non-functional properties of the system. Functional properties include throughput, mean execution time, reliability and so on. Non-functional properties include deadlock-freedom, usability, responsiveness and others.

Different kind of system models have been considered in the literature: analytical, simulation-based and mixed. Analytical modeling involves building a system description using some formal, mathematical notation. Closed-form solutions for some class of models have been studied extensively. Usually analytical models must satisfy some constraints in order to be solved exactly. In general cases, approximate numerical solutions can be computed.

A simulation is the imitation of the operation of a real-world process or system over time [17]. This is done by developing a simulation model of the system. The model is based on a set of assumptions on the real system behavior, and on the workload driving the system. A correct and validated simulation model in fact can substitute the real system as long as the underlying assumptions are met. A simulation model cannot be "solved", as for analytical models. Instead, a simulation model can be implemented as a simulation program, which is then executed. The output of the simulation program includes values for the parameters of interests which are being measured from the model.

3.2 Advantages and Disadvantages of Simulation

Simulation is a very general modeling technique, in that it is possible to develop a simulation model without making restrictive assumptions which are necessary to develop analytically solvable models. For example, product-form queuing network models only allows a limited set of scheduling policies for service centers, and only some kinds of random distributions are allowed to model the interarrival time of users or the service time. Simulation models, on the other hand, can reproduce the behavior of real system at an arbitrary level of detail, provided that enough details from the system are known.

Simulation models can be implemented using a number of existing simulation languages, libraries and tools [16, 17], many of which provides libraries of pre-modelled components. Also, extensive data-collection functions are often provided. This simplifies the simulation program development and reduces the possibility of introducing error while implementing the simulation model. Graphical interfaces allow modelers to actually "see" the evolution over time of the simulated system.

However, there are some disadvantages which should be considered when choosing between simulation and analytical modeling. Simulation results may be difficult to interpret, given that the output of simulation programs are usually streams of random variables. Special skills are required to analyze the simulation output with correct statistical techniques.

Moreover, given that simulation results are raw numbers, it is not possible to get results depending on one or more parameters. For example, it is not possible to obtain the mean response time of a system as a function of the number of concurrent users, unless a separate simulation is performed increasing each time the number of system users.

Finally, simulation modeling and analysis can be time consuming and thus expensive, especially for complex models.

On the positive side, some of the problems above do not constitute serious issues. Many simulation software vendors developed advanced output analysis functions in their packages, so users can expect to get meaningful results without detailed mathematical knowledge. Simulations can be performed faster as the current generation of hardware evolves. Moreover, parallel and distributed simulation techniques [40] allow simulation models to be executed concurrently on different processors, obtaining significant speedup in some cases.

To conclude, recall that closed-form analytical models can not be developed for many real-world systems which are commonly encountered. Such systems can only be simulated.

3.3 Types of Systems

We define the *state* of a system as the collection of variables describing the system at any given time. The state of the system should defined according to the goal of the modeling process; in this way it is possible not to consider all the system details which are not relevant to the model, which therefore is greatly simplified. An *event* is defined as an instantaneous change of the system state. Events are usually classified as *endogenous*, which denote those events occurring within the system, and *exogenous*, which are activities and events of the surrounding environment which alter the state of the system.

Systems may have discrete or continuous state space, and events may occur at discrete points in time or continuously. An example of discrete state variable is the number of customers waiting for service in a bank. An example of continuous state variable is the temperature of a room. Note that there may be both continuous and discrete variables in the same system state. Continuous-time systems are those in which the state variables change continuously over the time. An example of a continuous system is the transfer of heat on a medium. The state variables in this case are the temperatures measured on every point of the medium. The temperatures change continuously according to the general heat transfer model. On the other hand, a discrete-time system is one in which the state changes only at discrete point in time. A bank is an example of such system, as the number of customers waiting service changes only when a new customer enters the bank, or a customer completes service.

3.4 Discrete-Event Simulation

Discrete-event simulation is used to model systems in which the state variables change only at specific points in time, called *events*. Each event is labeled with its simulation time of occurrence, which is denoted as *timestamp*. Simulation events are kept in a data structure called Future Event List (FEL) or Sequencing Set (SQS) in the Simula [35] language. The FEL supports the following operations:

- Schedule (insert) a new event at time t;
- Unschedule (remove) an arbitrary event;
- Extract the event with smaller timestamp.

Executing the model is done by executing all the events in the FEL in nondecreasing timestamp order. In this way the simulation time advances by jumping to the value of the events timestamps as they are extracted from the FEL.

Discrete-event simulation models can be implemented according to an event-oriented or process-oriented paradigm. It should be observed that these paradigms (which can also be used together on the same simulation implementation) are in fact equivalent, in the sense that each one reduces to the other.

Event Orientation

Event-oriented simulation models are defined in the following way. The modeler defines the set of all events which may happen on the model. Events can perform any combination of the following operations:

- Update the state of the simulated system;
- Cancel one or more of already scheduled events;
- Schedule one or more new events.

Process Orientation

A process-oriented simulation model is represented as a collection of concurrently executing simulation processes, each one with its own thread of control. A simulation process is made of two parts: a set of local variables, and a sequence of actions to execute.

As a practical example, we consider now the way simulation processes are handled in the SIMULA language [35]. SIMULA has been the first object-oriented programming language, and also one of the first general-purposes programming languages with built-in simulation facilities. These same features can be found in most currently available process-oriented simulation tools. The simulation library described in Part III provides many of the functionalities of SIMULA using the C++ programming language.

Actions performed by simulation processes are grouped together into active phases, separated by periods of inactivity. Active phases are "instantaneous", in the sense that they take no simulation time to execute (of course they will take some wall-clock time to complete). A simulation process is suspended upon execution of a deactivation statement (which is language dependent). Deactivation statements terminate the current active phase, and cause control to leave the process. When it is later resumed, the process starts another active phase by continuing execution from the point it was stopped. The typical deactivation statement is the one which suspends execution of the current simulation process until some specific event occurs; in the simplest form, a procedure like `hold(`t`)` suspends the execution of the currently executing simulation process for t simulation time units (see Fig. 3.1).

Simulation processes can be implemented using Operating System (OS) threads, if available. A group OS thread operates on the same process memory space, each thread being independently scheduled from the others. All the threads of the same process share a common memory space. Simulation processes execute the actions in their active phases, and then pass control to some other simulation process. Note however that the context switching time can be non-negligible, especially for complex simulations involving hundreds or even thousands of simulation processes.

Another way to implement simulation processes efficiently is to use the *coroutine* programming construct. This approach has the advantage of being potentially more

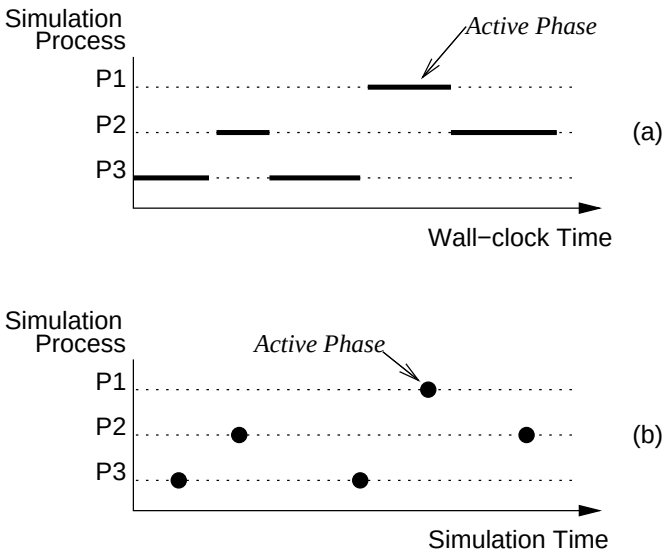


Figure 3.1: Execution of a process-oriented simulation model. In (a), active phases take wall-clock time to execute; only one process is in its active phase at any time. In (b) active phases take no simulation time to execute. Between active phases, each simulation process is deactivated.

efficient, as there is no context switch overhead caused by the OS. More details will be given in Section 8.2.

The Sequencing Seq

The Sequencing Set is a data structure which implements a Future Event List. The SQS is an ordered list of event notices; an event notice contains a timestamp and a reference to the simulation process to activate. Each simulation process maintains a copy of its status before being suspended, so it can be resumed from the exact point where it stopped. The structure of a SQS is depicted in Fig. 3.2.

Note that multiple processes may be linked to the same event notice. This is because multiple processes may be activated at exactly the same time. The SQS data structure supports exactly all the functionalities of the Future Event List described above.

The SQS can be implemented using different data structures. Most languages and libraries (including SIMULA) implement SQS as a doubly-linked list. Inserting and removing an event notice or a process from the list requires $O(n)$ time, n being the length of the SQS. Accessing the element with lower timestamp can be accomplished in $O(1)$ time. More efficient implementations have been proposed in the literature. Balanced search trees (such as B-Trees [20]) reduce the complexity of inserting and removing event notices to $O(\log(n))$; the computational complexity of finding the event notice with smallest timestamp is $O(\log(n))$ as well. More sophisticated data structures, such as Calendar Queues [30] perform the same operations

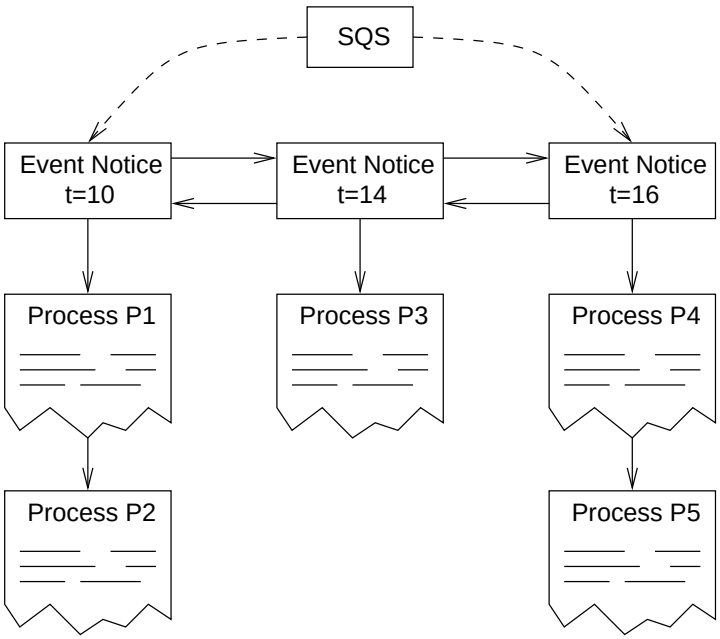


Figure 3.2: Sequencing Set structure

in $O(1)$ expected time. It should be noted that in practice the relative performances of the aforementioned data structures may be different than what expected from their computational complexity. In particular, simulation programs handling small event lists (on the order of 10-20 event notices) perform better with the simple doubly-linked list implementation [30].

The simulation engine works according to the following simple pseudocode:

```
procedure Simulation_Engine;
var
   s : SQS;
   p : process;
begin
   while ( not( s.empty ) ) do begin
      p := s.first( );
      p.resume( );
   end;
end;
```

Basically, the scheduler fetches the first event notice from the SQS and executes the associated process. If more than one process is associated to the same event notice, then one of those is selected according to some ordering criteria. When the SQS is empty, the simulation stops.

Scheduling primitives

We illustrate now the simulation primitives that will be used in Part II for describing the simulation model. These primitives are based on those provided by the SIMULA

language.

activate P The activate statement is used to insert a process P in the SQS. Without any parameter, P is inserted in front of the SQS. Optionally, it is possible to specify the simulated time when the P will be activated, or whether P should be activated immediately before or after another process Q.

hold(dt) Suspends the execution of the current process for a simulated time dt. The process is removed from the SQS and reinserted with timestamp increased by dt.

passivate Suspends the execution of the current process indefinitely, until another process reactivates it. The process is simply removed from the SQS.

cancel(P) Removes process P from the SQS. P needs to be activated explicitly by another process to resume execution.

P**.idle()** Returns true if the simulation process P is currently idle, ie, it is not present in the SQS. Returns false otherwise.

A simulation process may be in one of four different states during its execution (see Fig. 3.3). The states are:

Passive A simulation process is passive if it is not in the SQS, ie, if it has no associated event notice. A passive process gains control only when another process explicitly activates it.

Active An active simulation process is the one currently executing. Only one process can be in the active state at any time.

Suspended A suspended process has an associated event notice, but it is not currently executing. If it is not removed from the SQS, it will be resumed at the simulation time specified in its event notice.

Terminated A terminated simulation process finished its execution, and thus cannot be reactivated again.

A terminated (but still allocated) process can still be used, as it is possible to call its methods or access its attributes.

A newly created simulation process is in the passive state, as it has not yet been inserted into the SQS. When a process Y explicitly activates a passive process X, then X become suspended: X is associated with some event notice and thus is in the SQS. When a suspended process becomes the head of the SQS and is selected by the scheduler to execute, it enters the active phase. An active process may leave its active phase in three ways:

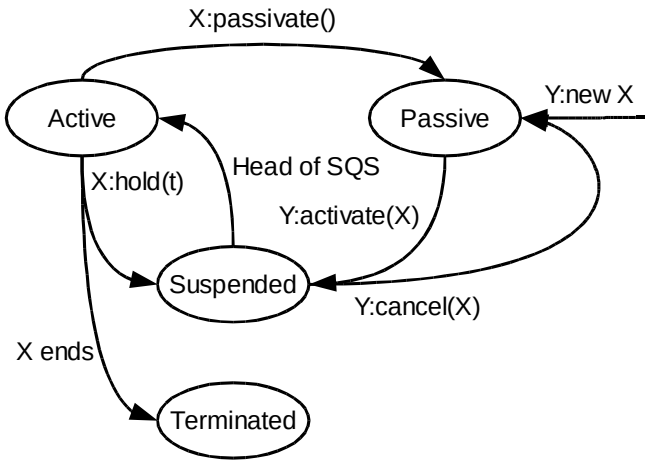


Figure 3.3: State transitions for a simulation process

- It executes an `hold(t)` operation, which reschedules it to be activated in the future;
- It executes the `passivate()` operation, which renders the process passive.
- It terminates.

Finally, a suspended process X may become passive if another process Y removes it from the SQS.

3.5 Steps in a simulation study

The steps which made a simulation study are depicted in Fig. 3.4, taken from [17].

Problem formulation The starting point is problem formulation. It is very important that the problem is well-stated and understood both by the parts involved in the simulation study.

Setting objectives and project plan . During this step the questions that must be answered by the simulation study are defined. According to the nature of both the questions and the problem to be studied, it should be checked whether simulation is the appropriate tool to use. If it is, the plan should include an estimate of the costs, and the number of people involved in the simulation study.

Model conceptualization During this phase a model of the system under study is developed. The model will usually depend on the goal of the simulation study. The model complexity should not exceed the minimum complexity required to satisfy the goals of the simulation study.

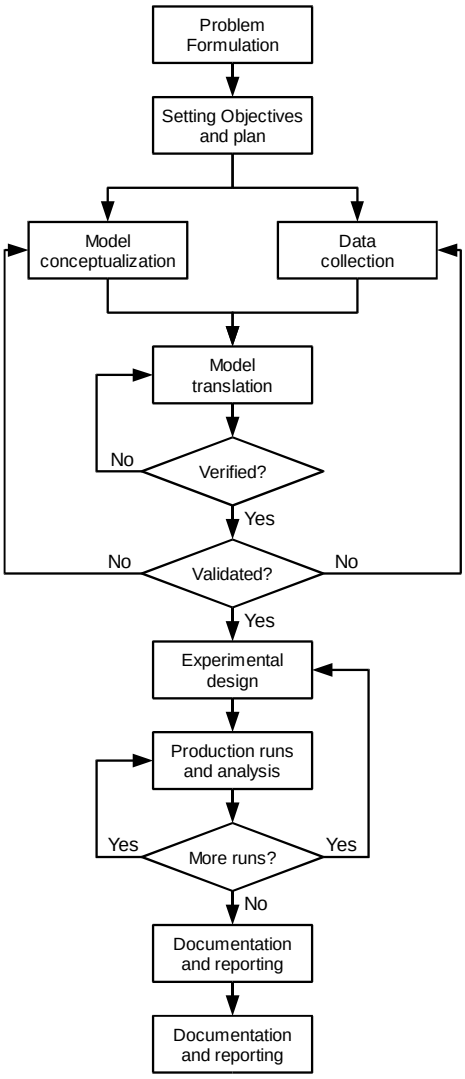


Figure 3.4: Steps of a simulation modeling study (from [17])

Data collection This phase requires simulation modelers to collect the data which will be used to drive the simulation program. Depending on the problem, this may require collecting traces from a running system, or finding distributions and parameters for generating synthetic workloads.

Model translation During this step the simulation model is translated into a simulation program. This can be done using a general-purpose programming language augmented with a simulation library, or using special-purposes simulation languages or environments. Simulation-specific tools usually provide a graphical user interfaces using which the modeler can build the simulation program by direct visual manipulation of graphical entities. Graphical interfaces are also used to

display the evolution of the simulation program as it executes.

Verified? A simulation program is verified if it contains no errors.

Validated? A simulation model is validated if it is an accurate representation of the real system, i.e. it can be substituted to the real system, with respect to the goals of the simulation study.

Experimental design During this step it is necessary to determine parameters such as the length of simulation runs, the number of replication of each run to execute and the length of the initialization period.

Production runs and analysis Production runs are used to estimate the parameters of interest of the simulation study.

More runs? Once production runs have been analyzed, additional runs may be needed. In this step it is necessary to decide what design those additional runs should follow.

Documentation and reporting At the end of simulation executions, it is necessary to document the whole modeling process. Documentation includes the simulation program documentation and the progress report. The progress report describes the decisions which have been taken and the work which has been done during the various phases. Of course, the result of all the analysis should be provided.

II

Description of the approach

4

Simulation Modeling of UML Specifications

4.1 General Principles

Traditional software development processes [93] do not take into account performance considerations about the software system being produced. The possible outcome is that performance problems arise only when the system has been at least partially implemented. This results in waste of time and resources, as fixing performance problems requires the system to be re-engineered, so the development process must start back from some earlier phase.

The starting point of many software development processes is the production of a high-level description of the system to be built. Such software models can be produced relatively fast using standard design tools. Also, software models are usually available since the early development phases. The question arises whether it is feasible to use a software model to estimate the performances of the real system, once such a real system is built.

The answer to this question is not trivial, but has already been answered positively [9, 91, 92]. Giving an extremely accurate performance estimation is, however, very difficult in general, for a number of reasons:

- The software model is, by definition, an abstract and simplified representation of a real system. Software models deliberately lack details. Starting from an incomplete model it is rarely possible to derive an accurate simulation model of the real system.

- Correct identification of parameters for the performance model can be difficult. These parameters include execution times of procedures, size distribution of packets sent to communication networks, network delays, and many others. Parameters can be easily estimated if a running system already exists, so that they can be directly measured; unfortunately, such a running system is seldom available.

- The performance model must be driven by a set of external workloads. Identifying these workloads during early design stages can be difficult, since little or no information is available. Again, if there is a similar system already in use, measurements on it can provide useful informations about the workload. However, not only a running system may not be available, but characterizing the workload is particularly difficult in many situations as the actual pattern of requests may be completely different from what expected.

For these reasons, we propose an approach which is not intended to predict the actual performances of software systems with complete accuracy. Instead, we want to provide the software developer a tool to explore and compare different design alternatives during early development steps. The proposed approach derives a simulation model from a UML specification of a software system. The simulation model is implemented as a simulation program and executed. Performance results are then fed back into the UML model.

Keeping the intrinsic limitations of the approach in mind, the issues described above can be answered as follows:

- Lack of details in the software model can certainly be an issue. However, this is a limitation of every approach deriving performance models from SA descriptions. In fact, the simulation model we develop is as accurate as the software model from which the performance model is derived. This means that it is clearly impossible to represent something which is not described in the SA model; thus it is responsibility of the software modeler to find a compromise between the level of detail of the architectural description, the knowledge available on the system, and the accuracy of the performance results desired. If performance modeling is carried out during the whole software development cycle, then more details can be entered later, when they become available.

- Parameters from the simulation model can be obtained in different ways. The system modeler may use part of the design from another system, for which parameters can be measured or accurately estimated. If no accurate estimation is available, an educate guess should be enough in order to get at least some insights.

- As the problem of workload characterization is concerned, SA performance studies are often used to predict the behavior of the system under varying workloads, as the exact arrival pattern of requests is often unknown. At early design stages it is useful to experiment with different workloads in order to understand the potential limitations of the system under development.

4.2 Overview of the approach

We consider the performance modeling cycle is depicted in Fig. 4.1, which is based on the general cycle depicted in Fig. 1.1, p. 8.

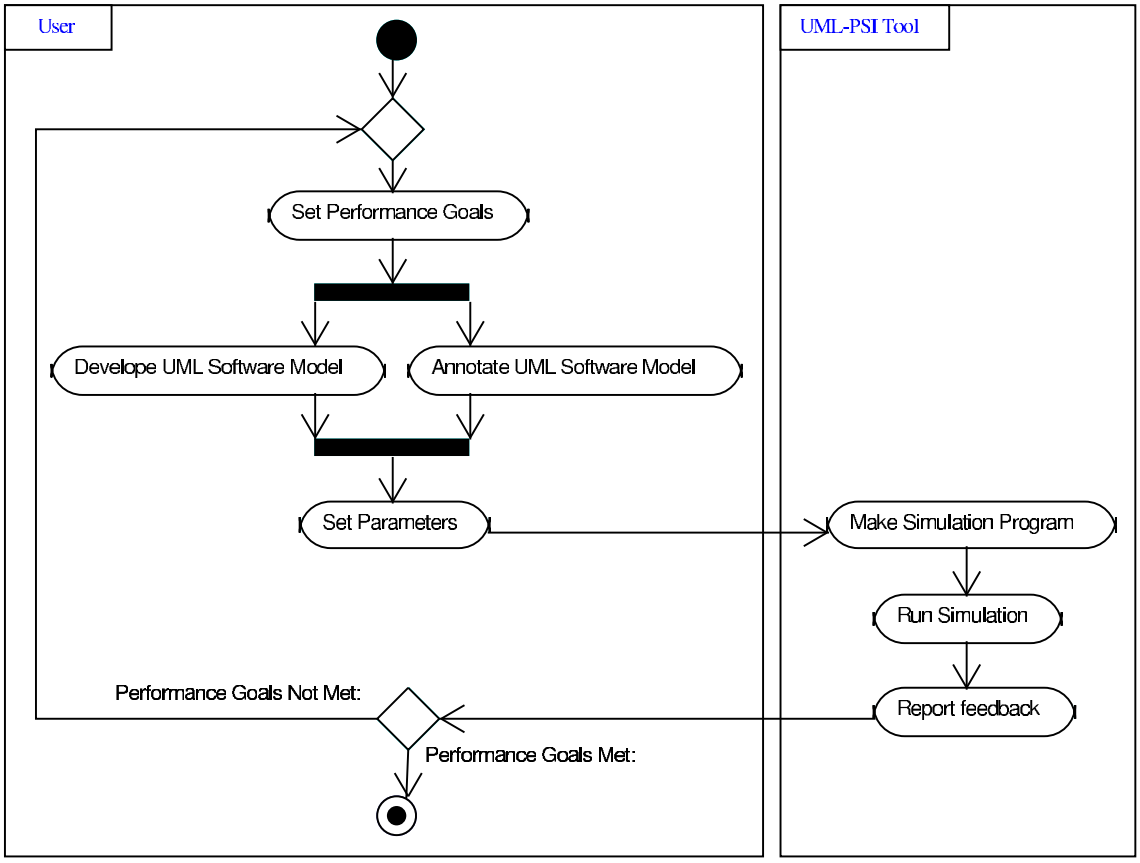


Figure 4.1: Performance modeling cycle

The first step requires the identification of the performance goals which the system should satisfy. This is put before all the other steps, as the SA will likely be structured and implemented according to its desired performances.

The next step is to develop a system model in UML; at the same time, UML diagrams are to be annotated with quantitative, performance-oriented informations which will be used to evaluate the software model. The software model can be visually drawn using a UML CASE tool. Annotations are inserted as stereotypes and tagged values, according to a subset of the annotations defined in the UML Performance Profile [75]. Details about which UML diagrams to use and what kind of informations can be specified will be given later in this chapter.

Our approach produces a simulation-based performance model of the SA. Once the UML SA is defined and annotated, the user must specifies both the model and

simulation parameters. Model parameters are required if the software architect inserted unbounded variables in the annotations. In this case, some of the informations needed to execute the simulation program are undefined, and must be instantiated at model implementation time. Simulation parameters are those parameters defining the termination criteria of the simulation program, such as the maximum length of the simulation, the confidence interval for the computed statistics and the interval relative width.

The next step is the derivation of the simulation program from the annotated UML diagrams. This step is done automatically by UML-Ψ, a prototype performance evaluation tool we built to demonstrate the proposed approach. UML-Ψ parses an XMI [76] representation of the annotated UML model, and builds a process-oriented, discrete simulation program for that model based on a general-purpose, C++ simulation library. The implementation details of UML-Ψ will be given in Part III.

The simulation program is finally executed, and the computed performance results are used to provide feedback at the SA design level. In particular, performance results are associated to the UML elements they refer as new tagged values. Simulation results provide feedback to the software developer, which is then able to check whether the SA developed so far satisfies the performance requirements, given the set of parameters used to perform the simulation. If performance requirements are not met, then the modeling process can be repeated, possibly changing the SA and/or the parameters used for the simulation. The software architect could also decide that the performance goals can not be reached. In this case, performance goals should be restated and the process restarted from the beginning.

In order to apply our proposed approach, the software modeler only needs to learn the notation used for specifying quantitative, performance-oriented informations to UML diagrams. The steps of generating the performance model from the software specification, implementing and executing the model, and reporting the results back at the SA level are done automatically and transparently by a prototype tool we describe in Part III. Thus, no specific knowledge is requested to the user about simulation modeling or output data analysis. In Fig. 4.1 we show who is responsible for each step by using *swimlanes*. Thus, the user (software modeler) is responsible for defining the performance goals, defining and annotating the UML diagrams and defining the parameters. The UML-Ψ tool automatically derives the simulation program, executes it and computes the results.

Note that each time the cycle is iterated, the performance model is computed from scratch, even if only small modifications are done on the software model. Building the simulation model in an incremental fashion, that is, reusing part of previously obtained models and changing only what is needed to accomodate the changes in the software model, could certainly be done with some effort. At the moment, we decided not to do so as the model generation step is performed in time linear with the number of UML elements in the software model. The additional complexity of incremental simulation model generation does not seem to justify the possible

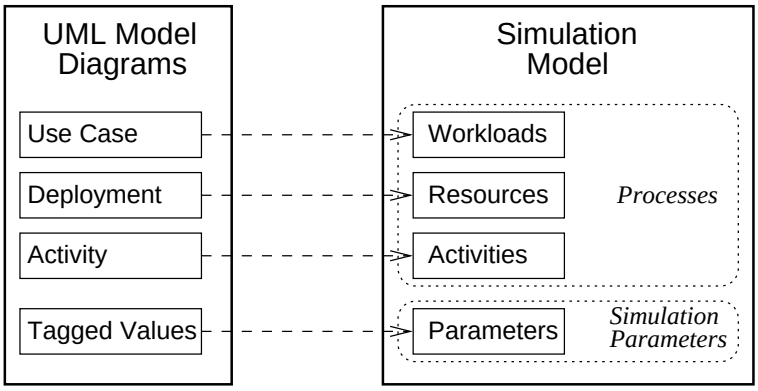


Figure 4.2: Mapping UML elements into simulation processes

improvements in the speed of the UML-Ψ tool.

We given a high-level overview of the simulation model in Section 4.3; we will describe all the different kind of processes and their interactions. In Section 4.4 we describe in detail the performance model. The mapping from performance domain concepts to the UML notation will be given in Chapter 5, and the tagged value types used to annotate the diagrams will be explained in Section 5.5.

4.3 Overview of the Performance Model

The simulation model has a structure which is very similar to that of the software model. In Fig. 4.2 we show how UML elements are mapped to simulation model elements. Simulation processes can be divided in three families, corresponding to processes representing workloads, resources and activities respectively. UML actors are translated into workloads; nodes of the deployment diagrams correspond to processes modeling resources, and action states in activity diagrams are translated into processes representing the actions. UML annotations are used as parameters for the simulation model.

In Table 4.1 we list all the processes types which can be used in the simulation model. The table shows the process names in the left column, and the list of other processes with which it can interact. A more detailed description of the behavior of each process will be given in Chapter 6 using a Pascal-like notation.

An informal description of each process is given as follows:

OpenWorkload This process performs an endless loop generating an infinite stream of processes of type OpenWorkloadUser. After each process is created, it pauses for a random amount of simulated time to simulate the interarrival time between requests.

OpenWorkloadUser Represents a request arriving to the software system. It triggers the activation of one of the use cases associated to the actor representing the

Process Name	Interacts with
OpenWorkload	OpenWorkloadUser
OpenWorkloadUser	CompositeAction
ClosedWorkload	ClosedWorkloadUser
ClosedWorkloadUser	CompositeAction
SimpleAction	Any Action, ActiveResource
CompositeAction	Any Action
ForkAction	Any Action
JoinAction	Any Action
AcquireAction	Any Action, PassiveResource
ReleaseAction	Any Action, PassiveResource
ActiveResource	SimpleAction
PassiveResource	AcquireAction, ReleaseAction

Table 4.1: The different kinds of simulation processes used to define the process-oriented simulation model

workload, and then terminates. The use case is modeled as a CompositeAction process.

ClosedWorkload This process creates a fixed population of system requests, represented by processes of class ClosedWorkloadUser. All the requests are activated, and the ClosedWorkload terminates.

ClosedWorkloadUser Represents a request of service to the software system. The request belongs to a finite set of requests generated from the same closed workload. This process behaves like a OpenWorkloadUser, except that after the activated use case terminates, it waits a random amount of time and then activates another use case.

SimpleAction Represents a computation performed on the system. The computation may be repeated a number of time, and may request service from an active resource. It interacts first with the associated active resource, and then with one of the successor actions.

CompositeAction Models a composite computation, made of a number of sub-computations. Its behavior consists on activating the first (root) substep.

ForkAction Represents the start of a parallel execution of two concurrent computations. Its actions consists on activating all the successor actions.

JoinAction Represents the end of a parallel computation. This process waits for all the predecessor actions to complete, and then activates one of its successors.

SimAcquireAction Represents an action requiring a passive resource. It interacts with the passive resource before activating one of the successor actions.

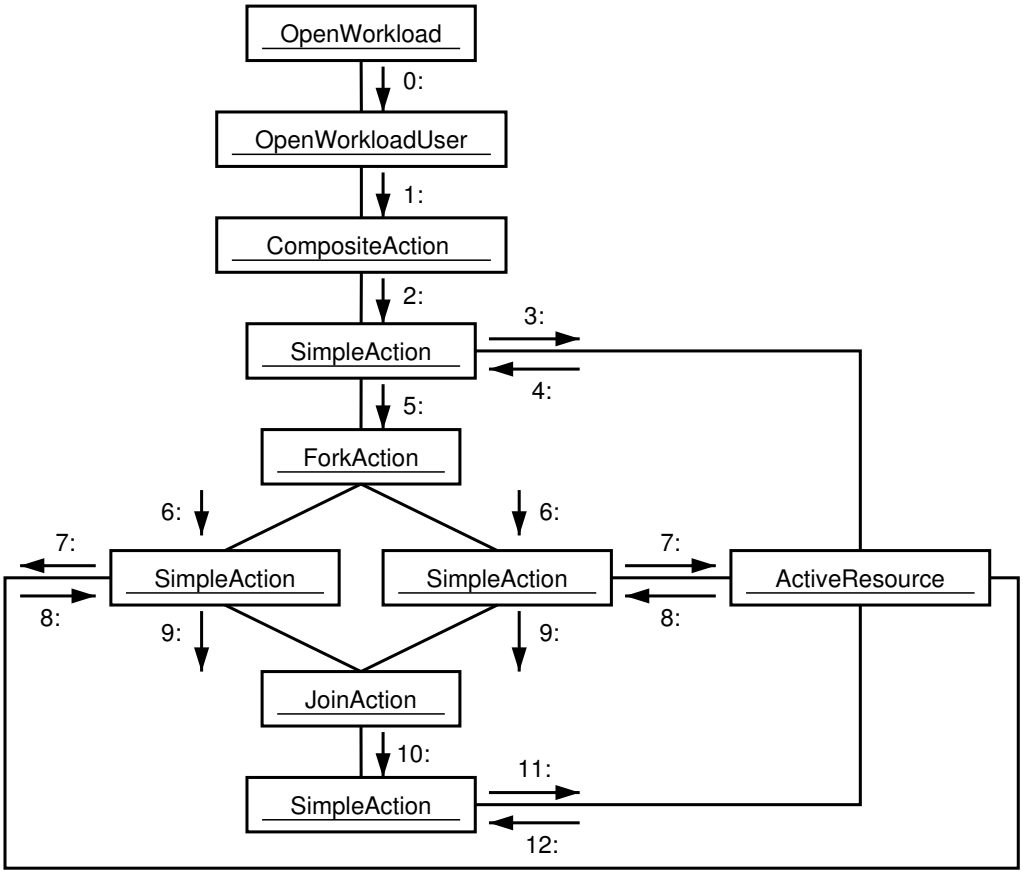


Figure 4.3: Collaboration diagram showing an example of simulation model instance

SimReleaseAction Represents an action releasing a passive resource. It interacts with the passive resource before activating one of the successor actions.

ActiveResource This process represents an active resource. It waits for requests, satisfying them according to its scheduling policy.

PassiveResource This process represents a passive resource. It waits for requests, checking wether they can be satisfied; thus, it interacts with processes of class simAcquireAction and SimReleaseAction.

We show in Fig. 4.3 a UML collaboration diagram describing an example on how the simulation model is instantiated. Nodes in the collaboration diagram are instances of simulation processes. The messages (arrows) show the order in which processes are activated.

The first simulation processes to start are those associated with workloads. They will periodically create and activate other processes representing workload users, which in turn select and activate one use case. The composite action process associated to the use case will cause the first action of the associated activity diagram to

be executed. At this point, actions will be activated in the correct order according to their successor relationship. In the case of fork nodes, they cause all successor actions to start simultaneously. Each action may require service from an active resource, or may require some tokens from a passive resource. In either case, the process representing the resource is activated to request service; it will then pass control back to the requesting step by activating it when service is completed.

All the different kind of simulation processes and their exact behavior will be explained in Chapter 6.

4.4 The Performance Model

An UML representation of the performance model we propose is given in Fig. 4.4. The structure of similar to the one described in [75]; however, there are some differences which will be analyzed in Sec. 4.6.

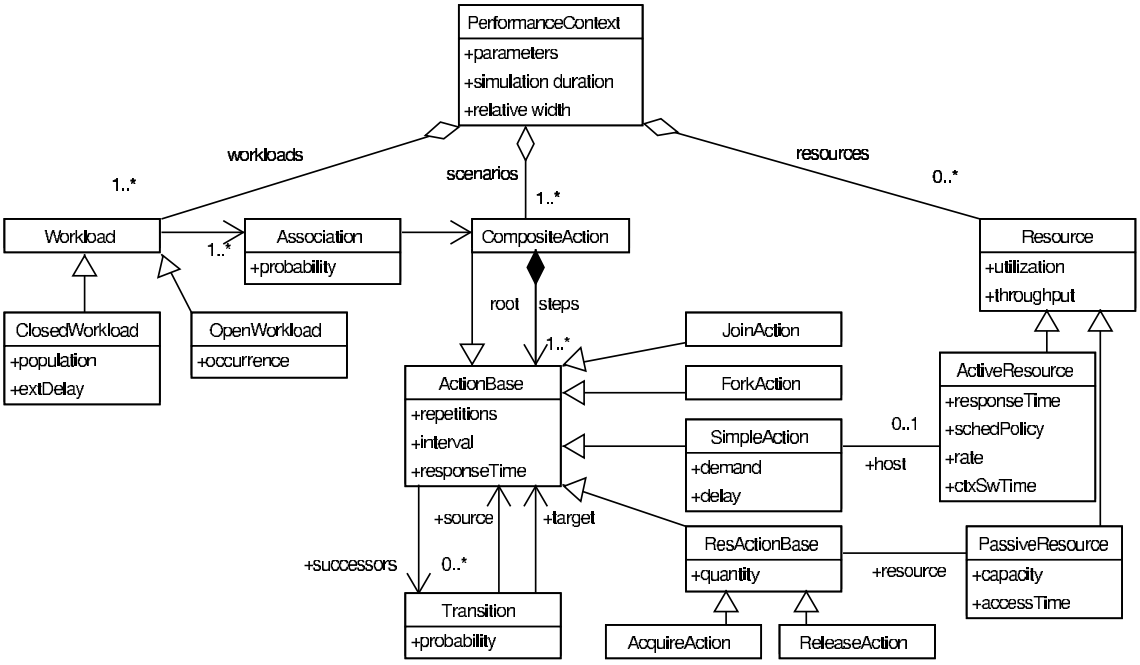


Figure 4.4: Structure of the simulation performance model

The model is explained as follows. We suppose that the software system is driven by a set of external workloads. Each workload is a stream of of users requesting service to the system. Users may be either physical people interacting with the system (e.g., users accessing a e-commerce site) or other kinds of requests or stimuli (e.g., sensors readouts triggering some system activity). In the following, we will use the terms “requests” or “users” interchangeably. Workloads may be either open or closed, depending if the request population is unlimited or fixed.

Each workload is connected to a set of scenarios. A scenario is a sequence of computational steps or activities with a given starting and ending point. Conceptually, a scenario is defined in [75] as a response path whose endpoints are externally visible.

Scenario steps are computational activities requiring service from resources. We distinguish between two kinds of requests: those for active resources (e.g., processors), and those for some amount of a passive resource (e.g., memory). Actions are connected with a predecessor-successor relationship; each action may have multiple predecessors and/or multiple successors, and may be repeated any number of time. Also, actions have a hierarchical structure in that a composite action may be represented, at a deeper level of detail, as a set of sub-actions.

As already introduced, the simulation model may include two kinds of resources. Active resources are modeled as servers, having a given processing rate and scheduling policy; active resources include processors and network links. Passive resources can be seized and released in fixed amounts by actions. Each passive resource exists a limited quantity. Actions may request and release an arbitrary amount of resource. If the requested amount is not available, the request action is suspended until it can be satisfied.

4.4.1 Performance Context

The simulation performance model is represented by an instance of class PerformanceContext, which acts as a container for the other model components. A performance model is made of a set of *workloads* driving *scenarios*, and a set of available *resources*. It is necessary to specify at least one workload and one scenario, as performance modeling would be pointless without any of them.

Associations

workloads	The set of workloads driving the system.
scenarios	The set of scenarios which can executed by users.
resources	The set of available active and passive resources.

Attributes

parameters	The list of definitions for variables used as values int simulation parameters. In general, the modeler may specify annotations using undefined variables. When the simulation model is executed, these free variables are instantiated with values specified in this attribute.

simulation duration The maximum duration of the simulation. The simulation program stops when it reaches the (simulated) time specified in this attribute.

confidence width The desired relative width of the confidence intervals for the computed performance results. Results are computed as confidence intervals (at 90% confidence level, by default); this attribute specifies the relative width of those intervals at which the simulation stops.

4.4.2 Workload

Workloads are modeled as instances of the Workload class. A workload is a stream of requests arriving to the system. We model two kind of workloads (see Fig. 4.5). If the number of requests is unlimited, it is called an open workload and is represented by class OpenWorkload. If the population of requests is fixed, that is there is a fixed number of requests each returning to the system once completed, then the workload is called a closed workload and is represented by class ClosedWorkload. The users of a closed workload behaves in the following way: they request service to the system, and when the service is completed they wait for some time (called think time) before starting another interaction.

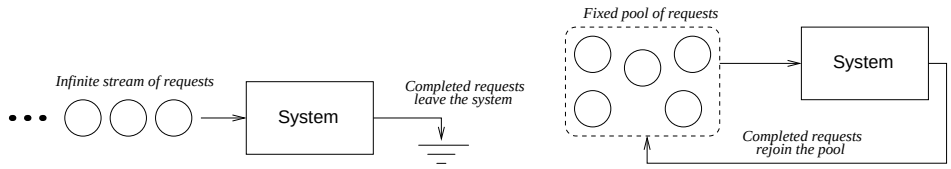


Figure 4.5: Schematic representation of open 4.5(a) and closed 4.5(b) workloads

Upon arriving to the system, all the requests activate a scenario, which is modeled as a composite action. A workload may be associated to multiple scenarios; each scenario is given a probability to be chosen. Workload-generated users randomly choose which scenario to execute according to these probabilities.

Associations

association The link from a workload to the set of scenarios it can activate. Each workload can be associated to multiple scenarios; every time a new request is generated, it will choose and activate one of them, randomly chosen.

4.4.3 Association

This class represents a link between a workload and a scenario which can be activated by that workload.

Attributes

probability The probability given to this association to be chosen by a request. If the workload is associated with only one scenario, then this value is ignored. If the workload is associated with multiple scenarios (via multiple associations), then each workload request will choose the scenario to execute according to the probability specified in the association. The constraint that the sum of all the probabilities of associations from the same workload must be 1.0 must hold.

Associations

workload The workload originating requests.

compositeAction The scenario (set of actions) the workload may execute if this association is chosen.

4.4.4 OpenWorkload

This class denotes a workload made of an infinite stream of requests arriving to the system.

Attributes

occurrence The occurrence pattern of workload users. This is generally expressed as a random variable of given distribution; the modeler specifies both the name of the distribution and its parameters.

4.4.5 ClosedWorkload

This class denotes a workload made of an a fixed number of requests circulating through system. After a request is serviced, it spends a given amount of time (think time) outside the system, before starting another interaction.

Attributes

population The total number of requests for this workload.

extDelay The external delay (think time) between successive requests from the same workload user.

4.4.6 ActionBase

This class represents a request for a resource. The request may be repeated many times, with an interval between repetitions. The average response time, which is one of the simulation results, is the average interval between the time the action starts and the time it finishes. Response time includes the time spent waiting for the resource to become available, and any other delay the action incurred. Actions may have a set of successors, that are other actions which may be activated after the current one finishes.

Attributes

repetitions	The number of times this action has to be repeated.
interval	The interval between repetitions. This is ignored unless repetitions is greater than one.
responseTime	The computed average response time of this action, measured as the average delay between the time the action starts and the last repetition finishes.

Associations

successors	The list of outgoing transitions from this action.

4.4.7 Transition

A transition associates a source action with a target action. The meaning of a transition is the following: when the last repetition of the source action terminates, the target action is immediately started. When a simple action or a composite action has multiple outgoing transitions, then the execution path can follow one of multiple routes. In this case, one transition is randomly selected according to its probability, and execution continues with the target action of the selected transition. Multiple outgoing transitions are treated differently when they originate from a fork action; in this case, all the target actions are started at the same time, representing the fact that the computation split in multiple concurrent execution paths.

Attributes

probability	The probability to select the transition. This attribute is ignored if the source action is a ForkAction, or if the source action has only one outgoing transition.

Associations

source	The action from which this transition originates.
target	The action to which this transition leads.

4.4.8 CompositeAction

A composite action is a container for a set of actions. It is used to model situations in which a behavior can be decomposed in simpler actions. One of the subactions is designated as the root step, which is the first action invoked when executing the composite action. The behavior of a composite action is the execution of its sub steps.

Associations

steps	The set of sub-actions making this composite action.
root	The initial action executed.

4.4.9 SimpleAction

This represents a processing step, which is modeled as a request for service from an active resource.

Attributes

demand	The service demand for this action, expressed as a random variable. If this action is associated with a host (processor), then service is requested to that host. If this action is not associated with any host, then the service is simulated by making the action wait for an amount of time equal to the value of this attribute.
delay	An inserted delay within this action, which does not correspond to any service request. It may be used to model user interaction.

Associations

host	The active resource on which this action is executed, that is, the host from which service will be requested. If missing, the action is assumed to be executed on a dedicated processor with processing rate of 1. There is one of such dedicated processors for each action in the UML model.

4.4.10 JoinAction, ForkAction

Fork and join actions are used to model concurrent execution of multiple threads. A fork action takes no time, and causes immediate activation of all successor actions, that are, all target actions of the outgoing transitions. A join action takes no time as well, and waits for all predecessor actions to terminate before activating the successor step.

4.4.11 Resource

This class represents an active or passive resource which is available in the system.

Attributes

utilization The computed utilization of this resource. This value is computed by the simulation program in different ways, depending whether the resource is active or passive. The exact definition of utilization is given below.

4.4.12 Active Resource

Active resources represent processors executing jobs. Each job has a service demand expressed in seconds of processing time for the reference processor; the reference processor is a processor which requires 1 second of simulation time to satisfy a service demand of 1 second.

The utilization of an active resource is computed as follows. For any (simulated) time $t \geq 0$, let $B(t)$ be defined as follows:

$$B(t) = \begin{cases} 1 & \text{if the resource is busy at time } t \\ 0 & \text{otherwise} \end{cases}$$

Then, the utilization over the simulated time interval $[0, T]$ of length T is defined as:

$$U(T) = \frac{1}{T} \int_0^T B(t)dt$$

If the system is stationary, then the mean utilization U is defined as:

$$U = \lim_{T \to \infty} U(T) = \lim_{T \to \infty} \frac{1}{T} \int_0^T B(t)dt$$

Attributes

responseTime The computed average response time of the resource. The response time is defined as the difference between the time service completes and the time the request for service is made. Response time include queuing and context-switch delays.

schedPolicy The scheduling policy for this resource. It defines how the resource is shared among multiple, concurrent requests. Supported values for this attribute are: "FIFO" (First-in-First-out), "LIFO" (Last-in-First-out), "PS" (Processor Sharing).

rate The processing rate of this resource. This denotes the speed of this processor with respect to a reference processor. The reference processor requires 1 second of simulated time to satisfy a service demand of 1 second. An arbitrary processor will require $1/rate$ seconds of simulated time to satisfy a service demand of 1 second.

ctxSwTime The time needed to perform a context switch. This has not effect if the scheduling policy of this processor has been set to Processor Sharing ("PS").

4.4.13 Passive Resource

A passive resource is a resource which needs to be acquired during the execution of an operation. Examples of passive resources include memory or energy. A passive resource consists of a finite number of items (or tokens), which can be acquired and released. Each passive resource has a maximum capacity, which is the maximum number of tokens it can hold. Requests can be made for an arbitrary number of tokens, not exceeding the total resource capacity. Once a request is granted, the number of available tokens is lowered accordingly. Requests are served in strict FIFO queue. The first request of the queue is examined; if it can be satisfied with the residual number of tokens then it can proceed. Otherwise, it remains in the queue at its current position. The request will be satisfied as soon as enough tokens are released into the resource.

The utilization of a passive resource is defined in the following way. Let us suppose that the resource has a maximum capacity of $N > 0$ tokens. For every $t \geq 0$ we define $A(t)$ as the number of tokens which are available at time t. A request for $0 \leq K \leq N$ tokens at time $s \geq 0$ works in the following way:

- If $K \leq A(s)$ then we set $A(s) := A(s) - K$ and the request succeeds, ie, the requesting action completes.
- If $K > A(s)$, then the request is put on hold in a FIFO queue.

Every time some tokens are released, the first request from the queue is examined. If it can be satisfied, the resource is granted (and $A(t)$ is updated accordingly) and the request terminates.

Note that the maximum number of tokens is always N, which means that if more tokens are released than those acquired (which is allowed), then the excess is lost. The utilization $U(T)$ over the simulated time interval $[0, T]$ of length T is defined as

$$U(T) = 1 - \frac{1}{NT} \int_0^T A(t)dt$$

If the system is stationary, then the mean steady-state utilization is defined as

$$U = \lim_{T \to \infty} U(T) = 1 - \lim_{T \to \infty} \frac{1}{NT} \int_0^T A(t)dt$$

Attributes

capacity The initial and maximum number of available instances (tokens) of this resource. This must be a positive integer. The number of instances of the resource will be always guaranteed to be in the range [0..*capacity*].

accessTime The access time of the resource. This is the time needed to physically obtain the resource, once it is granted. Releasing a resource takes no time.

4.4.14 ResActionBase

This class represents actions requesting or releasing a passive resource. The user specifies the amount to request/release, and on which resource the action is being executed.

Attributes

quantity The amount of resource to acquire or release.

Associations

resource The resource to acquire or release.

4.4.15 AcquireAction

This class models a request for acquiring some instances of a passive resource.

Attributes

quantity (inherited from ResActionBase). The amount of resource to acquire. Setting this value higher than the resource's capacity results in this action to block forever, as enough resource will never be available. If the requested quantity is less than or equal to the current available number of tokens, then the

request is satisfied, the available resource counter is updated and the step proceeds. If the request is for more than what currently available, the request is put on a FIFO queue.

4.4.16 ReleaseAction

This class models the release of tokens of a passive resource.

Attributes

quantity (inherited from ResActionBase). The amount of resource to release. Actions may release an arbitrary quantity, which will be added to the resource available tokens, constrained by the maximum resource capacity. Releasing a resource causes the following behavior. If there are no requests waiting for the resource, then the action terminates. If there are requests waiting in the queue, then the first request is taken (in FIFO order). If it can be satisfied with the currently available resource, then it is reactivated. If the request can not be satisfied, it is put again in front of the queue. Note that if the request in front of the queue can not be satisfied, it blocks all the remaining requests.

4.5 Simulation Results

The computed simulation results are the following:

- Utilization of resources;
- Throughput of resources;
- Mean execution times of actions and use cases.

Resource utilization and throughput are stored in the PAutilization and PAthroughput tags respectively. Mean execution times of use cases is stored in the PArespTime tag (a use case is seen by the simulator as a composite action, whose body is the associated activity graph); details will be given in Chapter 5.

At the moment the type of simulation results which can be computed are hard-coded into the simulation prototype tool, and users cannot currently define new measures of interest. Definition of a suitable notation to allow the software modeler to define different types of performance parameters are outside the scope of this work, and are subject of ongoing research.

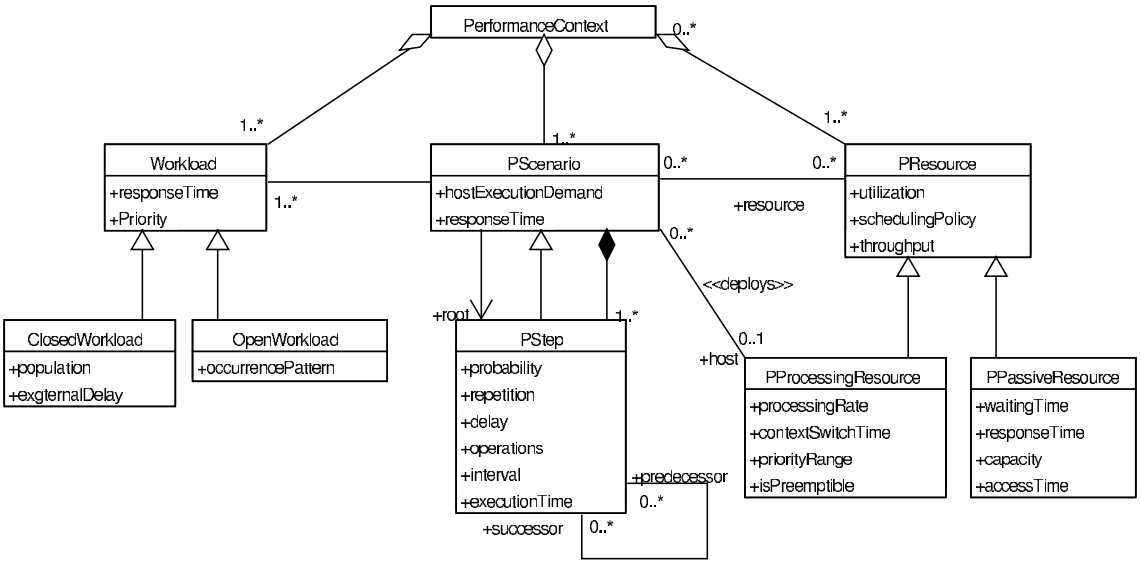


Figure 4.6: The performance model from [75], p. 8–4

4.6 Differences with the UML Performance Profile

We highlight now the differences between the performance model described above and the one proposed in the UML Performance Profile [75], which is reported in Fig. 4.6. We concentrate on the structural differences between that and the model we describe on Fig. 4.4, p. 46.

Modeling steps As can be seen, the main difference lies in the model of the processing steps is represented. The UML Profile defines a PScenario class from which a PStep class is derived, thus every step is a scenario. This is done to represent the fact that, at a deeper level of detail, a step could be modeled as a whole scenario, that is, a sequence of multiple sub-steps. However, making PStep inherit from PScenario would mean that every step is a scenario, which is not true in general.

We choose a different structure to model the hierarchy of actions, and keep atomic steps (actions) and scenarios (composite actions) as separate entities. We apply the Composite Pattern [41] to reflect the hierarchical nature of the processing steps. The root class of all actions is, in our case, ActionBase. This class contains the common features among all king of actions. From this class we derive, among others, atomic actions (SimpleAction class), and composite actions (CompositeAction class). The latter, in turn, is associated with a set of objects derived from ActionBase, which means that the content of a composite action can include both simple actions and other composite actions.

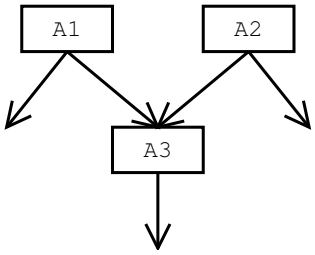


Figure 4.7: Actions $A1$ and $A2$ may have transitions with different probabilities to $A3$

This choice makes the construction of the simulation model easier, because each different kind of action (SimpleAction, CompositeAction, ForkAction and so on) is modeled as different simulation process types. Each process has its specific behavior.

Associating probabilities to transitions There is another difference between our proposal and the UML Performance Profile, namely the way we model nondeterminism in activity diagrams. Observe that in Fig. 4.6 there is a PAprob attribute associated with processing steps. This represents the fact that when a step has multiple successors, the thread of control may evolve nondeterministically by choosing one of them according to its probability. We model exactly the same situation, except that we associate the probability with action transitions, rather than with action themselves. The reason is the following. It is possible that one action has multiple predecessors. Each of those predecessors may choose to execute that action with different probability; however, this cannot be modeled with the UML Profile performance model, as the probability is associated with the action, and not with transitions.

Fig. 4.7 illustrates the problem. According to the UML Performance Profile, it is not clear whether actions $A1$ and $A2$ can evolve to action $A3$ with different probabilities. Our approach makes it clear to allow multiple predecessor actions to evolve toward the same action with different probabilities.

Modeling workloads Finally, we use use case diagrams to model workloads and their association with scenarios, whereas the UML Performance Profile uses the first stimulus of a scenario, the stimulus being stereotyped as a workload. We decided to make the interaction explicit by using use case diagrams (as proposed also in [33]) for different reasons. First, software architects often include use case diagrams in their software models, and they are usually one of the first diagrams developed; hence, it seems reasonable to make use of them. Second, the semantics of use case diagrams, although only informally specified by the UML standard [74], can be naturally extended to represent workloads.

5

Mapping Performance Model Elements to UML

We now illustrate how the performance domain concepts previously described can implemented in UML models. We define a UML profile which allows the software modeler to specify performance-related informations in UML diagrams.

The UML Performance Profile describes two different set of diagrams which can be used for performance evaluation purposes. These two different approaches are called *collaboration-based* and *activity-based* approach, and are based respectively on collaboration and on activity diagrams. The activity-based approach has the advantages of allowing easier modeling of hierarchical scenarios. Also, activity diagrams are usually more readable and more easy to understand than collaboration diagrams. For these reasons our approach is based on activity diagrams.

The proposed UML profile is based on a set of stereotypes and tagged values. For each stereotype we specify which UML element(s) it applies to; also, the list of the tagged values it provides will be shown. Each tagged value will then be described in detail.

Basically, annotated UML diagrams can be considered a high-level graphical simulation language. They not only describe the structure of the software model, but also describe the structure of the simulation model as there is an almost one-to-one mapping between UML elements and simulation processes. Such mapping will be described in detail in Chapter 6.

5.1 The Performance Context

A performance context is modeled as a UML model. Tagged values are used to specify simulation parameters and the file used to load definitions.

Stereotype	**Base Class**	**Tagged Values**
≪PAContext≫	Model	paramFileName simDuration confRelWidth

Tag definitions:

Tag	Type	Multiplicity	Domain Attribute Name
paramFileName	String	[0..1]	PerformanceContext::parameters
simDuration	Real	[0..1]	PerformanceContext::simulation duration
confRelWidth	Real	[0..1]	PerformanceContext::confidence width

In addition to the above tag definitions, the Performance Context allows users to specify a PAprob tag associated to Associations in use case diagrams (see 4.4.3) and Transitions in activity diagrams (see 4.4.7).

Tag	Type	Multiplicity	Domain Attribute Name
PAprob	Real	1	Association::probability
PAprob	Real	1	Transition::probability

5.2 Modeling Workloads

Workloads are modeled using UML use case diagrams. Actors represent workloads, and use cases represent scenarios which can be activated by the requests. Two different stereotypes, ≪ OpenWorkload ≫ and ≪ ClosedWorkload ≫, are defined to denote open and closed workloads respectively. These stereotypes are associated with actors.

Open workload: ≪ OpenWorkload ≫

This stereotype denotes an open workload.

Stereotype	Base Class	Tagged Values
≪ OpenWorkload ≫	Actor	PAoccurrence

Tag definitions:

Tag	Type	Multiplicity	Domain Attribute Name
PAoccurrence	RTarrivalPattern	[0..1]	ClosedWorkload::occurrence

Closed Workload: ≪ ClosedWorkload ≫

This stereotype denotes a closed workload.

Stereotype	Base Class	Tagged Values
≪ ClosedWorkload ≫	Actor	PApopulation PAextDelay

Tag definitions:

Tag	Type	Multiplicity	Domain Attribute Name
PApopulation	Integer	[0..1]	ClosedWorkload::population
PAextDelay	PAperfValue	[0..1]	ClosedWorkload::extDelay

5.3 Modeling Resources

Resources are modeled using UML deployment diagrams. Each diagram represents some resources which are available in the system. Resources correspond to node instances.

Active Resource: ≪PAhost≫

This stereotype denotes an active resource.

Stereotype	Base Class	Tagged Values
≪PAhost≫	Node	PAutilization PAthroughput PAschedPolicy PActxSwT PArate

Tag definitions:

Tag	Type	Multiplicity	Domain Attribute Name
PAutilization	mean	[0..1]	Resource::utilization
PAthroughput	mean	[0..1]	Resource::throughput
PAschedPolicy	Enumeration { 'LIFO', 'FIFO', 'PS' }	[0..1]	ActiveResource::schedPolicy
PActxSwT	PAperfValue	[0..1]	ActiveResource::ctxSwTime
PArate	Real	[0..1]	ActiveResource::rate

Passive Resource: ≪PAresource≫

This stereotype denotes a passive resource.

Stereotype	Base Class	Tagged Values
≪PAresource≫	Node	PAutilization PAthroughput PAcapacity PAaxTime

Tag definitions:

Tag	Type	Multiplicity	Domain Attribute Name
PAutilization	mean	[0..1]	Resource::utilization
PAthroughput	mean	[0..1]	Resource::throughput
PAcapacity	Real	[0..1]	PassiveResource::capacity
PAaxTime	PAperfValue	[0..1]	PassiveResource::accessTime

5.4 Modeling Scenarios

Scenarios are modeled as UML activity diagrams.

Simple Action: ≪PAstep≫

This stereotype denotes a simple (atomic) step.

Stereotype	Base Class	Tagged Values
≪PAstep≫	ActionState	PArep PAinterval PAdemand PAhost PAdelay PArespTime

Tag definitions:

Tag	Type	Multiplicity	Domain Attribute Name
PArep	Integer	[0..1]	ActionBase::repetitions
PAinterval	PAperfValue	[0..1]	ActionBase::interval
PAdemand	PAperfValue	[0..1]	SimpleAction::demand
PAhost	String	[0..1]	SimpleAction::host
PAdelay	PAperfValue	[0..1]	SimpleAction::delay
PArespTime	mean	[0..1]	ActionBase::responseTime

Composite Action: ≪PAcompositeStep≫

This stereotype models a composite action (step). This is necessary due to a limitation of the UML CASE tool used (ArgoUML) which at the moment does not support composite action states to be defined explicitly.

Stereotype	Base Class	Tagged Values
≪PACompositeStep≫	ActionState	PArep PAinterval PArespTime

Tag definitions:

Tag	Type	Multiplicity	Domain Attribute Name
PArep	Integer	[0..1]	ActionBase::repetitions
PAinterval	PAperfValue	[0..1]	ActionBase::interval
PArespTime	mean	[0..1]	ActionBase::responseTime

Resource Acquire: ≪GRMacquire≫

This stereotype models a request for a passive resource.

Stereotype	Base Class	Tagged Values
≪GRMacquire≫	ActionState	PAresource PAquantity

Tag definitions:

Tag	Type	Multiplicity	Domain Attribute Name
PAresource	String	1	ResActionBase::resource
PAquantity	PAperfValue	[0..1]	ResActionBase::quantity

Resource Release: ≪GRMrelease≫

This stereotype models the release of a passive resource.

Stereotype	Base Class	Tagged Values
≪GRMrelease≫	ActionState	PAresource PAquantity

Tag definitions:

Tag	Type	Multiplicity	Domain Attribute Name
PAresource	String	1	ResActionBase::resource
PAquantity	PAperfValue	[0..1]	ResActionBase::quantity

5.5 Tagged Value Types

The UML Performance Profile proposes the use of the Tag Value Language to express tag values; Tag Value Language (TVL) is a subset of the Perl [97] language. The use of some high-level language for tags helps the software modeler, which can use complex expressions and variables in value specifications. We decided to use the whole Perl language in order to express tag values. The reason is that the freely available Perl implementation [34] ships with an interpreter library, which allows the embedding of a Perl interpreter in a program. As it is easier and more convenient to use the full language interpreter rather than developing an ad-hoc one for a subset, the UML-Ψ tool described in III makes use of the Perl interpreter library to parse tag values.

In describing tagged value types we use the following syntactical conventions:

- Strings and characters in bold face are literals (e.g., **"element"**). Note that square brackets and commas are literals, and are used to define a Perl array.
- Strings in angular brackets are non terminals (e.g., < element >).
- A vertical bar | denotes alternative choices.

5.5.1 PAperfValue

This data type is used to express performance annotations which can be given as random variables with some suitable distribution.

An element of type PAperfValue is a Perl vector of three elements. The first element is used to describe whether a given parameter is assumed (for example, based on experience), predicted (for example, computed by a performance tool) or measures. This distinction is only used as a mnemonic aid for the modeler, as it is ignored by the performance tool. The second element of the PAperfValue vector is the keyword "dist", meaning that the value represents a distribution. The third element actually describes the type of distribution.

$$
\begin{aligned}
< \text{PAperfValue} > &:= [< \text{SourceModifier} > \textbf{,“dist”,} < \text{PDFstring} >] \\
< \text{SourceModifier} > &:= \textbf{“assm”}|\textbf{“pred”}|\textbf{“msrd”} \\
< \text{PDFstring} > &:= [< \text{constantPDF} > | < \text{uniformPDF} > \\
&\quad | < \text{exponentialPDF} > | < \text{normalPDF} > \\
&\quad | < \text{ltnormalPDF} > | < \text{rtnormalPDF} > \\
&\quad | < \text{lrtnormalPDF} >] \\
< \text{constantPDF} > &:= < \text{real} > |\textbf{“constant”,} < \text{real} > \\
< \text{uniformPDF} > &:= \textbf{“uniform”,} < \text{real} > \textbf{,} < \text{real} > \\
< \text{exponentialPDF} > &:= \textbf{“exponential”,} < \text{real} > \\
< \text{normalPDF} > &:= \textbf{“normal”,} < \text{real} > \textbf{,} < \text{real} > \\
< \text{ltnormalPDF} > &:= \textbf{“normal”,} < \text{real} > \textbf{,} < \text{real} > \textbf{,} < \text{real} > \\
< \text{rtnormalPDF} > &:= \textbf{“normal”,} < \text{real} > \textbf{,} < \text{real} > \textbf{,} < \text{real} > \\
< \text{lrtnormalPDF} > &:= \textbf{“normal”,} < \text{real} > \textbf{,} < \text{real} > \textbf{,} < \text{real} > \textbf{,} < \text{real} >
\end{aligned}
$$

The currently available distributions are the following:

Constant This represents a constant value, and may be expressed as a real number alone or preceding the value with the keyword "constant":

$$< \text{constantPDF} >:=< \text{real} > |\textbf{“constant”,} < \text{real} >$$

Exponential This represents a negative exponential distribution with given mean:

$$< \text{exponentialPDF} >:= \textbf{“exponential”,} < \textit{mean} >$$

Uniform This represents a uniform distribution over the interval $[a, b]$:

$$< \text{uniformPDF} >:= \textbf{“uniform”,} < a > \textbf{,} < b >$$

Normal This represents the normal distribution with given mean and standard deviation:

$$< \text{normalPDF} >:= \textbf{“normal”}, < \textit{mean} > , < \textit{stddev} >$$

It should be noted that normally distributed random variables may assume positive and negative values. This can cause problems if this distribution is used to express the duration of some action. For this reason, users are expected to use one of the Left or Left-and-Right Truncated Normal distributions to express durations, setting the left truncation point at some non negative value.

Truncated Normal These distributions represent truncated normal distribution with given mean and standard deviation. The normal distribution may be truncated to the left, to the right or both.

$$\begin{aligned}
< \text{ltnormalPDF} > &:= \textbf{“ltnormal”}, < \textit{mean} > , < \textit{stddev} >, < \textit{left_tr_point} > \\
< \text{ltnormalPDF} > &:= \textbf{“rtnormal”}, < \textit{mean} > , < \textit{stddev} >, < \textit{right_tr_point} > \\
< \text{ltnormalPDF} > &:= \textbf{“lrtnormal”}, < \textit{mean} > , < \textit{stddev} >, \\
&\quad < \textit{left_tr_point} >, < \textit{right_tr_point} >
\end{aligned}$$

5.5.2 RTarrivalPattern

This type is used to define arrival patterns for system workloads.

$$\begin{aligned}
< \text{RTarrivalPattern} > &:= \text{'['} < \text{bounded} > | < \text{unbounded} > | < \text{bursty} > \text{']'} \\
< \text{bounded} > &:= \textbf{“bounded”}, < \text{real} > , < \text{real} > \\
< \text{bursty} > &:= \textbf{“bursty”}, < \text{PDFstring} > , < \text{integer} > \\
< \text{unbounded} > &:= \textbf{“unbounded”}, < \text{PDFstring} >
\end{aligned}$$

We have three types of arrival patterns:

Bounded The inter-arrival time between two successive requests is uniformly distributed, and the lower and upper bounds are given by the first and second parameter respectively.

Bursty Requests arrive in burst of given maximum size. The time between successive burst is given by the first parameter, and the maximum burst size is given by the second parameter. The actual burst size is computed for each arrival as a random variable uniformly distributed over the interval $0 \ldots$ Maximum Burst Size.

Unbounded Requests arrive one at a time, with inter-arrival time given by the parameter. The bounded arrival pattern is a special case of this one, with inter-arrival time uniformly distributed over an interval.

5.6 An Example

In this section we illustrate with a simple example how the annotations previously introduced can be applied to UML models. The proposed example is very similar to the one appearing in Sec. 8 of [75]. It involves a web-based video streaming application. We assume a constant population of 10 users. Each user selects a video to view using a web browser and the browser contacts the remote web server for the video to be sent back through the Internet. The video server triggers the activation of a video player on the workstation of the user before sending the stream of frames. The UML use case, deployment and activity diagrams are depicted in Fig. 5.1.

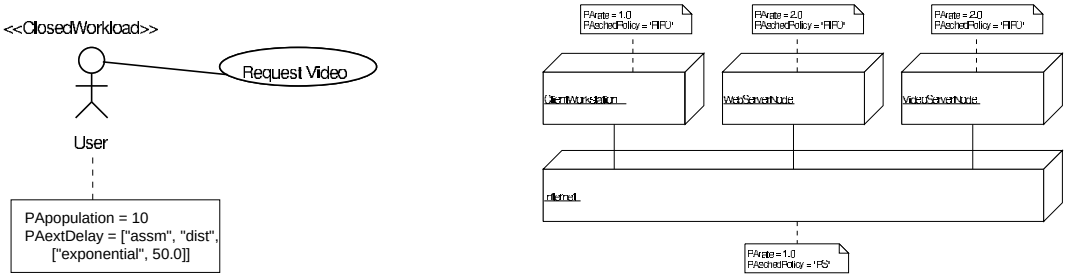
The annotations are shown in the figure inside note icons mainly for display purposes. In fact, CASE tools may provide different means of entering tagged values; for example, ArgoUML/Poseidon display them in the properties box on the bottom of the editing area, and are not shown directly into the UML model.

The use case diagram in Fig. 5.1(a) describes the workload driving the system. There is a single actor stereotyped as ≪ ClosedWorkload ≫ representing a closed workload. The workload consists of a population of 10 requests, and the assumed time each request spends waiting between end of service and start of the next service period is exponentially distributed with mean 50 seconds. The requests activate the activity diagram associated to use case "Request Video", which is shown in Fig. 5.1(c).

The activity diagram in Fig. 5.1(c) describes the computations performed when executing the "Request Video" use case. Each action is annotated with the name of the resource needed to service the request. In this example, all steps are stereotyped as ≪ PAstep ≫, and thus represent computations performed on some resource, whose name is specified in the PAstep tag. Those names correspond to node names in the deployment diagram. The service demands are specified as random variables of some given distribution. The "Send Video" action is repeated 100 times, since the PArep tag is assigned value 100.

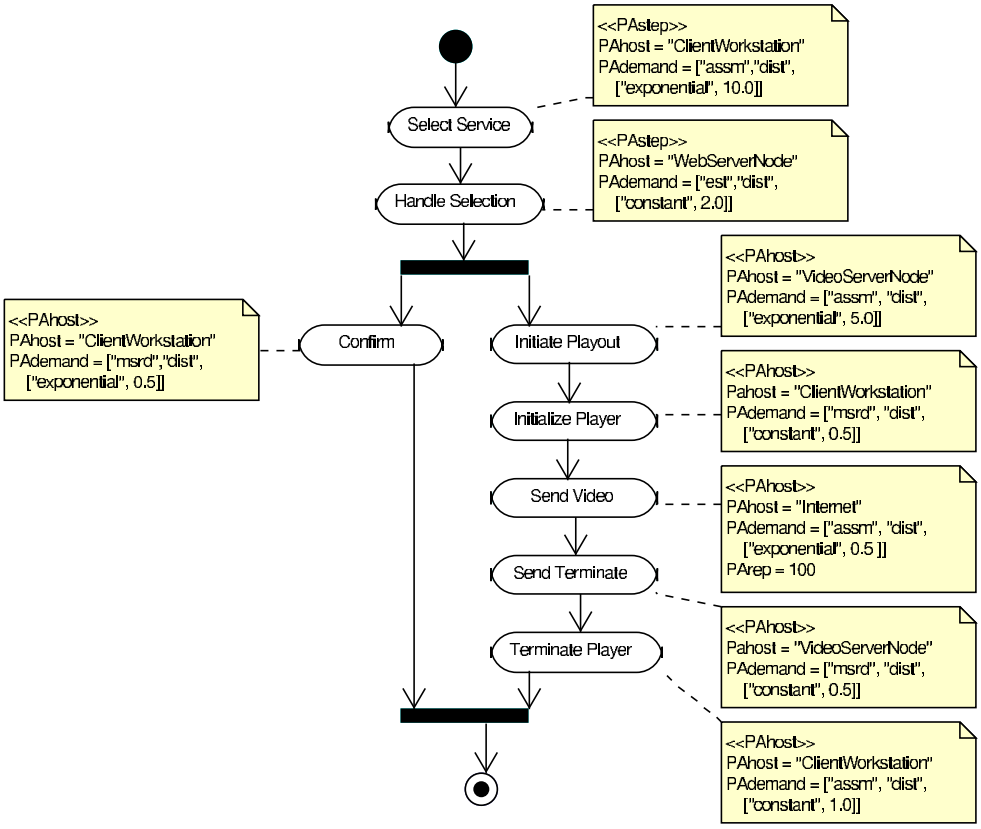
The deployment diagram in Fig. 5.1(b) is used to describe the resources available in the system. In this case, the resources are the following:

- A processor "ClientWorkstation", with FIFO scheduling policy and processing rate of 1.0 (meaning that a service demand of t seconds will be serviced in exactly t seconds of simulated time, in case of no resource contention);
- A processor "WebServerNode", with FIFO scheduling policy and processing rate of 2.0 (meaning that a service demand of t seconds will be serviced in $t/2$ seconds of simulated time, in case of no resource contention);
- A processor "VideoServerNode", with FIFO scheduling policy and processing rate of 2.0;
- A processor "Internet", representing the communication network. The network is modeled as an active resource with Processor Sharing scheduling policy



(a) Use Case Diagram

(b) Deployment Diagram



(c) Request Video Activities

Figure 5.1: UML representation of a Web video application.

and processing rate of 1.0.

6

The Simulation Model

In this chapter we describe how UML elements are mapped (translated) into simulation processes; we also show how the different types of simulation processes are implemented, by describing them in pseudocode. There is an almost one-to-one mapping between UML elements and simulation processes. This makes the model generation step almost straightforward; also, it is very easy to map simulation results from processes back to the UML elements they refer.

The simulation processes are described using an rather unconstrained Pascal-like notation. For simplicity and clarity we will use data types rather freely; also, we omit the instructions for computation of the performance measures. We denote with `X.some_tag` either the value of attribute `some_tag` of object `X`, or the value of the tag `some_tag` associated to the UML object `X`. Either way, the context will help in identifying which of the two alternatives hold.

6.1 Mapping the UML Model into the Performance Model

We describe briefly the transformation algorithm which is used for transforming annotated UML models into performance models. The algorithm is very simple, in that it builds a simulation model with almost the same structure as the software model.

The algorithm performs the following basic steps:

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

2. Each action in activity diagrams is translated into the appropriate 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 in a predecessor-successor relation, according to the transitions in the activity diagrams.

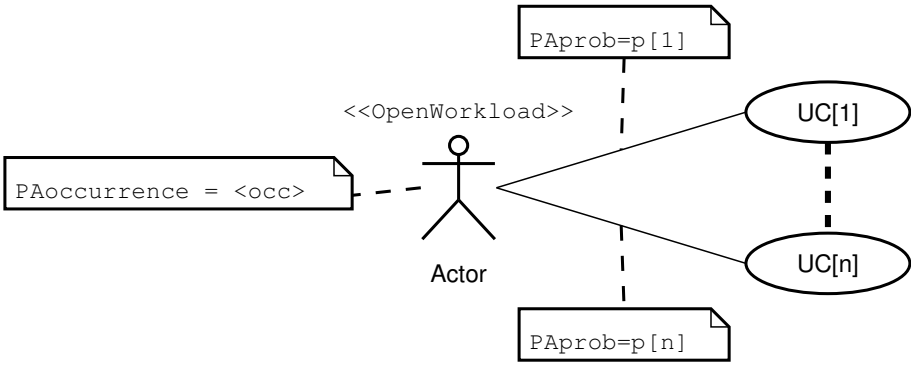
4. Finally, nodes in deployment diagrams are mapped into processing resources or passive resources, depending on their stereotype.

The pseudo code for the transformation algorithm is reported below. The algorithm refers to the simulation processes which will be described in more detail in the next sections.

```
{Process Use Case diagrams}
for all Use Case diagram U do
  for all Actor a ∈ U do
    if a is tagged as OpenWorkload then
      Ac ← Make new OpenWorkload(a) process
    else if a is tagged as ClosedWorkload then
      Ac ← Make new ClosedWorkload(a) process
    end if
    for all Use Case u associated with a do
      Sc ←Make new PScenario object
      Link Sc to Ac
      A ←Activity diagram associated with u
      {Process Activity diagram}
      for all Activity s ∈ A do
        if s is stereotyped as ≪PAstep≫ then
          if a is a simple action then
            P[a] ← new SimpleAction(a) process
          else
            P[a] ← new ComposieAction(a) process
          end if
        else if a is stereotyped as ≪GRMacquire≫ then
          P[a] ← new AcquireAction(a) process
        else if a is stereotyped as ≪GRMrelease≫ then
          P[a] ← new ReleaseAction(a) process
        else if a is a fork node then
          P[a] ← new ForkAction(a) process
        else if a is a join node then
          P[a] ← new JoinAction(a) process
        end if
        Add P[a] to Sc
      end for
      {Link activities}
      for all a, b ∈ A such that b is successor of a do
        Set P[b] as a successor of P[a]
      end for
    end for
  end for
```

end for
{Process Deployment diagrams}
for all Deployment diagram D **do**
  for all Node instance $n \in D$ **do**
    if n is tagged as $\ll$PAhost$\gg$ **then**
      $p \leftarrow$New `PRhost`(n) process
    else if n is tagged as $\ll$PAresource$\gg$ **then**
      $p \leftarrow$New `PResource`(n) process
    end if
  end for
end for

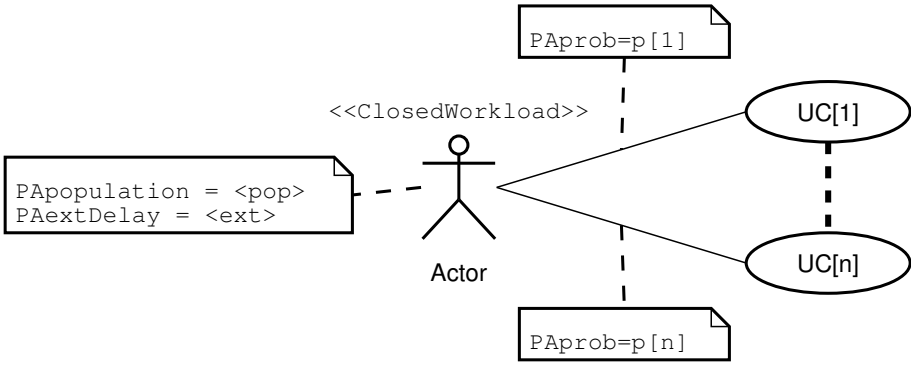
6.1.1 Workloads




```
process OpenWorkload( Actor A );
var
   u : OpenWorkloadUser;
begin
   while ( true ) do begin
      hold( A.PAoccurrence );
      u := new( OpenWorkloadUser( A ) );
      activate( u );
   end;
end;

process OpenWorkloadUser( Actor A );
var
   i  : integer;
   uc : UseCase;
begin
   { choose i in [1..n] with
   probability [A.p[1]..A.p[n]] }
   uc := new( CompositeAction( UC[i] ) );
   activate( uc );
end;
```

Figure 6.1: Mapping open workloads into simulation processes

The mapping of an open workload into a simulation process is shown in Fig. 6.1. Each actor which is stereotyped as ≪OpenWorkload≫ is mapped into a simulation process of type OpenWorkload. The behavior of this process consists of an infinite loop, during which it waits for an amount of time specified in the PAoccurrence tag and then creates a simulation process of type OpenWorkloadUser. The latter represents a workload user, that is, a request to the system. A workload user randomly chooses one of the use cases associated with the corresponding actor. A simulation process corresponding to the selected use case is created and activated.



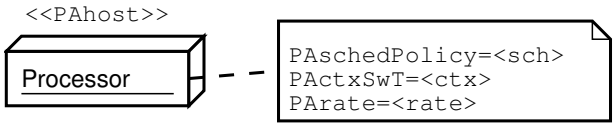

```
process ClosedWorkload( A: Actor )
var
   i : integer;
   u : ClosedWorkloadUser;
begin
   for i:=1 to A.pop do begin
      u := new( ClosedWorkloadUser( A ) );
      activate( u );
   end;
end;

process ClosedWorkloadUser( A: Actor )
var
   i  : integer;
   uc : UseCase;
begin
   while ( true ) do begin
      hold( A.PAextDelay );
      { choose i in [1..n] with
      probability [ A.p[1]..A.p[n] ] }
      uc := new( CompositeAction( UC[i] ) );
      activate( uc );
      { Wait for uc to terminate }
   end;
end;
```

Figure 6.2: Mapping closed workloads into simulation processes

The mapping of a closed workload to the simulation model is depicted in Fig. 6.2. Each actor which is stereotyped as ≪ ClosedWorkload ≫ is mapped into a simulation process of type ClosedWorkload. Its actions are simply to create as many ClosedWorkloadUser processes as specified in the PApopulation tag. Each process of type ClosedWorkloadUser represents a user (request); its action is to execute an infinite loop. First, the user waits for an amount of time corresponding to the PAextDelay tag (the "think time"). Then, it randomly selects one of the use cases associated with the actor. A simulation process corresponding to the selected use case is created and activated. When the simulation process executing the use case

terminates, the loop is iterated again and a new request appears to the system.

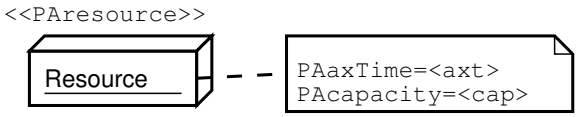
6.1.2 Resources




```
process ActiveResource( N: NodeInstance )
var
   j    : Action;
   jobq : queue of Action; { Queue of jobs }
begin
   while ( true ) do begin
      while ( not jobq.empty( ) ) do begin
         j := jobq.first( );
         jobq.remove_first( );
         hold( j.Pademand / N.PArate );
         activate( j );
      end;
      passivate( );
   end;
end;
```

Figure 6.3: Mapping active resources into simulation processes

The mapping between an active resource and the corresponding simulation processes is given in Fig. 6.3. Each UML node instance which is stereotyped as ≪PAhost≫ is translated into a simulation process of type Processor. For simplicity, only the "FIFO" and "LIFO" scheduling policies are considered in the simulation process of Fig. 6.3; implementation of the Processor Sharing ("PS") scheduling policy has been implemented in the UML-Ψ simulation tool, but is not reported here as it is only a variation of the same basic idea. Simulation processes of type Processor contain a queue of waiting jobs. Each job is extracted from the queue according to the scheduling policy (from the front of the queue in the case of "FIFO" scheduling, from the back of the queue in the case of "LIFO" scheduling). Each job is an action waiting for service (see the performance model on page 46). The active resource simulates the execution of the job by waiting for an amount of time equal to the service demand divided by the speed (processing rate) of the processor. At this point, the action corresponding to the completed job is resumed, and another job is extracted from the queue.

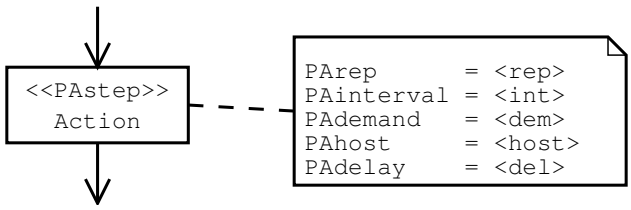



```
process PassiveResource( N: NodeInstance )
var
   r     : Action;
   reqq  : queue of Action; { Queue of requests }
   avail : integer;
begin
   avail := N.PAquantity;
   while ( true ) do begin
      while ( not jobq.empty( ) ) do begin
         r := reqq.first( );
         if ( r.quantity <= avail ) then begin
            avail := avail - r.quantity;
            reqq.remove_first( );
            hold( N.PAaxTime );
            activate( r );
         end;
      end;
      passivate( );
   end;
end;
```

Figure 6.4: Mapping passive resources into simulation processes

The mapping between a passive resource and the corresponding simulation processes is given in Fig. 6.4. A passive resource is modeled as a process with an associated queue of requests. Note that releasing tokens is performed by the process modeling a release action, which will be illustrated in a later section. A process of type Resource performs the following actions. It checks the first request in its queue. If the request can be satisfied, then the number of available tokens is decreased accordingly and the simulation process associated with the request is resumed. If the request cannot be satisfied, it is left in the queue.

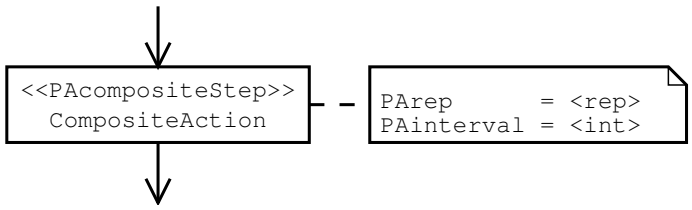
6.1.3 Actions




```
process SimpleAction( A: Action )
var
   i    : integer;
   next : ActionBase;
begin
   while ( true ) do begin
      hold( A.PAdelay );
      for i:=1 to A.PArep do begin
         if ( A.PAhost <> nil ) then
            { Request service to A.PAhost }
         else
            hold( A.PAdemand );
         if ( i < A.PArep )
            hold( A.PAinterval );
      end;
      { Activate next step }
      next := choose_next_action( );
      activate( next );
      { Wait for the next execution }
      passivate( );
   end;
end;
```

Figure 6.5: Mapping simple actions into simulation processes

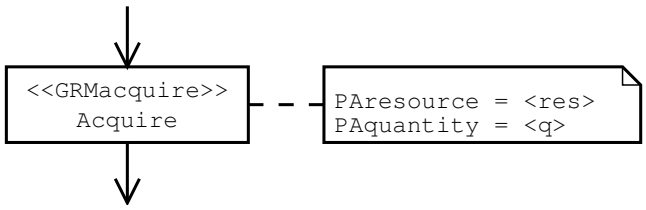
A simple action stereotyped as ≪PAstep≫ is mapped into a simulation process as shown in Fig. 6.5. The simulation process modeling a simple action executes a cycle for a number of times equal to the PArep attribute of the action state. For each iteration, service is requested to the active resource where the step is executed, if defined. If the host processor is not defined, then the simple action process waits for the time specified in the PAdemand attribute. Once complete, a SimpleAction process activates one of its successors, chosen randomly according to the probability of outgoing transitions.




```
process CompositeAction( A: Action )
var
   i    : integer;
   next : ActionBase;
begin
   while ( true ) do begin
      for i:=1 to A.PArep do begin
         activate( A.root );
         { Wait for the substeps to complete }
         if ( i < A.PArep )
            hold( A.PAinterval );
      end;
      { Activate next step }
      next := choose_next_action( );
      activate( next );
      passivate( ); { Wait for the next execution }
   end;
end;
```

Figure 6.6: Mapping composite actions into a simulation process

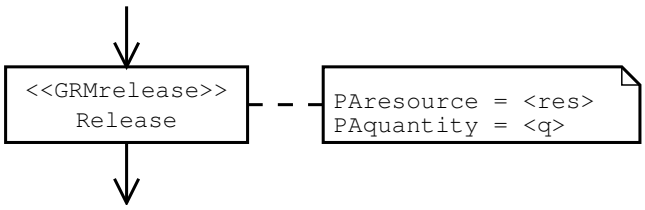
Fig. 6.6 shows the mapping from a composite action to the corresponding simulation process. The behavior of a CompositeAction process consists of activating its root step, waiting for all the sub-steps to terminate. This behavior is iterated as many times as required by the PArep tagged value. Similarly to SimpleAction processes, the successor step is finally selected and executed.




```
process AcquireAction( A: Action )
var
   res  : PassiveResource;
   next : ActionBase;
begin
   res := A.PAresource;
   while ( true ) do begin
      enqueue( res.reqq ); { Join the request queue }
      if ( res.idle( ) ) then
         activate( res );  { Wake up resource }
      passivate( );        { Wait to be served }
      { Activate next step, or owner }
      next := choose_next_action( );
      activate( next );
      { Wait for the next execution }
      passivate( );
   end;
end;
```

Figure 6.7: Mapping acquire actions into simulation processes

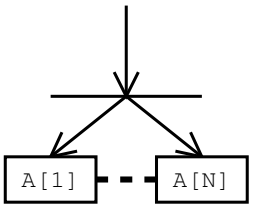
Fig. 6.7 shows the mapping from an action acquiring a passive resource and the corresponding simulation process. The process puts itself on the queue of requests association with the target resource. At this point, the simulation process modeling the passive resource is awaken (if idle), so it will examine the queue and decide whether some process can be serviced. Once the AcquireAction process gets the resource, it choses and activates the next action.



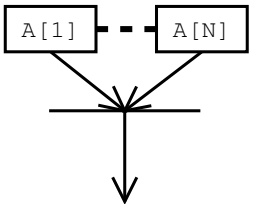

```
process ReleaseAction( A: Action )
var
   res  : PassiveResource;
   next : ActionBase;
begin
   res := A.PAresource;
   while ( true ) do begin
      res.avail := res.avail + A.PAquantity;
      { The maximum number of instances of this
      resource is res.PAquantity, so we clip off
      the excess }
      if ( res.avail > res.PAquantity )
         res.avail := res.PAquantity;
      if ( res.idle( ) ) then
         activate( res );
      { Activate next step }
      next := choose_next_action( );
      activate( next );
      { Wait for the next execution }
      passivate( );
   end;
end;
```

Figure 6.8: Mapping release actions into simulation processes

Fig. 6.8 illustrates how an action releasing a passive resource is mapped into the corresponding simulation process. It is very simple, consisting only in updating the available number of tokens of the resource, and reactivating the PassiveResource simulation process associated to the resource. In general, we allow actions to release an arbitrary amount of tokens for any resource. This means that at any given time, the number of available tokens may be greater to the total capacity of the resource. In order to cope with this, ReleaseAction processes ensure that the maximum available number of tokens does not exceed the resource capacity.




```
process ForkAction( A: Action )
var
   i    : integer;
   next : ActionBase;
begin
   while ( true ) do begin
      for i:=1 to N do
         activate( A[i] );
      next := choose_next_action( );
      activate( next );
      { Wait for the next execution }
      passivate( );
   end;
end;
```




```
process JoinAction( A: Action )
var
   i    : integer;
   next : ActionBase;
begin
   while ( true ) do begin
      { Wait to be activated exactly N
      times, one from each predecessor }
      for i:=1 to N do
         passivate( );
      next := choose_next_action( );
      activate( next );
      { Wait for the next execution }
      passivate( );
   end;
end;
```

Figure 6.9: Mapping fork and join actions into simulation processes

6.2 The validation issue

Validation of a simulation model deals with the accuracy of the modeling process, i.e. how accurately the simulation model represents the real system. Validation is usually an iterative process involving comparison of model and system behavior. Discrepancies between these behaviors are used to tune the model until its accuracy is considered acceptable [60].

In the context of SPE, the "real system" is an abstract representation of a SA. Specifically, we have considered here a UML representation of a SA. Such an abstract representation cannot be directly executed, so the standard validation techniques cannot be applied within this context.

Formal validation techniques [16] can be used to validate performance models when an already implemented system is not available. The idea is to develop a formal proof that the performance model is valid, according to the scope of the

simulation. To get such a formal proof of correctness both the system to simulate and the simulation model must be translated into a formal and precise mathematical notation. The model is considered validated if it is possible to formally prove that it behaves in the same way as the system and satisfies the requirements specification with sufficient accuracy. Note that this validation method does not require the simulation model to be actually implemented.

As observed in [16], developing formal proofs of correctness in realistic cases is not always possible under the current state of the art. If the system to be modeled is a set of UML diagrams the problem is even worse, because UML has no formal semantics. There are some research addressing this open problem, as discussed in [31, 85]. However, giving a formal description of a simulation model is difficult, and currently there is no standard notation to do so. A few works describe a semantics for a subset of a simulation language [25, 27, 95]. Thus, the simulation model could be formally described by first translating it into a simulation program for which there is a formal semantics.

A simpler validation approach is to develop a set of test cases, each one being an UML representation of an already implemented SA. Then we can validate the simulation model with the implemented SA.

Both the approaches above have some drawbacks. Formal validation is difficult given the current state of the art, since UML descriptions of SA and simulation models have no standard, formal representation. On the other hand, validation of simulation models against testing, using standard techniques, gives no guarantee that the transformation from UML model into performance model always preserves properties of the original SA. Validation by example can be used just to show that the transformation can produce valid models, and not that the transformation must always produce valid models.

We outline another possible approach to model validation. We observe that transforming UML models into simulation models is very similar to defining a semantics of a set of annotated UML diagrams. This semantics should depend on what UML diagrams are used and how they are annotated; we assume annotations based on the Performance Profile. The transformation $\mathcal{S} : U \rightarrow M$ defines a mapping from the space of the annotated UML diagrams U into the space M of the simulation models. The mapping $\mathcal{S}$ could be defined according to the informal UML semantics defined in the standard [74], and according to the equally informal semantics of the annotations described in the UML performance profile [75]. Function $\mathcal{S}$ could be defined in an arbitrary way. However, if we treat $\mathcal{S}$ as a semantics function, we can impose a constraint on it by requiring the soundness property to hold. Suppose that we define an equivalence relation $\approx_U \subseteq U \times U$ and $\approx_M \subseteq M \times M$. Given that, we can define a soundness property for the transformation $\mathcal{S}$ as follows:

$$\text{for each } X, Y \in U : \mathcal{S}(X) \approx_M \mathcal{S}(Y) \Rightarrow X \approx_U Y \tag{6.1}$$

This means that, for each pair of annotated sets of UML diagrams X and Y, if they map to equivalent performance models, then they represent similar SA.

Defining the equivalence relations $\approx_M$ and $\approx_U$ is very difficult. Two SA could be equivalent if they represent the same system at different levels of detail, or if they describe two systems providing the same functionalities, or if they are structurally equivalent, and so on. Two simulation models could be equivalent if they are structurally equivalent, or if they represent systems with the same performances under identical conditions. However, note that we could define an efficient performance model validation technique by using the definition of relations $\approx_M$ and $\approx_U$ so that the soundness property can be easily and possibly automatically verified.

7

An application to UML Mobility Modeling

7.1 Introduction

In this chapter we describe an application of the simulation-based UML performance modeling approach introduced earlier. We develop an integrated UML notation for describing and evaluating the performances of mobile systems described at a high level of detail. We combine the annotations from the profile described in Chapter 5 with a structured approach for describing mobile systems in term of use case, activity and deployment diagrams.

Motivations The current generation of network-centric applications exhibits an increasingly higher degree of mobility. From one side, wireless networks allow devices to move from one location to another without loosing connectivity. From the other side, new software technologies allow code fragments or entire running applications to migrate from one host to another. In this direction, design approaches based on location awareness and code mobility have been proposed where the application components can move to different locations during their execution. This should improve system quality, allowing a higher degree of flexibility and increasing the performance. Indeed, from the performance viewpoint, moving components of an application in a distributed environment could lead to transforming remote interaction into local ones. We consider software mobile system at a high level of abstraction and we refer to SA to describe system structure and behavior [18, 90]. Mobile software systems can be represented by SA [39]; in particular, one can define mobile SA with various mobility styles, whose definition depends on whether they require copies creation of components at new locations, or local change of components preserve their identity (mobile agent). In the former case we can further distinguish systems where the copy is created at the location of the component that starts the interaction (code on demand), or systems where it is created at the location of the component that accepts the interaction (remote evaluation). Many formalisms have been proposed to represent mobile software systems and for reasoning about mobility [37, 71, 72, 78, 100]. However, most of these formalisms

cannot be considered Architectural Description Language (ADL), since they do not explicitly model components and interactions as first class entities. Several models of SA have been proposed based on formal ADL with precise semantics and syntax, as presented and compared in [68]. On the other side, due to the difficulties in integrating formal ADL in the design practice, other approaches consider semi-formal widely used modeling languages such as UML, taking advantage of the availability of development tools [70].

We consider mobile software systems at the SA level, and an UML-based system specification. Performance modeling of mobile software systems is a difficult task, which should be carried out from the early design stages of system development. The integration of quantitative performance analysis with software system specification has been recognized to be a critical issue in system design. In particular, performance is one of the most influential factors that drive system design choices.

We address the problem of integrating system performance modeling and analysis with a specification of mobile software system based on UML. In particular we describe an integrated approach to modeling and performance evaluation of mobile systems at the architectural level based on simulation. Here we show how the performance-enabled UML specifications described in Chapter 4 can be applied to modeling and performance evaluation of mobile systems. We consider both physical mobility (devices which physically change their locations) and code mobility (code fragments which migrate from one execution host to another).

Previous Works Baumeister et al. propose in [19] an extension of UML class and activity diagrams to represent mobility. They define new stereotypes for identifying mobile objects and locations. Stereotypes are also defined for moving and cloning activities. Mobile systems are then represented by using activity diagrams using either a "responsibility centered notation", which focuses on who is performing actions, and a "location centered notation" which focuses on where actions are being done and how activities change their location. While this approach has the advantage of requiring only minor extensions to UML, a possible shortcoming is that it represents in the same activity diagram both the mobility model (how objects change their location) and the computation model (what kind of computations the objects do). For large models this could render the diagrams difficult to understand.

Some UML notation mechanisms can be used to represent mobile SA, as discussed in [87]. They are based on the tagged value `location` to express a component location, and the stereotypes `copy` and `become` to express the location change of a component. They can be used in Collaboration diagrams to model location changes of mobile components. Grassi and Mirandola [44] suggest an extension to UML to represent mobility using collaboration diagrams. Collaboration diagrams contain a location tagged value representing the physical location of each component. They define the `moveTo` stereotype, which can be applied to messages in the collaboration diagram. When the `moveTo` stereotype is present, it indicates that the source com-

ponent moves to the location of the destination component before interacting with it. Sequence diagrams are used to describe the interactions between components, regardless of the mobility pattern of the components.

Kosiuczenko [58] proposes a graphical notation for modeling mobile objects based on UML sequence diagrams. Mobile objects are modeled using an extended version of lifelines. Each lifeline is represented as a box that can contain other objects (lifelines). Stereotyped messages are used to represent various actions such as creating or destroying an object, or entering and leaving an object. This approach has the drawback of requiring a change in the standard notation of UML sequence diagrams, that is, lifelines should be represented as boxes, with possibly other sequence diagrams inside. Existing graphical UML editors and processors need to be modified in order to support the new notation.

7.2 The approach

We consider a mobile system as a collection of devices and processes running on them. Devices include both communication networks, processors with a fixed location (e.g., a desktop PC) and mobile devices (e.g., a PDA). We consider both physical mobility, that is the possibility that the devices change their physical location, and logical mobility, that is the fact that fragments of code can migrate from one execution host to another. A computation on the system is modeled as a set of activities carried out on the devices. A configuration of the system is a specific allocation of activities on processors. So, while a mobile entity travels through the system, it activates a sequence of configurations, each representing a specific system state. Once a configuration is activated, the mobile entity starts an interaction with the system. This typically includes requesting service to the devices (processors) or performing communications, which we also model as requesting service to network devices. We assume that while a mobile entity is interacting with the system, it cannot move, i.e., the system configuration cannot change. Further movements are possible when the interaction is completed. Changes of scenario (and hence mobility) may happen only at the end of an interaction. Even if possible in theory, it is complex to apply our approach to modeling mobile systems in which change of scenario are triggered by some external event, possibly interrupting the current activity at some point.

The proposed approach to mobile system modeling involves three main steps, which are depicted in Fig. 7.1

1. Enumerate the various mobility strategies of the mobile entities.
2. Model the sequence of configurations which are triggered in each mobility strategy.
3. Model the interactions performed by the mobile entities in each configuration.

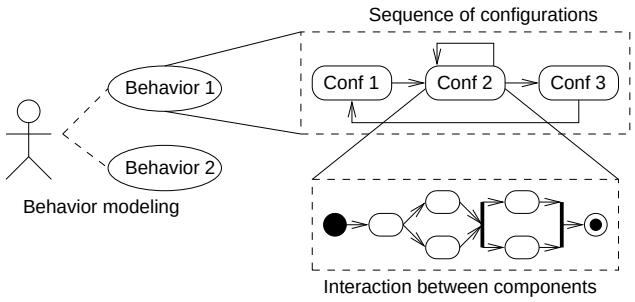


Figure 7.1: Overview of the mobility modeling steps

The first step deals with identifying the mobility pattern of the entities. For example, users of the system may exhibit a bigger or smaller probability to change their location, hence they may exhibit different degrees of mobility. Also, users may move through the system with different preferential patterns. We define a *mobility behavior* as the sequence of different configurations which are executed while the user moves. Such mobility behaviors need to be identified, and further described in the next steps. Mobility behaviors are represented by UML use case diagrams. Also, the set of resources (processors) which are present in the system are described using Deployment diagrams.

The next step involves the description of the sequence of configurations which are triggered while the mobile entities travel through the system. Such description can be easily expressed as a state transition diagram. We use activity diagrams for this purpose. Each activity represents a particular configuration of the system. Transitions describe the order in which configurations are triggered as the user moves. We will refer to these diagrams as "high level" activity diagrams.

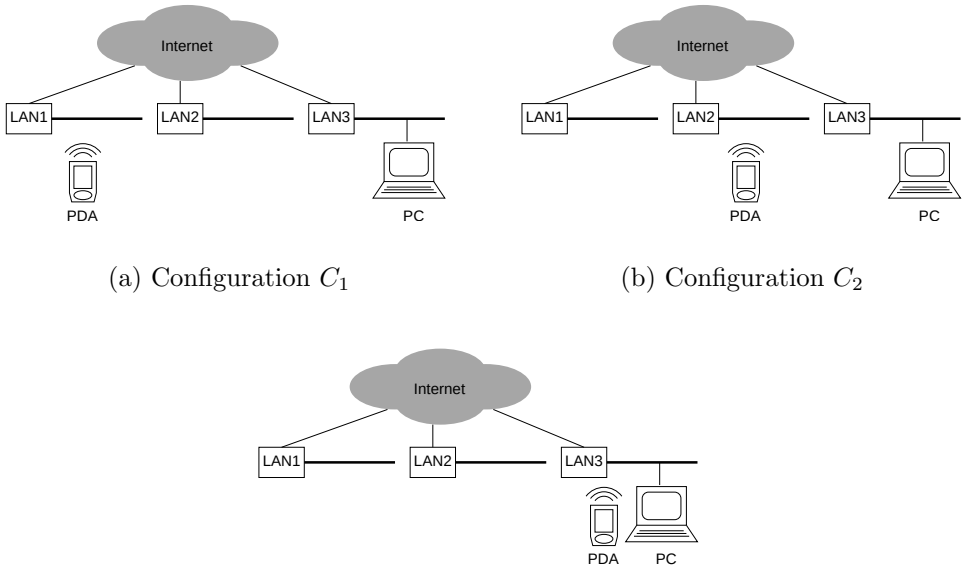
The last step involves detailing what happens while the system is in each configuration. This means specifying what are the interactions between the components while each configuration is active. This is done again using activity diagrams. Each Action state is associated to the Deployment node instance on which the activity takes place. Each node of the activity diagrams defined in the previous step is expanded as an interaction. This can be readily expressed using standard UML notation as activity diagrams have a hierarchical structure, that is, each action state may be exploded into another diagram. We will refer to these diagrams as "low level" activity diagrams.

In order to illustrate the proposed approach we introduce an application example of software mobile system.

Let us consider the example of software system illustrated in Fig. 7.2. There is a mobile user that is connected to a PC using a PDA with a wireless network card. The user is viewing a video stream, which is generated by a video server residing on the PC.

Three different Local Area Networks (LAN1, LAN2 and LAN3) are connected through the Internet. Each LAN allows wireless connections as well as wired ones.

The PC is connected to LAN3 and does not move, while the user with the PDA travels through the different LAN. In the configuration C_1 of Fig. 7.2(a) the communication between the PDA and the PC travels through the path LAN1–Internet–LAN3. In the configuration C_2 of Fig. 7.2(b) the communication is routed through the path LAN2–Internet–LAN3, and in the configuration C_3 of Fig. 7.2(c) the communication between the PC and the PDA is routed through LAN3 only.



(a) Configuration C_1

(b) Configuration C_2

(c) Configuration C_3

Figure 7.2: A mobile user travels through three different LAN

7.2.1 Modeling the choice of Mobility

As the very first step, it is necessary to provide the physical structure of the system. This can be done by using UML deployment diagrams, which describe the processing resources available on the system. Such resources include both CPUs and also communication links. The deployment diagram describing the system in the example is illustrated in Fig. 7.3.

In the example above, we suppose that the mobile user can behave in two different ways. In behavior B_1 he joins the system in LAN1, then travel to LAN2 and finally to LAN3 and leaves the system. In behavior B_2 the user joins the system in LAN2, travel to LAN1 and leaves the system. We model these behaviors with the use case diagram in Fig. 7.4.

The diagram shows an Actor (mobile entity) that can perform one of the associated use cases, each representing one possible mobility behavior. An Actor represents each class of mobile entity. The Use Cases associated with that Actor

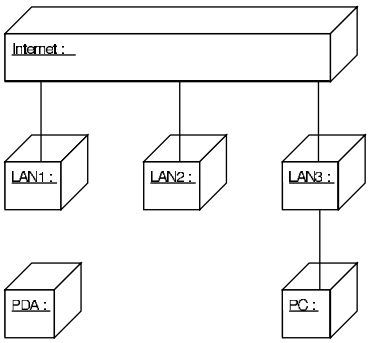


Figure 7.3: Deployment diagram for the example

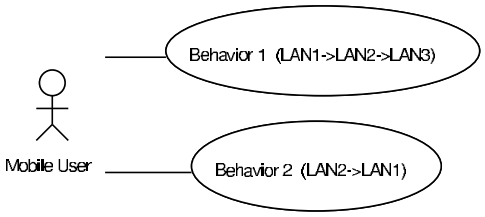
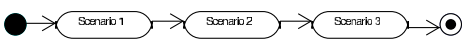


Figure 7.4: UML representation of different mobility possibilities

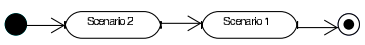
represent the different ways in which entities of the associated mobile entity class may interact with the system. Note that this is perfectly consistent with the UML semantics of use case diagrams [74], as they are used to specify the behavior of an entity without specifying its internal structure.

7.2.2 Modeling Mobility Behaviors

The next step is to describe the order in which configurations are activated in each behavior. To do that, we associate an activity diagram to each use case; each activity of the activity diagram represents a configuration of the system. If the mobile user triggers the configuration C_j immediately after the configuration C_i, then in the activity diagram there will be a transition between the activity representing C_i and the one representing C_j. Considering our example, the two behaviors B_1 and B_2 are represented as the activity diagrams of Fig. 7.5.



(a) Activity diagram associated to B_1

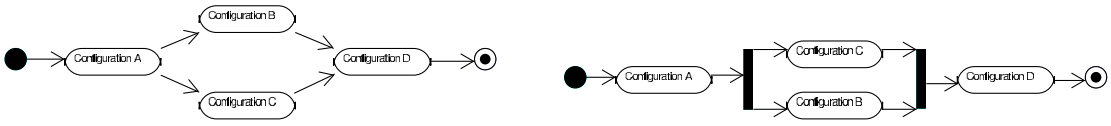


(b) Activity diagram associated to B_2

Figure 7.5: Activity diagrams associated to the mobility behaviors

Note that in this way it is possible to represent non-determinism that is a be-

havior can have multiple successors. Fig. 7.6(a) illustrates an example where the mobile entity starts by activating configuration A. Then it may proceed by activating one of configuration B and C. After that, configuration D is activated. It is important to observe that the activity diagrams associated to behaviors do not need to be acyclic. Thus, it is also possible to model situations in which the user triggers the same sequence of configurations for a number of times. Moreover, using fork and join nodes of activity diagrams it is possible to represent the concurrent execution of different configurations. This can be used to model situations in which the mobile entity generates copies of itself, each one traveling independently through the system. Fig. 7.6(b) shows an example where a mobile entity starts by entering configuration A. Next, it splits in two copies, one executing configuration B and the other executing configuration C in parallel. This means that the two copies of the mobile entity can move to different locations and perform different interactions with the system. After that, the two copies synchronize and collapse into one instance, which proceeds by executing configuration D.



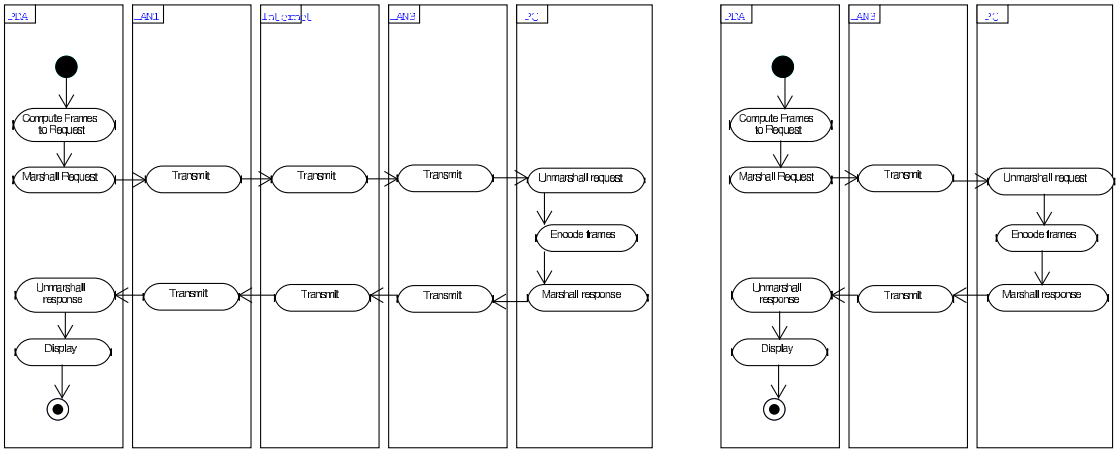
(a) (b)

Figure 7.6: Modeling nondeterministic mobility behavior 7.6(a) and mobile concurrent execution of multiple agent instances 7.6(b)

7.2.3 Modeling Interactions between Components

The final step is describing the activities carried out in each configuration. To do that, we use the hierarchical structure of the activity diagrams to associate the interactions to each action state identified in the previous step. Namely, we expand each action step representing a configuration into the activity graph describing the sequence of actions which are taken during the interaction between the mobile entity and the system. It is necessary to specify where the actions are executed. To do so it is possible to use "swimlanes", which are a means for specifying responsibility for actions. The name of the swimlanes denotes the Deployment node instance on which the actions execute. As some graphical UML editors do not support swimlanes, it is possible to tag each action with the PAhost tagged value, whose value is the name of the node instance of the deployment diagram corresponding to the host where the action is executed. Fig. 7.7 shows the interactions performed while the system we are considering is in configuration C_1 and C_3 using the swimlane-based notation.

The interaction between the PDA and the PC is very simple. Basically, first the



(a) Interaction in configuration C_1

(b) Interaction in configuration C_3

Figure 7.7: UML description of the interactions

PDA computes which frames it needs. Then a suitable request is encoded and sent through the communication networks to the PC. The request is unmarshalled, and the requested frames are encoded and packed into a reply message. This message is sent back through the network to the PDA, which finally displays the frames. Note that in Fig. 7.7 we omitted the description of the interaction in configuration C_2, as this is basically the same as in C_1, with the only difference that LAN2 is used instead of LAN1.

7.3 Summary of the UML Mobility Approach

The proposed mobility modeling approach can be summarized in the following steps:

1. Identify the processing resources (processors or networks) available in the system. Each resource is represented by a node instance in the UML deployment diagram.

2. Identify the classes of users of the system. Users represent workloads applied to the system. Each class of users is represented as an Actor in the use case diagram. Actor may represent either a fixed population of users (closed workload) or an unlimited stream of users (open workload).

3. Identify the mobility behaviors. For each class of users it is necessary to identify the different pattern of mobility they may exhibit. Each of such mobility patterns is represented by a use case associated to the Actor representing the class of users.

4. Provide a high level description of the mobility behaviors. An activity diagram is association to each of the use cases identified in the previous step. Such activity diagram represents the sequence of configuration changes which happens in the system while the mobile user moves.

5. Describe the interactions occurring in each system configuration. Each Action states defined in the previous step are expanded into a low-level activity diagram describing the interactions between system entities. Each Action of the low-level diagram represents a service requested to a specific processing resource. Code mobility is represented by associated Activities in the low-level diagram to different hosts. Physical mobility is represented by a possibly different interaction pattern associated to nodes of the high-level activity diagram.

Quantitative informations required by the simulator can be associated to UML elements as described in Chapter 4. Each use case can be tagged with the probability of its occurrence, that is, the probability that the associated Actor (motile entity) will execute that use case (mobility behavior) upon arriving to the system. Action states of the high-level activity diagram associated to each use case can be annotated with the probability of occurrence, the number of times they are repeated and the delay between repetitions. As each action state represents a configuration, the annotations allow the specify (nondeterministically) the pattern of mobility and how long the system remains in each configuration. When the simulator "executes" a configuration, it basically executes all the activities of the low-level activity diagram embedded in the configurations.

A mobile code fragment moving from host H_1 to host H_2 is represented as follows. Let C_1 be the configuration where the code executes in H_1 and C_2 the configuration where the code executed in H_2. The low level activity diagram describing C_1 and C_2 will contain an action state (or a whole subdiagram) corresponding to the computation. Such action state or subdiagram will be tagged with the PAhost tagged value, which describes the location where the activity is executed. In C_1 we set PAhost=H_1, and in C_2 we set PAhost=H_2.

Physical mobility is modeled in a similar way. If a mobile device travels through the system, as in our example, probably it will interact with different other nodes for communicating. Depending on the situation, it may even choose a different communication pattern, and hence a different interaction style with other entities. Such interaction styles will be represented by (possibly different different) structures of the low level activity diagrams.

7.4 A Simple Example

The case study illustrated in the previous sections has been simulated using the parameters reported on Table 7.1. A single user interacts with the system (closed

workload), and the probability p that the user triggers the behavior B_1 (see Fig. 7.5) has been set to $p = 0.3$, while the probability that the user triggers the behavior B_2 has been set to $1 - p$.

Parameter	**Value**
(Un)Marshalling Requests	Constant $0.1s$
(Un)Marshalling Responses	Exponential, mean=$5.0s$
Request Transmission Times	Exponential, mean=$1.0s$
Response Transmission Times	Exponential, mean=$10.0s$
Request Computation	Exponential, mean=$0.1s$
Frame Encoding Time	Exponential, mean=$20.0s$
Display time on the PDA	Exponential, mean=$20.0s$
PDA Speedup factor	0.2
PC Speedup factor	10.0
Processors Sched. Policies	FIFO

Table 7.1: Simulation Parameters for the Mobile System Example

The simulation results are shown in Table 7.2. They are the steady-state mean values computed at 90% confidence level; for simplicity only the central value of the confidence interval is shown.

Internet utilization	0.039
LAN1 utilization	0.010
LAN2 utilization	0.028
LAN3 utilization	0.072
PDA utilization	0.832
PC utilization	0.015
Behavior B_1 response time	445s
Behavior B_2 response time	312s

Table 7.2: Simulation Results for the Mobile System Example.

III

Implementation

8

`libcppsim`: A process-oriented simulation library

In this chapter we illustrate the design and implementation of `libcppsim`, a general-purpose, process-oriented simulation library written in C++. `libcppsim` provides the user with a set of classes for implementing coroutines, simulation processes, the SQS data structure, and basic statistical functions for collecting and processing simulation results. The basic simulation entity provided by the library is the *simulation process*: methods are provided for activating, stopping and rescheduling simulation processes. The library provides simulation primitives which are commonly implemented in many process-oriented simulation languages or libraries, such as the SIMULA language [35].

The choice of the C++ programming language was motivated by the availability of efficient compilers and utility libraries (such as libraries for XML processing [62]), which greatly accelerated the simulator development. We implemented a minimalistic set of simulation primitives in order to provide a simple and general framework on top of which more complex, high-level functionalities can be defined. Many existing simulation packages are either special-purpose or not very efficient. For that reason we decided to implement our own simulation toolkit which includes only the needed functionalities. This simplified the debug of the simulation tool as we had access to all the source code, and potential problems in the simulation library would reside in a restricted set of modules providing only simple functionalities.

8.1 Introduction and general concepts

The package structure of `libcppsim` is illustrated in Fig. 8.1; in the figure, UML packages denote sets of functionally related classes. There are two root packages: Coroutines and Variables. The Coroutines package provides two different implementations of coroutines, which are used for implementing both simulation processes and the Sequencing Set. The Variables package contains the basic definition of a "variable", that is an entity which has a an internal state which can be updated with new values; variables provides a method for computing a function of this internal state. Using the abstraction of a "variable" it is possible to derive random number

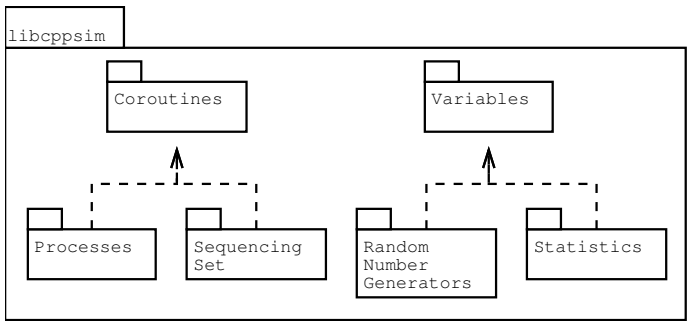


Figure 8.1: `libcppsim` package structure

generators and statistics classes. Random number generators are implemented as variables returning a different value every time it is computed, thus producing a stream of pseudo-random numbers. Statistics are special kind of variables which are repeatedly updated with a sequence of observations of some quantity of interest computed by the simulation, and whose value is the requested statistics (mean, confidence interval, variance and others).

8.2 Coroutines

Coroutines are a programming construct which can be very helpful for implementing process-oriented simulation languages and libraries; SIMULA had coroutines as a built-in facility. Unfortunately, the C and C++ programming languages, which are widely used and extremely efficient, do not provide coroutines natively.

Coroutines are blocks of code with associated state, which are activated according to the following rules:

- One coroutine at a time is active and is running; control is retained either until the coroutine terminates or until it explicitly resumes another coroutine.
- There exists a "main coroutine" which is the first coroutine being activated. The main coroutine gains control as soon as the program is executed.
- An active coroutine that gives up control by resuming another one retains its current execution context. When the suspended coroutine is resumed, it will execute from the point at which it last stopped.

Coroutines can be considered as a special kind of procedures, as defined in most structured programming languages. For procedures there is a strict caller-callee activation order, that is, the caller procedures can transfer control to the called one, and be reactivated only when the latter terminates. On the other hand control can flow arbitrarily for coroutines (see Fig. 8.2). Coroutines can be considered as

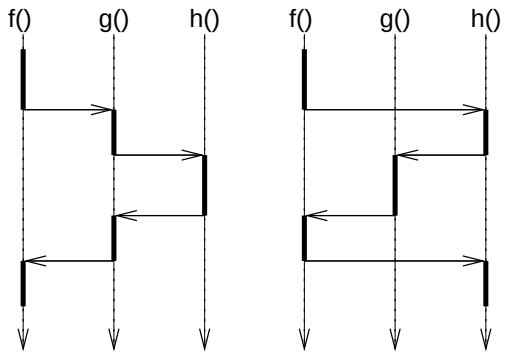


Figure 8.2: Flow control for routines (left) and coroutines (right). Time advances downward. Thick lines represent which (co)routine is active at any given time.

a cooperative multithreading environment, with coroutines acting as lightweight threads explicitly passing control each other.

Adding support for coroutines to the C++ programming language is not trivial. Each C++ function is associated to a data structure called *activation record* which is stored in the run-time stack. The activation record contains informations about the status of the routine, such as the value of the local variables. When a subroutine call occurs, a new activation record for the called function is created and put on the top of the stack. All the local variables, defined by the called routine, are stored on the newly created activation record. Similarly, when a routine terminates, its activation record is pushed from the stack. LIFO handling of the runtime stack does not work anymore with coroutines, because the currently active coroutine may not be the one associated with the topmost activation record. This implies that the order of activation of coroutines is not given by the LIFO stack handling.

This problem can be solved in different ways. The first approach is the "copy-stack implementation" described by Helsgaun [46]. The stack of the currently operating coroutine is kept in the C++ runtime stack. When a coroutine suspends, the runtime stack is copied in a buffer associated with the coroutine. The buffer is allocated in the dynamic heap, so it does not interfere with the normal stack operation. A coroutine is resumed by copying the content of that buffer to C++'s runtime stack and setting the program counter to the saved execution location. While this approach is fairly portable, it makes the assumption that the run-time stack is implemented as a contiguous block of memory, as opposed to a linked list of frames.

A second approach is simpler and more portable, and consists of making use of the context handling facilities provided by most Unix SysV-like environments. The Operating System provides functions allowing user-level context switching between different threads of control within a process. These functions are getcontext(), setcontext() and makecontext().

The getcontext() function is used to initialize a user-supplied data structure of

type ucontext_t with a copy of the current context. The ucontext_t data structure is defined as follows under the Linux OS [64] (in file /usr/include/asm/ucontext.h):

```
typedef struct ucontext
{
    unsigned long int uc_flags;
    struct ucontext *uc_link; /* Reference to the
                                 next context            */
    stack_t uc_stack;         /* Where the stack for
                                 this context is located */
    mcontext_t uc_mcontext;
    __sigset_t uc_sigmask;
    struct _fpstate __fpregs_mem;
} ucontext_t;
```

The relevant fields of the structure are uc_link which, if nonzero, points to the context to be activated when this one terminates, and uc_stack, which contains a pointer to a dynamically allocated user memory area which holds the context. stack_t is defined as follows:

```
typedef struct sigaltstack
{
    void *ss_sp;
    int ss_flags;
    size_t ss_size;
} stack_t;
```

ss_sp points to the stack.

The makecontext() procedure modifies an ucontext_t data structure; the user defines a function which will be executed when the context is activated, and an optional successor context to be activated when the function terminates. The swapcontext() function saves the current context and activates a new one.

Both approaches have advantages and disadvantages. The "copy-stack" approach requires a copy of the C++ runtime stack to be saved every time a coroutine passes control, which may cause a considerable overhead. Moreover, the approach requires that the run-time stack is internally represented as a contiguous block of memory (which however is the most common case in practice). The approach based on the ucontext_t data structure does not incur in this overhead, as the context each coroutine is allocated only once when the coroutine is activated. A coroutine switch (performed via the swapcontext() system call) just makes the CPU's stack pointer to point to the stack of the coroutine to be resumed. This approach does not depend on the layout of the run-time stack, but does require the Operating System to support the makecontext and swapcontext system calls (old versions of Linux did not implement them).

Unfortunately, the ucontext_t based approach does not allow the coroutine context to grow past the dimension defined when the context is allocated. The user is required to set the maximum dimension of the context in advance; choosing a too small buffer results in weird runtime behaviors caused by stack corruption which

are typically very difficult to debug. The "copy-stack" approach does not have this limitation. The `libcppsim` library implements both alternatives, and the user can choose one at compile time. As a general rule, it is useful to use the "copy-stack" variant when developing the simulation program, in order to ensure that stack corruption errors do not happen; then, when the program is checked and correct, production runs may be done with the more efficient context-based approach.

The coroutine class is defined as follows:

```
class coroutine
{
public:
    virtual ~coroutine ();
    inline bool terminated( void ) const;
    void resume( void );
    void call( void );
    static size_t cStackSize;
protected:
    coroutine( );
    void detach( void );
    virtual void main( void ) = 0;
private:
    inline void enter( void );
    coroutine   *_caller;
    coroutine   *_callee;
    static coroutine *_current_coroutine;
};
```

Note that it is a virtual class, since the main() is pure virtual. Users can implement their own coroutines by inheriting a class from coroutine. The meaning of user-accessible methods (public and protected) is the following:

main() This method represents the body of the coroutine. When the coroutine object is activated, the function main() is executed. The coroutine terminates when main() terminates.

bool terminated() Returns true if and only if the coroutine object represents a terminated coroutine, that is, a coroutine whose main() function finished.

call() This method is used for invoking a coroutine, setting the caller as the coroutine to activate when this one terminates or performs a detach() operation. This method also sets the _caller and _callee pointers accordingly, so that when the caller terminates, control is automatically passed to the caller.

resume() Activates the coroutine from the main() function, if this is a newly created coroutine, or from the point it was last suspended. Note that this method does not set a caller-callee relationship between the coroutines. This means that when the resumed coroutine terminates, control is not transferred to the one who performed the resume() operation.

detach() Suspends the execution of the coroutine.

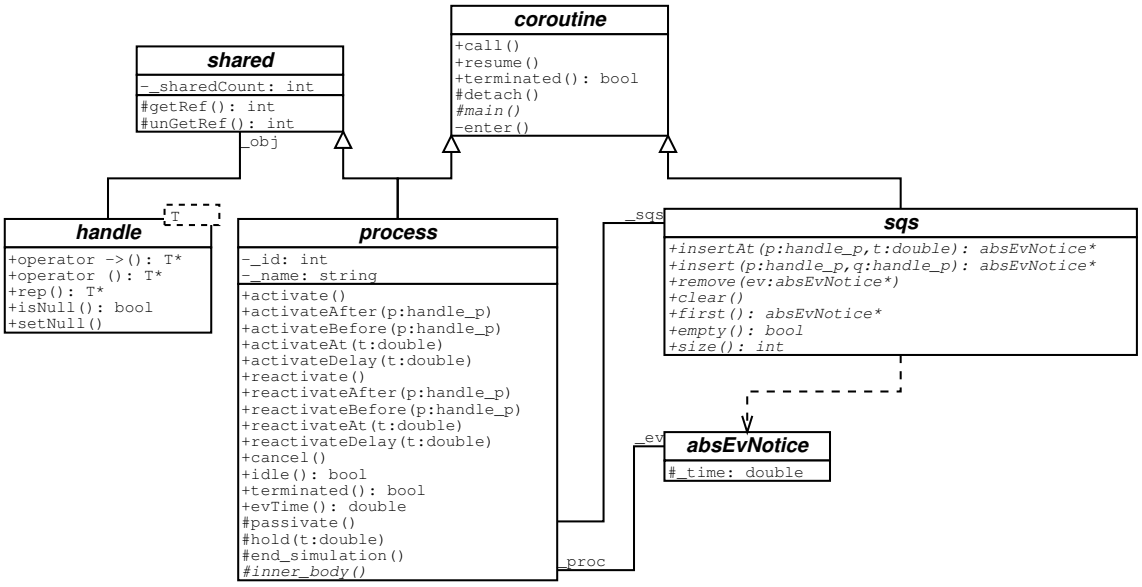


Figure 8.3: libcppsim process package class diagram

The enter() method is used to actually save save the context of the current coroutine and restore the context of the one control is passed to. Implementation details can be found in Helsgaun [46].

8.3 Simulation processes

Once coroutines are available, it is very easy to define simulation processes on top of them. Fig. 8.3 shows the class hierarchy related to simulation process implementation.

A simulation process is represented by the process class, and it inherits from coroutine. Simulation processes implement their functionalities in term of the basic activation primitives provided by the coroutine class.

Each simulation process has a unique identifier and a user-supplied (not necessarily unique) name. The process class implements methods providing all the scheduling primitives described in Section 3.4. The actions performed by the simulation process should be specified by redefining the pure virtual method inner_body().

One feature which was missing from the initial implementation of the process class was a mechanism for automatic garbage collection. This is a feature which is not present in the C/C++ languages, but is extremely valuable when writing simulation programs. The reason is that when many simulation processes are dynamically created, it is often desirable to have them automatically destroyed when they terminate and are not reachable from any live pointer.

It turns out that the implementation of a complete garbage collector was not

really necessary. Instead, we chose to implement a simpler mechanism based on *smart pointers*. A smart pointer is represented by the templated class handle⟩T⟨, which represents a reference to an object of type T which can be shared among multiple owners. When the last handle containing a reference to some object goes out of scope and is deallocated, then the referenced object is automatically destroyed and its memory reclaimed. The implementation of the handle class is based on the reference-count idea described in [94]. Basically, the handle class redefines the constructor, destructor and assignment operator in order to maintain a count of the number of active references to the shared object. When this counter goes to zero, the object is destroyed. Our implementation is however slightly different in that the reference count is kept inside the shared object. This requires that, in order to be put inside an handle⟨T⟩, the type T must be derived from the shared class, which contains just a counter and operators to increment and decrement it. This approach has the disadvantage that not every data type can be used inside an handle (e.g., it is impossible to have an handle⟨double⟩), but it has the advantage that the reference count is preserved even when a handle⟨T⟩ is dereferenced as an object of type "pointer to T" and then back to a different hanlde⟨T⟩. That is, in the following fragment of code, with our implementation of handles, the reference count is preserved correctly:

```
int foo( void )
{
    handle<process> p = get_some_process( );
    process* r = p.rep( );  // Dereference p
    handle<process> q( r ); // OK. reference count preserved
}
```

Situations like that happen in the implementation of some methods of the process and sqs classes, and are unavoidable. Note that sharing the same object between handles and ordinary C++ pointers may cause problems, as handles may cause the shared object to be destroyed, making pointers invalid.

Simulation processes may be associated with an event notice, represented by an instance of a class derived from absEvNotice. The event notice basically contains the simulation time the process must be activated, and its position in the SQS. The latter is clearly dependent upon how the SQS data structure is implemented. Hence, every SQS implementation defines its own specialized version of event notice.

8.4 Sequencing Set implementations

The Sequencing Set (SQS), as described in Section 3.4, is a data structure containing the set of simulation processes to activate sorted by nondecreasing timestamp order. Thus, it is basically a priority queue, where the priority is given by the process (re)activation time. Each simulation process contained in the SQS is associated with an event notice. We keep event notices as separate entities (derived from the

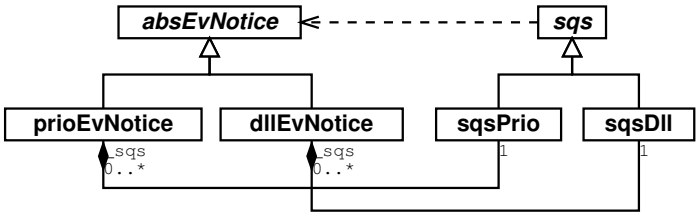


Figure 8.4: libcppsim Sequencing Set class diagram

absEvNotice class) in order to be able to provide different implementations of the sequencing set data structure, each one needing particular informations in its event notices. We provide two different implementations of the SQS data structure (see Fig, 8.4).

The first implementation is contained in class sqsDll, and is based on a simple doubly linked list. This data structure is very simple, but inserting an element in the list requires linear time on average. A more efficient implementation, based on balanced search trees, is given in class sqsPrio. The expected insertion time in this case is proportional to the logarithm of the SQS size.

8.5 Random variate generations

Each simulation engine should provide an efficient and statistically robust mechanism for producing streams of pseudo-random numbers with given distribution. libcppsim defines an abstract templated class rng⟨T⟩ representing a pseudo-random number generator producing a stream of numbers of type T. Thus, it is possible to generate streams of random integers, real or boolean values, by setting T to the appropriate datatype in a subclass. The rng⟨T⟩ class hierarchy is depicted in Fig. 8.5.

It turns out that the basic ingredient for generating stream of pseudo-random numbers is a good uniform generator RN$(0, 1)$ over the interval [0..1]. We chose to implement algorithm MRG32k3a from L'Ecuyer [61], which is known to have long period and very good statistical properties, other than being very efficient. In the pseudocode used below, rngUniform01() denotes the implementation of this random number generator.

The following random variate generators are currently implemented. The implementations are taken from [16, Ch. 5].

Uniform distribution: $U(a, b), a < b$

It is implemented by class rngUniform.

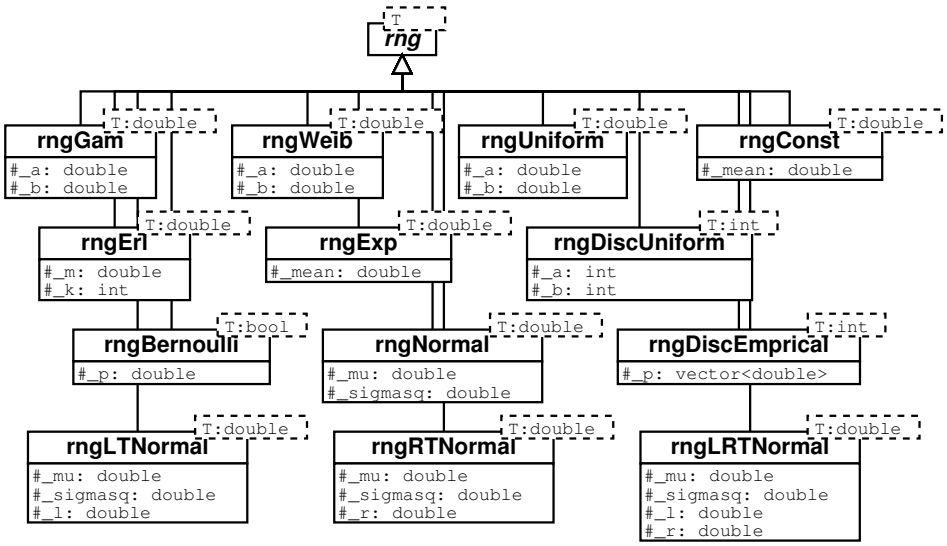


Figure 8.5: `libcppsim` Random Number Generators class diagram

Density:

$$f(x) = \begin{cases} \dfrac{1}{b-a} & a < x < b \\ 0 & \text{otherwise} \end{cases}$$

Distribution function:

$$F(x) = \begin{cases} 0 & x \le a \\ \dfrac{x-a}{b-a} & a < x < b \\ 1 & x \ge b \end{cases}$$

Generator:

```
rng<double> U1 = rngUniform01( );
return U1.value( )*(_b - _a) + _a;
```

Exponential distribution: $\mathbf{EXP}(a), a > 0$

It is implemented by class rngExp.

The parameter $a > 0$ is the mean of the distribution. The rate parameter is a^{-1}.
Density:

$$f(x) = \begin{cases} a^{-1} \exp\left(-\dfrac{x}{a}\right) & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Distribution function:

$$F(x) = \begin{cases} 1 - \exp\left(-\dfrac{x}{a}\right) & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Generator:

```
rng<double> U1 = rngUniform01( );
return (double)(- _mean*log( 1.0 - U1.value( ) ) );
```

Weibull distribution: WEIB$(a, b), a, b > 0$

It is implemented by class rngWeib.

Density:

$$f(x) = \begin{cases} ba^{-b}x^{b-1}\exp\left[-\left(\frac{x}{a}\right)^b\right] & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Distribution function:

$$F(x) = \begin{cases} 1 - \exp\left[-\left(\frac{x}{a}\right)^b\right] & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Generator:

```
rng<double> U1 = rngUniform01( );
// pow( a, b ) returns a raised to the b power
return _a*pow( -log( 1.0 - U1.value() ), 1.0/_b );
```

Gamma distribution: GAM$(a, b), a, b > 0$

It is implemented by class rngGam.

Density:

$$f(x) = \begin{cases} \frac{(x/a)^{b-1}}{a\Gamma(b)}\exp\left(-\frac{x}{a}\right) & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

where $\Gamma(b)$ is the gamma function:

$$\Gamma(b) = \int_0^\infty u^{b-1}e^{-}udu$$

The distribution function has no simple closed form.

Generator:

```
rng<double> U1 = rngUniform01();
rng<double> U2 = rngUniform01();
// M_E = e = 2.71828...
if ( b < 1.0 ) {
    const double beta = (M_E+b)/M_E;
    while( 1 ) {
        double U = U1.value( );
        double W = beta*U;
        double V = U2.value( );
        if ( W < 1.0 ) {
            double Y = pow( W, 1.0/b );
            if ( V <= exp( -Y ) )
                return a*Y;
        } else {
            double Y = -log( (beta-W)/b );
            if ( V <= pow(Y, b-1.0) )
                return a*Y;
        }
    }
} else if ( b < 5.0 ) {
    while( 1 ) {
        double U_1 = U1.value( );
        double U_2 = U2.value( );
        double V_1 = -log( U_1 );
        double V_2 = -log( U_2 );
        if ( V_2 > ( b-1.0 )*( V_1-log( V_1 )-1.0 ) )
            return a*V_1;
    }
} else {
    double alpha = 1.0/sqrt( 2.0*b - 1.0 );
    double beta = b - log(4.0);
    double gamma = b + 1.0/alpha;
    double delta = 1.0 + log( 4.5 );
    while( 1 ) {
        double U_1 = U1.value( );
        double U_2 = U2.value( );
        double V = alpha*log( U_1/(1.0-U_1) );
        double Y = b*exp( V );
        double Z = U_1 * U_1 * U_2;
        double W = beta + gamma*V - Y;
        if ( W+delta-4.5*Z >= 0 )
            return a*Y;
        else
            if ( W >= log( Z ) )
                return a*Y;
    }
}
```

k-Erlang distribution: ERL$(m, k), m > 0, k$ positive integer

It is implemented by class rngErl.

Density: the same as that of GAM$(m/k, k)$.

Distribution function: No closed form for the general case.

Generator:

```
if ( k < 10 ) {
    rng<double> U1 = rngUniform01();
    double prod = 1.0;
    for ( int i=0; i<k; i++ )
        prod *= ( 1.0 - U1.value() );
    return -( m/(double)k )*log( prod );
} else {
    rng<double> U2 = rngGam( m/(double)k, k );
    return U2.value();
}
```

Normal distribution: $N(\mu, \sigma^2), \sigma > 0$

It is implemented by class rngNormal.

μ is the mean and σ^2 is the variance of the distribution. Density:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right]$$

Distribution function: No closed form expression.

Generator:

```
rng<double> U1 = rngUniform01();
rng<double> U2 = rngUniform01();
// M_PI = pi = 3.1415...
double R = sqrt( -2.0*log( U1.value( ) ) );
double T = 2.0*M_PI*U2.value( );
val[0] = mu + sigma*( R*cos( T ) );
val[1] = mu + sigma*( R*sin( T ) );
return val[0] and val[1];
```

Truncated Normal distributions: $N_L(\mu, \sigma^2, l), N_R(\mu, \sigma^2, r), N_{LR}(\mu, \sigma^2, l, r), \sigma > 0$

These distributions are similar to the Normal distribution, except that the range of values they can assume is constrained to the left, to the right or both.

μ is the mean and σ^2 is the variance of the distribution. l is the the left truncation point, and r is the right truncation point.

Let

$$g(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right] \qquad -\infty < x < \infty$$

The density function f_L for the Left Truncated Normal distribution $N_L(\mu, \sigma^2, l)$ is defined as:

$$f_L(x) = \begin{cases} 0 & \text{if } x \geq l \\ \dfrac{g(x)}{\int_l^{+\infty} g(x)dx} & \text{otherwise} \end{cases}$$

Generator:

```
rng<double> U1 = rngNormal(mu, sigmasq);
double result;
do {
    result = U1.value();
} while ( result < l );
return result;
```

The density function f_R for the Right Truncated Normal distribution $N_R(\mu, \sigma^2, r)$ is defined as:

$$f_R(x) = \begin{cases} \dfrac{g(x)}{\int_{-\infty}^{r} g(x)dx} & \text{if } x \leq r \\ 0 & \text{otherwise} \end{cases}$$

Finally, the density function f_{LR} of the Left and Right Truncated Normal distribution $N_{LR}(\mu, \sigma^2, l, r)$ is defined as:

$$f_{LR}(x) = \begin{cases} \dfrac{g(x)}{\int_{l}^{r} g(x)dx} & \text{if } l \leq x \leq r \\ 0 & \text{otherwise} \end{cases}$$

Generator functions are very similar to the one for the Left Truncated Normal distribution.

Constant distribution: $C(\mu)$

It is implemented by class rngConst.

This distribution always returns the constant value μ, and is trivially implemented.

Bernoulli distribution: BER$(p), 0 < p < 1$

Probability mass function:

$$X = \begin{cases} 1 & \text{with probability } p \\ 0 & \text{with probability } 1 - p \end{cases}$$

Generator:

```
rng<double> U1 = rngUniform01();
return ( U1.value( ) <= p );
```

Empirical Distribution

This distribution is implemented by class rngDiscEmpirical. Let us consider a set of probabilities $p_0, p_1, \ldots p_{n-1}$ such that for each $i = 0, \ldots n-1$, $0 \leq p_i \leq 1$, and

$\sum_{i=0}^{n-1} p_i = 1$. The empirical distribution returns one of the integers $[0, 1, \ldots n-1]$ with probabilities $[p_0, p_1, \ldots p_{n-1}]$ respectively.

$$P(X = i) = \begin{cases} p_i & \text{if } i = 0, 1, \ldots n-1 \\ 0 & \text{otherwise} \end{cases}$$

The generator can then be expressed as:

```
rng<double> U1 = rngUniform01();
const double tmp = U1.value( );
double sum = 0.0;
int i;
// p is the vector of probabilities
for ( i=0; i<(int)p.size( )-1; i++ ) {
    sum += p[i];
    if ( tmp < sum ) break;
}
return (int)i;
```

Discrete Uniform DISCU$(a, b), a < b$ integers

This generator is implemented by class rngDiscUniform. The probability mass function is:

$$P(X = i) = \begin{cases} \dfrac{1}{b-a+1} & \text{if } i \in [a, a+1, \ldots b] \\ 0 & \text{otherwise} \end{cases}$$

8.6 Output Data Analysis

Simulation results are typically a sequence of observations of quantities of interest. For example, let us consider a road traffic simulation. Suppose that the modeler is interested in computing the mean number of cars crossing a bridge each day. The simulation produces a sequence of observations $X_1, X_2, \ldots X_n$ corresponding to the number of cars crossing the bridge on day $1, 2, \ldots n$. Unfortunately, the estimator $\hat{X} = \frac{1}{n}\sum X_i$ is in general a biased estimator of the mean $\bar{X}$. The reason is that the observations $X_1, X_2, \ldots X_n$ are in general autocorrelated, and thus not statistically independent. In addition to the autocorrelation problem, the initial conditions of the simulation must be accurately chosen, since they may effect all the computed observations.

`libcppsim` implements a set of classes dealing with collection of statistics. Figure 8.6 shows the relevant portion of the class diagram.

All types of statistical variables inherit from the var$\langle \mathsf{T}_{\text{in}}, \mathsf{T}_{\text{out}} \rangle$ abstract base class. Class var represents "variables" which can be updated many times by feeding variables of type T_{in}, and compute some function from the input data as a result of type T_{out}. Each variable has a (not necessarily unique) name, and supports two pure virtual methods: update() and reset(). These are used to insert a new value and reset

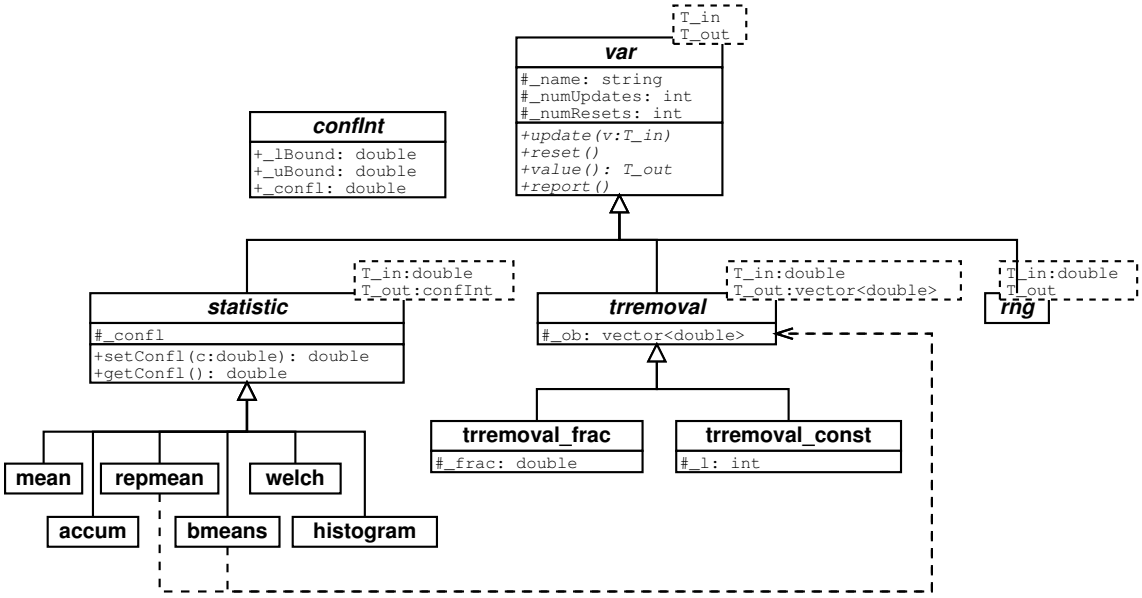


Figure 8.6: `libcppsim` statistics class diagram

the variable to a known state. The result can be computed by invoking the value() method. Variables are not stateless; thus, it is possible that successive repeated invocations of the value() method return different results. This is particularly useful, as a random stream can be represented as a special kind of variable which can be updated with the seed of the pseudo-random number generator, and whose value() method returns the next pseudo-random number from the stream, at the same time updating the seeds. Class var defines also a pure abstract report() method which can be used to display a report about the content of the class.

The statistics class represents a base class for a number of predefined statistic functions. In addition to the methods and attributes defined in its parent class var, statistics contains an attribute representing the confidence level of the computed result.

The trremoval class is used to model algorithms dealing with the removal of the initialization bias from a sequence of observations. Suppose that the simulation run produces the sequence of observations $Y_1, Y_2, \ldots Y_n$ for some parameter of interest. In general using the whole sequence to compute the statistics is dangerous as there is a bias associated to artificial or arbitrary initial conditions. One method to overcome this limitation is to divide the sequence of observations into two phases: first an initialization phase including observations $Y_1, Y_2, \ldots Y_d$, followed by a data-collection phase $Y_{d+1}, Y_{d+2} \ldots Y_n$. The parameter d, $0 < d \ll n$ is called *truncation point*. Classes inheriting from trremoval are used to identify the truncation point d according to some specific algorithm.

mean

This class is used to compute a confidence interval for the mean of a sequence of *statistically uncorrelated* observations $Y_1, Y_2, \ldots Y_n$. The point estimator $\hat{Y}$ of the mean $\bar{Y}$, is computed as [17]:

$$\bar{Y} = \frac{1}{n} \sum_{i=1}^{n} Y_i \tag{8.1}$$

The approximate $100(1-\alpha)\%$ confidence interval for $\bar{Y}$ is computed as

$$\bar{Y} - t_{\alpha/2,f}\sigma(Y) \leq E(Y) \leq \bar{Y} + t_{\alpha/2,f}\sigma(Y) \tag{8.2}$$

where $t_{\alpha/2,f}$ is the $100(1-\alpha/2)\%$ percentage point of a t-distribution with f degrees of freedom; that is, $t_{\alpha/2,f}$ is defined by $P(t \geq t_{\alpha/2,f} = \alpha/2$. Also,

$$\sigma^2(Y) = \frac{1}{n(n-1)} \sum_{i=1}^{n} (\bar{Y} - Y_i)^2$$

accum

This class is used to compute a time-weighted sum of observations $Y_1, Y_2, \ldots Y_n$, $n > 1$ with timestamps respectively $t_1, t_2, \ldots t_n$. The result is computed as

$$A = \sum_{i=1}^{n-1} Y_i(t_{i+1} - t_i)$$

repmean

One method of computing the mean of observations $Y_1, Y_2, \ldots Y_n$ is called the method of independent replications. The simulation is repeated a total of R times, each run using a different random number stream and independently chosen initial conditions. Let $Y_r i$ be the ith observation within replication r, for $i = 1, 2, \ldots n_r$ and $r = 1, 2, \ldots R$. For fixed replication r, $Y_{r1}, Y_{r2}, \ldots Y_{rn_r}$ is a possibly autocorrelated sequence of observations. However, for different replications $r \neq s$, Y_{ri} and Y_{sj} are statistically independent. We compute the sample mean $\bar{Y}_r$ within each replication r by

$$\bar{Y}_r = \frac{1}{n_r} \sum_{i=1}^{n_r} Y_{ri} \qquad r = 1, 2, \ldots R$$

The R sample means $\bar{Y}_1, \bar{Y}_2, \ldots \bar{Y}_n$ are statistically independent and identically distributed. Then, a confidence interval for the mean $\bar{Y}$ can be computed by Eq. 8.2. The method reset() is used in class repmean to start a new replication.

Note that this class depends on the trremoval class. This is because a strategy for removing the initialization bias must be provided when creating an object of

type repmean. By default a simple strategy consisting of deleting the first 20% of each replication is employed.

bmeans

The method of *batch means* divides the output data from one replication (after deleting an appropriate amount of the initial observations) into a number of batches, and then treating the means of these batches as if they were independent. Let us consider the sequence $Y_{d+1}, Y_{d+2}, \ldots Y_n$ after deleting the first d observations. We form k batches of size $m = (n-d)/k$ and compute the batch means as:

$$\bar{Y}_j = \sum_{i=(j-1)n+1}^{jn} Y_{i+d} \tag{8.3}$$

for $j = 1, 2, \ldots k$ (we assume k divides $n-d$ evenly). The variance of the sample mean is estimated by

$$\frac{S^2}{k} = \frac{1}{k}\sum_{j=1}^{k}\frac{(\bar{Y}_j - \bar{Y})^2}{k-1} = \frac{\sum_{j=1}^{k}\bar{Y}_j^2 - k\bar{Y}^2}{k(k-1)} \tag{8.4}$$

where $\bar{Y}$ is the overall sample mean of the data after deletion. The batch means $\bar{Y}_1, \bar{Y}_2, \ldots \bar{Y}_k$, even if not independent, are approximately independent for large batch sizes. The reader is referred to [16, 17, 60] for guidelines on setting appropriate values for the batch size k.

Within the class bmeans the user can call the method reset() is used to start a new replication.

Note that this class depends on the trremoval class. This is because a strategy for removing the initialization bias must be provided when creating an object of type bmeans. By default a simple strategy consisting of deleting the first 20% of the observations is employed.

welch

This class implements the graphical procedure of Welch [99] for identifying the length of the initial transient period. Let us consider R independent replications, of length n each: replication r produces the observations $Y_{r1}, Y_{r2}, \ldots Y_{rn}$, where $r = 1, 2, \ldots R$. First, we compute the average across the repetitions:

$$\bar{Y}i = \frac{1}{R}\sum_{j=1}^{R} Y_{ji} \qquad i = 1, 2, \ldots n \tag{8.5}$$

Ensemble average are smoothed by plotting a moving average, considering time window of length w:

$$Y_j(w) = \begin{cases} \dfrac{1}{2w+1} \displaystyle\sum_{m=-w}^{w} \bar{Y}_{j+m} & w+1 \le j \le n-w \\ \dfrac{1}{2j-1} \displaystyle\sum_{m=-j+1}^{j-1} \bar{Y}_{j+m} & 1 \le j \le w \end{cases} \tag{8.6}$$

The graph of $Y_j(w)$ is less variable as j increases. The truncation point is the value of j at which the plot definitely stabilizes.

The user should invoke the update() method of this class in order to add a new observation value for the current replication. The method reset() is used to terminate the current replication, and start a new one. The plot is produced by calling the method report(); it produces a file named `rep.<histogram_name>` which can be processed by the `gnuplot` tool [42].

histogram

This class computes an histogram profile from the inserted values $Y_1, Y_2, \ldots Y_n$. The user supplies an expected lower bound and upper bound for the observed values. Also, the user specifies the number of cells (bins) in which the histogram will be divided. Instances of the class histogram do not compute anything using the value() method. Users are expected to call the report() method to print the informations concerning an histogram. A sample printout is presented in Fig. 8.7.

The problem of initialization bias

One of the most difficult problems in steady-state simulation output analysis is the removel of the initialization bias [16]. The problem can be stated as follows. Let I the set of initial conditions for the simulation model and assume that as $n \to \infty$, $P(X_n \le x|I) \to P(X \le x)$, where $\{X_i\}$ is the stochastic process corresponding to the sequence of observed values from the simulation, and X is the steady-state random variable. The steady-state mean of the process $\{X_i\}$ is $\mu = \lim_{n\to\infty} E(X_n|I)$. The problem with the use of the estimator $\bar{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i$ for a finite n is that $E(X_n|I) \neq \mu$, and thus $X(\bar{X}_n|I) \neq \mu$.

As an example, consider the computation of the average waiting time on a simple $M/M/1$ queue shown in Fig. 8.8. This is a system made of a single server with an unlimited buffer (queue) of waiting customers served in FIFO order. The interarrival time of customers is modeled by an exponentially distributed random variable with rate $\tau = 0.09$ and service rate $\omega = 0.1$. Fig. 8.8 shows the mean waiting time computed after averaging 50 independent replications of the simulation experiment, and after the plot has been smoothed with Welch' approach, with window size $w = 500$. We can see that the first d observations of the mean waiting time, with

```
          Name /    Obs /   N /      min /      max /     mean /  std.dev
        Normal /  50000 /  22 /    -4.02 /     4.09 /    -0.00 /     1.00

bin/ low.bound/ bin% / cum% /   Histogram                        /count
  0/    -INFTY/  0.00/  0.00/                                    /0
  1/     -5.00/  0.00/  0.00/                                    /0
  2/     -4.50/  0.00/  0.00/.                                   /1
  3/     -4.00/  0.01/  0.01/.                                   /4
  4/     -3.50/  0.07/  0.08/.                                   /35
  5/     -3.00/  0.52/  0.60/.                                   /258
  6/     -2.50/  1.74/  2.34/***                                 /870
  7/     -2.00/  4.47/  6.81/********                           /2236
  8/     -1.50/  9.19/ 16.00/****************                   /4596
  9/     -1.00/ 15.05/ 31.05/**************************         /7524
 10/     -0.50/ 19.23/ 50.27/************************************/9613
 11/      0.00/ 18.84/ 69.12/*********************************** /9421
 12/      0.50/ 15.13/ 84.24/**************************          /7564
 13/      1.00/  9.17/ 93.42/****************                   /4586
 14/      1.50/  4.31/ 97.73/*******                              /2155
 15/      2.00/  1.66/ 99.39/***                                  /830
 16/      2.50/  0.48/ 99.87/.                                    /241
 17/      3.00/  0.11/ 99.98/.                                    /55
 18/      3.50/  0.02/100.00/.                                    /9
 19/      4.00/  0.00/100.00/.                                    /2
All remaining bins are 0
```

Figure 8.7: Sample histogram printout

$d \approx 1000$, are consistently lower than the real waiting time. If those values are included in the computation of the mean waiting time, the result will be highly biased and the confidence interval will be larger.

Currently, there is no automatic method for detecting the length of the initialization bias which can be proven correct in every situation. Different approaches have been proposed in the literature [101], although their general effectiveness is dubious, and strongly depends on the specific simulation model to which they are applied. For that reason, simulation packages usually implement one of the following simple strategies:

- Removing a prefix consisting of a fixed fraction (eg, 20%) from the sequence of observations;
- Performing a very long simulation run such that the effect of the initial transient period can be neglected.

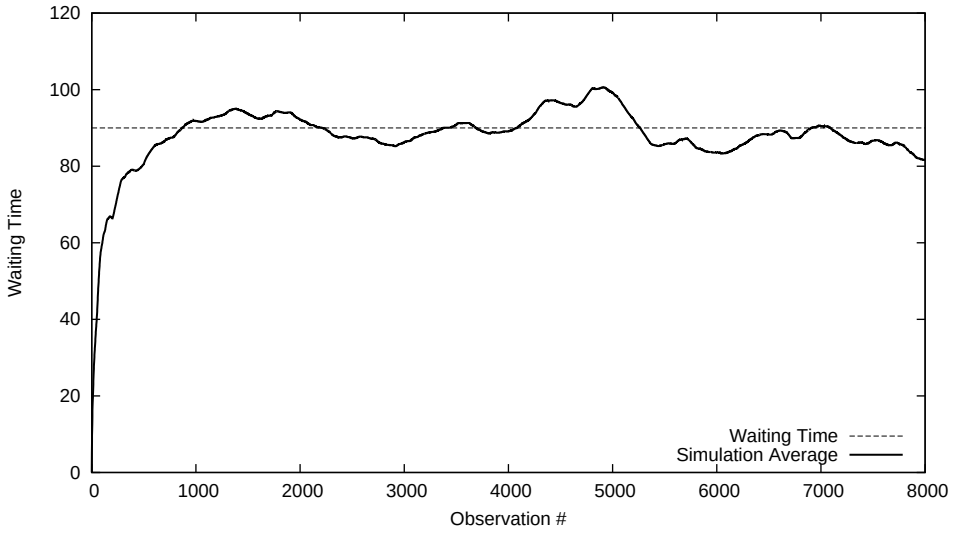


Figure 8.8: Average waiting time for the first 8000 customers of an $M/M/1$ queue from 50 independent replications. Moving averages with window size $w = 500$ are plotted

trremoval_const

This class represents a very simple strategy for removing the initialization bias. It basically discards the first d observations from the sequence $Y_1, Y_2, \ldots Y_n$, where d is specified by the user. The value() method returns the vector of values $Y_{d+1}, Y_{d+2} \ldots Y_n$, if more than d values have been inserted using the update() method.

trremoval_frac

This class implements the initialization bias removal strategy consisting in removing a prefix of the sequence of observations $Y_1, Y_2, \ldots Y_n$ of length $\lfloor n \times f \rfloor$, with $0 \leq f \leq 1$, and $\lfloor \cdot \rfloor$ denoting rounding to the nearest lower integer.

9

UML-Ψ Tool Description

In this chapter we describe the UML-Ψ prototype tool which has been built in order to show how the general approach described in Part II can be applied in practice.

9.1 Introduction

The UML-Ψ tool parses a software model annotated according to the UML profile described in Chapter 5. The model has to be exported in XMI format [76], which is an XML-based representation of the UML model. Unfortunately, the XMI notation described in the standard can be implemented in different (and incompatible) ways by CASE tool vendors. This constitutes an obstacle to large-scale adoption of XMI as platform-neutral representation of UML models. UML-Ψ understands the XMI dialect used by the freely available ArgoUML modeling tool [2], and its commercial counterpart Poseidon [80]. ArgoUML version 0.12 and Poseidon version 1.4 have been tested. Note that UML-Ψ is known *not* to work with Poseidon version 1.6, as the XMI format it employs is different and incompatible with previous versions.

UML-Ψ builds an internal representation of the UML model; only the portions of the model which are relevant for the performance evaluation model construction are considered. At this point the internal UML model representation is used to build the simulation performance model. The simulation model is based on the one presented in Fig, 4.4, p. 46.

Once the performance model is set up, UML-Ψ starts the simulation by activating the simulation processes representing the workloads and the resources. The workload processes create the requests arriving to the system, which in turn will activate the corresponding activity diagrams. The simulation stops when one of the following conditions becomes true first:

- The simulation time reached a user-defined maximum value;
- All the performance measures computed during the simulation converged within the given confidence interval width.

In Fig. 9.1 we depict the high-level structure of the UML-Ψ model processing framework. The UML model is created using a CASE tool. At the moment, UML-Ψ has been tested with the models generated by ArgoUML [2] version 0.12 and its

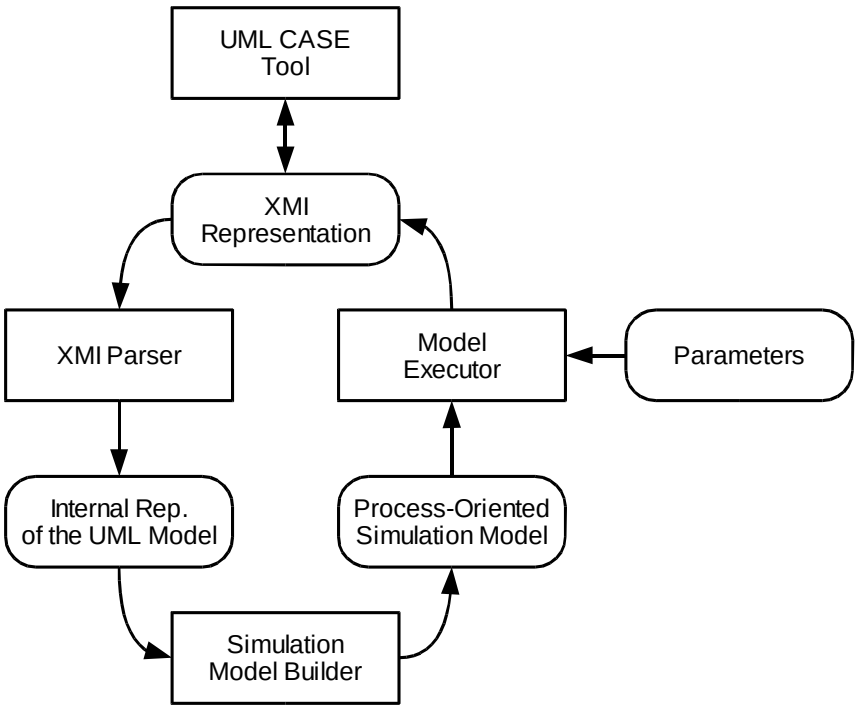


Figure 9.1: The UML-Ψ model processing framework

commercial counterpart Poseidon [80], version 1.4. The annotated UML model has to be exported in XMI format. Due to different implementations of the XMI format by different tool vendors, it is at the moment not possible to use UML-Ψ with other CASE tools. In fact the XMI specification [76] did not attempt to create an official metamodel for UML.

Once the XMI representation of the UML software system has been created, the UML-Ψ tool parses the XML document and builds an internal simplified representation of the relevant portions of the UML SA.

From this internal representation, a simulation model is derived. The model is executed by considering both the parameters specified by the user (tagged values associated to UML elements), and also by including a configuration file. This configuration file can be an arbitrary fragment of Perl code. After this code has been parsed, the Perl interpreter environment (modified by every declaration contained in the configuration file) is used to parse the tagged values. Hence, the configuration file may be used to define Perl variables which are used inside tagged values. The user may then explore different performance behaviors for different values of these variables by simply changing the configuration file, without affecting the UML model.

Finally, the simulation performance model is executed. The computed results are inserted into the XMI document as tagged values associated with the UML elements they refer. It is possible to use the CASE tool to reopen the UML model in order

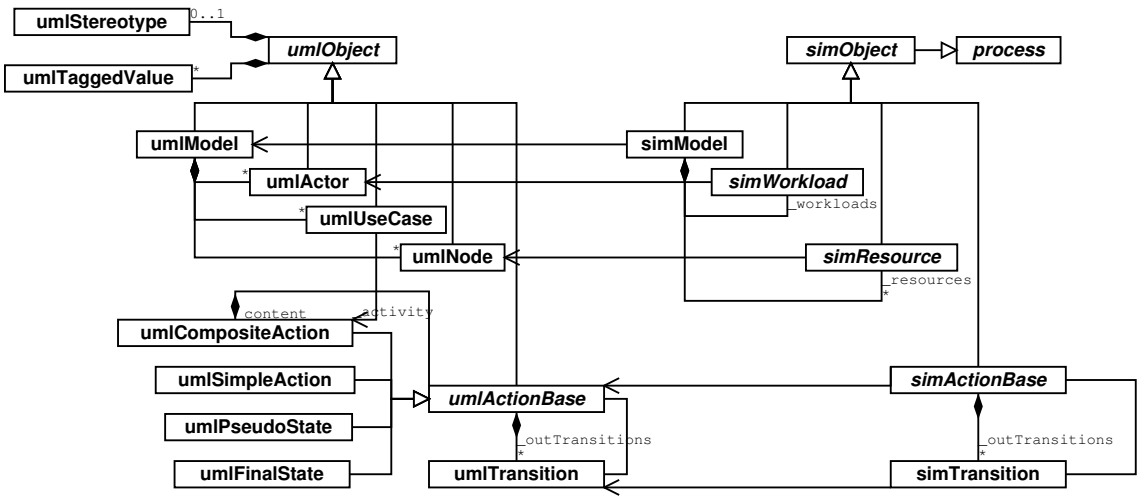


Figure 9.2: UML-Ψ class diagram, main part

to see the results. The performance modeling cycle may be repeated many times, until the performance goals have been satisfied.

9.2 UML-Ψ class diagram

A portion of the class diagram for the UML-Ψ tool is given in Fig. 9.2. The diagram shows on the left part the classes used for representing the UML model which is reconstructed from the XMI representation provided as input. On the right part, portion of the classes which are used to build the simulation model can be seen. We observe the almost direct correspondence between UML elements and simulation processes. There is a one-to-one mapping between a workload process and the corresponding UML actor; the simulation model is as a whole associated to the UML model. Finally, each resource (active or passive) corresponds to a node in the depoloyment diagram. Simulation actions are associated to action states in Activity diagrams, and transitions between actions are the simulation counterpart of transitions between action states.

9.3 UML model representation

The class hierarchy concerning the UML model has its root in the umlObject class. This class contains the common attributes and basic operations of all UML model elements. Each UML object can be stereotyped with at most one stereotype, and may have an arbitrary number of tagged values associated with it. Stereotypes and tagged values are represented as objects of class umlStereotype and umlTaggedValue

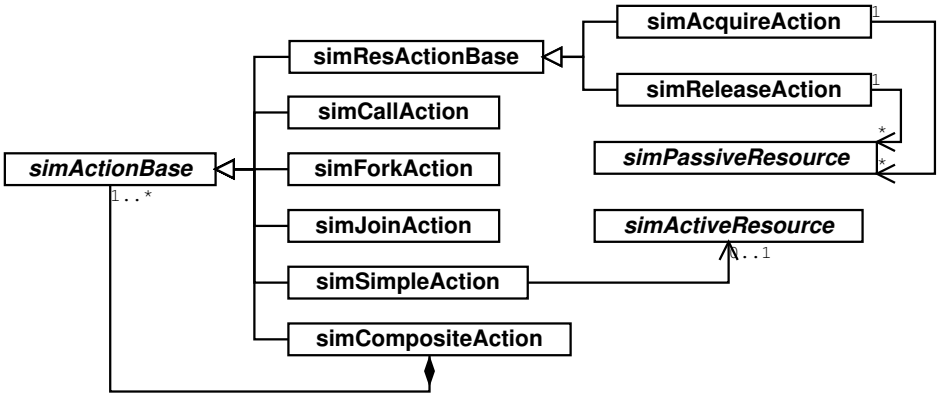


Figure 9.3: Class diagram for UML-Ψ simulation actions

respectively. A UML model is represented by class umlModel, which contains objects of class umlActor, umlUseCase and umlNode. They represent actors, use cases and deployment diagram nodes respectively. A use case is associated with a composite action which describes its behavior.

The common features of UML actions are represented in class umlActionBase. This abstract class contains a reference to a list of outgoing transitions (umlTransition). A transition, in turn, is associated to a source and a target action. The different kind of actions in UML models are represented by classes inheriting from umlActionBase. These are: composite actions (umlCompositeAction), simple actions (umlSimpleAction), pseudo states (umlPseudoState, which correspond to fork and join nodes, for example) and the final state (umlFinalState).

9.4 Simulation model representation

The structure of the simulation model, whose top portion is reported on the right side of Fig. 9.2, is similar to that just seen for the UML model representation. The root class of the hierarchy is simObject (derived from the process class of the `libcppsim` library). Simulation actions, workloads and resources are represented by class simActionBase, simWorkload and simResource respectively, all inheriting from the base simObject class.

The class hierarchy representing simulation actions is shown in Fig. 9.3. The class hierarchy follows a pattern similar to that of the corresponding UML model representation. The root class is simActionBase. Its dynamic behavior (that is, the body of the inner_body() method inherited from the process class) is not specified; this allows subclasses to implement specific behaviors, according to their type. Subclasses are defined to implement atomic actions (simSimpleAction), composite actions (simCompositeAction), fork and join nodes (simForkAction and simJoinAction respectively), actions dealing with passive resources (simResActionBase), and the special call action (simCallAction). A call action is needed to overcome a limitation of Ar-

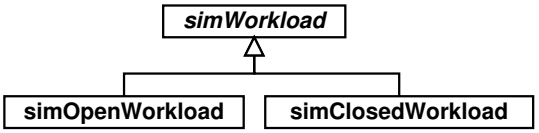


Figure 9.4: Class diagram for UML-Ψ simulation workloads

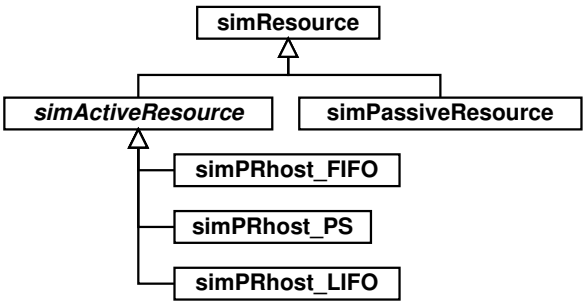


Figure 9.5: Class diagram for UML-Ψ simulation resources

goUML and Poseidon, which at the moment do not provide functionalities to define composite actions, even if they are defined in the UML standard. As a workaround, a composite action is modeled as follows. The user must specify a use case associated with the content of the composite action. This is necessary as ArgoUML/Poseidon do not allow users to create an activity diagram without it being used to specify a class or use case behavior. At this point, a simple (atomic) action is used where the composite action would have been put. This atomic action is stereotyped as ≪PAcompositeStep≫. A tagged value labeled `ActivateUC` must contain the name of the use case to activate when the action is to be executed.

The class diagram representing simulation workloads, depicted in Fig. 9.4, is very simple; a simWorkload virtual base class is used to represent a generic workload, whereas specialized simOpenWorkload and simClosedWorkload classes are used to model open and closed simulation workload processes respectively.

Finally, Fig. 9.5 depicts the class diagram of UML-Ψ processes representing active and passive resources. The simResource virtual base class is the root of the resource processes hierarchy. Resources are specialized in active resources, or processors (simActiveResource) and passive resources (simPassiveResource). Active resources are specialized again depending on their scheduling policy. Hence, we define a class representing a simulation process of a processor with FIFO scheduling policy (simPRHost_FIFO), one for the LIFO scheduling policy (simPRHost_LIFO) and one for the Processor Sharing (PS) scheduling policy (simPRHost_PS). Adding more scheduling policies to the simulation engine requires the user to derive a suitable class from simActiveResource.

10

The NICE Case Study

10.1 Introduction

In this chapter we apply the performance evaluation approach and the UML-Ψ tool previously introduced to the NICE case study from [7]. We show how performance analysis can be performed on early UML models in order to verify if the SA satisfies its performance requirements set by the system commissioner. This case study is in part motivated by the fact that it proved to be difficult to handle using analytical software performance evaluation techniques (in particular, using Process Algebras) due to the state-space explosion problem [24, 32]. The simulation-based approach we apply is of course immune to such problem.

Naval Integrated Communication Environment (NICE) is a project developed by Marconi Selenia. It provides communications for voice, data and video in a naval communication environment. It also provides remote control and monitoring in order to detect equipment failures in the transmission/reception radio chain. It manages the system elements and data distribution services, enables system aided message preparation, processing, storage, retrieval distribution and purging, and it implements radio frequency transmission and reception, variable power control and modulation and communications security techniques. The system involves several operational consoles that manage the heterogeneous system equipment including the ATM based Communication Transfer System (CTS) through blind Proxy computers.

On a gross grain the Software Architecture is composed of the NICE Management subsystem (NICE MS), CTS and EQUIP subsystems, as highlighted in Fig. 10.1. The WORKSTATION subsystem represents the management entity, while the PROXY AGENT and the CTS PROXY AGENT subsystems represent the interface to control the EQUIP and the CTS subsystems, respectively.

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

The more critical component is the NICE MS subsystem. It controls both internal and external communications and it satisfies the following class of requirements:

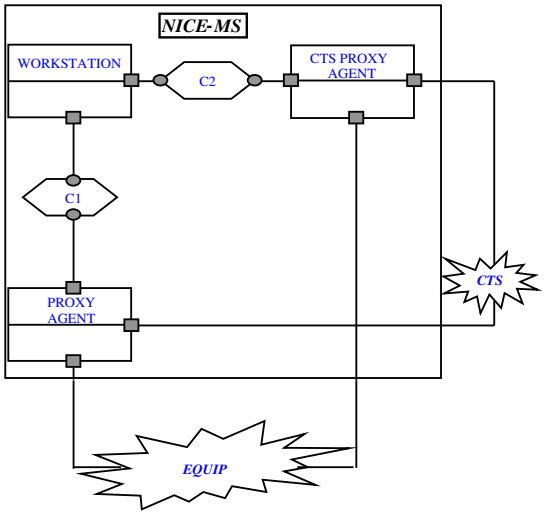


Figure 10.1: NICE static software description

fault and damage management, system configuration, security management, traffic accounting and performance management. All these class of requirements must satisfy some particular performance constraints. For the sake of the presentation, in this work we focus on two scenarios representing two crucial activities of the NICE system: *Equip Configuration* and *Recovery* activities. The two scenarios are described in Fig. 10.2. The estimated execution times of the various actions are exponentially distributed random variables; the mean values were provided by the system developers and are reported in Fig. 10.3.

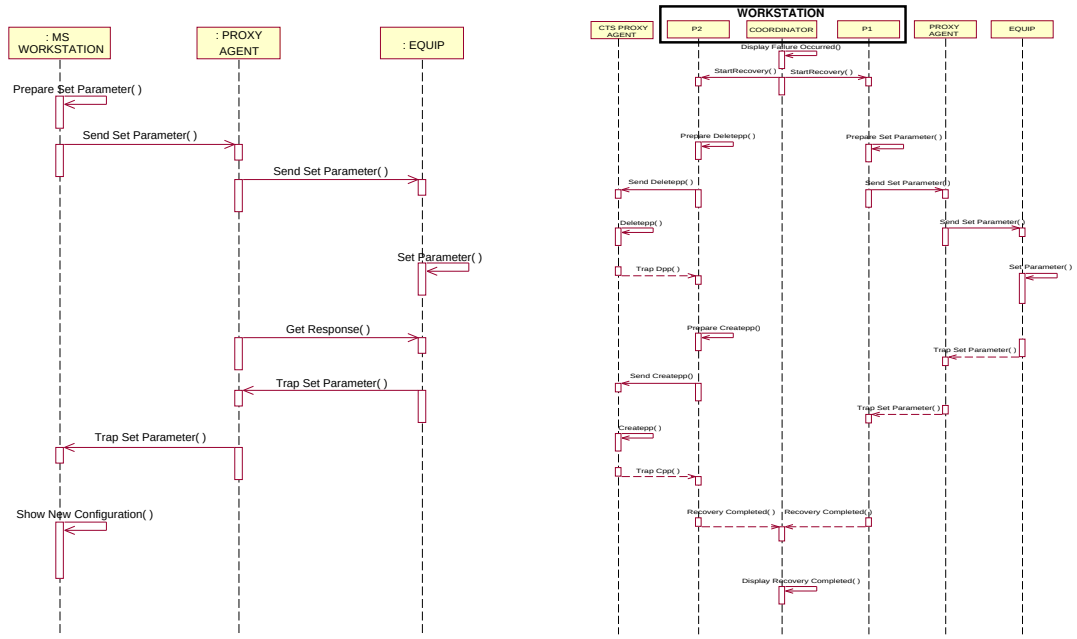
10.2 Equip Configuration Scenario

The configuration scenario is activated by an operator when new parameter setting of one or more equipments is required. The system configuration activity consists in a set of actions having the aim the reconfiguration of any equipment (see Fig. 10.2(a)).

The performance constraint for this activity, as required by the system developers, is: *"The equipment configuration has to be performed within 5 seconds (with a tolerance of 1 second)"*.

10.3 Recovery Scenario

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



(a) Configuration Scenario (b) Recovery Scenario

Figure 10.2: NICE configuration and recovery scenarios

The performance requirement for this activity, as required by the system developers, is: *"A recovery has to be performed within 5 seconds (with a tolerance of 1 second), when a failure occurs"*.

For the sake of the modeling, as shown in Fig. 10.2(b),the WORKSTATION subsystem is decomposed in three main components: COORDINATOR, P1 and P2 where P1 and P2 are auxiliary components interacting with PROXY AGENT subsystem and CTS PROXY AGENT subsystem, respectively, while COORDINATOR represents the control logics of the WORKSTATION subsystem.

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

10.4 Simulation Modeling

In order to apply the simulation-based modeling technique, it is necessary to translate the sequence diagrams of Fig. 10.2 into activity diagrams. This can be done easily, resulting in the activity diagrams depicted in Fig. 10.4(a) (which corresponds to the Configuration scenario of Fig. 10.2(a)) and the activity diagram of Fig. 10.4(b)

Action	Mean Execution time (sec.)
PrepareSetParameter	1.00
SendSetParameter	0.01
SetParameter	2.00
GetParameter	0.01
TrapSetParameter	0.01
ShowNewConfiguration	1.00

(a) Configuration Scenario

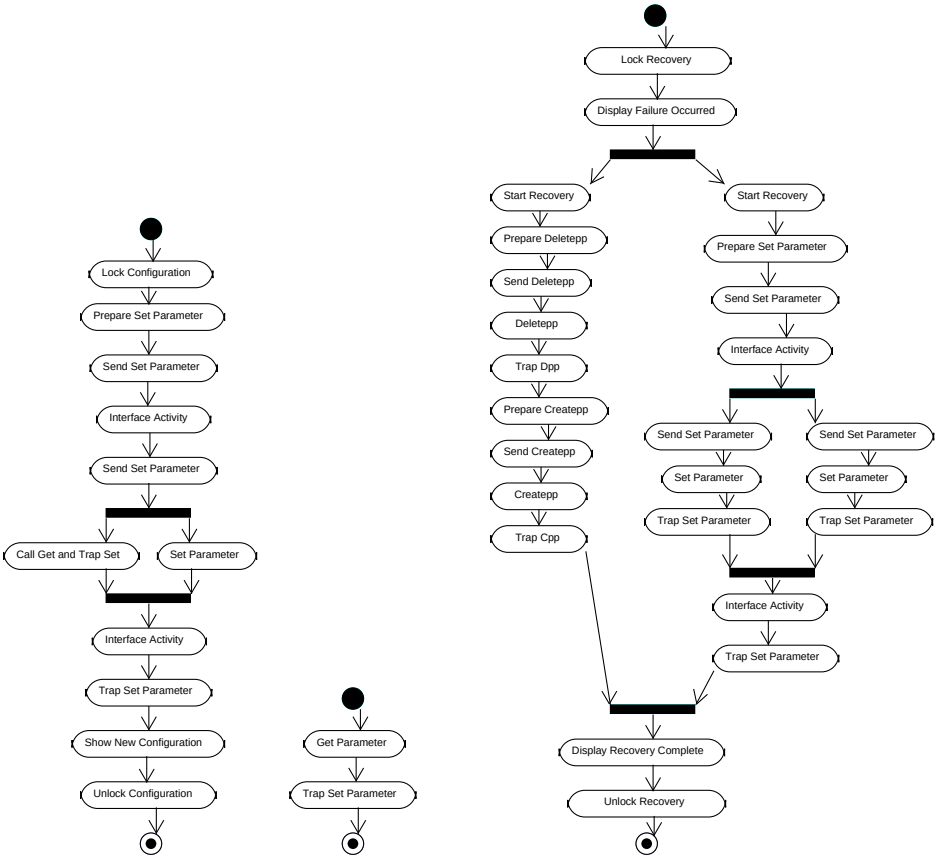
Action	Mean Execution time (sec.)
Display Failure Occurred	0.00
Start Recovery	0.00
PrepareSetParameter	1.00
SendSetParameter	0.01
SetParameter	2.00
GetParameter	0.01
TrapSetParameter	0.01
TrapCreatepp	0.01
TrapDeletepp	0.01
PrepareDeletepp	1.00
SendDeletepp	0.01
Deletepp	1.00
PrepareCreatepp	1.00
SendCreatepp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

(b) Recovery Scenario

Figure 10.3: Mean execution times

(corresponding to the Recovery scenario of Fig. 10.2(b)).

Note that the system we are simulating is synchronous, meaning that when an equipment is being configured (resp. repaired), then no other equipment can be configured (resp. repaired) at the same time, but must wait until the current corresponding operation has been completed. In order to simulate this behavior it is necessary to use two passive resources in order to simulate a lock on the scenarios. When executing a scenario it is first necessary to get the lock; if no configuration



(a) Configuration Scenario (b) Recovery Scenario

Figure 10.4: Activity diagram for the Configuration and Activity Scenario. The Configuration diagram (on the left) has a composite state ("Get and Trap Set") whose content is described by the small diagram of Fig. 10.4(a)

(recovery) operation is currently running, then the lock is granted immediately. If the lock is not available, the configuration (recovery) request is put on a queue. We compute the mean execution time of the scenarios, including the contention time spent waiting for another running scenario to complete. The service demand for each action state was set as in Fig. 10.3.

A simplified representation of the resulting simulation model is given in Fig. 10.5. Each node represents a simulation process and arrows indicate the process activation order. This graph shows the correspondence between the activity diagrams and the simulation processes. The dotted arrows indicate requests or releases of passive resources, which in our example are denoted as borderless gray boxes labeled as "Lock Configuration Scenario" and "Lock Recovery Scenario". As explained above, these resources are used to guarantee that only a single configuration and recovery

scenario is executed at the same time.

We simulate the system considering an increasing number N of equipments, for $N = 1 \ldots 6$. We assume that the time between successive configuration or recovery operations on the same equipment are exponentially distributed with mean $15s$. Simulation results in terms of average execution times of the Configuration and Recovery scenarios are shown in Table 10.1, which are also plotted in Fig. 10.6 as a function of N. The table displays also the total execution time of the simulations on a Linux/Intel machine running at 900Mhz, with 256MB or RAM.

N	**Configuration Scenario**		**Recovery Scenario**		**Simulation**
	Mean Exec. Time (s)	Requirement Satisfied?	Mean Exec. Time (s)	Requirement Satisfied?	**Exec. Time**
1	4.02	yes	6.61	no	$56s$
2	4.68	yes	7.64	no	$1m09s$
3	5.50	yes	10.26	no	$1m37s$
4	6.87	no	13.70	no	$1m29s$
5	8.95	no	17.66	no	$2m44s$
6	10.94	no	23.97	no	$2m51s$

Table 10.1: Computed mean execution times for the Configuration and Recovery scenarios, for different number N of equipments. The rightmost column reports the execution time of the simulation program

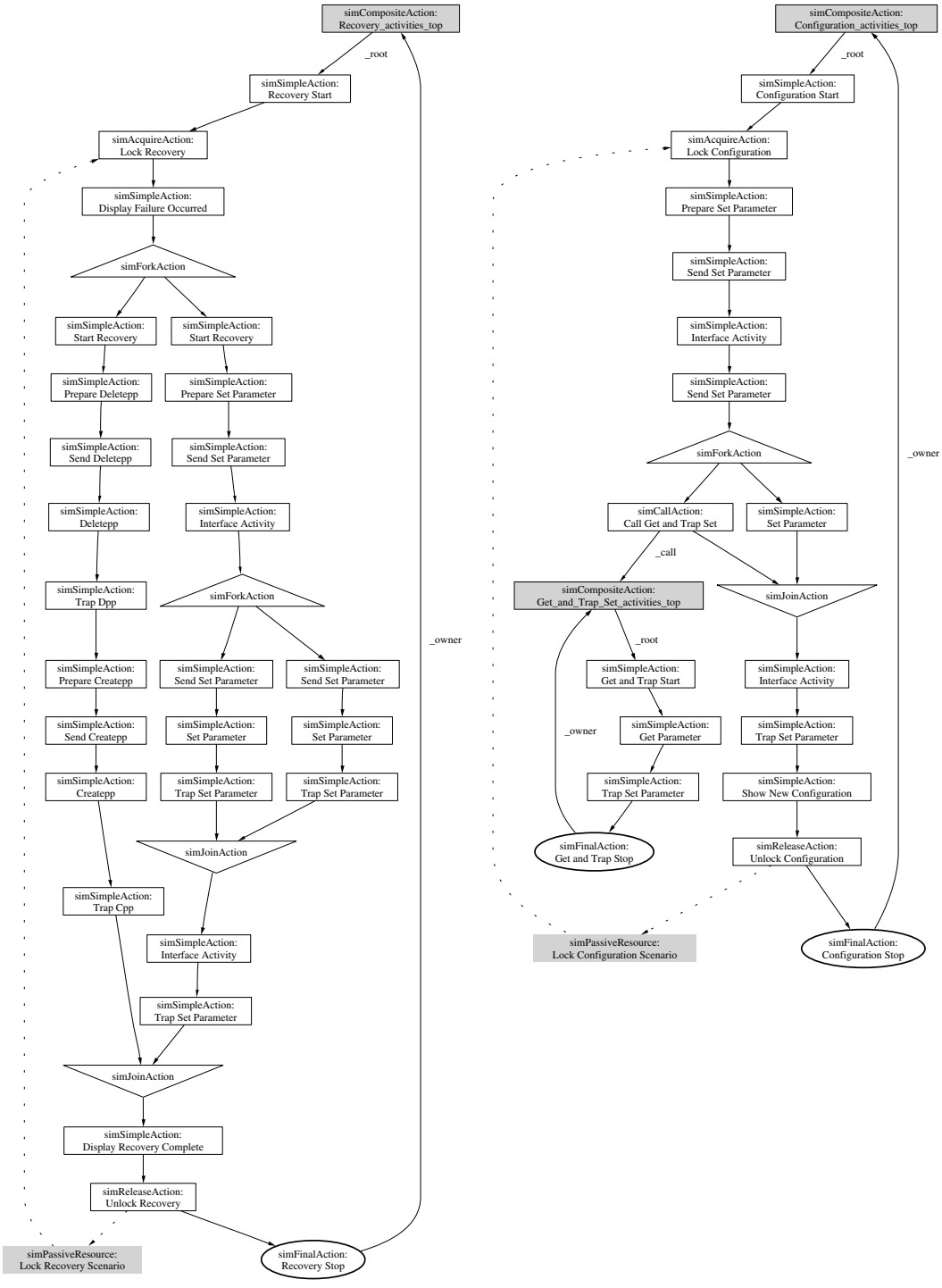


Figure 10.5: Structure of the simulation model

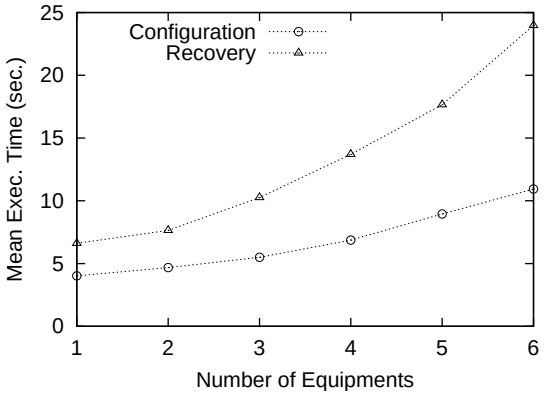


Figure 10.6: Graph of the execution times from Table 10.1

Conclusions

11 Conclusions

11.1 Contributions of this work

In this work we considered performance evaluation of software systems during the early development stages. We proposed an approach for quantitative evaluation of the performances of software systems described at a high level of detail with annotated UML diagrams. We developed a prototype tool for automatic translation of the software model into a process-oriented simulation model. The tool executes the simulation model and inserts performance results into the original UML diagram.

We developed a UML notation based on the UML Performance Profile [75] to add quantitative, performance-oriented informations to UML use case, activity and deployment diagrams. Use case diagrams are used to represent workloads driving the system. Activity diagrams describe the computations which are performed, including acquiring/releasing passive resources. Finally, deployment diagrams describe the resources available: they include active resources (processors), and passive resources. UML is the de-facto standard for specifying and describing software artifacts; it is widely used and easy to learn and understand. Software performance evaluation techniques using UML as a specification notation have the advantage of not requiring users to learn a completely new and often ad-hoc notation.

We defined a process-oriented simulation model for representing the dynamic behavior of the software system described by the UML diagrams. Choosing a simulation model as the performance model has been motivated by several factors. First, simulation is a very general modeling technique which allows unconstrained model representation. Then, the structure of the performance model is very similar to that of the corresponding UML model. This makes the mapping process between software and performance models very easy, and thus less error-prone. Moreover, this has the advantage that it is extremely easy to report performance results back into the original software model. This is a particularly viable feature, which is not provided by many software performance evaluation approaches present in the literature. We defined the simulation (meta)model in Chapter 4, and more detailed informations are given in Chapter 6. We showed in Chapter 5 how UML elements can be annotated with quantitative, performance-oriented informations.

We developed a prototype tool, called UML-Ψ, for parsing annotated UML mod-

els and translating them into the simulation model. UML-Ψ is written in the C++ programming language, and is based on a general-purpose process-oriented simulation library which has been developed. The simulation library can be used to implement arbitrary simulation models, as it provides SIMULA-like process scheduling facilities and basic statistical functions for analyzing simulation results. The UML-Ψ tool parses an XMI description of the UML model, and builds the corresponding simulation model. Model parameters are obtained from the annotations (stereotypes and tagged values) associated with the UML elements. Tag values can be given according to a simple syntax; they are evaluated using the Perl interpreter library, which allows users to include arbitrary Perl expressions. The simulation model is executed until all the performance measures can be computed with the requested accuracy, or until the simulation time reaches a user-defined termination value. Results are associated to the UML elements they refer, so the user can easily get feedback. The `libcppsim` library has been described in Chapter 8, and the UML-Ψ tool in Chapter 9.

It should be noted that, in order to use our approach, the software modeler only needs to learn the syntax used for specifying the performance annotations, as described in Chapter 5. No knowledge of performance evaluation techniques are requested, as the steps of performance model generation, execution and feedback are done automatically by the UML-Ψ tool.

Moreover, we defined an extension of the performance evaluation approach which can be applied to modeling and evaluation of mobile systems described in UML. We use annotated UML activity diagrams to describe how the configuration of the mobile system evolves as the result of location changes; both physical mobility (devices changing their physical location), and logical mobility (code fragments migrating from one processor to another) can be represented. Describing the extension of the performance evaluation approach to mobility modeling was the topic of Chapter 7.

Finally, we presented a simple case study in which an early performance assessment of an integrated naval communication system was performed. The case study was partly motivated by the fact that it proved to be difficult to handle using Process Algebra-based performance modeling, due to the state space explosion problem [32]. Simulation does not suffer the state space explosion problem, and we were able to compute the performance measures with a very limited resource usage on the host machine. The case study has been described in Chapter 10.

This work is, to the best of our knowledge, the first one applying the ideas presented in the UML Performance Profile to the derivation of a simulation performance model (Petriu and Shen in [77] derive LQN performance models from UML diagrams annotated according with the profile). We propose some modifications to the UML profile in order to better adapt it to direct implementation of the performance model. We contribute a general-purpose simulation library (`libcppsim`) and the prototype performance evaluation tool UML-Ψ. Using the prototype we can argue that the proposed software performance evaluation approach is sound and potentially useful in practice. Both the `libcppsim` library and the UML-Ψ tool can

be easily extended to add new functionalities due to their modular nature. They will be released as open source software, so we hope that people will be able to use them as building blocks for other projects.

Comparison with previous approaches Other existing simulation-based performance modeling approaches to software performance evaluation (some of which have been described in Sec. 1.2) differ from our proposal as they make use of custom-defined annotations of UML models and/or custom representations of the UML model. This is expected, as the UML Performance Profile was not available until recently. Our aim was to test the effectiveness of what is the proposed standard for UML-based performance evaluation. In particular, our approach differs from that of Arief and Speirs [6] in that we use different kind of UML diagrams to describe the software system; also, we use the standard XMI notation to export the software description in XML. With respect to the approach by de Miguel et al. [36], we do not focus explicitly on real-time applications and constraints modeling; we considered a more generic performance evaluation framework, which however could be specialized with minimum effort to real-time applications (the UML Performance Profile [75] alreay defines a sub-profile for real-time modeling). Finally, with respect to the approach proposed by Hennig [49], our proposal is fully based on simulation rather than emulation of the system behavior. We do not generate synthetic components emulating the behavior of real components, but instead built a whole simulated system. Both approaches have their merits: building a simulation model of the whole system does not require the software architect to develop a "skeleton" application on which emulated components must be plugged, and thus simplifies the performance evaluation steps. On the other hand, using a partly developed system on which (emulated) parts can be changed may yield more accurate performance results.

11.2 Relevant publications

The work described in this thesis has been the subject of some publications, which we now list in chronological order.

In [8] we discussed the problems behind simulation-based performance evaluation of software systems. In [12] we illustrated the basic ideas behind the approach proposed here. The use of use case and activity diagrams for performance evaluation purposes was introduced, along with the simulation-based performance model and an early prototype of the UML-Ψ tool. In [10, 11] deployment diagrams were introduced for representing active resources, and the approach was extended accordingly. The integrated approach for mobility and performance evaluation modeling was introduced in [13]; finally, in [7] the Naval Integrated Communication Environment (NICE) case study was presented, together with a new extension of the performance model including passive resources.

11.3 Open Problems

The work presented in this thesis can be extended in several directions. We propose here some considerations on possible future extensions of the approach previously described.

UML-related improvements Currently we do not use all types of UML diagrams which are defined. While the subset we consider proved to be expressive enough to represent many real-world situations, different users may be more comfortable with different kind of diagrams for expressing the same things. Thus, the UML profile could be extended to include more types of UML diagrams. In particular, it would be very useful to include sequence diagrams and state charts in the performance modeling approach. Sequence diagrams provide a different way to express computations with respect to activity diagrams. However, sequence diagrams make communications between elements explicit, while in activity diagrams communication is modeled as an action requesting service to a network resource. State chart diagrams are defined in the UML standard as a superset of activity diagrams. However, state charts put more emphasis on the notion of "event", while activity diagrams put more emphasis on the notion of "activity" taking some amount of time to complete before starting the next one. The simulation model should be easy to expand to include these different views of the system.

Simulation-related improvements Other possible improvements may be related to the performance indices which are currently computed by the UML-Ψ tool. At the moment, only a limited set of measures is computed; more performance indices (for example quantiles or distributions) would certainly be useful. The UML performance profile already defines some of them, so it is just a matter of extending our simulator. The `libcppsim` library can also be extended both by providing a richer set of statistical data analysis functions, and by providing some more high-level simulation entities on top of the simulation process abstraction in order to facilitate the modeling process. Useful simulation entities may include condition variables, resources, more advanced communication primitives (such as the mailboxes provided by CSIM [89]). Some ideas may be derived from the excellent DEMOS package [26], originally written for the Simula language, which has already inspired the development of the `libcppsim` library.

Further improvements It would be extremely useful to integrate the proposed approach into a general framework consisting of different environments for software systems specification and automated evaluation and prediction of quantitative and qualitative characteristics of the system. Currently the UML-Ψ tool can be used as a stand-alone method for performance evaluation of UML specifications. However, our approach may also be considered part of an integrated software development plat-

form allowing users to specify a software system using different notations (including UML), and providing different kind of analysis to be performed on the system: reliability analysis, performance analysis with possibly different approaches, security analysis, and others.

Overall, the proposed simulation-based approach to software performance modeling demonstrated that UML can be used also for quantitative software analysis. The next logical step would be to develop another UML profile for evaluating other quantitative characteristics of the software, such as reliability. Hopefully, the UML-Ψ tool could be extended to include different modules able to perform different kind of analysis on the same SA.

Bibliography

[1] D. Al-Dabass, editor. *Proc. of ESM'03, the 17th European Simulation Multiconference*, Nottingham, UK, June 9–1 2003. SCS–European Publishing House.

[2] ArgoUML – Object-oriented design tool with cognitive support. http://www.argouml.org/.

[3] L. B. Arief. *A Framework for Supporting Automatic Simulation Generation from Design*. PhD thesis, Dept. of Computing Science, University of Newcastle Upon Tyne, July 2001.

[4] L. B. Arief and N. A. Speirs. Automatic generation of distributed system simulations from UML. In *Proc. of ESM '99, 13th European Simulation Multiconference*, pages 85–91, Warsaw, Poland, June 1999.

[5] L. B. Arief and N. A. Speirs. Using SimML to bridge the transformation from UML to simulation. In *Proc. of One Day Workshop on Software Performance and Prediction extracted from Design*, Heriot-Watt University, Edinburgh, Scotland, Nov. 1999.

[6] L. B. Arief and N. A. Speirs. A UML tool for an automatic generation of simulation programs. In Proceedings of WOSP 2000 [83], pages 71–76.

[7] S. Balsamo, A. Di Marco, P. Inverardi, and M. Marzolla. Experimenting different software architectures performance techniques: a case study. Tech. Rep. TR SAH/46, MIUR Sahara Project, Sept. 2003. To Appear in Proc. WOSP 2004.

[8] S. Balsamo, M. Grosso, and M. Marzolla. Towards simulation-based performance modeling of UML specifications. Tech. Rep. CS-2003-2, Dipartimento di Informatica, Università Ca' Foscari di Venezia, Mestre, Italy, Jan. 2003.

[9] S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. Software performance: state of the art and perspectives. Tech. Rep. CS-2003-1, Dip. di Informatica, Università Ca' Foscari di Venezia, Jan. 2003.

[10] S. Balsamo and M. Marzolla. A simulation-based approach to software performance modeling. Tech. Rep. TR SAH/44, MIUR Sahara Project, Mar. 2003.

[11] S. Balsamo and M. Marzolla. A simulation-based approach to software performance modeling. In P. Inverardi, editor, *Proc. ESEC/FSE 2003*, pages 363–366, Helsinki, FI, Sept. 1–5 2003. ACM Press. An extended version appeared as [10].

[12] S. Balsamo and M. Marzolla. Simulation modeling of UML software architectures. In Al-Dabass [1], pages 562–567.

[13] S. Balsamo and M. Marzolla. Towards performance evaluation of mobile systems in UML. In B. di Martino, L. T. Yang, and C. Bobenau, editors, *The European Simulation and Modelling Conference 2003*, pages 61–68, Naples, Italy, Oct. 27–29 2003. EUROSIS-ETI.

[14] S. Balsamo, V. D. N. Personè, and R. Onvural. *Analysis of Queueing Networks with Blocking*. Kluwer Academic Publishers, 2001.

[15] S. Balsamo and M. Simeoni. Deriving performance models from software architecture specifications. In Proceedings of ESM'01 [82].

[16] J. Banks, editor. *Handbook of Simulation*. Wiley–Interscience, 1998.

[17] J. Banks, J. S. Carson, B. L. Nelson, and D. M. Nicol. *Discrete-Event System Simulation*. Prentice Hall, 2001.

[18] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[19] H. Baumeister, N. Koch, P. Kosiuczenko, and M. Wirsing. Extending activity diagrams to model mobile systems. In M. Aksit, M. Mezini, and R. Unland, editors, *NetObjectDays*, volume 2591 of *Lecture Notes in Computer Science*. Springer, 2003.

[20] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

[21] F. Belina and D. Hogrefe. The CCITT specificatoin and description language SDL. *Computer Networks and ISDN Systems*, 16(4):311–341, 1988.

[22] A. Benzekri, A. Valderruten, O. Hjiej, and D. Gazal. Deriving queueing networks performance models from annotated LOTOS specifications. In R. Pooley and J. Hillston, editors, *Computer Performance Evaluation: Modelling Techniques and Tools, 6th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation, TOOLS '92*, number 10 in Edits, pages 120–130, Edinburgh, Sept. 1992. Edinburgh University Press.

[23] S. Bernardi, S. Donatelli, and J. Merseguer. From UML sequence diagrams and statecharts to analysable Petri net models. In Proceedings of WOSP 2002 [84].

[24] M. Bernardo, P. Ciancarini, and L. Donatiello. ÆMPA: A process algebraic description language for the performance analysis of software architectures. In Proceedings of WOSP 2000 [83], pages 1–11.

[25] G. Birthwistle and C. Tofts. Getting Demos models right. (I) Practice. *Simulation Practice and Theory*, 8(6–7):377–393, Mar. 2001.

[26] G. Birtwistle. *DEMOS–A system for discrete event modelling on Simula*. MacMillan Press, 1979.

[27] G. Birtwistle and C. Tofts. Getting Demos models right. (II) ... and theory. *Simulation Practice and Theory*, 8(6–7):395–414, Mar. 2001.

[28] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation With Computer Science Applications*. Wiley–Interscience, 1998.

[29] T. Bolognese and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[30] R. Brown. Calendar queues: A fast $O(1)$ priority queue implementation for the simulation event set problem. *Commun. ACM*, 31(10), Oct. 1988.

[31] T. Clark, A. Evans, S. Kent, S. Brodsky, and S. Cook. A feasibility study in rearchitecting UML as a family of languages using a precise OO meta-modeling approach - version 1.0. Tech rep., Department of Computer Science, University of York, Sept. 2000. http://www.cs.york.ac.uk/puml/mmf.pdf.

[32] D. Compare, A. D. Marco, A. D'Onofrio, and P. Inverardi. Our experience in the integration of process algebra based performance validation in an industrial context. Tech. Rep. TR SAH/047, MIUR Sahara Project, Oct. 2003.

[33] V. Cortellessa and R. Mirandola. PRIMA–UML: a performance validation incremental methodology on early UML diagrams. In Proceedings of WOSP 2002 [84].

[34] Comprehensive Perl Archive Network (CPAN). http://www.cpan.org/.

[35] O.-J. Dahl and K. Nygaard. SIMULA–an ALGOL-based simulation language. *Commun. ACM*, 9(9):671–678, Sept. 1966.

[36] M. De Miguel, T. Lambolais, M. Hannouz, S. Betgé-Brezetz, and S. Piekarec. UML extensions for the specifications and evaluation of latency constraints in architectural models. In Proceedings of WOSP 2000 [83], pages 83–88.

[37] R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. KLAIM: a kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[38] G. Franks, A. Hubbard, S. Majumdar, D. C. Petriu, J. Rolia, and C. M. Woodside. A toolset for performance engineering and software design of client-server systems. *Performance Evaluation*, 24(1–2):117–135, 1995.

[39] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[40] R. M. Fujimoto. *Parallel and Distributed Simulation Systems*. Wiley–Interscience, 1999.

[41] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of reusable Object-Oriented programming*. Addison-Wesley, 1995.

[42] gnuplot home page. http://www.gnuplot.info.

[43] H. Gomaa and D. A. Menascé. Performance engineering of component-based distributed software systems. In *Performance Engineering – State of the Art and Current Trends*, volume 2047 of *Lecture Notes in Computer Science*, pages 40–55. Springer, 2001.

[44] V. Grassi and R. Mirandola. UML modelling and performance analysis of mobile software architectures. In M. Gogolla and C. Kobryn, editors, *UML*, volume 2185 of *Lecture Notes in Computer Science*. Springer, 2001.

[45] G. Gu and D. C. Petriu. XSLT transformation from UML models to LQN performance models. In Proceedings of WOSP 2002 [84].

[46] K. Helsgaun. A portable C++ library for coroutine sequencing. Datalogiske Skrifter (Writings on Computer Science) 87, Roskilde University, 1999.

[47] A. Hennig and H. Eckardt. Challenges for simulation of systems in software performance engineering. In Proceedings of ESM'01 [82], pages 121–126.

[48] A. Hennig, A. Hentschel, and J. Tyack. Performance prototyping – generating and simulating a distributed IT-system from UML models. In Al-Dabass [1].

[49] A. Hennig, D. Revill, and M. Ponitsch. From UML to performance measures – simulative performance predictions of IT-systems using the Jboss application server with OMNET++. In Al-Dabass [1].

[50] H. Hermanns, U. Herzog, and J.-P. Katoen. Process algebra for performance evaluation. *Theoretical Computer Science*, 274(1-2):43–87, 2002.

[51] F. Hoeben. Using UML models for performance calculation. In Proceedings of WOSP 2000 [83], pages 77–82.

[52] D. Hogrefe, E. Heck, and B. Möller-Clostermann. Hierarchical performance evaluation based on formally specified architectures. *IEEE Transactions on Computers*, 40(4):500–513, April 1991.

[53] G. Iazeolla, editor. *Performance Evaluation, Special Issue on Performance Valudation of Software Systems*, volume 45. Elsevier, July 2001.

[54] P. Kähkipuro. UML-based performance modeling framework for component-based distributed systems. In R. R. Dumke, C. Rautenstrauch, A. Schmietendorf, and A. Scholz, editors, *Performance Engineering*, volume 2047 of *LNCS*, pages 167–184. Springer-Verlag, 2001.

[55] K. Kant. *Introduction to Computer System Performance Evaluation*. McGraw-Hill, Int. Editions, 1992.

[56] P. J. B. King and R. J. Pooley. Derivation of Petri net performance models from UML specifications of communications software. In B. R. Haverkort, H. C. Bohnenkamp, and C. U. Smith, editors, *Computer Performance Evaluation / TOOLS*, volume 1786 of *Lecture Notes in Computer Science*, pages 262–276. Springer, 2000.

[57] L. Kleinrock. *Queueing Systems*, volume 1. J. Wiley, 1975.

[58] P. Kosiuczenko. Sequence diagrams for mobility. In J. Krogstie, editor, *Proc. of MobIMod workshop*, Tampere, Finland, Oct. 2002. Springer.

[59] S. S. Lavenberg. *Computer Performance Modeling Handbook*. Academic Press, New York, USA, 1983.

[60] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. McGraw–Hill, 3rd edition, 2000.

[61] P. L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47:159–164, 1999.

[62] `libxml`: the XML C library for Gnome. http://xmlsoft.org/.

[63] C. Lindemann, A. Thümmler, A. Klemm, M. Lohmann, and O. P. Waldhorst. Performance analysis of time-enhanced UML diagrams based on stochastic processes. In Proceedings of WOSP 2002 [84].

[64] The Linux kernel archive. http://www.kernel.org/.

[65] M. C. Little. JavaSim Home page. http://javasim.ncl.ac.uk/.

[66] M. C. Little and D. L. McCue. Construction and use of a simulation package in C++. Tech. Rep. 437, Department of Computing Science, University of Newcastle upon Tyne, July 1993.

[67] N. M. Macintyre, S. R. Kershaw, and N. Xenios. A toolset to support the performance evaluation of systems described in SDL. In *Proc. of TOOLS '94, Vienna, Austria*, pages 23–26. Speinger–Verlag, May 1994.

[68] N. Madvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[69] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modeling with Generalized Stochastic Petri Nets*. J. Wiley, 1995.

[70] N. Medvidovic, D. S. Rosenblum, and D. F. Redmiles. Modeling software architectures in the Unified Modeling Language. *ACM Trans. Soft. Engineering*, 11(1):2–57, 2002.

[71] R. Milner. *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, 1999.

[72] C. Nottegar, C. Priami, and P. Degano. Performance evaluation of mobile processes via abstract machines. *IEEE Transactions on Software Engineering*, 27(10):338–395, 2001.

[73] Object Management Group (OMG). http://www.omg.org/.

[74] Object Management Group (OMG). Unified modeling language (UML), version 1.4, Sept. 2001.

[75] Object Management Group (OMG). UML profile for schedulability, performance and time specification. Final Adopted Specification ptc/02-03-02, OMG, Mar. 2002.

[76] Object Management Group (OMG). XML Metadata Interchange (XMI) specification, version 1.2, Jan. 2002.

[77] D. C. Petriu and H. Shen. Applying the UML performance profile: graph grammar-based derivation of LQN models from UML specifications. In T. Field, P. G. Harrison, J. Bradley, and U. Harder, editors, *Computer Performance Evaluation / TOOLS*, volume 2324 of *Lecture Notes in Computer Science*. Springer, 2002.

[78] G. P. Picco, G.-C. Roman, and P. J. McGann. Reasoning about code mobility in mobile UNITY. *ACM Trans. On Software Engineering and Methodology*, 10(3):338–395, 2001.

[79] R. J. Pooley and P. J. B. King. The Unified Modeling Language and performance engineering. In *IEE Proceedings–Software*, volume 146, pages 2–10, Feb. 1999.

[80] Poseidon for UML. http://www.gentleware.com/.

[81] Precise UML Group. http://www.cs.york.ac.uk/puml.

[82] *Proc. of ESM'01, the 15th European Simulation Multiconference*, Prague, Czech Republic, June 6–9 2001.

[83] *Proc. of the Second International Workshop on Software and Performance (WOSP 2000)*, Ottawa, Canada, Sept. 2000. ACM Press.

[84] *Proc. of the Third International Workshop on Software and Performance (WOSP 2002)*, Rome, Italy, July 2002. ACM Press.

[85] G. Reggio, M. Cerioli, and E. Astesiano. Towards a rigorous semantics of UML supporting its multiview approach. In H. Hussmann, editor, *FASE 2001*, volume 2029 of *LNCS*, pages 171–186. Springer–Verlag, 2001.

[86] J. A. Rolia and K. C. Sevcik. The method of layers. *IEEE Transactions on Software Engineering*, 21(8):689–700, Aug. 1995.

[87] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[88] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[89] H. Schwetman. CSIM19: a powerful tool for building system models. In *Proceedings of the 33nd conference on Winter simulation*, pages 250–255. IEEE Computer Society, 2001.

[90] M. Shaw and D. Garlan. *Software Architecture: perspectives on an emerging disciline*. Prentice-Hall, 1996.

[91] C. U. Smith. *Performance Engineering of Software Systems*. Addison-Wesley, 1990.

[92] C. U. Smith and L. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, 2002.

[93] I. Sommerville. *Software Enginnering, 6th edition*. Addison-Wesley, 2000.

[94] B. Stroustrup. *The C++ Programming Language*. Addison–Wesley, third edition, 1997.

[95] C. Tofts and G. Birtwistle. A denotational semantics for a process-based simulation language. *ACM Trans. on Modeling and Computer Simulation*, 8(3):281–305, July 1998.

[96] A. Varga. The OMNET++ discrete event simulation system. In Proceedings of ESM'01 [82], pages 319–324.

[97] L. Wall, T. Christiansen, and J. Orwan. *Programming Perl*. O'Reilly & Associates, third edition, July 2000.

[98] G. Waters, P. Linington, D. Akehurst, P. Utton, and G. Martin. Permabase: predicting the performance of distributed systems at the design stage. *IEE Proceedings–Software*, 148(4):113–121, Aug. 2001.

[99] P. D. Welch. The statistical analysis of simulation results. In S. Lavenberg, editor, *The Computer Performance Modeling Handbook*, pages 268–328. Academic Press, New York, 1983.

[100] M. Wermelinger and J. L. Fiadeiro. Connectors for mobile programs. *IEEE Transactions on Software Engineering*, 24(5):331–341, 1998.

[101] K. P. White Jr., M. J. Cobb, and S. C. Spratt. A comparison of five steady-state truncation heuristics for simulation. In *Proceedings of the 32nd Winter Simulation Conference*, pages 755–760, Orlando, Florida, 2000. Society for Computer Simulation International.

[102] L. G. Williams and C. U. Smith. PASA: a method for the performance assessment of software architectures. In Proceedings of WOSP 2002 [84].

[103] C. M. Woodside, J. Neilson, S. Petriu, and S. Mjumdar. The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. *IEEE Transactions on Computers*, 44:20–34, 1995.

[104] World Wide Web Consortium (W3C). XSL Transformations (XSLT), version 1.0, Nov. 1999. http://www.w3.org/TR/1999/REC-xslt-19991116.

List of PhD Thesis

TD-2004-1 Moreno Marzolla
"Simulation-Based Performance Modeling of UML Software Architectures"

TD-2004-2 Paolo Palmerini
"On performance of data mining: from algorithms to management systems for data exploration"