Performance modelling for system-level design

Performance Modelling for System-Level Design

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van
de Rector Magnificus, prof.dr. R.A. van Santen,
voor een commissie aangewezen door het College
voor Promoties in het openbaar te verdedigen op
donderdag 25 november 2004 om 16.00 uur

door

Bartholomeus Desiderius Theelen

geboren te Heerlen

Dit proefschrift is goedgekeurd door de promotoren:

prof.ir. M.P.J. Stevens, prof.dr.ir R.H.J.M. Otten

en

prof.dr. H. Corporaal

Copromotor:

dr.ir. J.P.M. Voeten

Cover design: R.W.C. Theelen

Printed by: Universiteitsdrukkerij Technische Universiteit Eindhoven

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Theelen, Bartholomeus D.

Performance Modelling for System-Level Design / by Bartholomeus Desiderius Theelen. - Eindhoven : Technische Universiteit Eindhoven, 2004.

Proefschrift. - ISBN 90-386-1633-3

NUR 992

Trefw.: systeemanalyse / ingebedde systemen / object-georienteerd modelleren / Markov-ketens / stochastische simulatie.

Subject headings: performance evaluation / hardware-software codesign / modelling / Markov processes / discrete event simulation.

Summary

Industry faces the challenge of designing hardware/software systems of increasing complexity within ever-shortening design times. Commonly, the design process involves the consideration of several design alternatives for realising the desired functionality. Early in the design process, the choice for a specific design alternative may have a deep impact on the performance of the final implementation. To assist the designer in taking well-founded design decisions, system-level design methods can be applied. System-level design methods are frameworks for structuring the earliest phases of the design process with the intention to find a feasible design within minimal time. They focus on constructing models that allow the analysis of functional and non-functional properties before actually realising the system in hardware and software. This thesis contributes to developing such a system-level design method and focuses on aspects of performance modelling and performance evaluation.

Markov chains are mathematical structures for which classical techniques exist that allow both analytical computation and simulation-based estimation of long-run average performance metrics. These techniques are however not directly applicable to more common performance metrics, which requires taking the occurrence of a certain event into account. The reduction technique proposed in this thesis allows evaluating conditional long-run averages based on applying the classical techniques to a reduced version of the Markov chain that formalises the behaviour of a system. An important advantage of using the reduction technique is that performance evaluation speed is improved considerably in case of simulation-based estimation.

Many performance metrics for industrial hardware/software systems concern algebraic combinations of several long-run averages. Both analytical computation and simulation-based estimation of such complex long-run averages can be accomplished by first evaluating the constituent long-run averages and then combining their results. For the case of simulation-based estimation, this thesis proposes an algebra of Confidence Intervals that also allows analysing the accuracy of point estimates for complex long-run averages. This algebra is an alternative to the complicated process of directly deriving point estimators for complex long-run averages.

Although Markov chains form a suitable basis for evaluating performance properties, their construction does not match with the compositional way designers reason about the working of hardware/software systems. System-level design methods use expressive modelling languages such as the Parallel Object-Oriented Specification Language (POOSL) to construct models. A prerequisite for obtaining credible performance results based on models developed with a modelling language is that the models are amenable to Markov-chain based performance analysis techniques. The semantics of a POOSL model defines a probabilistic timed labelled transition system, which can be transformed into a Markov chain. Based on this result, a framework for reflexive performance analysis with POOSL is proposed, which involves extending a POOSL model with additional variables and behaviour to define performance monitors. By simulating the extended model, such performance monitors enable reasoning about the behaviour of the system, which is specified in that same model.

Although the framework for reflexive performance analysis provides a sound basis for obtaining credible performance results, a practical problem arises when applying the Markov-chain based performance analysis techniques for simulation-based estimation. The problem is related to the requirement of identifying a recurrent state of the system. Checking whether the currently visited state is equal to a previously visited state commonly requires too much compute time. Two different solutions are proposed. The first utilises knowledge that a designer may have about the reoccurrence of a starting point in the local behaviour of a component. The second is more generally applicable and matches a commonly used approach for simulation. Based on these approaches, POOSL library classes have been developed, which assist in defining performance monitors for (complex) long-run averages. They allow user-friendly application of the proposed mathematical techniques for accuracy analysis and contribute to obtaining credible performance results.

Originally, POOSL was introduced for the Software/Hardware Engineering (SHE) method. This system-level design method is extended in this thesis with guidelines for performance modelling. Performance modelling with SHE is structured in three stages. In the formulation stage, the concepts and requirements for a system are documented with various diagrams expressed in the Unified Modelling Language (UML). These diagrams comply with an intuitively applicable UML profile for SHE. This UML profile facilitates utilisation of POOSL in the formalisation stage, where the UML diagrams are unified into an executable POOSL model. Several useful modelling patterns are presented, which focus on making adequate abstractions with respect to performance properties. In addition, guidelines are provided for utilising the POOSL library classes for accuracy analysis. These guidelines assist in extending a POOSL model with performance monitors in accordance with the performance properties that are to be evaluated. In the evaluation stage, the actual evaluation of the performance properties is performed. The obtained analysis results then enable to judge whether the proposed concepts for realising the desired functionality satisfy the performance requirements.

The applicability of the proposed techniques and performance modelling framework has been investigated with several case studies. Two of them are briefly discussed in this thesis. The first was performed in cooperation with Alcatel Bell Antwerp and concerns the performance modelling of an Internet Router. The second is concerned with the performance modelling of a network processor and was performed in cooperation with IBM Research Laboratory Zürich. These case studies have provided valuable experiences with the proposed performance modelling framework and proved its suitability for analysing the performance of very complex systems.

Samenvatting

De industrie staat voor de uitdaging hardware/software systemen van toenemende complexiteit te ontwerpen in steeds kortere ontwerptijd. Tijdens het ontwerpproces worden gewoonlijk verschillende ontwerpalternatieven voor het realiseren van de gewenste functionaliteit tegen elkaar afgewogen. De keuze voor een bepaald ontwerpalternatief, welke vroeg in het ontwerpproces wordt genomen, kan een grote invloed hebben op de uiteindelijke prestatie van het systeem. Om de ontwerper te ondersteunen in het nemen van weloverwogen ontwerpbeslissingen kunnen systeemniveau ontwerpmethoden worden toegepast. Dit zijn raamwerken om de vroegste fasen van het ontwerpproces te structureren met als doel een geschikt ontwerp te vinden in minimale tijd. Zulke ontwerpmethoden concentreren zich op de ontwikkeling van modellen waarmee functionele en niet-functionele eigenschappen kunnen worden geanalyseerd voordat het systeem wordt gerealiseerd in hardware en software. Dit proefschrift draagt bij aan de ontwikkeling van een dergelijke systeemniveau ontwerpmethode en concentreert zich op aspecten omtrent het ontwikkelen van modellen voor prestatieanalyse en de prestatieanalyse zelve.

Markov ketens zijn wiskundige structuren waarvoor klassieke technieken bestaan om lange-termijn gemiddelden zowel analytisch te berekenen als te schatten met behulp van simulatie. Deze technieken zijn echter niet direct toepasbaar voor de meer gebruikelijke vorm van prestatiemetrieken waarbij rekening moet worden gehouden met het optreden van een bepaalde gebeurtenis. Dit proefschrift introduceert een reductietechniek die het mogelijk maakt om conditionele lange-termijn gemiddelden te evalueren door het toepassen van de klassieke technieken op een gereduceerde versie van de Markov keten die het gedrag van een systeem formaliseert. Een belangrijk voordeel van deze techniek is een sterke afname van de tijd die nodig is voor het schatten van prestatiemetrieken met behulp van simulatie.

Veel prestatiemetrieken voor industriële hardware/software systemen bestaan uit een algebraïsche combinatie van verschillende conditionele lange-termijn gemiddelden. Zowel analytische berekening als het schatten met behulp van simulatie van zulke complexe lange-termijn gemiddelden kan worden uitgevoerd door eerst de individuele conditionele lange-termijn gemiddelden separaat te bepalen en dan de resultaten te combineren. Voor het geval van simulatie maakt de in dit proefschrift geïntroduceerde algebra van Betrouwbaarheidsintervallen het nu ook mogelijk om de nauwkeurigheid van puntschattingen voor complexe lange-termijn gemiddelden te analyseren. Deze algebra is een alternatief voor het ingewikkelde process van het direct afleiden van puntschatters voor complexe lange-termijn gemiddelden.

Hoewel Markov ketens een geschikte basis vormen voor het evalueren van prestatie-eigenschappen, komt de manier waarop ze gespecificeerd moeten worden niet overeen met de compositionele manier waarop ontwerpers redeneren over de werking van systemen. Systeemniveau ontwerpmethoden gebruiken expressieve modelleertalen zoals de Parallel Object-Oriented Specification Language (POOSL) om modellen te ontwikkelen. Een voorwaarde om betrouwbare prestatieresultaten te krijgen bij het gebruik van een modelleertaal, is dat de ontwikkelde modellen geschikt zijn voor het toepassen van prestatieanalysetechnieken die op Markov ketens zijn gebaseerd. De semantiek van een POOSL model definieert een door stochasticiteit en tijd gekenmerkt transitiesysteem, dat kan worden omgezet in een Markov keten. Op basis van dit resultaat wordt een raamwerk voor reflexieve prestatieanalyse met POOSL geïntroduceerd. Dit raamwerk gaat uit van het uitbreiden van een POOSL model met extra variabelen en gedrag om prestatiemonitoren te definiëren. Door het uitgebreide model te simuleren, kunnen de prestatiemonitoren uitspraken doen over het gedrag van het systeem dat in datzelfde model is gespecificeerd.

Alhoewel het raamwerk voor reflexieve prestatieanalyse een goede basis is voor het verkrijgen van betrouwbare prestatieresultaten, is er een praktisch probleem in het geval van schatting met behulp van simulatie. Het probleem betreft de noodzaak om een toestand te identificeren waarin het systeem terugkeert. Het kost vaak te veel rekentijd om gedurende een simulatie te bepalen of de huidig bezochte toestand gelijk is aan een eerder bezochte toestand. Er worden twee oplossingen voorgesteld. De eerste maakt gebruik van kennis die een ontwerper heeft over een terugkerend startpunt in het lokale gedrag van een component. De tweede oplossing is meer generiek toepasbaar en komt overeen met een aanpak die veelal wordt gebruikt voor simulatie. Op basis hiervan zijn POOSL bibliotheekklassen ontwikkeld die de ontwerper assisteren in het definiëren van prestatiemonitoren voor (complexe) langetermijn gemiddelden. Deze klassen maken een gebruiksvriendelijke toepassing van de voorgestelde wiskundige technieken voor nauwkeurigheidsanalyse mogelijk en dragen bij tot het verkrijgen van betrouwbare prestatieresultaten.

POOSL is oorspronkelijk geïntroduceerd voor de Software/Hardware Engineering (SHE) methode. Dit proefschrift breidt deze systeemniveau ontwerpmethode uit met richtlijnen voor het ontwikkelen van modellen voor prestatieanalyse. SHE structureert de ontwikkeling van zulke modellen in drie fasen. In de formulatie fase worden de concepten en eisen voor een systeem gedocumenteerd met verschillende diagrammen die uitgedrukt worden in de Unified Modelling Language (UML). Deze diagrammen zijn in overeenstemming met een intuïtief UML profiel voor SHE. Dit UML profiel vergemakkelijkt het gebruik van POOSL in de formalisatie fase, waarin de UML diagrammen verenigd worden tot een uitvoerbaar POOSL model. Dit proefschrift reikt een aantal praktische modelleerpatronen aan die nadruk leggen op het maken van adequate abstracties met betrekking tot prestatie-eigenschappen. Daarnaast worden richtlijnen gegeven voor het gebruik van de POOSL bibliotheekklassen voor nauwkeurigheidsanalyse. Deze richtlijnen assisteren in het uitbreiden van een POOSL model met prestatiemonitoren. De daadwerkelijke evaluatie van prestatie-eigenschappen vindt plaats in de evaluatie fase. De resultaten daarvan onderbouwen de beoordeling of de voorgestelde concepten voor het realiseren van de gewenste functionaliteit leiden tot een systeem dat voldoet aan de prestatie-eisen.

De toepasbaarheid van de besproken technieken en het raamwerk voor het ontwikkelen van modellen voor prestatieanalyse is onderzocht met een aantal casussen. Dit proefschrift beschrijft twee daarvan in meer detail. De eerste casus is uitgevoerd in samenwerking met Alcatel Bell in Antwerpen en betreft de prestatieanalyse van een Internet Router. De tweede casus betreft de prestatieanalyse van een netwerkprocessor en is uitgevoerd in samenwerking met IBM Research Laboratory in Zürich. De casussen hebben de geschiktheid aangetoond van de in dit proefschrift geïntroduceerde theoriën voor het analyseren van de prestatie van zeer complexe systemen.

Acknowledgement

Completing my PhD project with writing this thesis would not have been possible without the support of my supervisors, many colleagues, friends and my family.

First of all, I would like to thank my first promotor Mario Stevens for giving me the opportunity to perform research in a very enthusiastic research group. Unfortunately, doctors diagnosed a terminal disease about three and a half months before my PhD defence. Although he still tried to support me in finishing this thesis, his condition rapidly declined. On 17 September 2004, he died at the age of 60. My colleagues and I miss his joyful conversations and his great enthusiasm in working with young people. I wish his family strength in dealing with the loss of a great husband, father and grandfather.

Next to Mario Stevens, my PhD project was supported by Ad Verschueren, Jeroen Voeten and Piet van der Putten. I am thankful to all of them for the many useful suggestions they gave. Especially the discussions with Jeroen Voeten were very inspiring and I am really grateful for being coached by him. His enthusiasm and ingeniousness helped a lot in solving many of the theoretical issues. Furthermore, I thank Marc Geilen, Leo van Bokhoven, Frank van Wijk and Mark Verhappen for the many discussions we had on all kinds of topics that are more or less related to this thesis. I also thank Marc Geilen and especially Leo van Bokhoven for further developing the SHESim and Rotalumis tools based on the requirements for my project. Rian van Gaalen is thanked for her support as secretary of our research group.

I thank Mario Stevens, Jeroen Voeten, Piet van der Putten for their efforts to arrange the case studies performed in cooperation with Alcatel Bell and IBM Research Laboratory. These case studies profoundly contributed to the identification of theoretical deficiencies of classic performance analysis techniques and helped in focussing on the applicability of the solutions proposed in this thesis. I would like to thank Alex Niemegeers from Alcatel Bell and Gjalt de Jong, who is now affiliated to Research in Motion, for granting the case studies on the Internet Router and Dataflow System and for explaining the working of these system. From IBM Research Laboratory, I owe gratitude to Ton Engbersen, Patricia Sagmeister and Andreas Herkersdorf (who is now at the University of Munich) for their support in the Network Processor case study. It was really interesting to share experiences with people from industry.

The last part of my PhD project was funded by PROGRESS through the ”Modelling and Performance Analysis of Telecommunication Systems” project. Within this project, I worked together with Zhangqin Huang, Ana Sokolova and Erik de Vink. I thank them for the discussions on both practical and theoretical topics. Furthermore, I thank Jinfeng Huang, who worked on a related PROGRESS project, for his interest in my work and for the discussions we had. Thanks go to two of my master students, Arjan van Ewijk and Rene Kramer, for their assistance in developing the performance models of the Dataflow System and Network Processor respectively.

The sudden death of Mario Stevens had impact on the final stages of my PhD project and caused some delay. I thank Ralph Otten for his immediate decision to take care and for his willingness to succeed Mario Stevens as my first promotor. I am really grateful to the members of the reading committee including Henk Corporaal, Jeroen Voeten, Onno Boxma and Andreas Herkersdorf for their willingness to sacrifice their leisure time in order to read and judge the final draft of this thesis. I thank the other committee members Martin Rem, Hans de Stigter and Johan Lukkien for their interest in my work and for the discussions we had.

Finally, I would like to thank my family and all my friends for their mental support. I am grateful to the Limburgse Mini Freaks for being patience with me when I did not spend the time that I should have spent in my task as the club's secretary.

Bart Theelen, September 2004

The research presented in this thesis is partly supported by PROGRESS, the Program for Research on Embedded Systems and Software of the Dutch organisation for Scientific Research NWO, the Dutch Ministry of Economic Affairs and the Technology Foundation STW through the funding of project EES.5202 "Modelling and Performance Analysis of Telecommunication Systems"

Contents

Chapter 1

Introduction

Hardware/software systems have become an essential backbone of human society. Well-known examples are telecommunication systems like telephony systems and computer networks allowing more and more people to communicate, automated control systems improving the efficiency of machinery, monitoring systems that observe the status of patients in a hospital and multi-media systems enhancing our ways of entertainment. The increasing amount of functionality that is to be performed by such systems with more stringent performance requirements has made their design an extremely complex task. Distributing functionality over a larger and often more heterogeneous set of components and a higher degree of dynamism in the data that is to be processed are examples of aspects causing an increasing difficulty in understanding and specifying the desired behaviour of a system. In order to remain competitive, industry faces the challenge of managing this difficulty within ever-shortening design times.

The growing design complexity, the increasing implementation costs, and the time-to-market imperative have urged the need for frameworks that structure and largely automate the design process. Such design methods intend to eliminate the risk of expensive design/implementation iterations by assisting designers in constructing models. A model enables to analyse the properties of a proposed design solution for a system before actually realising it in hardware and software. Based on evaluation results obtained for alternative design solutions, it can be decided which solution will serve as a basis for realising the system. Working out a design solution into a complete implementation with hardware and software, which then turns out not to have the desired performance, can bankrupt a company these days [148].

This thesis contributes to the development of a design method that allows the assessment of alternative concepts for realising a system during the earliest phases of its design. Particular focus is on the area of performance modelling, which concerns the construction of models for the purpose of evaluating performance properties. The next section introduces the problem domain of designing hardware/software systems in more detail and section 1.2 identifies several aspects that are to be considered when developing design methods. Section 1.3 elaborates on the issues addressed in this thesis, while an overview of the thesis structure is given in section 1.4.

1.1 Design of Hardware/Software Systems

The trajectory from product idea to tangible system involves the development of a desirable but a priori unknown realisation based on an initial specification. Such a *specification* prescribes in a comprehensible, precise and verifiable way the requirements for a system, its functional behaviour and other characteristics [86]. Often, a number of alternative solutions can be proposed for realising the desired functionality. Such *design alternatives* may differ in various ways, ranging from differently partitioning the system in components to using other algorithms for performing a certain functionality. Also, other values for *system parameters* like the size of a memory or the bandwidth of a communication medium imply different design alternatives. The virtually unlimited number of all possible design alternatives that can be realised with known technology is usually referred to as the *design space*.

Only a limited number of designs in the design space can satisfy all requirements for a system. These include functional and non-functional requirements. *Functional* requirements concern the correctness of the behaviour exposed by a system. An example of a desirable correctness property is the absence of deadlock. Not succeeding in meeting functional requirements may result in failures. A design is said to be *correct* if it satisfies all functional requirements. *Non-functional* requirements are related to the remaining characteristics of a system. Examples are cost, maintainability and performance properties like latency and throughput. A correct design that also satisfies all non-functional requirements will be called a *feasible* design. The design process can be described as the search for a feasible design. The next sections outline how this search can be structured and indicate the role of performance modelling.

1.1.1 System-Level Design

A common approach for managing the complexity of designing hardware/software systems is to distinguish a number of design phases. Consecutive design phases focus on refining the amount of detail with which the question of how to realise the specified functionality such that the desired non-functional properties are satisfied is answered. It is also common to say that the level of abstraction from implementation details decreases. The relation between design phases and levels of abstraction is often explicitly indicated by using the term *design level* instead of design phase.

At each design level, several options can be considered for adding details on how to realise certain functionality. This implies the need for exploring design alternatives and deciding which one most likely leads to satisfying the requirements (in an optimal way). Figure 1.1 shows a 3D-version of the abstraction pyramid originally introduced in [110]. It illustrates how *design decisions* taken at consecutive levels may lead from the initial specification for a system to its realisation. The main concepts for realising the functionality thought of when starting from the initial specification immediately restrict the number of designs that can result from the design process. This is because certain designs are based on main concepts that are not considered during the first level. Decisions taken during consecutive levels continue to restrict the number of possible resulting designs until a single realisation is found.

Design decisions taken earlier in the design process have in general a bigger impact

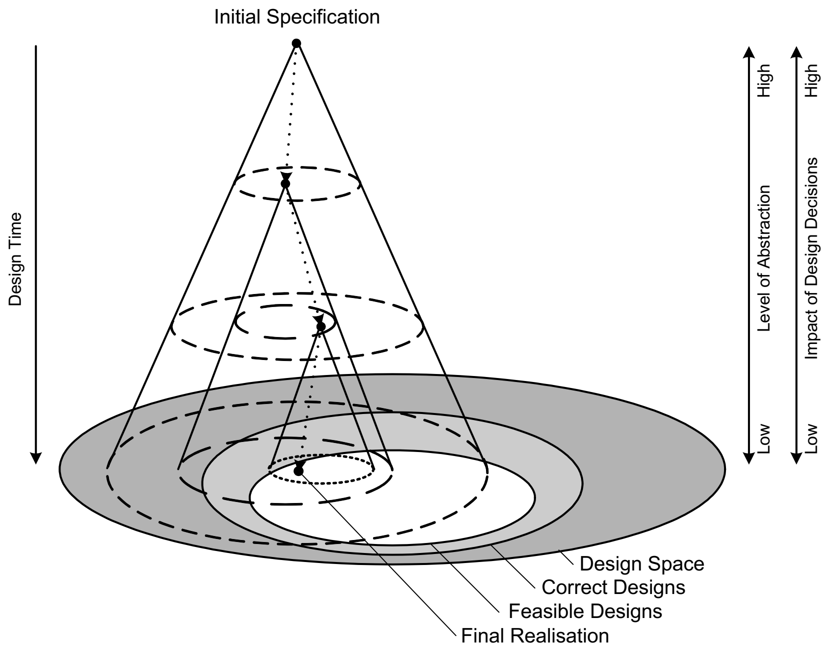


Figure 1.1: Exploring alternatives and taking decisions at different design levels.

on the properties of the final realisation than those taken during later phases [138]. For example, wrongly choosing between concepts like a bus-based architecture or a switch-based architecture when designing a router for some communication system can have more serious consequences in not satisfying the performance or cost requirements compared to choosing wrong physical dimensions of the system's output connectors. One reason for the high impact of wrong design decisions made at higher levels is that a higher reduction in the number of possible resulting designs is obtained due to these early decisions. Notice also that in case a wrong decision is made, there may be no feasible designs attainable anymore. Correction of these *design errors* is particularly expensive when they occurred early in the design process and remain undetected until later design phases [131]. This is due to the necessity of redoing (parts of) the design steps performed after taking the wrong decision.

This thesis concentrates on *system-level design*, which is an umbrella term for the earliest design phases. System-level design is concerned with developing the main concepts for a system and with evaluating whether these concepts will lead to the satisfaction of all requirements. Structured approaches for performing system-level design are of utmost importance for tackling the difficulty of making the right decisions during the earliest design phases and hence, for minimising design time.

1.1.2 Exploring Design Alternatives

Each design phase involves developing one or more alternatives for realising certain functionality. To investigate the feasibility of design alternatives, models can be used. A *model* is an approximate or abstract representation of a system, which is

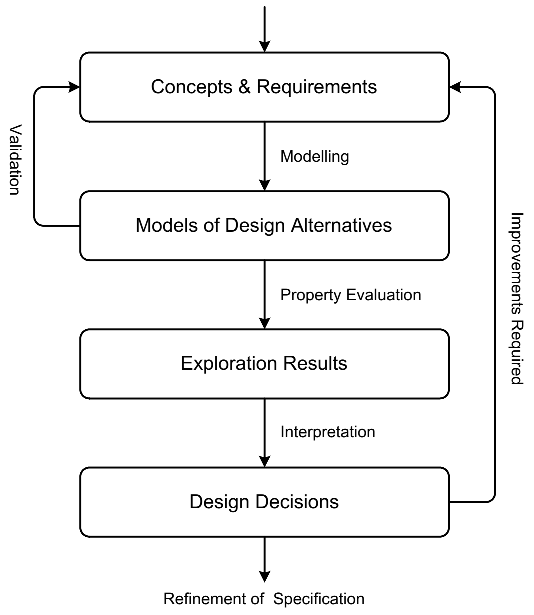


Figure 1.2: Exploring design alternatives based on models.

intended for the analysis of certain properties [177]. Hence, models allow investigating the implications of new ideas before actually realising the system according to these ideas. The evaluation results obtained based on models enable to judge what design alternative satisfies the requirements (in an optimal way).

A general approach for structuring the development of design alternatives and evaluating their properties based on models is shown in figure 1.2. Starting from some specification of the desired functionality, designers may come up with several concepts for realising this functionality such that the specified requirements are expected to be satisfied. To investigate whether the effects of the proposed concepts are indeed as expected, models are developed. The construction of models during the design process inherently involves making abstractions from implementation details since these details cannot be known in advance. Hence, modelling involves checking the *adequacy* of such abstractions in order to become convinced that those details that are thought to be essential for investigating the properties of interest are captured by a model. Moreover, a model must be *validated* against the proposed concepts for realising the specified functionality to ensure that they are adequately represented.

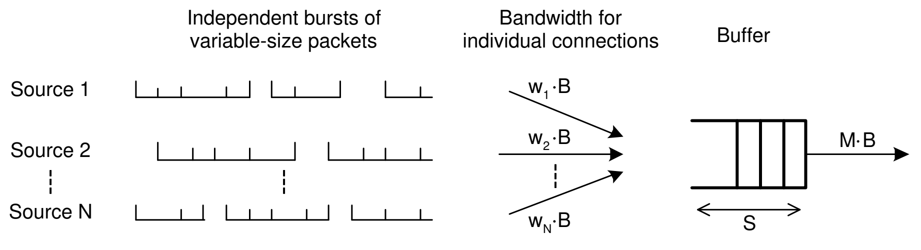
After evaluating the properties of interest, interpretation of the exploration results allows properly founding design decisions. In case all requirements are satisfied for certain design alternatives, these alternatives may serve as a basis for realising the system. A specific alternative may be favourable if it satisfies the requirements in a better way compared to other alternatives. Choosing for a feasible alternative leads to refining the design. However, if not all requirements are met, improvements for the proposed concepts are required or other concepts need to be thought of. In this case, certain steps in the involved design phase need to be redone, see also figure 1.2.

This thesis focuses on developing a framework for structuring the exploration of design alternatives during system-level design. Generally, the *actual* modelling of such alternatives and the evaluation of their properties is less time consuming at higher levels than at lower levels. This originates from the smaller number of implementation details that are taken into account during higher levels when developing models and evaluating properties. Although such abstraction is indispensable for handling the system's complexity, it also causes system-level design to be very difficult since the designer must be convinced that the used abstractions are adequate.

1.1.3 Performance Modelling

Models are intended for investigating only specific aspects of a system. The construction of models for the evaluation of performance properties is called *performance modelling*. The actual assessment of performance properties is known as performance evaluation or performance analysis. Although performance analysis is sometimes considered to be a part of verifying that a system has certain desirable properties, the term *verification* will be primarily used in the context of evaluating correctness properties. This thesis focuses on performance modelling for system-level design and the accompanying evaluation of performance properties. An illustrative performance modelling example is given below.

Example 1.1 *Consider the example of figure 1.3, where N Internet-traffic sources send packets independently from each other to a shared buffer in a telecommunication system. The total bandwidth B for receiving packets into the buffer is fixed by the environment in which the telecommunication system will operate. The bandwidth for removing packets from the buffer, which is a factor M larger than B, is a parameter of the system and its value depends on the used implementation technology. The size S of the buffer is also a parameter and the design task is to find a feasible combination of M and S such that the probability of loosing a packet (which occurs in case the buffer is full) is smaller than a certain bound p without implying a too costly implementation (which might for example be the case if M is much larger than 1).*


Independent bursts of variable-size packets
Bandwidth for individual connections
Buffer
Source 1
Source 2
Source N
w1·B
w2·B
wN·B
M·B
S

N = Number of Sources
B = Input Bandwidth
S = Buffer Size
M = Output Bandwidth Factor
w_i = Weight Factor for Source i due to Arbitration Mechanism

Figure 1.3: Buffering packets from independent Internet-traffic sources.

Experienced designers will recognise that implementing the shared buffer requires using some arbitration mechanism (a weighted round robin scheduler for example). Such an arbitration *mechanism determines the order in which the sources access the buffer for storing a certain amount of data. The implementation of the arbitration mechanism accomplishes this by assigning a number of access time-slots to each source, which corresponds to the weight factor for that source. The effect is that bandwidth B is subdivided into bandwidths for the individual connections. During system-level design, the exact details of how and in what order access time-slots are assigned is of less importance for analysing whether the loss probability is smaller than p. What is essential are the weights for subdividing B. Decisions on the exact implementation of the arbitration mechanism should be postponed until designing the buffer in more detail. To evaluate the loss probability for alternative combinations of M and S, a performance model can be developed that takes the weight factors for subdividing B into account without the necessity of specifying how this is accomplished (example 4.4 on page 126 illustrates how such a model may look like). Possibly, the evaluation results reveal that different combinations of M and S will satisfy the performance requirements. After deciding which combination will be used as a starting point for realising the system, the exact details of how the arbitration mechanism subdivides B can be defined in a next design phase.*

1.2 On Formalisms, Techniques, Methods and Tools

This thesis contributes to the development of a *system-level design method* that provides a framework for constructing performance models during the earliest phases of the design. The following areas of *design methodology*, which is the science that studies the way in which design problems can be solved, are involved:

- The research area of *formalisms* examines notations for writing down models, the desired properties of a system and their semantics. Formalisms used for constructing models are called *modelling languages*. A formalism for capturing the properties of a system will be called a *property specification language*.
- Research on *techniques* concerns the investigation of mathematically founded ways to transform or analyse models. Examples include techniques for model refinement, simulation of models, verification and performance evaluation.
- Structured frameworks for applying formalisms and techniques are studied in the research area of *design methods*. Design methods assist the designer in developing feasible designs by providing a number of design steps that have to be completed in a certain order. Each design step is supported by *guidelines*, which capture experience with applying certain formalisms and techniques. Determining which formalisms and techniques are suitable for a design process therefore forms an essential starting point for developing a design method.
- Research on *(design) tools* investigates how user-friendly computer support can be provided to enable effective and efficient application of formalisms, techniques and design methods.

The suitability of formalisms, techniques, methods and tools is subject to the characteristics of the application domain and to the design level. This section identifies some general prerequisites for these aspects in the context of performance modelling for system-level design. In addition, the state of the art is briefly discussed.

1.2.1 Modelling Languages

Models developed during system-level design should be:

- **Abstract** Developing models involves *abstraction* from characteristics that are not relevant for analysing the properties of interest and/or from implementation details that are still unknown at the time of contriving the main concepts for the system. Important advantages of abstraction are the prospect of gaining insight in the working of a system more quickly and faster simulation compared to executing a description including all implementation details. However, the main advantage is that it allows postponing design decisions related to details of the final implementation [172] (see also example 1.1).
- **Adequate** A model is said to be *adequate* in case all essential details necessary for properly evaluating the properties of interest are captured by the model.
- **Intuitive** To accelerate the validation of a model, it should be intuitive. *Intuitiveness* is important in the context of minimising the time needed for explaining how a system operates. This means that any designer contributing to the construction of a model should be able to understand quickly the intended interpretation of the model and how the proposed concepts for realising the desired functionality are expected to satisfy the requirements.
- **Executable** Models can be categorised in non-executable and executable. *Executable* models enable a structured and automatic application of techniques for analysing the properties of interest by means of (simulation) tools.

The requirements for models constructed during system-level design have important implications for modelling languages. They should be:

- **Expressive** The *expressive power* of a modelling language determines how intuitively, succinctly and readable the provided primitives enable representing the characteristics of hardware/software systems. Intuitiveness, succinctness and readability are indispensable for accelerating the validation of models and for checking the adequacy of abstractions. Important aspects that determine the expressive power of a modelling language include the following:
  - To capture the characteristics of hardware/software systems, a modelling language should provide primitives for expressing architectural *structure*, *concurrency* of behaviour, *time*, *data*, and *communication* [58, 172], without the necessity to specify all implementation details. For modelling the uncertainty in the dynamism of the behaviour of a system, modelling languages should also allow expressing *non-determinism* [77] and *stochasticity*. Stochasticity, as opposed to non-determinism, requires the designer to have knowledge about the probability that certain behaviour occurs.
  - Another aspect affecting the expressive power is compositionality. The primitives of a modelling language are called *compositional* if they can be combined in an orthogonal fashion such that they do not influence each others semantics [134, 79]. Supporting compositionality results in a small set of powerful primitives, which greatly improves expressive power.

- **Formal** In this thesis, a modelling language is called *formal* in case the semantics of its primitives is defined mathematically [145, 7, 134]. A *formal semantics* advances the compositionality of primitives because the semantics of combinations of them can be mathematically derived from the semantics of the individual primitives. A formal semantics has two essential advantages:

  – A formal semantics enables rigorous reasoning about the properties of a model [57]. If no formal semantics is available, only verbal reasoning about the working of a system is possible. Verbal reasoning is insufficient for analysing the properties of interest due to the difficulty of taking the effects of all relevant aspects into account. A more credible evaluation is achievable in case a formal modelling language is used because its semantics allows mathematically relating models to analysis techniques.

  – A formal modelling language allows for unambiguous execution of models based on a mathematical framework that is derived from the semantics [28, 48, 30]. In case the semantics of a modelling language is defined informally, ambiguities may arise about the precise meaning of certain combinations of primitives [134] and hence, it is difficult to guarantee that no semantical freedom exists when executing models. If such freedom exists, compilers or execution engines may interpret certain combinations of primitives differently [163]. Hence, this semantical freedom may result in obtaining different analysis results. Instead of bothering the designer with the way possible ambiguities are resolved, a formal semantics defines the meaning of any possible combination of primitives unambiguously [145].

State of the Art In recent years, much research has been performed on modelling languages for hardware/software systems. It revealed the difficulty of defining an expressive language with a formally defined semantics. Different choices for expressing the characteristics of hardware/software systems like the used paradigm for modelling concurrency of behaviour (see for example [145, 22, 172]) have resulted in a large diversity of modelling languages. Some well-known examples are UML[1] [149] and SystemC 2.0 [68]. Although providing support for expressing many characteristics of hardware/software systems, such de facto applied modelling languages often lack a mathematically defined semantics. Academic modelling languages like LOTOS [29] do have a formal semantics but are less often used for designing industrial hardware/software systems due to a lack of expressive power. The modelling languages SDL [88] and POOSL [145, 28] are examples that combine excellent expressive power with a formal semantics.

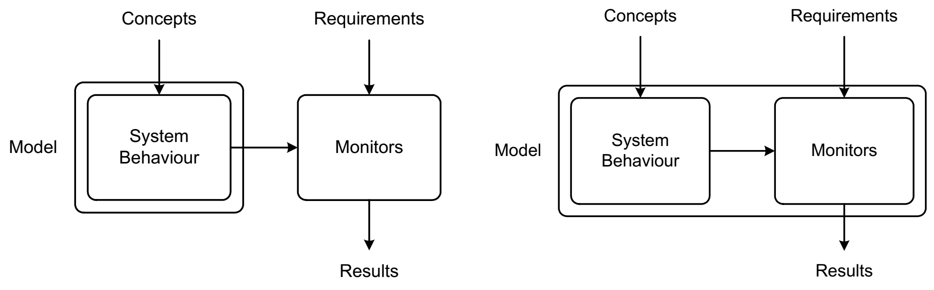
1.2.2 Property Specification Languages

To enable evaluating the feasibility of a design, it is necessary to specify the properties that are to be satisfied by the resulting system. Property specification languages for denoting correctness and performance properties should be:

[1]It is remarked that UML models are in principle not executable. Nevertheless, UML tools often allow supplementing UML models with executable code using a host language such as C++.

- **Formal** A property specification language may have a syntax that is similar to the syntax of a natural language or programming language. The evaluation of correctness and performance properties is commonly based on mathematical analysis techniques, which require the properties to be expressed in terms of mathematical formulae. A property specification language is called *formal* in case its semantics is mathematically defined. Only a formal property specification language allows to rigorously match property specifications with the mathematical formulae based on which the analysis techniques can be applied.
- **Expressive** The expressive power of a property specification language depends on how intuitively, succinctly and readable the provided primitives can denote a wide range of properties.

Specifying desired properties leads to defining monitors. A *monitor* observes the behaviour of a system and gathers information about a certain property of the system. Figure 1.4 illustrates the two approaches that can be identified for specifying monitors:



Model-Checking Approach Reflexive Approach

Figure 1.4: Approaches for specifying monitors.

- **Model-Checking Approach** With this approach, the model only represents the behaviour of the system. Monitors are specified *separately* from the model, see figure 1.4. Together, the model and monitor define a mathematical structure that allows analysing the properties of interest. The model-checking approach is often used in the context of formal verification of correctness properties.
- **Reflexive Approach** If the modelling language is sufficiently expressive, a model can be explicitly extended with additional behaviour to define monitors *within* the model. The extended model enables analysing the properties of interest in a reflexive way, i.e., by reasoning about its own behaviour. An important disadvantage is that the original model is polluted with behaviour which is only needed for evaluating the properties of interest but which has nothing to do with the inherent behaviour of the system. Hence, it becomes more difficult to validate the model. The reflexive approach is often used for evaluating performance and correctness properties by means of simulating models.

Although the model-checking approach is favorable, only a few formalisms exist that allow separately specifying monitors for evaluating performance properties. Unfortunately, these formalisms lack the power to express many common performance properties. Consider for example the average latency of transferring packets through the system of example 1.1. Formalisms like temporal rewards [180] cannot associate the time at which a packet left the system with the time at which that packet entered the system. Due to this lack of expressive power, it is common to use the reflexive approach for specifying monitors, taking the pollution of the model for granted.

State of the Art Research on formalisms for specifying performance and correctness properties has followed different paths. Much research has been performed on logics for denoting correctness properties. Examples of these mathematical notations, which are usually applied in accordance with the model-checking approach, are LTL [112] and several real-time temporal logics [74]. The research on expressive formalisms for specifying performance properties is initiated more recently. An example is the already mentioned formalism of temporal rewards [180], which allows computing performance properties analytically in accordance with the model-checking approach. On the other hand, modelling languages that support a reflexive way of performance analysis exist in abundance. Nevertheless, they usually lack a formal semantics. An example is the modelling language UML [149] for which the UML profile for performance specification [130] can be used to specify monitors.

1.2.3 Performance Analysis Techniques

Properly analysing the performance of hardware/software systems requires the application of mathematical analysis techniques. Such performance analysis techniques can be categorised as follows:

- **Analytical Computation** Techniques for computing the performance of a system analytically are exhaustive in the sense that all possible behaviours of the system are taken into account. The advantage is that *exact* performance results are obtained. However, such performance results can only be credible if:

  - the model is adequate;
  - the model is amenable to mathematical analysis techniques.

  A practical problem of this approach is that existing techniques for computing the performance analytically are often not suitable for industrial hardware/software systems. This is due to the exponential growth in possible behaviours when complexity increases (the state-space explosion problem [126]).

- **Simulation-Based Estimation** Simulation (or execution) of a model allows to investigate a limited number of all possible behaviours of a system. As a consequence, the obtained performance results are *estimates* of the true performance of the system. Such performance results can only be credible if:

  - the model is adequate;
  - the model is amenable to mathematical analysis techniques;
  - the model is executed unambiguously;

- the *accuracy* of the performance estimation results [135, 136, 171] is provided, which indicates how close the results are to the true performance;
- the used simulation tool relies on good random number generators[2] [136] for emulating the dynamism expressed as probabilistic behaviour.

State of the Art Many techniques exist for both analytical computation and simulation-based estimation of performance properties. Commonly, they require the behaviour of the system to be modelled using mathematical structures like queueing networks [5, 102], Petri nets [113, 114] or Markov processes [78, 175]. To enable computing performance properties analytically, such mathematical structures usually support only exponential distributions for expressing probabilistic behaviour. However, the imperative of using exponential distributions sometimes yields inadequate performance models. In addition, techniques for computing performance properties often do not scale with the complexity of industrial hardware/software systems. Due to these limitations, simulation-based estimation of performance properties is used more often [160]. Performance estimation is based on statistical analysis of simulation results [105, 3]. Unfortunately, estimation techniques are commonly applied without rigorously matching the model to the mathematical structure required for using these techniques. Moreover, the accuracy of simulation results is rarely indicated [136] and often bad random number generators are used [136, 47].

1.2.4 System-Level Design Methods

System-level design methods structure the design of hardware/software systems in a number of design steps. These design steps include the aspects of capturing the concepts and requirements, constructing and validating models, actually evaluating the properties of interest and making design decisions. Each design step is accompanied with guidelines that assist the designer in achieving the ultimate goal of developing a feasible design. Most of such guidelines strongly depend on the provided modelling languages, property specification languages and analysis techniques. An example of a more conceptual choice for design methods concerns principally separating the application from the architecture when developing models, like in [98, 99]. Another conceptual choice is supporting either or both analytical computation and estimation of either or both functional and non-functional properties.

To assist the designer in constructing models, system-level design methods provide guidelines for applying certain modelling language(s). Such guidelines include for example modelling patterns. *Modelling patterns* are templates for modelling typical aspects of systems with a specific modelling language [177]. Other guidelines assist in validating the constructed models and in defining monitors for analysing the properties of interest with the provided analysis techniques. The actual application of those techniques should not require the designer to be an expert on the underlying mathematical details. In case the analysis results are obtained by means of simulation, the designer should also not be bothered with any detail of how the compiler or execution engine operates. Regarding the final design decision making, guidelines should be provided that assist in properly interpreting the analysis results.

[2]Detailed research on random number generation to facilitate simulation-based estimation is beyond the scope of this thesis. Appendix B briefly discusses the approach used for executing POOSL models.

State of the Art Many different system-level design methods have been developed [186, 115, 67]. In a purely industrial context, design methods often merely concern a brief documentation of the design process for a specific hardware/software system. Design methods developed in an academic context do tend to bundle discovered guidelines in separate documents that can be consulted when designing other systems. Some examples of system-level design methods that have such reference documents are the Y-chart approach [99], the POLIS framework [12], the design space exploration framework in [174], the SpecC-based method in [53], the Artemis framework [137], the SystemC 2.0 method in [68] and the SHE [145, 58] method. Opposed to the numerous guidelines for applying certain modelling languages, guidelines for the validation of models, the specification of requirements, the application of analysis techniques and the interpretation of analysis results are usually much scarcer.

1.2.5 Design Tools

(Partly) automating the application of modelling languages, property specification languages and analysis techniques in accordance with the guidelines provided by a design method largely accelerates the design process. The following prerequisites can be identified for tools that provide computer support for design methods:

- Of essential importance is that tools comply with the formalisms, techniques and guidelines provided by the design method. For example, to obtain credible analysis results, tools should act according to the mathematical framework that underlies models and analysis techniques. This also implies that if a model is considered to be inadequate, then it should not be possible to influence the behaviour of that model by adapting the execution engine. Instead, the model should be changed. To support this, tools should clearly separate the code that represents the system behaviour from the execution engine [160].
- Tools for constructing models should provide sufficient means for validating such models. For example, informative feedback that allows validating whether a model appropriately represents the proposed ideas for realising the desired functionality, may be provided automatically.
- In the case of using the reflexive approach for defining monitors, tools should support isolating the code that models the system behaviour from the code denoting monitor behaviour (see also section 1.2.2).
- Tools allowing automated analysis of performance properties should execute models efficiently. The efficiency can for example be expressed in how well the execution scales for different values of the system parameters (in example 1.1, execution speed depends for example on the number of sources).
- Finally, to ease the use of tools, they should have intuitive user-interfaces.

State of the Art The need for suitable system-level design tools has been addressed before [44]. Recently developed commercial tools like the Simulink tool suite [116], the System Designer and System Verifier tools [42], the Virtual Component Codesign (VCC) tool [37], the System Studio tool suite [164] and RoseRT [146] offer very sophisticated user-interfaces. However, such tools are usually based on modelling languages that do not have a formal semantics and hence, they do not execute models according to a mathematical framework. Some of such tools, which for example rely on programming languages such as C++, even do not prohibit to adapt the execution engine when resolving inadequacies in a model. Examples of academic tools are SPADE [110], Metropolis [13], EXPO [174] and Rotalumis [28]. Academic tools are sometimes based on modelling languages with a formal semantics but they are rarely applied for designing industrial systems due to a lack of customer support and steep learning curves. Most modern design tools provide one or more ways to validate models, ranging from simple debugging facilities to graphical visualisation of an executing model. Defining monitors is usually based on the reflexive approach but code for monitors is rarely strictly separated from the code representing system behaviour. In efforts to reduce design time, simulation efficiency is a key selling point of many commercial tools. Some academic tools, like Rotalumis, have similar execution performance and scalability characteristics as commercial tools.

1.3 Objectives and Contributions

Evaluating performance properties without a proper mathematical foundation is insufficient for obtaining credible performance results. Many expressive modelling languages lack a formal semantics on which a mathematical framework for applying performance analysis techniques and for executing models can be based. On the other hand, existing performance analysis techniques do not scale with the complexity of today's hardware/software systems. This thesis contributes to decreasing the gap between existing performance analysis techniques and models constructed with expressive modelling languages. Hence, the first objective is to provide a mathematical framework for obtaining credible performance results based on models developed with an expressive modelling language. Relying on this framework, the second objective is to provide a system-level design method that assists the designer in constructing adequate performance models with that modelling language.

Figure 1.5 shows an overview of the mathematical framework proposed in this thesis to achieve the first objective. A suitable mathematical structure for evaluating performance properties are Markov chains [175]. They capture all possible behaviours of a system in a probabilistic way by means of a probability space [39]. Based on the relation between Markov chains and probability spaces, classical performance analysis techniques for both analytical computation and simulation-based estimation of performance properties have been defined. Hence, extending these classical techniques in order to make them more suitable for handling complex systems requires to take the relation between Markov chains and probability spaces into account.

Relying on the relation between Markov chains and probability spaces, this thesis extends the classical performance analysis techniques in two ways. First, it is recognised that many common performance properties require to take the occurrence of a certain event into account. A simple example of such a conditional performance metric is the average duration that a packet is in the buffer of example 1.1. Only when a packet is removed from the buffer, the value of the variable denoting the storage time of the most recently removed packet must be taken into account. The

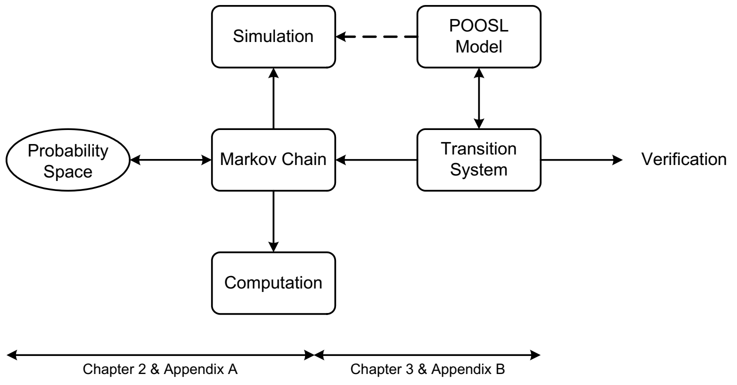


Figure 1.5: Relation between models and analysis techniques proposed in this thesis.

classical techniques cannot evaluate this metric efficiently by requiring to consider the value of the variable in all states of the system. Chapter 2 introduces a technique that allows applying the classical techniques to evaluate the average storage time without the necessity to take all the states into account. This technique is applicable in both the case of analytical computation and simulation-based estimation of the average storage time. A second observation is that common performance properties can be expressed as an algebraic combination of simple performance metrics. An example is the average occupancy of the buffer in example 1.1, which requires to take the duration of each occurring occupancy into account. Although analytical computation of the average occupancy can be accomplished with the classical techniques, no general technique exists for analysing the accuracy of estimation results for such complex performance metrics obtained by simulation. Chapter 2 introduces a generally applicable technique that allows applying the classical techniques for analysing the accuracy of estimation results for complex performance metrics. The proposed techniques improve the applicability of the classical techniques for complex systems.

Directly specifying the behaviour of a hardware/software system as a Markov chain requires to identify the states of the system and possible transitions between the states explicitly. Considering that the state is amongst others determined by the value of all variables, this approach is rather impractical for complex systems. Instead, it is more convenient to use a modelling language, which allows to specify the complex behaviour of a system by combining several components that expose some simple behaviour. In this thesis, the expressive Parallel Object-Oriented Specification Language (POOSL) [145, 28] is presumed to be used for constructing models. This formal modelling language has proven to be very suitable for modelling industrial hardware/software systems (see for example [111, 171, 172]) by its capability to express their characteristics in an intuitive and succinct way. A detailed comparison of POOSL with other modelling languages can for example be found in [172] and [58].

Based on the formal semantics, a POOSL model defines a transition system (see figure 1.5). This mathematical structure forms the basis for unambiguously executing a POOSL model [57, 28], for formal verification of correctness properties as presented in [57] and for performance analysis. In [180], it is shown how the transition system defined by a POOSL model can be transformed into a Markov chain in order to compute performance properties in accordance with the model-checking approach. Unfortunately, actually performing this transformation is often infeasible for industrial hardware/software systems due to their complexity. Relying on the relation between POOSL models and Markov chains, chapter 3 introduces a framework for the estimation of performance properties by means of simulating a POOSL model (indicated with the dotted arrow in figure 1.5). This framework involves extending a POOSL model with monitors in accordance with the reflexive approach. The use of the simulation-based estimation techniques proposed in chapter 2 for four common types of performance metrics is implemented in POOSL library classes. These performance monitor templates enable user-friendly application of the performance analysis techniques without the necessity to know all their mathematical details.

Performance modelling with an expressive modelling language like POOSL can be a complicated task. This is because it is often not immediately clear which combination of primitives can be used to specify the desired behaviour. For achieving the second objective of this thesis, the general approach for structuring the evaluation of design alternatives during system-level design (shown in figure 1.2) is refined. To assist designers in constructing adequate performance models with POOSL, the proposed refinement should fit the Software/Hardware Engineering (SHE) [145] method for which POOSL was originally developed. This system-level design method offers several diagram types that facilitate discussions on proposed concepts and that facilitate the use of POOSL for developing executable models. Since the Unified Modelling Language (UML) [149] is a de facto standard for the first purpose, chapter 4 defines a profile for stereotyping UML diagrams, which enables to replace the original SHE diagram types. In addition, generally applicable modelling patterns for constructing adequate performance models are presented as well as guidelines for validating them against the stereotyped UML diagrams. Furthermore, guidelines are supplied for extending a model with monitors (by using the performance monitor templates for example), which ensure preserving the intuitiveness of the model.

The proposed extensions to SHE assist in how POOSL can be used for constructing performance models of design alternatives during system-level design. Chapter 5 assesses the applicability of the proposed performance analysis techniques and system-level design method by means of representative industrial case studies.

1.4 Thesis Overview

This thesis consists of six chapters, which consecutively discuss the following topics:

1. **Introduction** This chapter introduces the problem domain of performance modelling for system-level design and identifies the prerequisites for formalisms, techniques, methods and tools that ensure obtaining credible performance results. In addition, an overview of the objectives of this thesis is given.

2. **Mathematical Techniques for Performance Evaluation** Chapter 2 gives an introduction on the relation between Markov chains and probability spaces and discusses classical performance analysis techniques for both analytical computation and simulation-based estimation of long-run average performance

metrics. To improve applicability of these classical techniques, a technique for evaluating the more common conditional form of such performance metrics is introduced. It aims at reducing the number of states that has to be taken into account. Moreover, a technique is introduced for analysing the accuracy of simulation results obtained for combinations of long-run averages. Four different types of such complex performance metrics are identified. With these, a large range of frequently required performance properties can be expressed.

3. **Reflexive Performance Analysis with POOSL** Chapter 3 discusses how models constructed with the modelling language POOSL relate to Markov chains. Based on this relation, a framework is defined for extending a POOSL model with monitors. The framework forms the basis of evaluating performance properties by means of simulating the extended POOSL model. Performance monitor templates in the form of POOSL library classes are introduced, which allow accuracy analysis of simulation results obtained for the four types of common complex performance metrics identified in chapter 2. Finally, the quality of the accuracy obtained using these POOSL library classes is assessed.

4. **Extending the SHE Method for Performance Modelling** Chapter 4 extends the system-level design method SHE with guidelines for performance modelling. Starting with developing UML models, guidelines for constructing and validating POOSL models are given. To facilitate the construction of POOSL models, a UML profile and several generally applicable modelling patterns are presented. Furthermore, guidelines for extending a POOSL model with monitors are given, which aim at preserving the intuitiveness of the model when applying for example the POOSL library classes of chapter 3.

5. **Case Studies** To assess the proposed performance analysis techniques and system-level design method, chapter 5 elaborates on two industrial case studies. The first has been performed in cooperation with Alcatel Bell Antwerp and concerned the performance modelling of a backbone Internet Router. The second case study is about the performance modelling of a network processor and has been performed in cooperation with IBM Research Laboratory Zürich.

6. **Conclusions** Chapter 6 summarises the conclusions of this thesis in relation to the objectives presented in chapter 1 and discusses directions for future work.

In addition, two appendices are included:

A. **Mathematical Preliminaries** To make this thesis self-contained, appendix A gives an overview of the mathematical concepts of set theory, probability theory and statistics.

B. **Parallel Object-Oriented Specification Language** To ease understanding of the numerous examples of performance models given in chapters 3, 4 and 5, appendix B gives an introductory overview of the POOSL modelling language. Next to presenting the syntax, both the semantical framework and the framework for executing POOSL models are discussed. Moreover, some aspects of the tools for creating, validating and executing POOSL models are presented.

Chapter 2

Mathematical Techniques for Performance Evaluation

A prerequisite for reasoning about the properties of a system using mathematical analysis techniques is formalising the system's behaviour with a mathematical structure. Such mathematical structures often regard the behaviour as a collection of states between which transitions can occur. Stochastic processes, which assume that transitions occur with certain probabilities, provide a suitable means for formalising the behaviour of a system for the purpose of performance evaluation. Appendix A rehearses several aspects of stochastic processes and underlying probability theory, which are presumed to be understood before reading this chapter.

The mathematical techniques for performance evaluation developed in this chapter presume the behaviour of a system to be formalised with a Markov chain, which is a particular type of stochastic process. Markov chains can be used for analysing the behaviour initially exposed after starting it up [3, 78] and its long-run behaviour. This thesis focuses on evaluating performance properties for the long-run behaviour of hardware/software systems, which are expressed as long-run average performance metrics. Simple examples of such performance metrics are the latency of transferring data through a system and the probability of loosing data due to an unreliable transmission medium. The next section summarises essential properties of Markov chains and discusses classical performance analysis techniques that allow both analytical computation as well as accuracy analysis for simulation-based estimation of simple long-run average performance metrics.

Many long-run average performance metrics for industrial hardware/software systems require to take a certain condition on the states into account. An example is the condition of receiving data when evaluating the probability of loosing such data due to a full buffer. Although conditional long-run average performance metrics can in principle be evaluated with the classical performance analysis techniques, directly applying them does not benefit from the commonly relatively small number of states for which the condition holds. Section 2.2 presents a technique for indirect application of the classical performance analysis techniques, which allows disregarding states for which the condition is invalid. It is furthermore shown that the proposed technique may reduce the computational complexity of such an evaluation.

Next to simple (conditional) long-run average performance metrics, more complex ones are often defined for industrial hardware/software systems. Examples are the variance in the latency of transferring data through a system, the average utilisation of a processor and the variance in the occupation of a buffer. Section 2.3 shows that many of these complex (conditional) long-run average performance metrics can be expressed as an algebraic combination of simple (conditional) long-run average performance metrics and can therefore be computed analytically using the classical performance analysis techniques. Accuracy analysis of simulation results for such performance metrics is however less straightforward. Section 2.3 presents a technique for analysing the accuracy of simulation results obtained for performance metrics that are composed of simple (conditional) long-run performance metrics.

2.1 Performance Evaluation with Markov Chains

Markov processes are named after the Russian A.A. Markov, who introduced the concept in 1907. A Markov process is a stochastic process satisfying the Markovian property, which states that the future behaviour of a system merely depends on the current state of the system. The case where the number of states is countable, which was launched by the Russian A.N. Kolmogorov in 1936, is also referred to as Markov chain. The techniques for analysing performance properties presented in this chapter are developed for discrete-time Markov chains, where the set of all time-epochs at which state transitions occur is countable. This section presents preliminaries on Markov-chain based performance evaluation inspired by [39, 24, 175] and [43].

2.1.1 Markov Chains

Consider a discrete-time stochastic process $\{X_i \mid i \geq 1\}$ on an appropriate probability space $(\Omega, \mathcal{F}, \mathbb{P})$, where each random variable[1] X_i is discrete and assumes values in a countable set of states $\mathcal{S}$. Set $\mathcal{S}$ is called the *state space* of $\{X_i \mid i \geq 1\}$ and for each $S \in \mathcal{S}$, $\mathbb{P}(X_i = S)$ indicates the probability that $\{X_i \mid i \geq 1\}$ is in S or visits S at time-epoch i. If for all $i \geq 1$ and $S_1, \ldots, S_{i+1} \in \mathcal{S}$

$$\mathbb{P}(X_{i+1} = S_{i+1} \mid X_k = S_k \text{ for all } 1 \leq k \leq i) = \mathbb{P}(X_{i+1} = S_{i+1} \mid X_i = S_i) \tag{2.1}$$

then $\{X_i \mid i \geq 1\}$ is said to satisfy the *Markovian property* and is called a *discrete-time Markov chain*. For any $S, T \in \mathcal{S}$, the probability $\mathbb{P}(X_{i+1} = T \mid X_i = S) = \mathcal{P}_{S,T}(i)$ is called the *(one-step) transition probability* from S to T and indicates the probability that discrete-time Markov chain $\{X_i \mid i \geq 1\}$ transfers from S to T at time-epoch i.

Of special interest are *time-homogeneous* discrete-time Markov chains, for which transition probability $\mathcal{P}_{S,T}(i)$ is independent of i for all $S, T \in \mathcal{S}$. If $\mathcal{P}_{S,T}(i) = \mathcal{P}_{S,T}(j)$ for any two time-epochs i, j and all $S, T \in \mathcal{S}$, then the discrete-time Markov chain is

[1]Strictly speaking, the X_i's are no random variables in the sense of appendix A because $\mathcal{S}$ is not a subset of $\bar{\mathbb{R}}$. This is however immaterial since a complete function can be defined, which uniquely assigns an element of $\bar{\mathbb{R}}$ to each element of $\mathcal{S}$.

said to have *stationary transition probabilities*. In that case, $\mathcal{P}_{S,T}$ is defined as

$$\mathcal{P}_{S,T} = \mathbb{P}(X_{i+1} = T \mid X_i = S)$$

for all time-epochs $i \geq 1$ and $S, T \in \mathcal{S}$. For any fixed enumeration of the states in $\mathcal{S}$, the matrix $\mathcal{P} = (\mathcal{P}_{S,T})$ with $S, T \in \mathcal{S}$ is called the *transition matrix* of $\{X_i \mid i \geq 1\}$. Denoting $\mathbb{P}(X_1 = S)$ with I_S for $S \in \mathcal{S}$, the n-dimensional joint probabilities of a time-homogeneous discrete-time Markov chain $\{X_i \mid i \geq 1\}$ can be written as

$$\mathbb{P}(X_i = S_i \text{ for all } 1 \leq i \leq n) = I_{S_1} \cdot \prod_{i=1}^{n-1} \mathcal{P}_{S_i,S_{i+1}}$$

The probability I_S is called the *initial probability* of S and indicates the probability that $\{X_i \mid i \geq 1\}$ departs from state S. The distribution I of initial probabilities over $\mathcal{S}$ is referred to as the *initial distribution* of $\{X_i \mid i \geq 1\}$. In this thesis, time-homogeneous discrete-time Markov chains are conveniently called *Markov chains*.

Definition 2.1 (Markov Chain) *A Markov chain is a discrete-time stochastic process $\{X_i \mid i \geq 1\}$, where each discrete random variable X_i assumes values in a countable state space $\mathcal{S}$ such that, for each time-epoch $i \geq 1$, the transition probabilities*

$$\mathbb{P}(X_{i+1} = S_{i+1} \mid X_k = S_k \textit{ for all } 1 \leq k \leq i) = \mathbb{P}(X_{i+1} = S_{i+1} \mid X_i = S_i)$$

are independent of i, for any $S_1, \ldots, S_{i+1} \in \mathcal{S}$.

An important theorem in Markov theory is the *existence theorem* [39, 24]. It states that for any countable set of states $\mathcal{S}$, sequence $\{I_S \mid S \in \mathcal{S}\}$ and matrix $(\mathcal{P}_{S,T})$ with $S, T \in \mathcal{S}$, satisfying respectively

$$\begin{aligned} I_S \geq 0 \quad &\text{and} \quad \sum_{S \in \mathcal{S}} I_S = 1 \\ \mathcal{P}_{S,T} \geq 0 \quad &\text{and} \quad \sum_{T \in \mathcal{S}} \mathcal{P}_{S,T} = 1 \end{aligned}$$

there exists a probability space $(\Omega, \mathcal{F}, \mathbb{P})$ and a Markov chain $\{X_i \mid i \geq 1\}$ defined on it with state space $\mathcal{S}$, initial distribution I and transition matrix $\mathcal{P} = (\mathcal{P}_{S,T})$. As a result, the Markov chain $\{X_i \mid i \geq 1\}$ is completely determined by the triple $(\mathcal{S}, I, \mathcal{P})$, with which it is therefore often conveniently represented. If the state space $\mathcal{S}$ is finite, the Markov chain can be visualised as a graph. Each state in $\mathcal{S}$ is represented as a node labelled with the corresponding name of the state. For every non-zero transition probability $\mathcal{P}_{S,T}$, a directed arrow is drawn from the node representing S to the node representing T, labelled with transition probability $\mathcal{P}_{S,T}$. For any state $S \in \mathcal{S}$ with non-zero initial probability, a symbol $>$ directed towards the node representing S is drawn, labelled with initial probability I_S. In case $I_S = 1$ for state S, the label 1 to the symbol $>$ is usually omitted. This is illustrated with an example.

Example 2.1 *Let $(\mathcal{S}, I, \mathcal{P})$ represent a Markov chain, where $\mathcal{S} = \{A, B, C\}$, I is defined as $I_A = 1$, $I_B = I_C = 0$, and where the transition matrix $\mathcal{P}$ is given by*

$$\mathcal{P} = \begin{pmatrix} \mathcal{P}_{A,A} & \mathcal{P}_{A,B} & \mathcal{P}_{A,C} \\ \mathcal{P}_{B,A} & \mathcal{P}_{B,B} & \mathcal{P}_{B,C} \\ \mathcal{P}_{C,A} & \mathcal{P}_{C,B} & \mathcal{P}_{C,C} \end{pmatrix} = \begin{pmatrix} 0 & \frac{1}{3} & \frac{2}{3} \\ 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 1 & 0 \end{pmatrix}$$

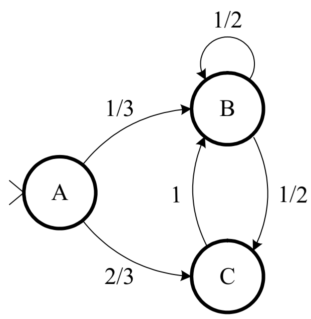


Figure 2.1: Visualisation of a Markov chain as a graph.

A graphical representation of this Markov chain is depicted in figure 2.1.

Probability Space The remainder of this section elaborates on the probability space $(\Omega, \mathcal{F}, \mathbb{P})$ on which a Markov chain $\{X_i \mid i \geq 1\}$ represented by $(\mathcal{S}, I, \mathcal{P})$ is defined.

The sample space Ω is the set of all infinite sequences $\underline{S} = (S_1, S_2, \ldots)$, where $S_i \in \mathcal{S}$ for all $i \geq 1$. Hence, Ω is the set $\mathcal{S}^\infty$ of all infinite *state sequences*. A *cylinder* of rank n is a subset C_n of Ω of the form $C_n = \{\underline{S} \in \Omega \mid \underline{S}_{1..n} \in A\}$ where $A \subseteq \mathcal{S}^n$. In case A is a singleton set $\{\underline{S}\}$, the set $\{\underline{T} \in \Omega \mid \underline{T}_{1..n} = \underline{S}\}$, which is denoted by $\underline{S}^{\rightsquigarrow}$, is called a *thin* cylinder of rank n. The σ-algebra $\mathcal{F}$ is the σ-algebra generated by the set of cylinders and the probability measure $\mathbb{P}$ is defined as

$$\mathbb{P}(\underline{S}^{\rightsquigarrow}_{1..n}) = I_{S_1} \cdot \prod_{i=1}^{n-1} \mathcal{P}_{S_i, S_{i+1}}$$

for each thin cylinder $\underline{S}^{\rightsquigarrow}_{1..n} \in \mathcal{F}$ with $\underline{S} = (S_1, S_2, \ldots)$ in Ω. Since any cylinder C_n can be written as a countable union of disjoint thin cylinders of the same rank,

$$\mathbb{P}(C_n) = \sum_{\underline{S}^{\rightsquigarrow}_{1..n} \subseteq C_n} \mathbb{P}(\underline{S}^{\rightsquigarrow}_{1..n})$$

by the property of countable additivity. Furthermore, it can be shown that any subset C of Ω can be written as a countable intersection of cylinders in $\mathcal{F}$. For example,

$$\{\underline{S}\} = \bigcap_{n=1}^{\infty} \underline{S}^{\rightsquigarrow}_{1..n}$$

for any $\underline{S} \in \Omega$. Since $\mathcal{F}$ is closed under countable intersection, it follows that $C \in \mathcal{F}$. As a result $\mathcal{F} = \mathbf{2}^\Omega$, which implies that any discrete random variable defined on $(\Omega, \mathcal{F}, \mathbb{P})$ is measurable. For brevity, the probability $\mathbb{P}(\{\underline{S}\})$ on a singleton set $\{\underline{S}\} \in \mathcal{F}$ is also denoted by $\mathbb{P}(\underline{S})$.

Next to the probability on infinite state sequences, the probability on finite state sequences is frequently used in this chapter. Although probability measure $\mathbb{P}$ is only defined for (sets of) infinite state sequences, the notation $\mathbb{P}(\underline{S})$ is also used for any finite state sequence $\underline{S} \in \mathcal{S}^n$, which is justified by defining that $\mathbb{P}(\underline{S}) = \mathbb{P}(\underline{S}^{\rightsquigarrow})$, and is referred to as the probability on finite state sequence $\underline{S}$.

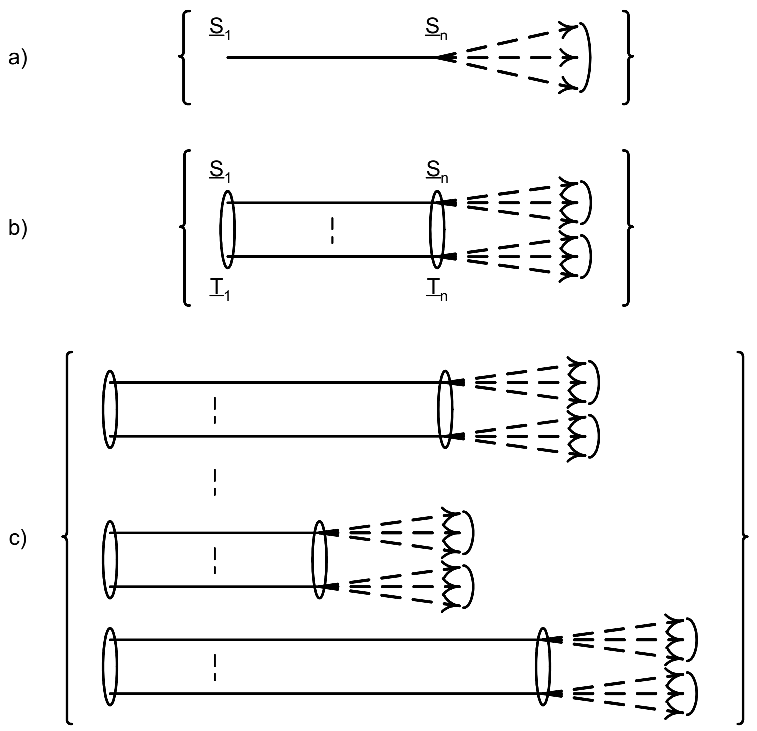


Figure 2.2: Visualisation of different cylinder types. a) Thin cylinder $\underline{S}_{1..n}^{\leadsto}$ of rank n. b) Cylinder of rank n. c) Generalised cylinder.

Now, a generalisation of the probability on cylinders is introduced. Let U be a set of finite state sequences (possibly of different lengths). Set U will be called *proper* if there exists no state sequence in U that is a prefix of any other state sequence in U. In case U is proper, the probability $\mathbb{P}(U)$ on U is defined as $\mathbb{P}(U) = \mathbb{P}(U^{\leadsto})$, where

$$U^{\leadsto} = \bigcup_{\underline{S} \in U} \underline{S}^{\leadsto}$$

Such a set $U^{\leadsto}$ will be called a *generalised* cylinder. Figure 2.2 illustrates the differences between thin cylinders, cylinders and generalised cylinders.

A proper set of finite state sequences U will be called an *initial set* if all the state sequences in U have the same initial state. For initial set U, $\mathbb{P}^*(U)$ is defined as

$$\mathbb{P}^*(U) = \frac{1}{I_{\underline{S}_1}} \sum_{\underline{S} \in U} \mathbb{P}(\underline{S})$$

and will be referred to as the *conditional probability* on U. Intuitively, $\mathbb{P}^*(U)$ denotes the sum of the probabilities on the state sequences $\underline{S}$ in U conditional on departing from state $\underline{S}_1$. A proper set of finite state sequences U is called a *final set* if all state sequences in U have the same final state.

Consider a final set of finite state sequences U and an initial set of finite state sequences V such that the initial state of the state sequences in V is equal to the final state of the state sequences in U. Then the *concatenation* of U and V, denoted by $U \circ V$, is defined as

$$U \circ V = \{(S_1, \ldots, S_{n-1}, S_n = T_1, T_2, \ldots, T_m) \mid (S_1, \ldots, S_n) \in U, (T_1, \ldots, T_m) \in V\}$$

Because both U and V are proper, the concatenation $U \circ V$ is also proper. Hence, $\mathbb{P}(U \circ V) = \mathbb{P}(U) \cdot \mathbb{P}^*(V)$. If U is also an initial set, then $\mathbb{P}^*(U \circ V) = \mathbb{P}^*(U) \cdot \mathbb{P}^*(V)$.

It can now be observed that the initial probability $I_S = \mathbb{P}(X_1 = S)$ indicating that the Markov chain $\{X_i \mid i \geq 1\}$ departs from state S represents in fact the probability $\mathbb{P}((S))$ on the finite state sequence (S). On the other hand, the probability $\mathcal{P}_{S,T} = \mathbb{P}(X_{i+1} = T \mid X_i = S)$ indicating that the Markov chain $\{X_i \mid i \geq 1\}$ transfers from state S to state T (at any time-epoch $i \geq 1$) represents in fact the probability $\mathbb{P}^*((S,T))$ on the finite state sequence (S,T) by the property of time-homogeneity.

2.1.2 State Classification

For any two states S and T of a Markov chain defined on probability space $(\Omega, \mathcal{F}, \mathbb{P})$, let $\mathcal{Z}_S^T$ represent the initial and final set of finite state sequences of lengths greater than 0 with initial state S and final state T and without any intermediate visits to T. The probability that the Markov chain ever makes a transition to T conditional on departing from S is equal to $\mathbb{P}^*(\mathcal{Z}_S^T)$. In case $\mathbb{P}^*(\mathcal{Z}_S^T) > 0$, then state T is said to be *reachable* from state S with probability $\mathbb{P}^*(\mathcal{Z}_S^T)$.

Definition 2.2 (Reachable State) *A state T is said to be reachable from state S with probability $\mathbb{P}^*(\mathcal{Z}_S^T)$ in case $\mathbb{P}^*(\mathcal{Z}_S^T) > 0$.*

An important aspect of Markov chains is that states may be repeatedly visited at different time-epochs. If a state S_r is visited more than once with probability 1 (i.e., $\mathbb{P}^*(\mathcal{Z}_{S_r}^{S_r}) = 1$) state S_r is called *recurrent*. Any non-recurrent state is called *transient*.

Definition 2.3 (Recurrent State) *A state S_r is called recurrent if S_r is reachable from S_r with probability 1.*

The sequentially visited states or *behaviour* of a Markov chain with recurrent state S_r is said to be *regenerated* after revisiting S_r. The time span between two visits of S_r is called a *(regenerative) cycle*. The behaviour exposed during a cycle through S_r is independent from the behaviour exposed in any previous cycle through S_r [39].

Of special interest are recurrent states for which the expected time until revisiting them is finite. Consider for any recurrent state S_r, the initial and final set $\mathcal{Z}_{S_r}^{S_r}$. The expected time until revisiting S_r or *expected recurrence time* of S_r is equal to the expected length of the state sequences in $\mathcal{Z}_{S_r}^{S_r}$, which is given by [2]

$$\sum_{\underline{S} \in \mathcal{Z}_{S_r}^{S_r}} |\underline{S}| \cdot \mathbb{P}^*(\underline{S}) \tag{2.2}$$

[2]Notice that equation 2.2 does not really express the expected value of a random variable. Let $X : \Omega \to \mathbb{R}$ be the discrete random variable that assigns to each $\underline{S} \in \Omega$ the absolute difference between the time-epochs of the first two visits to recurrent state S_r if the first visit of S_r occurs at time-epoch 1 and the second visit to S_r exists, and $X(\underline{S}) = 0$ otherwise. Then, it can be shown that equation 2.2 represents in fact the conditional expectation [5] of $X(\underline{S})$ given that $\underline{S}_1 = S_r$.

If the result of (2.2) is finite, S_r is called a positive recurrent state, which will be abbreviated to *positive state*. A Markov chain with a positive state that is reachable from any other state with probability 1 is called *ergodic* [5].

Definition 2.4 (Positive State) *A recurrent state S_r is called a positive state if the expected recurrence time of S_r is finite.*

Definition 2.5 (Ergodic Markov Chain) *A Markov chain is said to be ergodic if it has a positive state that is reachable from any other state with probability* 1.

Ergodic Markov chains have several interesting properties. The state space of an ergodic Markov chain can be partitioned in a set of positive states and a set of transient states. By the definition of ergodic Markov chains, each positive state is reachable from any transient state. As a result, ergodic Markov chains visit transient states only finitely many times with probability 1, whereas all positive states are visited infinitely many times with probability 1.

Another important property of ergodic Markov chains is that they have a unique *equilibrium distribution*. For an ergodic Markov chain with state space $\mathcal{S}$ and transition matrix $\mathcal{P}$, the *equilibrium probability* π_T of a state $T \in \mathcal{S}$ is defined as

$$\pi_T = \sum_{S \in \mathcal{S}} \pi_S \cdot \mathcal{P}_{S,T} \quad \text{such that} \quad \sum_{S \in \mathcal{S}} \pi_S = 1$$

For any $S \in \mathcal{S}$, the equilibrium probability π_S represents the long-run fraction of time-epochs that the Markov chain is in S. As a result, if S is a positive state, the equilibrium probability π_S equals the reciprocal of the expected recurrence time of S, whereas if S is a transient state then $\pi_S = 0$ [39, 175]. Finally, the expected number of visits to a state $S \in \mathcal{S}$ between two visits to a positive state $S_r \in \mathcal{S}$ equals π_S multiplied by the expected recurrence time of S_r, which is equal to $\frac{\pi_S}{\pi_{S_r}}$ [175].

2.1.3 Classical Performance Analysis Techniques

Markov chains enable analysing performance properties based on information that is contained in the states. Such information is assigned by *reward functions*. The idea is that each time a state is visited, the *reward value* as specified by the reward function is earned. Performance properties can be expressed as a certain combination of such reward values, which are referred to as *performance metrics*, and can be evaluated by analysing the behaviour of the Markov chain.

Definition 2.6 (Reward Function) *A function $r : \mathcal{S} \to \mathbb{R}$ defined for a Markov chain with state space $\mathcal{S}$ is called a reward function.*

This thesis focuses on evaluating performance metrics that can be expressed as a certain *long-run average* of reward values. Consider a Markov chain $\{X_i \mid i \geq 1\}$ on probability space $(\Omega, \mathcal{F}, \mathbb{P})$ for which reward function r is defined. For each $n \geq 1$,

$$\frac{1}{n} \sum_{i=1}^{n} r(X_i) \tag{2.3}$$

is a discrete random variable on $(\Omega, \mathcal{F}, \mathbb{P})$ assuming values in a countable subset of $\mathbb{R}$. The elementary form of long-run average performance metrics is the *long-run sample average* of r, which is defined as [3]

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} r(X_i) \tag{2.4}$$

Performance metric (2.4) is the long-run average of the reward values $r(X_i)$ weighted over the number of visited states. Since the discrete random variables X_i are not independent (due to the transition probabilities of the Markov chain), the discrete random variables $r(X_i)$ are not independent either. As a consequence, the strong law of large numbers and the central limit theorem are not directly applicable for analysing almost sure convergence or convergence in distribution of (2.3) respectively. The remainder of this section presents classical performance analysis techniques that indirectly apply these limit theorems to allow both analytical computation and simulation-based estimation of long-run sample averages. An intuitive interpretation of some theoretical details from [39] and [43] is included to indicate the origin of the conditions that need to be satisfied when applying these techniques.

Analytical Computation To investigate computing long-run sample averages analytically, some additional notation is introduced. Consider an ergodic Markov chain $\{X_i \mid i \geq 1\}$ with state space $\mathcal{S}$ and equilibrium distribution π, for which reward function r is defined. Let $(\Omega, \mathcal{F}, \mathbb{P})$ denote its probability space and let S_r be a recurrent state in $\mathcal{S}$. Define for $i \geq 1$, the functions $v^i_{S_r}$ from Ω to $\mathbb{N}$ as the number of times that S_r was visited up to but not including time-epoch i. Furthermore, let for $k \geq 0$, the functions $t^k_{S_r}$ from Ω to $\{1, 2, \ldots\}$ denote the time-epoch of the k^{th} visit to S_r, where $t^0_{S_r}$ is set to 1. Notice that because the Markov chain is ergodic, S_r is in fact a positive state such that each infinite state sequence $\underline{S} \in \Omega$ incorporates infinitely many visits to S_r with probability 1. As a result, the functions $v^i_{S_r}$ and $t^k_{S_r}$ are properly defined discrete random variables on $(\Omega, \mathcal{F}, \mathbb{P})$ when discarding the null set of infinite state sequences in Ω that do not visit S_r infinitely often [39].

Now, define for $k \geq 1$, the discrete random variables $Y^k_{S_r}$ on $(\Omega, \mathcal{F}, \mathbb{P})$ as

$$Y^k_{S_r} = \sum_{i=t^k_{S_r}}^{t^{k+1}_{S_r}-1} r(X_i)$$

which presents the sum of reward values earned during the k^{th} cycle through S_r based on reward function r. Since the behaviour exposed during a cycle through S_r is independent from any other cycle through S_r, the discrete random variables $Y^k_{S_r}$ are independent and identically distributed (i.i.d.) [39]. The expected value $\mathbb{E}[Y_{S_r}]$ of the discrete random variables $Y^k_{S_r}$ equals (see also a remark at the end of section 2.1.2)

$$\mathbb{E}[Y_{S_r}] = \sum_{S\in\mathcal{S}} \frac{\pi_S}{\pi_{S_r}} \cdot r(S) \tag{2.5}$$

[3]The limit notation used in equation 2.4 refers to almost sure convergence unless explicitly stated that convergence in distribution is meant, see also appendix A.

With the introduced notation, for any recurrent state S_r, the long-run sample average of r as defined in (2.4) can be rewritten as

$$\lim_{n\to\infty}\left(\frac{1}{n}\sum_{i=1}^{t^1_{S_r}-1} r(X_i)+\frac{1}{n}\sum_{k=1}^{v^n_{S_r}-1} Y^k_{S_r}+\frac{1}{n}\sum_{i=t^{v^n_{S_r}}_{S_r}}^{n} r(X_i)\right) \tag{2.6}$$

To enable computing long-run sample averages, almost sure convergence of (2.6) is investigated. The first term in (2.6) concerns the addition of reward values $r(X_i)$ until visiting S_r for the first time, which may therefore include reward values earned during the visit of transient states. Since S_r is a positive state, the time-epoch at which S_r is visited for the first time $t^1_{S_r}$ is finite with probability 1. Hence,

$$\frac{1}{n}\sum_{i=1}^{t^1_{S_r}-1} r(X_i) \xrightarrow{\text{a.s.}} 0$$

The second term in (2.6) can be rewritten as

$$\frac{v^n_{S_r}-1}{n}\cdot\frac{1}{v^n_{S_r}-1}\sum_{k=1}^{v^n_{S_r}-1} Y^k_{S_r}$$

which converges almost surely to $\pi_{S_r}\cdot\mathbb{E}[Y_{S_r}]$ by the strong law of large numbers in case the discrete random variables $Y^k_{S_r}\in\mathcal{L}^1$. This can be understood by realising that $\frac{1}{n}\cdot(v^n_{S_r}-1)$ converges almost surely to the long-run fraction of time-epochs that Markov chain is in S_r. In order to investigate the last term in (2.6), define for $k\geq 1$, the i.i.d. discrete random variables $U^k_{S_r}$ on probability space $(\Omega,\mathcal{F},\mathbb{P})$ as

$$U^k_{S_r}=\sum_{i=t^k_{S_r}}^{t^{k+1}_{S_r}-1} |r(X_i)|$$

which represents the sum of absolute reward values earned during the k^{th} cycle through S_r. In [39], it is proven that

$$\frac{1}{n}\sum_{i=t^{v^n_{S_r}}_{S_r}}^{n} r(X_i) \xrightarrow{\text{a.s.}} 0$$

provided that the expected value $\mathbb{E}[U_{S_r}]$ of the discrete random variables $U^k_{S_r}$ is finite[4]. Notice that this condition is stronger than requiring that the discrete random variables $Y^k_{S_r}\in\mathcal{L}^1$, which is equivalent to requiring that $\mathbb{E}[|Y_{S_r}|]<\infty$.

As each term of (2.6) individually converges almost surely, performance metric (2.4) converges almost surely as well. The final result, which is known as the *ergodic theorem*, is that if a Markov chain $\{X_i \mid i\geq 1\}$ with state space $\mathcal{S}$ is ergodic and its

[4]When considering convergence in probability [95] of equation (2.3), this condition can be relaxed [39].

equilibrium distribution is denoted by π, then for a reward function r defined on $\{X_i \mid i \geq 1\}$

$$\frac{1}{n}\sum_{i=1}^{n} r(X_i) \xrightarrow{\text{a.s.}} \sum_{S \in \mathcal{S}} \pi_S \cdot r(S) \tag{2.7}$$

provided that $\mathbb{E}[U_{S_r}] < \infty$ or equivalently [39] that the series in (2.7) converges absolutely. The ergodic theorem is suitable for computing the long-run sample average analytically after determining the equilibrium distribution of $\{X_i \mid i \geq 1\}$. This requires solving a system of $|\mathcal{S}| + 1$ linear equations as illustrated in example 2.2.

Example 2.2 *Consider again the Markov chain of example 2.1 on page 19 and let reward function r be defined for this Markov chain by $r(A) = -1$, $r(B) = -2$ and $r(C) = 3$. Of interest is the long-run sample average of r.*

Because the Markov chain is ergodic, its equilibrium distribution can be determined by solving the system of linear equations

$$\begin{aligned}
\pi_A &= \pi_A \cdot \mathcal{P}_{A,A} + \pi_B \cdot \mathcal{P}_{A,B} + \pi_C \cdot \mathcal{P}_{A,C} \\
\pi_B &= \pi_A \cdot \mathcal{P}_{B,A} + \pi_B \cdot \mathcal{P}_{B,B} + \pi_C \cdot \mathcal{P}_{B,C} \\
\pi_C &= \pi_A \cdot \mathcal{P}_{C,A} + \pi_B \cdot \mathcal{P}_{C,B} + \pi_C \cdot \mathcal{P}_{C,C} \\
&\pi_A + \pi_B + \pi_C = 1
\end{aligned}$$

which gives that $\pi_A = 0$, $\pi_B = \frac{2}{3}$ and $\pi_C = \frac{1}{3}$. Application of the ergodic theorem gives that the long-run sample average of r equals $\pi_A \cdot r(A) + \pi_B \cdot r(B) + \pi_C \cdot r(C) = -\frac{1}{3}$.

Simulation-based Estimation Consider an ergodic Markov chain $\{X_i \mid i \geq 1\}$ defined on probability space $(\Omega, \mathcal{F}, \mathbb{P})$ with state space $\mathcal{S}$. Let a reward function r be defined such that the long-run sample average μ_s of r exists. If $|\mathcal{S}|$ is large, determining the equilibrium distribution of $\{X_i \mid i \geq 1\}$ for computing μ_s analytically can be prohibitively complex. In case $|\mathcal{S}| = \infty$, computing μ_s is in general even impossible. In such cases, μ_s can however be estimated using simulation. *Simulation* of $\{X_i \mid i \geq 1\}$ concerns the generation of a finite state sequence $\underline{S}$ for which $\underline{S}^{\rightsquigarrow} \in \Omega$. Simulation tools support the latter by their capability of generating traces. A *trace* is a finite sequence of consecutively visited states in accordance with positive transition probabilities starting from a state with a positive initial probability.

Definition 2.7 (Trace) *A finite state sequence $\underline{S} = (S_1, \ldots, S_n)$ for a Markov chain represented by $(\mathcal{S}, I, \mathcal{P})$ is called a trace, if $I_{S_1} > 0$ and $\mathcal{P}_{S_i,S_{i+1}} > 0$ for each $1 \leq i < n$.*

To investigate simulation-based estimation of μ_s, some more notation is introduced. Let S_r be a recurrent state in $\mathcal{S}$. For $k \geq 1$, let the functions $L^k_{S_r}$ from Ω to $\{1, 2, \ldots\}$ indicate the length of the k^{th} cycle through S_r by defining $L^k_{S_r} = t^{k+1}_{S_r} - t^k_{S_r}$. Similarly as argued for $Y^k_{S_r}$, the functions $L^k_{S_r}$ are i.i.d. discrete random variables on $(\Omega, \mathcal{F}, \mathbb{P})$. Notice that the expected value $\mathbb{E}[L_{S_r}]$ of the discrete random variables $L^k_{S_r}$ is equal to the expected recurrence time of S_r (which equals $\frac{1}{\pi_{S_r}}$, see the remark at the end of section 2.1.2). Hence, by equation (2.5) and the ergodic theorem,

$$\mu_s = \frac{\mathbb{E}[Y_{S_r}]}{\mathbb{E}[L_{S_r}]} \tag{2.8}$$

for any recurrent state S_r. As a result, estimation of μ_s can be based on estimating the quotient of expected values (2.8). The expected values $\mathbb{E}[Y_{S_r}]$ and $\mathbb{E}[L_{S_r}]$ can be estimated using the strongly consistent (and unbiased) point estimators

$$Y_{S_r}(n) = \frac{1}{n}\sum_{k=1}^{n} Y_{S_r}^{k} \quad \text{and} \quad L_{S_r}(n) = \frac{1}{n}\sum_{k=1}^{n} L_{S_r}^{k} \tag{2.9}$$

respectively. Hence,

$$\hat{\mu}_s = \frac{Y_{S_r}(n)}{L_{S_r}(n)} \tag{2.10}$$

is a strongly consistent point estimator for μ_s if $Y_{S_r}^{k} \in \mathcal{L}^1$. Notice that the necessary conditions of $L_{S_r}^{k} \in \mathcal{L}^1$ and $\mathbb{E}[L_{S_r}] > 0$ are satisfied by the ergodicity of the Markov chain. Unfortunately, the *classical point estimator* for μ_s defined in (2.10) is biased in general. Several other point estimators such as the Beale, Jacknife, Fieller and Tin point estimators [87] were proposed in an effort to reduce the bias of the classical point estimator. Nevertheless, the classical point estimator is the easiest one to integrate in simulation tools, making it an attractive candidate for estimating μ_s [43].

Although any trace generated during simulation may serve as a basis for estimating μ_s, it is necessary to indicate the accuracy of an obtained point estimate $\bar{\mu}_s$ for μ_s because traces are finite. The *accuracy* of a point estimate $\bar{\mu}_s$ is defined as 1 minus the relative error of $\bar{\mu}_s$ with respect to μ_s. This relative error is defined as

$$\left| \frac{\bar{\mu}_s - \mu_s}{\mu_s} \right|$$

Point estimates generally become more accurate for longer traces (longer simulations). It is however difficult to determine in advance how long a simulation should run to ensure accurate point estimates. By specifying a desired *accuracy bound*, a simulation can be terminated *automatically* in case the relative error of an obtained point estimate becomes smaller than 1 minus that bound.

To determine the accuracy of a point estimate $\bar{\mu}_s$ based on the classical point estimator $\hat{\mu}_s$, a Confidence Interval is derived for μ_s. For this purpose, define for $k \geq 1$, the i.i.d. discrete random variables $Z_{S_r}^{k} = Y_{S_r}^{k} - \mu_s \cdot L_{S_r}^{k}$ on $(\Omega, \mathcal{F}, \mathbb{P})$. Notice that the expected value $\mathbb{E}[Z_{S_r}]$ equals 0. $\mathbb{E}[Z_{S_r}]$ can be estimated using the (strongly consistent and unbiased) point estimator

$$Z_{S_r}(n) = \frac{1}{n}\sum_{k=1}^{n} Z_{S_r}^{k} = \frac{1}{n}\sum_{k=1}^{n} Y_{S_r}^{k} - \mu_s \cdot \frac{1}{n}\sum_{k=1}^{n} L_{S_r}^{k} = Y_{S_r}(n) - \mu_s \cdot L_{S_r}(n)$$

In case the discrete random variables $Z_{S_r}^{k} \in \mathcal{L}^2$ (or equivalently $\mathbb{E}[|Z_{S_r}|^2] < \infty$) and their variance $\tau^2 = \text{var}[Z_{S_r}] > 0$, application of the central limit theorem gives that

$$\frac{\sqrt{n}}{\tau} \cdot Z_{S_r}(n) \xrightarrow{\text{d.}} \text{N}(0,1)$$

This means that for sufficiently large n and every $\kappa \in \mathbb{R}$

$$\mathbb{P}\left(-\kappa \leq \frac{\sqrt{n}}{\tau} \cdot Z_{S_r}(n) \leq \kappa\right) = \\ \mathbb{P}\left(\frac{Y_{S_r}(n)}{L_{S_r}(n)} - \frac{\kappa\tau}{\sqrt{n}} \cdot \frac{1}{L_{S_r}(n)} \leq \mu_s \leq \frac{Y_{S_r}(n)}{L_{S_r}(n)} + \frac{\kappa\tau}{\sqrt{n}} \cdot \frac{1}{L_{S_r}(n)}\right)$$

is approximately $2\mathfrak{R}(\kappa) - 1$. Hence, the stochastic interval

$$\left[\hat{\mu}_s - \frac{\kappa\tau}{\sqrt{n}} \cdot \frac{1}{L_{S_r}(n)}, \hat{\mu}_s + \frac{\kappa\tau}{\sqrt{n}} \cdot \frac{1}{L_{S_r}(n)}\right]$$

is an (approximate) $2\mathfrak{R}(\kappa) - 1$ Confidence Interval for μ_s. To obtain a confidence interval based on this Confidence Interval, the variance τ^2 must be known. With simulation, τ^2 can be estimated using the (unbiased) point estimator

$$\frac{1}{n-1}\sum_{k=1}^{n}(Y_{S_r}^k - \mu_s \cdot L_{S_r}^k)^2$$

in which μ_s can be replaced by point estimator $\hat{\mu}_s$ since $\hat{\mu}_s \xrightarrow{\text{a.s.}} \mu_s$. Hence,

$$\begin{aligned}\hat{\tau}^2 &= \frac{1}{n-1}\sum_{k=1}^{n}(Y_{S_r}^k - \hat{\mu}_s \cdot L_{S_r}^k)^2 \\ &= \frac{1}{n-1}\sum_{k=1}^{n}(Y_{S_r}^k)^2 - 2\hat{\mu}_s \cdot \frac{1}{n-1}\sum_{k=1}^{n}(Y_{S_r}^k \cdot L_{S_r}^k) + \hat{\mu}_s^2 \cdot \frac{1}{n-1}\sum_{k=1}^{n}(L_{S_r}^k)^2\end{aligned} \tag{2.11}$$

is a strongly consistent point estimator for τ^2. By Slutsky's theorem [95],

$$\frac{\sqrt{n}}{\hat{\tau}} \cdot Z_{S_r}(n) \xrightarrow{\text{d.}} \mathrm{N}(0,1)$$

and hence, the stochastic interval

$$\left[\hat{\mu}_s - \frac{\kappa\hat{\tau}}{\sqrt{n}} \cdot \frac{1}{L_{S_r}(n)}, \hat{\mu}_s + \frac{\kappa\hat{\tau}}{\sqrt{n}} \cdot \frac{1}{L_{S_r}(n)}\right] \tag{2.12}$$

is an (approximate) $2\mathfrak{R}(\kappa) - 1$ Confidence Interval for μ_s.

The bounds of Confidence Interval (2.12) for μ_s have an important property. Let

$$\varphi_1 = \hat{\mu}_s - \frac{\kappa\hat{\tau}}{\sqrt{n}} \cdot \frac{1}{L_{S_r}(n)} \quad \text{and} \quad \varphi_2 = \hat{\mu}_s + \frac{\kappa\hat{\tau}}{\sqrt{n}} \cdot \frac{1}{L_{S_r}(n)}$$

Since

$$\lim_{n\to\infty}\frac{\kappa\hat{\tau}}{\sqrt{n}} = 0$$

both $\varphi_1 \xrightarrow{\text{a.s.}} \mu_s$ and $\varphi_2 \xrightarrow{\text{a.s.}} \mu_s$. Hence, the bounds of a confidence interval obtained based on $[\varphi_1, \varphi_2]$ tend to μ_s for longer traces (longer simulations).

Notice that if $\tau^2 = 0$, the central limit theorem cannot be applied. For $\tau^2 = 0$, it follows by lemma A.2 that $Z_{S_r} = \mathbb{E}[Z_{S_r}] = 0$ and hence $Y_{S_r}(n) = \mu_s \cdot L_{S_r}(n)$. As a result, stochastic interval (2.12) reduces to $[\mu_s, \mu_s]$, which is still a valid $2\Re(\kappa) - 1$ Confidence Interval for μ_s according to definition A.1. Therefore, the condition $\tau^2 > 0$ for obtaining valid confidence intervals based on (2.12) can and will be discarded.

Based on Confidence Interval (2.12), a $2\Re(\kappa) - 1$ confidence interval $[\bar{\varphi}_1, \bar{\varphi}_2]$ can be determined for μ_s during simulation with

$$\bar{\varphi}_1 = \bar{\mu}_s - \frac{\kappa\bar{\tau}}{\sqrt{N}} \cdot \frac{1}{L_{S_r}(N)} \quad \text{and} \quad \bar{\varphi}_2 = \bar{\mu}_s + \frac{\kappa\bar{\tau}}{\sqrt{N}} \cdot \frac{1}{L_{S_r}(N)}$$

where N denotes the number of completed regenerative cycles. Now, if $\mu_s \in [\bar{\varphi}_1, \bar{\varphi}_2]$, then an upper bound for the relative error of point estimate $\bar{\mu}_s$ is given by

$$\left|\frac{\bar{\mu}_s - \mu_s}{\mu_s}\right| \leq \begin{cases} \frac{\bar{\varphi}_2 - \bar{\varphi}_1}{2\bar{\varphi}_1} & \text{if } 0 < \bar{\varphi}_1 < \infty \\ \frac{\bar{\varphi}_1 - \bar{\varphi}_2}{2\bar{\varphi}_2} & \text{if } -\infty < \bar{\varphi}_2 < 0 \\ \infty & \text{otherwise} \end{cases} \tag{2.13}$$

Notice that because $|\varphi_1 - \varphi_2| \xrightarrow{\text{a.s.}} 0$ by the property that the bounds of Confidence Interval (2.12) converge almost surely to μ_s, the upper bound (2.13) for the relative error for $\bar{\mu}_s$ tends to 0 for longer traces (longer simulations) if $\mu_s \neq 0$.

From the derivation above, it is concluded that simulation-based estimation of the long-run sample average μ_s using point estimator (2.10) and Confidence Interval (2.12) requires that $\mathbb{E}[|Y_{S_r}|] < \infty$ and $\mathbb{E}[|Z_{S_r}|^2] < \infty$. The latter condition is satisfied if both $\mathbb{E}[|Y_{S_r}|^2] < \infty$ and $\mathbb{E}[|L_{S_r}|^2] < \infty$. Since $\mathbb{E}[|Y_{S_r}|^2] < \infty$ implies $\mathbb{E}[|Y_{S_r}|] < \infty$, the conditions reduce to requiring that $\mathbb{E}[|Y_{S_r}|^2] < \infty$ and $\mathbb{E}[|L_{S_r}|^2] < \infty$.

Summary of Conditions Let r be a reward function defined for an ergodic Markov chain with recurrent state S_r. Computing the long-run sample average of r analytically using the ergodic theorem (2.7) requires that $\mathbb{E}[U_{S_r}] < \infty$. On the other hand, simulation-based estimation of the long-run sample average of r using point estimator (2.10) and Confidence Interval (2.12) requires $\mathbb{E}[|Y_{S_r}|^2] < \infty$ and $\mathbb{E}[|L_{S_r}|^2] < \infty$. In this thesis, r is called *proper* if all three conditions are satisfied for both analytical computation and simulation-based estimation of the long-run sample average of r.

Definition 2.8 (Proper Reward Function) *Let r be a reward function defined for an ergodic Markov chain $\{X_i \mid i \geq 1\}$ with a recurrent state S_r. If $\mathbb{E}[U_{S_r}] < \infty$, $\mathbb{E}[|Y_{S_r}|^2] < \infty$ and $\mathbb{E}[|L_{S_r}|^2] < \infty$, then r is called proper for $\{X_i \mid i \geq 1\}$.*

Although definition 2.8 indicates the precise conditions for a reward function r that enable both analytical computation and simulation-based estimation of the long-run sample average of r, it is sometimes convenient to use slightly stronger conditions. Instead of requiring that $\mathbb{E}[U_{S_r}] < \infty$ and $\mathbb{E}[|Y_{S_r}|^2] < \infty$, the condition $\mathbb{E}[|U_{S_r}|^2] < \infty$ (or equivalently $U_{S_r} \in \mathcal{L}^2$) can be used. When doing so, the three conditions in definition 2.8 can be replaced by $\mathbb{E}[|U_{S_r}|^2] < \infty$ and $\mathbb{E}[|L_{S_r}|^2] < \infty$. It is remarked that reward values for performance metrics of industrial hardware/software systems usually are all non-negative, in which case $Y_{S_r} = U_{S_r}$ and hence the condition $\mathbb{E}[|U_{S_r}|^2] < \infty$ then boils down to requiring that $\mathbb{E}[(Y_{S_r})^2] < \infty$.

2.2 Evaluating Conditional Long-Run Sample Averages

The classical performance analysis techniques discussed in section 2.1.3 provide ample means for analytical computation and simulation-based estimation of long-run sample averages. However, long-run average performance metrics for industrial hardware/software systems are generally *conditional* in the sense that reward values must only be taken into account in case a certain condition is satisfied.

Example 2.3 *Consider a telecommunication system that receives variable-sized packets from a network environment before processing them. Of interest is the evaluation of the expected packet size. A common performance modelling approach for this evaluation is to define a variable r, which holds the size of the last packet received, and assign a new value to r each time a new packet is received. When formalising the behaviour of the telecommunication system with a Markov chain, the values of variable r represent the result of a reward function that assigns the size of the last packet received to each state of the Markov chain. Now, directly applying the classical performance analysis techniques will not yield the desired expected packet size since the condition that the value of r should only be included for those states of the Markov chain where a new packet is received, is not taken into account.*

This section investigates the evaluation of conditional long-run sample averages by developing a performance analysis technique that allows taking only those states of a Markov chain into account for which the condition is satisfied. This so-called reduction technique, which was initially introduced in [142], involves deriving a Markov chain with a reduced state space from the original Markov chain that formalises the behaviour of a hardware/software system. By preserving ergodicity and long-run average performance results, the proposed reduction technique enables applying the classical performance analysis techniques on the obtained reduced Markov chain for computing or estimating conditional long-run sample averages.

Because formalising the behaviour of industrial hardware/software systems using Markov chains easily results in huge[5] state spaces [171], considering a reduced number of states may additionally alleviate rapid evaluation of performance properties. Therefore, the potential of increasing performance evaluation speed when applying the reduction technique is investigated for both analytical computation and simulation-based estimation of conditional long-run sample averages.

2.2.1 Markov Chain Reduction

Consider a Markov chain $\{X_i \mid i \geq 1\}$ with state space $\mathcal{S}$ for which reward function r is defined. Long-run sample averages for which a certain condition must be taken into account are long-run average performance metrics of the form

$$\lim_{n\to\infty} \frac{\displaystyle\sum_{i=1}^{n} r(X_i)\cdot c(X_i)}{\displaystyle\sum_{i=1}^{n} c(X_i)} \tag{2.14}$$

[5]A small 10×10 Internet router has already a control space of at least 10^{100} states, see also section 5.1.

where c is a reward function $c : \mathcal{S} \to \{0, 1\}$. A reward function that assigns reward value 0 or 1 to the states of a Markov chain will be called a *conditional* reward function. Conditional reward functions can be used to define the condition for which a reward value $r(X_i)$ must be taken into account or not by taking $c(X_i) = 1$ if the condition holds and $c(X_i) = 0$ otherwise. Observe that the condition of whether a new packet is received in example 2.3 can be formalised with this approach. Performance metric (2.14) is referred to as the *conditional* long-run sample average of r (with condition c) and can be explained as the long-run average of r weighted over the number of visited states at which the condition is satisfied.

Definition 2.9 (Conditional Reward Function) *A function $c : \mathcal{S} \to \{0, 1\}$ defined for a Markov chain with state space $\mathcal{S}$ is called a conditional reward function. State $S \in \mathcal{S}$ is called relevant (with respect to c) if $c(S) = 1$ and irrelevant (with respect to c) otherwise.*

Notice that performance metric (2.14) is equivalent to the long-run sample average (2.4) in case all states of the Markov chain are relevant. The following theorem identifies the conditions for directly computing (2.14) based on the ergodic theorem.

Theorem 2.1 *Let r be a proper reward function and let c be a conditional reward function defined for an ergodic Markov chain $\{X_i \mid i \geq 1\}$ with state space $\mathcal{S}$ and equilibrium distribution π. If this Markov chain has a recurrent state that is relevant with respect to c, then the conditional long-run sample average of r (with respect to c) is equal to*

$$\frac{\displaystyle\sum_{S \in \mathcal{S}} \pi_S \cdot r(S) \cdot c(S)}{\displaystyle\sum_{S \in \mathcal{S}} \pi_S \cdot c(S)}$$

Proof Because r is a proper reward function, it follows from definition 2.8 and a remark to the ergodic theorem (2.7) that[6]

$$\sum_{S \in \mathcal{S}} \pi_S \cdot |r(S) \cdot c(S)| \leq \sum_{S \in \mathcal{S}} \pi_S \cdot |r(S)| < \infty$$

Hence, by the ergodic theorem,

$$\frac{1}{n} \sum_{i=1}^{n} r(X_i) \cdot c(X_i) \xrightarrow{\text{a.s.}} \sum_{S \in \mathcal{S}} \pi_S \cdot r(S) \cdot c(S)$$

On the other hand, for conditional reward function c,

$$\sum_{S \in \mathcal{S}} \pi_S \cdot |c(S)| \leq 1$$

and hence, a second application of the ergodic theorem gives that

$$\frac{1}{n} \sum_{i=1}^{n} c(X_i) \xrightarrow{\text{a.s.}} \sum_{S \in \mathcal{S}} \pi_S \cdot c(S)$$

[6]Observe that it would actually be sufficient to require that $r \cdot c$ is proper.

Now, since there exists a recurrent state $S_r \in \mathcal{S}$ for which $c(S_r) = 1$,

$$\sum_{S \in \mathcal{S}} \pi_S \cdot c(S) > 0$$

Hence, the final result is that

$$\frac{\sum_{i=1}^{n} r(X_i) \cdot c(X_i)}{\sum_{i=1}^{n} c(X_i)} \xrightarrow{\text{a.s.}} \frac{\sum_{S \in \mathcal{S}} \pi_S \cdot r(S) \cdot c(S)}{\sum_{S \in \mathcal{S}} \pi_S \cdot c(S)} \qquad \blacksquare$$

The application of theorem 2.1 is illustrated with an example.

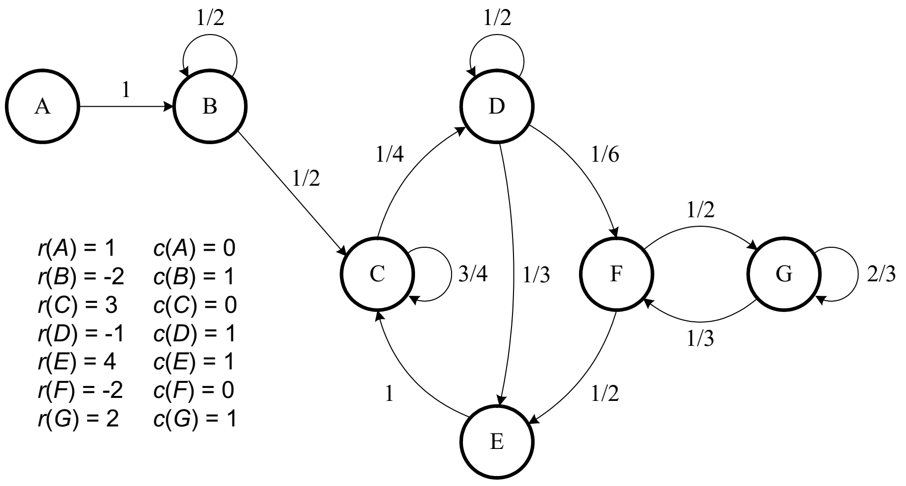


Figure 2.3: Example of a Markov chain.

Example 2.4 *Consider the Markov chain with state space $\mathcal{S} = \{A, B, C, D, E, F, G\}$ depicted in figure 2.3. For this ergodic Markov chain, reward function r and conditional reward function c are defined as indicated. The equilibrium probabilities for this Markov chain are $\pi_A = 0$, $\pi_B = 0$, $\pi_C = \frac{12}{26}$, $\pi_D = \frac{6}{26}$, $\pi_E = \frac{3}{26}$, $\pi_F = \frac{2}{26}$, $\pi_G = \frac{3}{26}$. Application of theorem 2.1 gives $\pi_A \cdot r(A) \cdot c(A) + \pi_B \cdot r(B) \cdot c(B) + \pi_C \cdot r(C) \cdot c(C) + \pi_D \cdot r(D) \cdot c(D) + \pi_E \cdot r(E) \cdot c(E) + \pi_F \cdot r(F) \cdot c(F) + \pi_G \cdot r(G) \cdot c(G) = \frac{12}{26}$ and $\pi_A \cdot c(A) + \pi_B \cdot c(B) + \pi_C \cdot c(C) + \pi_D \cdot c(D) + \pi_E \cdot c(E) + \pi_F \cdot c(F) + \pi_G \cdot c(G) = \frac{12}{26}$. Hence, the conditional long-run sample average of r equals* 1.

Theorem 2.1 shows that a conditional long-run sample average merely depends on the reward values earned in relevant states. It makes sense to investigate whether conditional long-run sample averages can be evaluated without considering irrelevant states and whether such an approach increases performance evaluation speed.

Consider a Markov chain $\{X_i \mid i \geq 1\}$ on probability space $(\Omega, \mathcal{F}, \mathbb{P})$ for which reward function r and conditional reward function c are defined. In case the state space of this Markov chain is denoted by $\mathcal{S}$, the set of states that are relevant with respect to c is represented by $\mathcal{S}^{\restriction c}$ and is defined by $\mathcal{S}^{\restriction c} = \{S \in \mathcal{S} \mid c(S) = 1\}$. In this section, a Markov chain $\{X_i^{\restriction c} \mid i \geq 1\}$ with state space $\mathcal{S}^{\restriction c}$ is introduced, which is defined on the same probability space $(\Omega, \mathcal{F}, \mathbb{P})$ as the Markov chain $\{X_i \mid i \geq 1\}$. To this end, define for each $i \geq 1$ and $\underline{S} \in \Omega$, the functions $X_i^{\restriction c}$ from Ω to $\mathcal{S}^{\restriction c}$ as

$$X_i^{\restriction c}(\underline{S}) = \begin{cases} \text{the } i^{\text{th}} \text{ state in } \underline{S} \text{ that is relevant with respect to } c & \text{if it exists} \\ \perp & \text{otherwise} \end{cases}$$

Notice that $\{X_i^{\restriction c} \mid i \geq 1\}$ is not necessarily a discrete-time stochastic process, let alone a Markov chain, because the functions $X_i^{\restriction c}$ are in general not complete. However, the following lemma states that $\{X_i^{\restriction c} \mid i \geq 1\}$ is a discrete-time stochastic process in case $\{X_i \mid i \geq 1\}$ is ergodic and if it has a relevant recurrent state.

Lemma 2.2 *Let conditional reward function c be defined for an ergodic Markov chain $\{X_i \mid i \geq 1\}$. If this Markov chain has a recurrent state that is relevant with respect to c, then $\{X_i^{\restriction c} \mid i \geq 1\}$ is a discrete-time stochastic process.*

Proof Assume that the Markov chain $\{X_i \mid i \geq 1\}$ is defined on probability space $(\Omega, \mathcal{F}, \mathbb{P})$ and that its state space is denoted by $\mathcal{S}$. Let $S_r \in \mathcal{S}$ be a recurrent state that is relevant with respect to c. Since the Markov chain is ergodic, S_r is a positive state. Each infinite state sequence $\underline{S} \in \Omega$ therefore incorporates infinitely many visits to S_r with probability 1. When discarding the null event of infinite state sequences not including infinitely many visits to S_r [39], the i^{th} state in $\underline{S}$ that is relevant with respect to c exists for all $i \geq 1$ and $\underline{S} \in \Omega$. Hence, the functions $X_i^{\restriction c} : \Omega \to \mathcal{S}^{\restriction c}$ are defined for all $i \geq 1$.

It remains to be shown that the functions $X_i^{\restriction c}$ are discrete random variables for all $i \geq 1$. However, this follows immediately from the fact that any set of infinite state sequences is measurable according to probability measure $\mathbb{P}$. ■

The following theorem states that the discrete-time stochastic process $\{X_i^{\restriction c} \mid i \geq 1\}$ satisfies both the Markovian property and the property of time-homogeneity for the same conditions as in lemma 2.2, such that $\{X_i^{\restriction c} \mid i \geq 1\}$ is actually a Markov chain.

Theorem 2.3 (Reduction Theorem) *Let conditional reward function c be defined for an ergodic Markov chain $\{X_i \mid i \geq 1\}$. If this Markov chain has a recurrent state that is relevant with respect to c, then $\{X_i^{\restriction c} \mid i \geq 1\}$ is also a Markov chain.*

Proof Assume that the state space of the Markov chain $\{X_i \mid i \geq 1\}$ is denoted by $\mathcal{S}$. Define for any $T \in \mathcal{S}^{\restriction c}$, the final set $\mathcal{X}^T$ of finite state sequences with final state T, for which all preceding states are irrelevant with respect to c. Notice that $\mathcal{X}_T \neq \varnothing$. $\mathbb{P}(\mathcal{X}^T)$ is the probability that the first relevant state visited by the Markov chain $\{X_i \mid i \geq 1\}$ is T. Furthermore, define for any $S \in \mathcal{S}$ and $T \in \mathcal{S}^{\restriction c}$, the initial and final set $\mathcal{X}_S^T$ of finite state sequences of lengths greater than 0 with initial state S and final state T, for which all intermediate states are irrelevant with respect to c.

$\mathbb{P}^*(\mathcal{X}_S^T)$ is the probability that the Markov chain $\{X_i \mid i \geq 1\}$ ever makes a transition to state T on the condition of departing from state S.

By lemma 2.2, $\{X_i^{\restriction c} \mid i \geq 1\}$ is a discrete-time stochastic process, where each discrete random variable $X_i^{\restriction c}$ assumes values in $\mathcal{S}^{\restriction c}$. To prove that $\{X_i^{\restriction c} \mid i \geq 1\}$ is a Markov chain, the Markovian property

$$\mathbb{P}(X_{i+1}^{\restriction c} = S_{i+1} \mid X_1^{\restriction c} = S_1, \ldots, X_i^{\restriction c} = S_i) = \mathbb{P}(X_{i+1}^{\restriction c} = S_{i+1} \mid X_i^{\restriction c} = S_i) \tag{2.15}$$

must be satisfied for each time-epoch i and all possible values of $S_1, \ldots, S_{i+1} \in \mathcal{S}^{\restriction c}$. Using the definitions of $\mathcal{X}^T$ and $\mathcal{X}_S^T$, the left hand side of (2.15) can be rewritten as

$$\begin{aligned}
\mathbb{P}(X_{i+1}^{\restriction c} = S_{i+1} \mid X_1^{\restriction c} = S_1, \ldots, X_i^{\restriction c} = S_i) &= \frac{\mathbb{P}(X_1^{\restriction c} = S_1, \ldots, X_{i+1}^{\restriction c} = S_{i+1})}{\mathbb{P}(X_1^{\restriction c} = S_1, \ldots, X_i^{\restriction c} = S_i)} \\
&= \frac{\mathbb{P}(\mathcal{X}^{S_1} \circ \mathcal{X}_{S_1}^{S_2} \circ \ldots \circ \mathcal{X}_{S_{i-1}}^{S_i} \circ \mathcal{X}_{S_i}^{S_{i+1}})}{\mathbb{P}(\mathcal{X}^{S_1} \circ \mathcal{X}_{S_1}^{S_2} \circ \ldots \circ \mathcal{X}_{S_{i-1}}^{S_i})} \\
&= \frac{\mathbb{P}(\mathcal{X}^{S_1} \circ \mathcal{X}_{S_1}^{S_2} \circ \ldots \circ \mathcal{X}_{S_{i-1}}^{S_i}) \cdot \mathbb{P}^*(\mathcal{X}_{S_i}^{S_{i+1}})}{\mathbb{P}(\mathcal{X}^{S_1} \circ \mathcal{X}_{S_1}^{S_2} \circ \ldots \circ \mathcal{X}_{S_{i-1}}^{S_i})} \\
&= \mathbb{P}^*(\mathcal{X}_{S_i}^{S_{i+1}})
\end{aligned}$$

Similarly, the righthand side of (2.15) can be rewritten as

$$\begin{aligned}
\mathbb{P}(X_{i+1}^{\restriction c} = S_{i+1} \mid X_i^{\restriction c} = S_i) &= \frac{\mathbb{P}(X_i^{\restriction c} = S_i, X_{i+1}^{\restriction c} = S_{i+1})}{\mathbb{P}(X_i^{\restriction c} = S_i)} \\
&= \frac{\mathbb{P}(\mathcal{X}^{S_i} \circ \mathcal{X}_{S_i}^{S_{i+1}})}{\mathbb{P}(\mathcal{X}^{S_i})} \\
&= \frac{\mathbb{P}(\mathcal{X}^{S_i}) \cdot \mathbb{P}^*(\mathcal{X}_{S_i}^{S_{i+1}})}{\mathbb{P}(\mathcal{X}^{S_i})} \\
&= \mathbb{P}^*(\mathcal{X}_{S_i}^{S_{i+1}})
\end{aligned}$$

Hence, (2.15) holds for each time-epoch i and all possible values $S_1, \ldots, S_{i+1} \in \mathcal{S}^{\restriction c}$.

The second condition that has to be satisfied is the property of time-homogeneity; the one-step state transition probabilities $\mathbb{P}(X_{i+1}^{\restriction c} = T \mid X_i^{\restriction c} = S)$ must, for each $S, T \in \mathcal{S}^{\restriction c}$, be independent of time-epoch i. However, this follows immediately from the fact that $\mathbb{P}(X_{i+1}^{\restriction c} = T \mid X_i^{\restriction c} = S) = \mathbb{P}^*(\mathcal{X}_S^T)$ is independent from time-epoch i. ■

For an ergodic Markov chain $\{X_i \mid i \geq 1\}$ having a relevant recurrent state (with respect to c), the Markov chain $\{X_i^{\restriction c} \mid i \geq 1\}$ is frequently referred to as the *reduced* Markov chain (with respect to c).

2.2.2 Preservation of Performance Results

Section 2.2.1 revealed that the reduction of an ergodic Markov chain with a relevant recurrent state yields another Markov chain. To enable an investigation on whether conditional long-run sample averages can be evaluated by considering only the relevant states, some properties of the reduced Markov chain must be known. The following theorem states that reduction preserves ergodicity.

Theorem 2.4 (Preservation of Ergodicity) *Let conditional reward function c be defined for an ergodic Markov chain $\{X_i \mid i \geq 1\}$. If this Markov chain has a recurrent state that is relevant with respect to c, then the reduced Markov chain $\{X_i^{\restriction c} \mid i \geq 1\}$ is ergodic.*

Proof Assume that the state space of the original Markov chain $\{X_i \mid i \geq 1\}$ is denoted by $\mathcal{S}$ and let S_r be a recurrent state that is relevant with respect to c. Since the original Markov chain is ergodic, S_r is a positive state of this Markov chain. The proof involves showing that S_r is also a positive state of the reduced Markov chain $\{X_i^{\restriction c} \mid i \geq 1\}$ and that it is reachable from any other state in $\mathcal{S}^{\restriction c}$ with probability 1.

According to definition 2.2, state S_r is reachable from any other state in $S \in \mathcal{S}^{\restriction c}$ if $\mathbb{P}^*(\mathcal{Z}_S^{S_r}) > 0$. But this is obviously true since $\mathbb{P}^*(\mathcal{Z}_S^{S_r}) > 0$ holds for the original Markov chain. Moreover, since S_r is a recurrent state of the original Markov chain, $\mathbb{P}^*(\mathcal{Z}_{S_r}^{S_r}) = 1$. Hence, S_r is also a recurrent state of the reduced Markov chain.

To show that S_r is a positive state of the reduced Markov chain, let $|\underline{S}|^{\restriction c}$ denote the number of states in $\underline{S} \in \mathcal{Z}_{S_r}^{S_r}$ that is relevant with respect to c minus 1. It must be shown that

$$\sum_{\underline{S} \in \mathcal{Z}_{S_r}^{S_r}} |\underline{S}|^{\restriction c} \cdot \mathbb{P}^*(\underline{S}) < \infty$$

Because S_r is a positive state in the original Markov chain

$$\sum_{\underline{S} \in \mathcal{Z}_{S_r}^{S_r}} |\underline{S}| \cdot \mathbb{P}^*(\underline{S}) < \infty$$

and since $|\underline{S}|^{\restriction c} \leq |\underline{S}|$ for any $\underline{S} \in \mathcal{Z}_{S_r}^{S_r}$,

$$\sum_{\underline{S} \in \mathcal{Z}_{S_r}^{S_r}} |\underline{S}|^{\restriction c} \cdot \mathbb{P}^*(\underline{S}) \leq \sum_{\underline{S} \in \mathcal{Z}_{S_r}^{S_r}} |\underline{S}| \cdot \mathbb{P}^*(\underline{S}) < \infty$$

Hence, S_r is also a positive state of the reduced Markov chain. ■

Lemma 2.5 *Let reward function r and conditional reward function c be defined for an ergodic Markov chain $\{X_i \mid i \geq 1\}$ with state space $\mathcal{S}$. If r is proper and $\{X_i \mid i \geq 1\}$ has a recurrent state S_r that is relevant with respect to c, then*

$$\mathbb{E}[U_{S_r}^{\restriction c}] < \infty \tag{2.16}$$

where $U_{S_r}^{\restriction c}$ denotes the sum of absolute reward values earned during a cycle of the reduced Markov chain $\{X_i^{\restriction c} \mid i \geq 1\}$ through S_r.

Proof Because reward function r is proper, the expected sum of absolute reward values $\mathbb{E}[U_{S_r}]$ earned during a cycle of the original Markov chain $\{X_i \mid i \geq 1\}$

through S_r is finite (see definition 2.8). This can be expressed as

$$\mathbb{E}[U_{S_r}] = \sum_{\underline{S} \in Z_{S_r}^{S_r}} \mathbb{P}^*(\underline{S}) \cdot \sum_{i=1}^{|\underline{S}|} |r(\underline{S}_i)| < \infty$$

Expectation $\mathbb{E}[U_{S_r}^{\restriction c}]$ equals the expected sum of absolute reward values earned in the relevant states that are visited during a cycle of the original Markov chain $\{X_i \mid i \geq 1\}$ through S_r. Hence,

$$\mathbb{E}[U_{S_r}^{\restriction c}] = \sum_{\underline{S} \in Z_{S_r}^{S_r}} \mathbb{P}^*(\underline{S}) \cdot \sum_{i=1}^{|\underline{S}|} |r(\underline{S}_i) \cdot c(\underline{S}_i)|$$

Since

$$\sum_{\underline{S} \in Z_{S_r}^{S_r}} \mathbb{P}^*(\underline{S}) \cdot \sum_{i=1}^{|\underline{S}|} |r(\underline{S}_i) \cdot c(\underline{S}_i)| \leq \sum_{\underline{S} \in Z_{S_r}^{S_r}} \mathbb{P}^*(\underline{S}) \cdot \sum_{i=1}^{|\underline{S}|} |r(\underline{S}_i)|$$

it follows that equation (2.16) holds indeed. ■

From the proof of lemma 2.5, it can be concluded that (2.16) also holds in case $r \cdot c$ is proper. Observe furthermore that (2.16) also holds when requiring the stronger condition $\mathbb{E}[|U_{S_r}|^2] < \infty$ for the original Markov chain (see also section 2.1.3).

Now, the following main theorem can be proven. It states that a conditional long-run sample average for a Markov chain evaluates to the same result as the long-run sample average for the reduced Markov chain.

Theorem 2.6 (Preservation of Long-Run Averages) *Let proper reward function r and conditional reward function c be defined for an ergodic Markov chain $\{X_i \mid i \geq 1\}$. If this Markov chain has a recurrent state that is relevant with respect to c, then*

$$\frac{\sum_{i=1}^{n} r(X_i) \cdot c(X_i)}{\sum_{i=1}^{n} c(X_i)} \xrightarrow{a.s.} \mu \quad \textit{if and only if} \quad \frac{1}{n} \sum_{i=1}^{n} r(X_i^{\restriction c}) \xrightarrow{a.s.} \mu$$

Proof Observe first that the conditional long-run sample average of r for $\{X_i \mid i \geq 1\}$ exists by theorem 2.1. On the other hand, the long-run sample average of r exists for $\{X_i^{\restriction c} \mid i \geq 1\}$ by theorem 2.4, lemma 2.5 and the ergodic theorem.

Assume that the Markov chain $\{X_i \mid i \geq 1\}$ is defined on probability space $(\Omega, \mathcal{F}, \mathbb{P})$. Let $\underline{S} = (S_1, S_2, \ldots)$ be an infinite state sequence in Ω that includes infinitely many visits to relevant states (the null event of infinite state sequences that do not include infinitely many visits to relevant states is discarded [39]). Then,

$$\lim_{n\to\infty} \frac{\sum_{i=1}^{n} r(X_i(\underline{S})) \cdot c(X_i(\underline{S}))}{\sum_{i=1}^{n} c(X_i(\underline{S}))} = \lim_{n\to\infty} \frac{\sum_{i=1}^{n} r(S_i) \cdot c(S_i)}{\sum_{i=1}^{n} c(S_i)}$$

Since $c(S_i) = 1$ if and only if S_i is relevant with respect to c, this expression equals

$$\lim_{m\to\infty} \frac{\displaystyle\sum_{i=1}^{m} r(i^{\text{th}} \text{ state in } \underline{S} \text{ that is relevant with respect to } c)}{m} = \lim_{m\to\infty} \frac{1}{m} \sum_{i=1}^{m} r(X_i^{\restriction c}(\underline{S}))$$

which completes the proof. ■

Theorem 2.6 provides the means for evaluating conditional long-run sample averages by considering relevant states only. In the remainder of this thesis, evaluating conditional long-run sample averages based on the application of theorems 2.3 and 2.6 is referred to as the *reduction technique*.

Although focussing on conditional long-run sample averages, the reduction technique is applicable for combinations of such long-run average performance metrics as well. Of special interest are quotients of conditional long-run sample averages. Consider a Markov chain $\{X_i \mid i \geq 1\}$ for which reward functions g and h and conditional reward function c are defined. The following corollary generalises the result of theorem 2.6 by considering performance metrics of the form

$$\lim_{n\to\infty} \frac{\displaystyle\sum_{i=1}^{n} g(X_i)\cdot c(X_i)}{\displaystyle\sum_{i=1}^{n} h(X_i)\cdot c(X_i)} \tag{2.17}$$

Performance metric (2.17) can be explained as the quotient of cumulated reward values $g(X_i)$ and $h(X_i)$ for those states that are relevant with respect to c.

Corollary 2.7 *Let proper reward functions g, h and conditional reward function c be defined for an ergodic Markov chain $\{X_i \mid i \geq 1\}$. If this Markov chain has a recurrent state that is relevant with respect to c, while*

$$\frac{1}{n}\sum_{i=1}^{n} g(X_i^{\restriction c}) \xrightarrow{a.s.} \mu \quad \textit{and} \quad \frac{1}{n}\sum_{i=1}^{n} h(X_i^{\restriction c}) \xrightarrow{a.s.} \lambda$$

with $\lambda \neq 0$, then the quotient in (2.17) converges almost surely to $\frac{\mu}{\lambda}$.

Proof The result follows immediately from theorem 2.6 by considering that

$$\frac{\displaystyle\sum_{i=1}^{n} g(X_i)\cdot c(X_i)}{\displaystyle\sum_{i=1}^{n} c(X_i)} \xrightarrow{\text{a.s.}} \mu \quad \text{and} \quad \frac{\displaystyle\sum_{i=1}^{n} h(X_i)\cdot c(X_i)}{\displaystyle\sum_{i=1}^{n} c(X_i)} \xrightarrow{\text{a.s.}} \lambda$$

■

2.2.3 Reduction and Analytical Computation

Application of the reduction technique relies on the relation that exists between a conditional long-run sample average defined for a Markov chain and the accompanying long-run sample average for the reduced Markov chain. Analytically computing a conditional long-run sample average can be accomplished indirectly by applying the ergodic theorem for the accompanying long-run sample average defined for the reduced Markov chain. However, this requires knowing the equilibrium distribution of the reduced Markov chain. Two different approaches can be identified for obtaining this equilibrium distribution, see also [142]. The first approach derives it directly from the equilibrium distribution of the original Markov chain. The second approach is indirect and computes it after constructing the reduced Markov chain.

Direct Computation of Equilibrium Distribution To compute the equilibrium distribution of the reduced Markov chain, the following theorem establishes the relation between the equilibrium distributions of the original and reduced Markov chains.

Theorem 2.8 *Let conditional reward function c be defined for an ergodic Markov chain $\{X_i \mid i \geq 1\}$ with state space $\mathcal{S}$ and equilibrium distribution π. If this Markov chain has a recurrent state that is relevant with respect to c, then the equilibrium distribution $\pi^{\restriction c}$ of the reduced Markov chain $\{X_i^{\restriction c} \mid i \geq 1\}$ is given by*

$$\pi_T^{\restriction c} = \frac{\pi_T}{\sum\limits_{S \in \mathcal{S}^{\restriction c}} \pi_S} \quad \textit{for all } T \in \mathcal{S}^{\restriction c}$$

Proof Let T be a state that is relevant with respect to c. Define reward function r_T as $r_T(S) = 1$ if $S = T$ and $r_T(S) = 0$ for any $S \in \mathcal{S} \setminus \{T\}$. Notice that r_T is proper and that $r_T(S) \cdot c(S) = r_T(S)$ for all $S \in \mathcal{S}$. By theorem 2.1,

$$\lim_{n\to\infty} \frac{\sum\limits_{i=1}^{n} r_T(X_i) \cdot c(X_i)}{\sum\limits_{i=1}^{n} c(X_i)} = \frac{\sum\limits_{S \in \mathcal{S}} \pi_S \cdot r_T(S)}{\sum\limits_{S \in \mathcal{S}} \pi_S \cdot c(S)} = \frac{\pi_T}{\sum\limits_{S \in \mathcal{S}^{\restriction c}} \pi_S}$$

On the other hand, by theorem 2.4 and the ergodic theorem

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} r_T(X_i^{\restriction c}) = \sum_{S \in \mathcal{S}^{\restriction c}} \pi_S^{\restriction c} \cdot r_T(S) = \pi_T^{\restriction c}$$

The result now follows directly from theorem 2.6. ■

Analytical computation of conditional long-run sample averages using the reduction technique and theorem 2.8 is illustrated with the following example.

Example 2.5 *Recall the Markov chain of example 2.4 on page 32. Satisfying the conditions of theorem 2.3, this Markov chain can be reduced with respect to conditional reward function* *c. The state space of the reduced Markov chain is $\mathcal{S}^{\restriction c} = \{B, D, E, G\}$. By theorem 2.8, the equilibrium distribution of the reduced Markov chain can be computed as $\pi_B^{\restriction c} = 0$, $\pi_D^{\restriction c} = \frac{1}{2}$, $\pi_E^{\restriction c} = \frac{1}{4}$, $\pi_G^{\restriction c} = \frac{1}{4}$. By theorem 2.6 and the ergodic theorem, the conditional long-run sample average of r equals $\pi_B^{\restriction c} \cdot r(B) + \pi_D^{\restriction c} \cdot r(D) + \pi_E^{\restriction c} \cdot r(E) + \pi_G^{\restriction c} \cdot r(G) = 1$. Opposed to the approach in example 2.4, the result is now obtained by considering the reward values of the relevant states only.*

Although the combination of the reduction technique and theorem 2.8 allows analytical computation of conditional long-run sample averages in a way that only reward values earned in states for which the condition is satisfied must be taken into account, the computationally complex task of determining the equilibrium distribution of the original Markov chain is still required. It can therefore be concluded that no improvement in evaluation speed is obtained with this approach compared to directly computing conditional long-run sample averages by means of theorem 2.1.

Construction Method Using the reduction technique in combination with theorem 2.8 to compute conditional long-run averages is not a constructive approach in the sense that the transition matrix and initial distribution of the reduced Markov chain are not determined. In case the reduced Markov chain is constructed, its equilibrium distribution can be computed in the classical way of solving a system of linear equations.

Consider an ergodic Markov chain represented by $(\mathcal{S}, I, \mathcal{P})$ for which conditional reward function c is defined. The transition matrix of the Markov chain obtained after reduction with respect to c is denoted by $\mathcal{P}^{\restriction c}$ and its initial distribution by $I^{\restriction c}$. Similar to the original Markov chain, the reduced Markov chain is completely determined by the set of states $\mathcal{S}^{\restriction c}$, the initial distribution $I^{\restriction c}$ and transition matrix $\mathcal{P}^{\restriction c}$. Therefore, it is also represented by the triple $(\mathcal{S}^{\restriction c}, I^{\restriction c}, \mathcal{P}^{\restriction c})$ or alternatively by $(\mathcal{S}, I, \mathcal{P})^{\restriction c}$.

To relate the transition matrix of the reduced Markov chain with the transition matrix of the original Markov chain, matrix M is introduced. For any $S \in \mathcal{S}$ and $T \in \mathcal{S}^{\restriction c}$, define $M_{S,T}$ as the probability that the original Markov chain ever makes a transition to T conditional on departing from S such that any intermediately visited state is irrelevant with respect to c. Notice that for $S \in \mathcal{S}^{\restriction c}$, $M_{S,T}$ equals the probability $\mathcal{P}_{S,T}^{\restriction c}$ that the reduced Markov chain transfers from S to T.

Theorem 2.9 *Let conditional reward function c be defined for an ergodic Markov chain with state space $\mathcal{S}$ and transition matrix $\mathcal{P}$. Then the elements $M_{S,T}$ of matrix M satisfy the system of linear equations*

$$M_{S,T} = \mathcal{P}_{S,T} + \sum_{Q \in \mathcal{S} \setminus \mathcal{S}^{\restriction c}} \mathcal{P}_{S,Q} \cdot M_{Q,T} \quad \textit{for all } S \in \mathcal{S} \textit{ and } T \in \mathcal{S}^{\restriction c} \tag{2.18}$$

Proof It follows from the proof of theorem 2.3 that the probability $M_{S,T}$ that the Markov chain ever transfers to any state $T \in \mathcal{S}^{\restriction c}$ under the condition of departing from any state $S \in \mathcal{S}$ is equal to $\mathbb{P}^*(\mathcal{X}_S^T)$, where $\mathcal{X}_S^T$ is the initial and final set of finite state sequences of lengths greater than 0 with initial state S and final state T, for which all intermediate states are irrelevant with respect to c. Set $\mathcal{X}_S^T$ can be written as the disjoint union of the set of state sequences in $\mathcal{X}_S^T$ of length 1, which is singleton set $\{(S,T)\}$, with the set of state sequences in $\mathcal{X}_S^T$ of lengths greater than 1. So,

$$\mathcal{X}_S^T = \{(S,T)\} \cup \bigcup_{Q \in \mathcal{S} \setminus \mathcal{S}^{\dagger c}} \{(S,Q)\} \circ \mathcal{X}_Q^T$$

and hence

$$\mathbb{P}^*(\mathcal{X}_S^T) = \mathcal{P}_{S,T} + \sum_{Q \in \mathcal{S} \setminus \mathcal{S}^{\dagger c}} \mathcal{P}_{S,Q} \cdot \mathbb{P}^*(\mathcal{X}_Q^T)$$

which completes the proof. Remark that for a fixed state $T \in \mathcal{S}^{\dagger c}$, the probabilities $\mathbb{P}^*(\mathcal{X}_S^T)$ do not depend on the probabilities $\mathbb{P}^*(\mathcal{X}_S^Q)$ with $Q \neq T$ for all $S \in \mathcal{S}$. ■

Theorem 2.9 states that the transition probabilities of the reduced Markov chain are a solution of the system of linear equations (2.18). However, (2.18) has in general no unique solution due to the recursive definition of these linear equations. The following theorem states that in case the conditions of the reduction theorem are satisfied, then the system of linear equation (2.18) has a unique *bounded* solution (see appendix A for a definition).

Theorem 2.10 *Let conditional reward function c be defined for an ergodic Markov chain with state space $\mathcal{S}$ and transition matrix $\mathcal{P}$. If this Markov chain has a recurrent state that is relevant with respect to c, then the system of linear equations (2.18) has a unique bounded solution.*

Proof By theorem 2.9 there exists a solution for the system of linear equations (2.18). This solution is bounded (by 1). To prove that this bounded solution is unique, fix a state $T \in \mathcal{S}^{\dagger c}$ and assume for any fixed enumeration of the states in $\mathcal{S}$ that the sequence $B_{S_1,T}, B_{S_2,T}, \ldots$ is a bounded solution of the system of linear equations (2.18). It is first remarked that the countable sum in each of the linear equations

$$B_{S,T} = \mathcal{P}_{S,T} + \sum_{Q \in \mathcal{S} \setminus \mathcal{S}^{\dagger c}} \mathcal{P}_{S,Q} \cdot B_{Q,T} \tag{2.19}$$

converges absolutely for all $S \in \mathcal{S}$. Consequently, the countable sum in (2.19) converges and the order in which the terms of the sum are added is irrelevant.

It has to be shown that $B_{S,T} = \mathbb{P}^*(\mathcal{X}_S^T)$ for all $S \in \mathcal{S}$, where $\mathcal{X}_S^T$ is the initial and final set of finite state sequences of lengths greater than 0 with initial state S and final state T, for which all intermediate states are irrelevant with respect to c (see proof of theorem 2.9). To this end, define $U_{S,T}^n$ and $V_{S,T}^n$ for all $n \geq 1$ and $S \in \mathcal{S}$ as

$$U_{S,T}^n = \begin{cases} \mathcal{P}_{S,T} & \text{if } n = 1 \\ \displaystyle\sum_{Q \in \mathcal{S} \setminus \mathcal{S}^{\dagger c}} \mathcal{P}_{S,Q} \cdot U_{Q,T}^{n-1} & \text{otherwise} \end{cases}$$

$$V_{S,T}^n = \begin{cases} \displaystyle\sum_{Q \in \mathcal{S} \setminus \mathcal{S}^{\dagger c}} \mathcal{P}_{S,Q} \cdot B_{Q,T} & \text{if } n = 1 \\ \displaystyle\sum_{Q \in \mathcal{S} \setminus \mathcal{S}^{\dagger c}} \mathcal{P}_{S,Q} \cdot V_{Q,T}^{n-1} & \text{otherwise} \end{cases}$$

Intuitively, $U^n_{S,T}$ denotes the conditional probability of the set of all finite state sequences in $\mathcal{X}^T_S$ of length n. $V^n_{S,T}$ indicates the conditional probability of the set of all finite state sequences in $\mathcal{X}^T_S$ of lengths greater than n. Remark that both $U^n_{S,T}$ and $V^n_{S,T}$ are well defined because the terms $U^{n-1}_{Q,T}$ and $V^{n-1}_{Q,T}$ respectively in their definitions above are bounded by 1 for all $n > 1$.

It is first proven by induction on n that the right hand side of (2.19) can be written as

$$\left(\sum_{i=1}^{n} U^i_{S,T}\right) + V^n_{S,T} \tag{2.20}$$

for all $n \geq 1$. For $n = 1$, equation (2.20) reduces to

$$U^1_{S,T} + V^1_{S,T} = \mathcal{P}_{S,T} + \sum_{Q \in \mathcal{S} \setminus \mathcal{S}^{\uparrow c}} \mathcal{P}_{S,Q} \cdot B_{Q,T}$$

which equals $B_{S,T}$ by (2.19). Now assume that $B_{S,T}$ equals (2.20) for some $n \geq 1$. Then for $n+1$, it follows that

$$\begin{aligned}
\sum_{i=1}^{n+1} U^i_{S,T} + V^{n+1}_{S,T} &= U^1_{S,T} + \sum_{i=1}^{n} U^{i+1}_{S,T} + \sum_{Q \in \mathcal{S} \setminus \mathcal{S}^{\uparrow c}} \mathcal{P}_{S,Q} \cdot V^n_{Q,T} \\
&= \mathcal{P}_{S,T} + \sum_{i=1}^{n} \left(\sum_{Q \in \mathcal{S} \setminus \mathcal{S}^{\uparrow c}} \mathcal{P}_{S,Q} \cdot U^i_{Q,T} \right) + \sum_{Q \in \mathcal{S} \setminus \mathcal{S}^{\uparrow c}} \mathcal{P}_{S,Q} \cdot V^n_{Q,T} \\
&= \mathcal{P}_{S,T} + \sum_{Q \in \mathcal{S} \setminus \mathcal{S}^{\uparrow c}} \mathcal{P}_{S,Q} \cdot \sum_{i=1}^{n} U^i_{Q,T} + \sum_{Q \in \mathcal{S} \setminus \mathcal{S}^{\uparrow c}} \mathcal{P}_{S,Q} \cdot V^n_{Q,T} \\
&= \mathcal{P}_{S,T} + \sum_{Q \in \mathcal{S} \setminus \mathcal{S}^{\uparrow c}} \mathcal{P}_{S,Q} \cdot \left(\sum_{i=1}^{n} U^i_{Q,T} + V^n_{Q,T} \right) \\
&= \mathcal{P}_{S,T} + \sum_{Q \in \mathcal{S} \setminus \mathcal{S}^{\uparrow c}} \mathcal{P}_{S,Q} \cdot B_{Q,T}
\end{aligned}$$

which again equals $B_{S,T}$ by (2.19). This completes the inductive proof and hence,

$$B_{S,T} = \lim_{n \to \infty} \left(\sum_{i=1}^{n} U^i_{S,T} \right) + V^n_{S,T} \tag{2.21}$$

To prove that (2.19) is a unique bounded solution for the system of linear equations (2.18), it will be shown that the right hand side of equation (2.21) equals $\mathbb{P}^*(\mathcal{X}^T_S)$ for all $S \in \mathcal{S}$. Observe first that for any $n \geq 1$ and $S \in \mathcal{S}$, $U^n_{S,T}$ is the conditional probability of the set $\{\underline{S} \in \mathcal{X}^T_S \mid |\underline{S}| = n\}$. Hence,

$$\lim_{n \to \infty} \left(\sum_{i=1}^{n} U^i_{S,T} \right) = \mathbb{P}^*\Big(\bigcup_{i=1}^{\infty} \{\underline{S} \in \mathcal{X}^T_S \mid |\underline{S}| = i\}\Big) = \mathbb{P}^*(\mathcal{X}^T_S)$$

and thus, it follows from (2.21) that

$$B_{S,T} = \mathbb{P}^*(\mathcal{X}^T_S) + \lim_{n \to \infty} V^n_{S,T}$$

As a result, it remains to be shown that

$$\lim_{n\to\infty} V^n_{S,T} = 0 \tag{2.22}$$

To proof (2.22), define W^n_S for all $n \geq 1$ and $S \in \mathcal{S}$ as

$$W^n_S = \begin{cases} \displaystyle\sum_{Q\in\mathcal{S}\setminus\mathcal{S}^{\restriction c}} \mathcal{P}_{S,Q} & \text{if } n = 1 \\ \displaystyle\sum_{Q\in\mathcal{S}\setminus\mathcal{S}^{\restriction c}} \mathcal{P}_{S,Q} \cdot W^{n-1}_Q & \text{otherwise} \end{cases}$$

Remark that W^n_S is well defined for each $n \geq 1$. It can be shown by induction on n (and using the boundedness of (2.19)) that $-bW^n_S \leq V^n_{S,T} \leq bW^n_S$ for all $n \geq 1$. Hence, to prove (2.22), it suffices (by the squeeze law) to show that

$$\lim_{n\to\infty} W^n_S = 0$$

Observe that W^n_S equals, for all $n \geq 1$ and $S \in \mathcal{S}$, the conditional probability of the set $\mathcal{W}^n_S$ of infinite state sequences $\underline{S}$ for which the initial state is S and $\underline{S}_i \in \mathcal{S} \setminus \mathcal{S}^{\restriction c}$ for $i = 2, 3, \ldots, n+1$. Since $\mathcal{W}^1_S \supseteq \mathcal{W}^2_S \supseteq \ldots$ is a decreasing sequence of events,

$$\lim_{n\to\infty} \mathbb{P}^*(\mathcal{W}^n_S) = \mathbb{P}^*(\bigcap_{i=1}^{\infty} \mathcal{W}^i_S)$$

The countable intersection of sets $\mathcal{W}^n_S$ is the set of all infinite state sequences for which the initial state is S and all other states are irrelevant with respect to c. In case the conditional probability on the countable union of the sets $\mathcal{W}^n_S$ would be greater than 0, then the probability that some relevant recurrent state S_r is reached from S would be strictly smaller than 1. However, this contradicts the assumption that the Markov chain has such a relevant recurrent state S_r, which is reachable from any other state with probability 1. Hence,

$$\mathbb{P}^*(\bigcap_{i=1}^{\infty} \mathcal{W}^i_S) = 0$$

and therefore (2.22) holds. This completes the proof of theorem 2.10. ■

Notice that theorem 2.10 implies that (2.18) has a unique solution in case the state space is finite. The application of theorem 2.9 for computing conditional long-run sample averages using the reduction technique is illustrated with example 2.6.

Example 2.6 *Recall the Markov chain of example 2.4 on page 32. Applying theorem 2.9 gives that matrix M equals*

$$M = \begin{pmatrix} M_{A,B} & M_{A,D} & M_{A,E} & M_{A,G} \\ M_{B,B} & M_{B,D} & M_{B,E} & M_{B,G} \\ M_{C,B} & M_{C,D} & M_{C,E} & M_{C,G} \\ M_{D,B} & M_{D,D} & M_{D,E} & M_{D,G} \\ M_{E,B} & M_{E,D} & M_{E,E} & M_{E,G} \\ M_{F,B} & M_{F,D} & M_{F,E} & M_{F,G} \\ M_{G,B} & M_{G,D} & M_{G,E} & M_{G,G} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & \frac{1}{2} & \frac{5}{12} & \frac{1}{12} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & \frac{1}{6} & \frac{5}{6} \end{pmatrix}$$

where for instance the computation of $M_{C,D}$ follows from solving $M_{C,D} = \mathcal{P}_{C,D} + \mathcal{P}_{C,C} \cdot M_{C,D} = \frac{1}{4} + \frac{3}{4} \cdot M_{C,D}$. Other examples of equations that need to be solved for obtaining matrix M are $M_{D,G} = \mathcal{P}_{D,G} + \mathcal{P}_{D,F} \cdot M_{F,G} = \frac{1}{6} \cdot M_{F,G}$ and $M_{G,G} = \mathcal{P}_{G,G} + \mathcal{P}_{G,F} \cdot M_{F,G} = \frac{2}{3} + \frac{1}{3} \cdot M_{F,G}$. The transition matrix $\mathcal{P}^{\upharpoonright c}$ of the reduced Markov chain is derived from matrix M by using that $\mathcal{P}^{\upharpoonright c}_{S,T} = M_{S,T}$ for any $S, T \in \mathcal{S}^{\upharpoonright c}$. Omitting the indication of initial probabilities, figure 2.4 depicts a graphical representation of this reduced Markov chain.

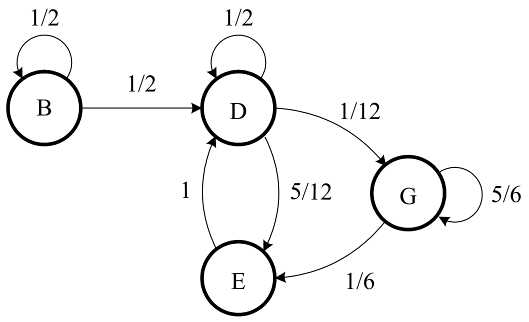


Figure 2.4: Reduced Markov chain corresponding to the Markov chain in figure 2.3.

After computing the equilibrium distribution of the reduced Markov chain using transition matrix $\mathcal{P}^{\upharpoonright c}$, the conditional long-run sample average of r can be computed by applying the ergodic theorem, similarly as in example 2.5.

Although the initial distribution of the reduced Markov chain is not needed for computing conditional long-run sample averages, constructing the reduced Markov chain requires deriving it. The following theorem enables computing the initial distribution of the reduced Markov chain based on the initial distribution of the original Markov chain and the transition matrix of the reduced Markov chain.

Theorem 2.11 *Let conditional reward function c be defined for an ergodic Markov chain represented by $(\mathcal{S}, I, \mathcal{P})$. If this Markov chain has a recurrent state that is relevant with respect to c, then the initial distribution $I^{\upharpoonright c}$ of the reduced Markov chain is given by*

$$I^{\upharpoonright c}_T = I_T + \sum_{Q \in \mathcal{S} \setminus \mathcal{S}^{\upharpoonright c}} I_Q \cdot M_{Q,T} \quad \textit{for all } T \in \mathcal{S}^{\upharpoonright c}$$

Proof From the proof of theorem 2.3, it follows that the probability $I^{\upharpoonright c}_T$ that the reduced Markov chain initially resides in state $T \in \mathcal{S}^{\upharpoonright c}$ equals $\mathbb{P}(\mathcal{X}^T)$, where $\mathcal{X}^T$ is the final set of finite state sequences with final state T and for which all preceding states are irrelevant with respect to c. The result follows from writing $\mathcal{X}^T$ as

$$\mathcal{X}^T = \{(T)\} \cup \bigcup_{Q \in \mathcal{S} \setminus \mathcal{S}^{\upharpoonright c}} \{(Q)\} \circ \mathcal{X}^T_Q$$

where $\mathcal{X}^T_Q$ is the initial and final set of finite state sequences of lengths greater than 0 with initial state Q and final state T and for which all intermediate states are irrelevant with respect to conditional reward function c. ■

Example 2.7 *Let the initial distribution of the Markov chain in example 2.4 on page 32 be defined as $I_A = \frac{3}{4}$, $I_C = \frac{1}{4}$ and $I_S = 0$ for $S \in \{B, D, E, F, G\}$. After deriving matrix M as shown in example 2.6, the initial distribution of the reduced Markov chain can be computed as $I_B^{\restriction c} = I_B + I_A \cdot M_{A,B} = \frac{3}{4}$, $I_D^{\restriction c} = I_D + I_C \cdot M_{C,D} = \frac{1}{4}$, $I_E^{\restriction c} = I_E + I_F \cdot M_{F,E} = 0$ and $I_G^{\restriction c} = I_G + I_F \cdot M_{F,G} = 0$.*

Combining the reduction technique with the construction method allows analytical computation of conditional long-run sample averages in such a way that only reward values earned in states for which the condition holds must be taken into account. By deriving the equilibrium distribution of the reduced Markov chain after constructing it, a system of $|\mathcal{S}^{\restriction c}| + 1$ (which is commonly much smaller than $|\mathcal{S}| + 1$) lineair equations has to be solved. Nevertheless, by the remark to the proof of theorem 2.9, constructing the reduced Markov chain requires solving $|\mathcal{S}^{\restriction c}|$ independent systems of $|\mathcal{S}|$ linear equations for deriving its transition probabilities from those of the original Markov chain. As a consequence, there is in general no improvement in performance evaluation speed when applying this approach compared to directly computing conditional long-run sample averages by means of theorem 2.1.

In some cases, an improvement in performance evaluation speed can be obtained[7]. For example 2.4, applying theorem 2.1 requires solving a system of linear equations to determine the equilibrium distribution of the original Markov chain. This system includes five (non-trivial) recursively defined equations (namely, $\pi_C = \mathcal{P}_{E,C} \cdot \pi_E + \mathcal{P}_{C,C} \cdot \pi_C$, $\pi_D = \mathcal{P}_{C,D} \cdot \pi_C + \mathcal{P}_{D,D} \cdot \pi_D$, $\pi_E = \mathcal{P}_{D,E} \cdot \pi_D + \mathcal{P}_{F,E} \cdot \pi_F$, $\pi_F = \mathcal{P}_{D,F} \cdot \pi_D + \mathcal{P}_{G,F} \cdot \pi_G$ and $\pi_G = \mathcal{P}_{F,G} \cdot \pi_F + \mathcal{P}_{G,G} \cdot \pi_G$). On the other hand, deriving the transition probabilities of the corresponding reduced Markov chain as shown in example 2.6 involves solving only a single (trivial) recursively defined equation (namely, $M_{C,D} = \mathcal{P}_{C,D} + \mathcal{P}_{C,C} \cdot M_{C,D}$). As a result, using the reduction technique with the construction method is less complex than applying theorem 2.1 in this case. From this example, it can be concluded that the degree in which the linear equations are entangled seems to affect the profit of using the reduction technique.

2.2.4 Simulation-Based Estimation

Consider an ergodic Markov chain $\{X_i \mid i \geq 1\}$ on probability space $(\Omega, \mathcal{F}, \mathbb{P})$ with state space $\mathcal{S}$. Let a conditional reward function c be defined and let $S_r \in \mathcal{S}$ be a recurrent state that is relevant with respect to c. Furthermore, let a proper reward function r be defined such that the conditional long-run sample average of r exists and equals μ_{cs}. By the reduction technique, μ_{cs} is equal to the accompanying long-run sample average $\mu_s^{\restriction c}$ defined for the reduced Markov chain $\{X_i^{\restriction c} \mid i \geq 1\}$. Hence, estimation of μ_{cs} can be accomplished using the classical point estimator for $\mu_s^{\restriction c}$. To examine this approach, define for all $k \geq 1$, the random variables ${Y_{S_r}^k}^{\restriction c}$ on $(\Omega, \mathcal{F}, \mathbb{P})$ as the sum of reward values $r(X_i^{\restriction c})$ earned during the kth cycle of the reduced Markov chain through S_r and let the random variables ${L_{S_r}^k}^{\restriction c}$ on $(\Omega, \mathcal{F}, \mathbb{P})$

[7]Hence, using the reduction technique may enable computing conditional long-run sample averages for Markov chains with larger state spaces than what would be feasible in case of applying theorem 2.1.

denote the length of that cycle. Then, the classical point estimator $\hat{\mu}_s^{\restriction c}$ given by

$$\hat{\mu}_s^{\restriction c} = \frac{Y_{S_r}^{\restriction c}(n)}{L_{S_r}^{\restriction c}(n)} \tag{2.23}$$

where

$$Y_{S_r}^{\restriction c}(n) = \frac{1}{n}\sum_{k=1}^{n} Y_{S_r}^{k\ \restriction c} \quad \text{and} \quad L_{S_r}^{\restriction c}(n) = \frac{1}{n}\sum_{k=1}^{n} L_{S_r}^{k\ \restriction c}$$

can be used to estimate μ_{cs}. To analyse the accuracy of a point estimate $\bar{\mu}_{cs}$, Confidence Interval

$$\left[\hat{\mu}_s^{\restriction c} - \frac{\kappa \hat{\tau}^{\restriction c}}{\sqrt{n}} \cdot \frac{1}{L_{S_r}^{\restriction c}(n)}, \hat{\mu}_s^{\restriction c} + \frac{\kappa \hat{\tau}^{\restriction c}}{\sqrt{n}} \cdot \frac{1}{L_{S_r}^{\restriction c}(n)}\right] \tag{2.24}$$

where

$$(\hat{\tau}^{\restriction c})^2 = \frac{1}{n-1}\sum_{k=1}^{n} (Y_{S_r}^{k\ \restriction c} - \hat{\mu}_s^{\restriction c} \cdot L_{S_r}^{k\ \restriction c})^2$$

can be used for obtaining a $2\Re(\kappa) - 1$ confidence interval for μ_{cs}. The conditions for using (2.23) and (2.24) require that $\mathbb{E}[|Y_{S_r}^{\restriction c}|^2] < \infty$ and $\mathbb{E}[|L_{S_r}^{\restriction c}|^2] < \infty$.

Notice that point estimator (2.23) and Confidence Interval (2.24) are expressed in random variables defined for the reduced Markov chain. Straightforwardly applying these estimators requires simulation of the *reduced* Markov chain, which would be impractical if it had to be constructed explicitly in order to do so. Since the reduced Markov chain is defined on the same probability space as the original Markov chain, μ_{cs} can also be estimated based on simulation of $\{X_i \mid i \geq 1\}$ by expressing (2.23) and (2.24) in random variables defined for the original Markov chain.

Observe that the sum of reward values $r(X_i^{\restriction c})$ earned during the k^{th} cycle of the reduced Markov chain through S_r is equal to the sum of reward values $r(X_i) \cdot c(X_i)$ earned during the k^{th} cycle of the original Markov chain through S_r. Similarly, the length of the k^{th} cycle of the reduced Markov chain through S_r equals the sum of reward values $c(X_i)$ earned during the k^{th} cycle of the original Markov chain through S_r. Using $t_{S_r}^k$ to denote the time-epoch of the k^{th} visit of the original Markov chain to S_r, it follows that

$$Y_{S_r}^{k\ \restriction c} = \sum_{i=t_{S_r}^k}^{t_{S_r}^{k+1}-1} r(X_i) \cdot c(X_i) \quad \text{and} \quad L_{S_r}^{k\ \restriction c} = \sum_{i=t_{S_r}^k}^{t_{S_r}^{k+1}-1} c(X_i) \tag{2.25}$$

Equations (2.25) can easily be evaluated during simulation of $\{X_i \mid i \geq 1\}$ by performing an addition with $r(X_i)$ respectively 1 *only* in case the condition holds.

To estimate μ_{cs} by simulation of the original Markov chain, it has to be investigated what conditions have to be imposed on the original Markov chain in order to satisfy $\mathbb{E}[|Y_{S_r}^{\restriction c}|^2] < \infty$ and $\mathbb{E}[|L_{S_r}^{\restriction c}|^2] < \infty$. The following theorem states that these conditions hold in case reward function r is proper for the original Markov chain.

Theorem 2.12 *Let reward function r and conditional reward function c be defined for an ergodic Markov chain $\{X_i \mid i \geq 1\}$. If r is proper and $\{X_i \mid i \geq 1\}$ has a recurrent state S_r that is relevant with respect to c, then*

$$\mathbb{E}[|Y_{S_r}^{\restriction c}|^2] < \infty \quad \textit{and} \quad \mathbb{E}[|L_{S_r}^{\restriction c}|^2] < \infty \tag{2.26}$$

Proof It will first be shown that $\mathbb{E}[|Y_{S_r}^{\restriction c}|^2] < \infty$. Because reward function r is proper, $\mathbb{E}[|Y_{S_r}|^2] < \infty$ by definition 2.8. This means that

$$\sum_{\underline{S} \in \mathcal{Z}_{S_r}^{S_r}} \mathbb{P}^*(\underline{S}) \cdot |\sum_{i=1}^{|\underline{S}|} r(\underline{S}_i)|^2 < \infty$$

Using (2.25), it follows for $\mathbb{E}[|Y_{S_r}^{\restriction c}|^2]$ that

$$\mathbb{E}[|Y_{S_r}^{\restriction c}|^2] = \sum_{\underline{S} \in \mathcal{Z}_{S_r}^{S_r}} \mathbb{P}^*(\underline{S}) \cdot |\sum_{i=1}^{|\underline{S}|} r(\underline{S}_i) \cdot c(\underline{S}_i)|^2 \leq \sum_{\underline{S} \in \mathcal{Z}_{S_r}^{S_r}} \mathbb{P}^*(\underline{S}) \cdot |\sum_{i=1}^{|\underline{S}|} r(\underline{S}_i)|^2 < \infty$$

For proving that $\mathbb{E}[|L_{S_r}^{\restriction c}|^2] < \infty$, observe that since reward function r is proper, $\mathbb{E}[|L_{S_r}|^2] < \infty$ by definition 2.8. This means that

$$\sum_{\underline{S} \in \mathcal{Z}_{S_r}^{S_r}} |\underline{S}|^2 \cdot \mathbb{P}^*(\underline{S}) < \infty$$

Now, let $|\underline{S}|^{\restriction c}$ denote the number of states in the finite state sequence $\underline{S} \in \mathcal{Z}_{S_r}^{S_r}$ that is relevant with respect to c minus 1. Notice that

$$|\underline{S}|^{\restriction c} = \sum_{i=1}^{|\underline{S}|} c(\underline{S}_i)$$

and hence

$$\mathbb{E}[|L_{S_r}^{\restriction c}|^2] = \sum_{\underline{S} \in \mathcal{Z}_{S_r}^{S_r}} \mathbb{P}^*(\underline{S}) \cdot |\sum_{i=1}^{|\underline{S}|} c(\underline{S}_i)|^2 \leq \sum_{\underline{S} \in \mathcal{Z}_{S_r}^{S_r}} |\underline{S}|^2 \cdot \mathbb{P}^*(\underline{S}) < \infty \qquad \blacksquare$$

From this proof, it can be concluded that (2.26) also holds if $r \cdot c$ is proper. Observe furthermore that (2.26) also holds when requiring the stronger conditions $\mathbb{E}[|U_{S_r}|^2] < \infty$ and $\mathbb{E}[|L_{S_r}|^2] < \infty$ for the original Markov chain (see also section 2.1.3).

Based on the reduction theorem and equations (2.25), the conditional long-run sample average μ_{cs} equals the expected sum of reward values obtained in relevant states visited during a cycle of the original Markov chain through S_r divided by the expected number of relevant states visited during such a cycle. By theorem 2.12, this quotient of expectations can be estimated with point estimator (2.23) and Confidence Interval (2.24) in case the conditions of the reduction theorem are satisfied. With (2.25), the estimation can be based on simulation of the original Markov chain without the necessity to take irrelevant states into account. Only in case the condition holds, intermediate estimation results for (2.23) and (2.24) have to be updated.

Without utilising the reduction technique, estimation of μ_{cs} by simulation of the original Markov chain can be based on a point estimator and Confidence Interval for the quotient of expectations mentioned above. However, deriving an unbiased point estimator and suitable Confidence Interval for a quotient of expectations is rather difficult (see also section 2.1.3). Another approach is to estimate the numerator and denominator separately and derive the estimation result for μ_{cs} by dividing the obtained point estimates for the numerator and denominator. For this approach, the algebra of Confidence Intervals presented in section 2.3 can be used to enable analysing the accuracy of the estimation result for μ_{cs}. However, compared with applying the reduction technique, this approach yields a smaller confidence level of the obtained confidence intervals. Another disadvantage of both these approaches is that they require to update intermediate estimation results in *all* states visited during simulation of the original Markov chain. Because the number of relevant states is commonly much smaller than the total number of states, performance evaluation speed may improve considerably when applying the reduction technique. Hence, a larger portion of all possible behaviours can be simulated in the same amount of simulation time as would be feasible if the reduction technique is not applied. Applying the reduction technique therefore gives a more accurate estimation of μ_{cs}.

2.2.5 Related Research

Several other approaches for improving the applicability of Markov-chain based analysis techniques have been proposed. Many focus on reducing the complexity of applying classical analysis techniques, just as the reduction technique does.

Analytical Computation An interesting technique is the algorithm proposed independently in [66] and [158] for computing equilibrium distributions. It constructs a sequence of stochastic matrices, each concerning the transition matrix of a Markov chain with a state space that is one state smaller than that of its predecessor. Basically, the technique is a variant of Gaussian elimination for solving a system of linear equations [75]. In [120], another variant of Gaussian elimination was proposed, which presumes the transition matrix to have certain properties that are believed to be common in the context of performance analysis of telecommunication networks. Although the reduction technique requires to compute an equilibrium distribution, this computation is not accelerated. The techniques of [66, 158] or [120] can however be combined with the reduction technique to accelerate computation of conditional long-run sample averages compared to when using standard Gaussian elimination.

Other approaches for reducing the state space propose techniques for lumping states into aggregate states [34, 162]. In [35] and [16] for example, states are lumped in accordance with the use of symmetrical or equal components in the system under investigation. States can also be lumped based on the reward values assigned to them, see for example [18] and [76]. The idea is that properties like egodicity and long-run averages are preserved such that the long-run averages can be computed based on the lumped Markov chain with reduced state space. Although reducing the state space as well, the reduction technique does not take the reward values themselves into account. Lumpability of Markov chains can however be combined with the reduction technique to accelerate computation of conditional long-run averages.

Instead of assuming particular properties for the Markov chain or reward values, the reduction technique is intended for evaluating a certain form of long-run average performance metrics. It enables evaluating conditional long-run sample averages by means of applying the classical performance analysis techniques on a Markov chain with reduced state space. The transition matrix of the reduced Markov chain specifies the probabilities of visiting a relevant state when starting from a relevant state but without intermediate visits to relevant states, see section 2.2.3. These transition probabilities can also be expressed with taboo probabilities. The general form of taboo probabilities is explained in [39], where they are in fact used to describe the classical technique for analytical computation of long-run sample averages. Opposed to [39], this chapter uses state sequences with certain properties for that purpose. Such state sequences are used for developing the reduction technique as well, which is useful in the context of simulation-based estimation of conditional long-run sample averages.

Simulation-Based Estimation Several approaches have been proposed to accelerate simulation-based performance estimation. Many focus on improving the statistical efficiency of the classical analysis techniques by means of reducing the variance of the point estimator for a performance metric, without disturbing its expected value [101, 27]. Such variance reduction techniques allow to obtain more accurate results (smaller confidence intervals) for the same simulation time or, alternatively, achieve a predefined accuracy within less simulation time [105]. However, the suitability of a variance reduction technique and its potential effect on the performance evaluation speed usually depends on the system under investigation. The variance reduction techniques of stratified sampling [101] and importance sampling [40, 61] require for example to give reward values certain weights and involve more computational intensive updates of intermediate estimation results. Although more generally applicable variance reduction techniques exist (for example the one of common random numbers [105]), they all require some knowledge about the behaviour of the system. Opposed to variance reduction techniques, the reduction technique allows a considerable improvement in performance evaluation time without any knowledge about the system's behaviour based on reducing the amount of computer time needed for updating intermediate estimation results for conditional long-run sample averages.

2.3 Accuracy Analysis of Complex Long-Run Averages

The reduction technique presented in the preceding section provides a means for both analytical computation and simulation-based estimation of conditional long-run sample averages. However, many long-run average performance metrics for industrial hardware/software systems are more complex.

Example 2.8 *Consider a telecommunication system that sends packets over some communication medium after processing them. The (fixed) bandwidth B of the communication medium indicates the maximum number of packets that can be communicated per time unit. Of interest is the expected utilisation (throughput) of the communication medium. When formalising the telecommunication system with a Markov chain, a reward function r can be defined, which assigns the current utilisation of the communication medium (either* 0 *if it is* *not used or B if it is in use) to each state of the Markov chain. Observe that the condition of taking reward values for r into account is a change in using the communication medium. The expected utilisation is however not equal to the long-run sample average of r (which equals $\frac{1}{2}B$) because the duration of each occurring utilisation is not taken into account.*

Section 2.3.1 identifies various forms of complex conditional long-run averages that are common for industrial hardware/software systems. These performance metrics are algebraic combinations of several conditional long-run sample averages. In case all conditions are satisfied, such a performance metric can be evaluated (either by analytical computation or by simulation-based estimation) with the reduction technique by separately determining the constituent conditional long-run sample averages and then combining the obtained results. However, in case of simulation-based estimation, it is unclear how the accuracy of a point estimate for a complex conditional long-run average obtained by combining point estimates for the constituent conditional long-run sample averages, could be analysed. Section 2.3.2 discusses a technique for analysing the accuracy of point estimates of complex conditional long-run averages. It is based on a number of operations defined on Confidence Intervals for estimating conditional long-run sample averages. Combined with the reduction technique, this algebra of Confidence Intervals, which was initially introduced in [173], allows analysing the accuracy of estimated complex conditional long-run averages using the classical performance analysis technique of section 2.1.3.

2.3.1 Common Complex Long-Run Averages

Conditional Long-Run Sample Variances Let proper reward function r and conditional reward function c be defined for an ergodic Markov chain $\{X_i \mid i \geq 1\}$. If this Markov chain has a recurrent state that is relevant with respect to c, then the conditional long-run sample average μ_{cs} of r (with condition c) exists. Now, the *conditional long-run sample variance* of r (with condition c) is defined as

$$\lim_{n\to\infty} \frac{\displaystyle\sum_{i=1}^{n} (r(X_i) - \mu_{cs})^2 \cdot c(X_i)}{\displaystyle\sum_{i=1}^{n} c(X_i)} \tag{2.27}$$

which can be rewritten[8] as

$$\lim_{n\to\infty} \frac{\displaystyle\sum_{i=1}^{n} r(X_i)^2 \cdot c(X_i)}{\displaystyle\sum_{i=1}^{n} c(X_i)} - \mu_{cs}^2 \tag{2.28}$$

by using the definition of conditional long-run sample average (2.14). As a result, performance metric (2.27) is equal to the conditional long-run sample average of r^2 minus the square of the conditional long-run sample average of r. Evaluating (2.27) with the reduction technique therefore requires r^2 to be proper.

[8] Notice that (2.28) matches the classical way in which a variance can be rewritten (see appendix A).

Example 2.9 *Recall the Markov chain of example 2.4 on page 32. Assume analytical computation of the conditional long-run sample variance of r. Using the results of example 2.5 on page 38, application of the reduction technique (which is allowed since r^2 is proper) gives that the conditional long-run sample average of r^2 equals $\pi_B^{\restriction c} \cdot r^2(B) + \pi_D^{\restriction c} \cdot r^2(D) + \pi_E^{\restriction c} \cdot r^2(E) + \pi_G^{\restriction c} \cdot r^2(G) = \frac{7}{2}$. Because the conditional long-run sample average of r is 1 (see example 2.5), the conditional long-run sample variance of r equals $\frac{5}{2}$.*

Example 2.10 *Consider a video system that processes video frames and the time needed for processing a video frame depends on its content. Of interest are the expected processing time as well as the variance in processing time. When formalising the behaviour of the video system with a Markov chain, a reward function r can be defined, which assigns the processing time of the last processed video frame to each state of the Markov chain. The expected processing time can be expressed as the conditional long-run sample average of r, while the variance in processing time can be formalised as the conditional long-run sample variance of r. For both these long-run average performance metrics, the condition for taking reward values obtained with r into account is the termination of processing of a video frame.*

Conditional Long-Run Time Averages An important class of complex conditional long-run averages concerns conditional long-run time averages. Let reward function r and conditional reward function c be defined for an ergodic Markov chain $\{X_i \mid i \geq 1\}$ with state space $\mathcal{S}$. Define reward function $\top : \mathcal{S} \to [0, \infty)$ such that $\top(S)$ is the progress in model time[9] in state $S \in \mathcal{S}$. Furthermore, define for any time-epoch $i \geq 1$, the reward function Δ as

$$\Delta(X_i) = \begin{cases} 0 & \text{if } c(X_j) = 0 \text{ for all } j = 1, \ldots, i \\ \top(X_i) & \text{if } c(X_i) = 1 \\ \displaystyle\sum_{k=j}^{i} \top(X_k) & \text{if } c(X_j) = 1 \text{ for } j < i \text{ and} \\ & \quad c(X_k) = 0 \text{ for all } k = j+1, \ldots, i \end{cases} \tag{2.29}$$

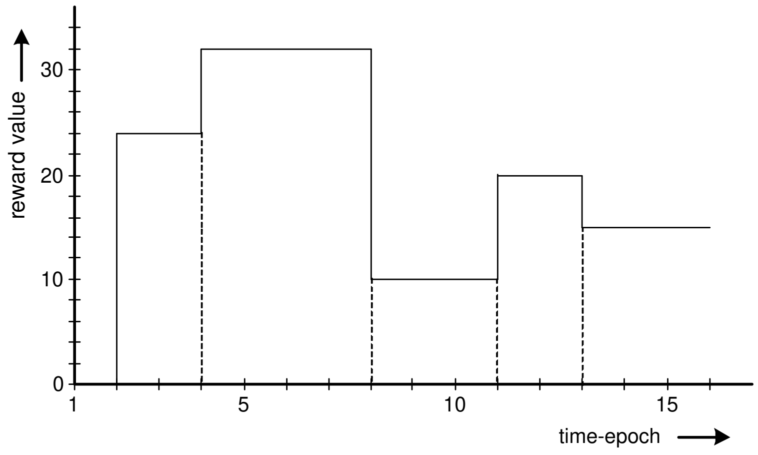
The definition of reward function Δ is illustrated in figure 2.5. Δ equals 0 until c evaluates to 1 for the first time. For any time-epoch at which c evaluates to 1, Δ is equal to the current progress in model time. In all other cases, Δ equals the total progress in model time since the last time-epoch at which c evaluated to 1.

Now, the *conditional long-run time average* of r (with respect to Δ and condition c) is defined as

$$\lim_{n \to \infty} \frac{\displaystyle\sum_{i=2}^{n} r(X_{i-1}) \cdot \Delta(X_{i-1}) \cdot c(X_i)}{\displaystyle\sum_{i=2}^{n} \Delta(X_{i-1}) \cdot c(X_i)} \tag{2.30}$$

Performance metric (2.30) can be explained as the long-run average of r weighted over the duration of each occurring reward value. Observe that the expected utilisation (throughput) of the communication medium in example 2.8 is of this form.

[9]The model time is an instance in the time domain of a model (presumed to be the non-negative real numbers) and may differ from the time domain of the underlying Markov chain, see also section 3.1.2.



i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$r(X_i)$	0	22	22	32	32	32	32	10	10	10	20	20	15	15	15
$c(X_i)$	0	1	0	1	0	1	1	1	0	0	1	0	1	1	0
$\top(X_i)$	2	0	1	3	2	0	0	0	2	0	0	2	3	4	0
$\Delta(X_i)$	0	0	1	3	5	0	0	0	2	2	0	2	3	4	4

Figure 2.5: Illustration of reward function Δ.

To enable rewriting (2.30) as a combination of conditional long-run sample averages, a *previous-reward operator* Θ is introduced similarly as in [180]. For any reward function r and Markov chain $\{X_i \mid i \geq 1\}$, previous-reward operator Θ is defined as

$$\Theta(r(X_i)) = \begin{cases} 0 & \text{if } i = 1 \\ r(X_{i-1}) & \text{otherwise} \end{cases}$$

With this notation, performance metric (2.30) can be rewritten as

$$\lim_{n\to\infty} \frac{\sum_{i=1}^{n} \Theta(r(X_i)) \cdot \Theta(\Delta(X_i)) \cdot c(X_i)}{\sum_{i=1}^{n} \Theta(\Delta(X_i)) \cdot c(X_i)} \tag{2.31}$$

which is the conditional long-run sample average of $\Theta(r) \cdot \Theta(\Delta)$ or $\Theta(r \cdot \Delta)$ divided by the conditional long-run sample average of $\Theta(\Delta)$. Remark that only requiring $\{X_i \mid i \geq 1\}$ to have a recurrent state that is relevant with respect to c and that both $\Theta(r \cdot \Delta)$ and $\Theta(\Delta)$ are proper is in general not sufficient for evaluating the conditional long-run time average of r. This is because, for any reward function r, the evaluation of $\Theta(r)$ in a state $S \in \mathcal{S}$ depends on the reward value of r in a state $S' \in \mathcal{S}$ that is visited before S. The problem is that S may have more than one such preceding state with different values for r. In [180], a technique is introduced for evaluating $\Theta(r)$ based on an extended Markov chain that does not suffer from this problem. Each state of this extended Markov chain encodes the previous state from which it was entered. This allows in fact defining Θ as a reward function on the extended Markov chain. In case certain mild conditions for $\{X_i \mid i \geq 1\}$ are satisfied, the technique for deriving the extended Markov chain preserves ergodicity and long-run averages [180]. Hence, applying the reduction technique after applying the technique of [180] allows evaluating the constituent conditional long-run sample averages of (2.30). To compute (2.30) analytically, corollary 2.7 can then be used. It is remarked that the condition for applying corollary 2.7 (there must exist a time-epoch i for which $\Theta(\Delta(X_i)) \cdot c(X_i) > 0$) is commonly satisfied for models that include timing aspects.

Conditional Long-Run Time Variances Let reward function r and conditional reward function c be defined for an ergodic Markov chain $\{X_i \mid i \geq 1\}$. Furthermore, let reward function Δ be defined as in equation (2.29). Assuming that all conditions for applying the technique presented in [180], the reduction technique and corollary 2.7 are satisfied, the conditional long-run time average μ_{ct} of r exists. Now, the *conditional long-run time variance* (with respect to Δ and condition c) is defined as

$$\lim_{n\to\infty} \frac{\sum_{i=2}^{n} (r(X_{i-1}) - \mu_{ct})^2 \cdot \Delta(X_{i-1}) \cdot c(X_i)}{\sum_{i=2}^{n} \Delta(X_{i-1}) \cdot c(X_i)} \tag{2.32}$$

which can be rewritten as

$$\lim_{n\to\infty} \frac{\sum_{i=2}^{n} r(X_{i-1})^2 \cdot \Delta(X_{i-1}) \cdot c(X_i)}{\sum_{i=2}^{n} \Delta(X_{i-1}) \cdot c(X_i)} - \mu_{ct}^2 \tag{2.33}$$

by using the definition of conditional long-run time average (2.30). Hence, performance metric (2.32) is equal to the conditional long-run time average of r^2 minus the square of the conditional long-run time average of r (both with respect to Δ). To evaluate (2.32) using the reduction technique, it is required that $\Theta(r^2 \cdot \Delta)$ is proper.

Example 2.11 *The video system of example 2.9 may include some buffer for queueing video frames before processing them. To evaluate the expected buffer occupancy and the variance in buffer occupancy, the duration of each occurring buffer occupancy must be taken into account. When formalising the behaviour of the video system with a Markov chain, a reward function r can be defined, which assigns the current buffer occupancy to each state of the Markov chain. In addition, reward function Δ can be defined in accordance with equation (2.29), which assigns the duration of the current buffer occupancy to each state of the Markov chain. The expected buffer occupancy can then be formalised as the conditional long-run time average of r, while the variance in the buffer occupancy is the conditional long-run time variance of reward function r. For both these complex conditional long-run averages, the condition for taking reward values for r into account is a change in the buffer occupancy.*

2.3.2 Algebra of Confidence Intervals

Section 2.3.1 illustrated common complex conditional long-run averages that are composed of several conditional long-run sample averages. After separately estimating these constituent conditional long-run sample averages, a point estimate can easily be derived for the complex conditional long-run average. To enable analysing the accuracy of point estimates obtained in such a way, this section defines a number of operations on the set of Confidence Intervals (see appendix A). These operations allow combining the Confidence Intervals for the constituent conditional long-run sample averages similarly to the way in which point estimates are combined, resulting in a Confidence Interval for the complex conditional long-run average.

Unary Operators The negation, square and reciprocal operations are defined on the set of Confidence Intervals according to theorems 2.13, 2.14 and 2.15 respectively.

Theorem 2.13 (Negation) *If the stochastic interval $[\varphi_1, \varphi_2]$ is a γ Confidence Interval for μ, then the stochastic interval $[-\varphi_2, -\varphi_1]$ is a γ Confidence Interval for $-\mu$.*

Proof Let $(\Omega, \mathcal{F}, \mathbb{P})$ denote the probability space on which the random variables φ_1 and φ_2 are defined. For any $\omega \in \Omega$, it follows that $-\varphi_2(\omega) \leq -\mu \leq -\varphi_1(\omega)$ if and only if $\varphi_1(\omega) \leq \mu \leq \varphi_2(\omega)$. Hence,

$$\mathbb{P}(\{\omega \in \Omega \mid -\mu \in [-\varphi_2(\omega), -\varphi_1(\omega)]) = \mathbb{P}(\{\omega \in \Omega \mid \mu \in [\varphi_1(\omega), \varphi_2(\omega)]) \geq \gamma \qquad \blacksquare$$

Theorem 2.14 (Square) *If the stochastic interval $[\varphi_1, \varphi_2]$ is a γ Confidence Interval for μ, then the stochastic interval $[\psi_1, \psi_2]$ is a γ Confidence Interval for μ^2, where*

$$\psi_1, \psi_2 = \begin{cases} \varphi_1^2, \varphi_2^2 & \textit{if } \varphi_1 \geq 0 \\ 0, \max(\varphi_1^2, \varphi_2^2) & \textit{if } \varphi_1 < 0 \textit{ and } \varphi_2 \geq 0 \\ \varphi_2^2, \varphi_1^2 & \textit{if } \varphi_2 < 0 \end{cases}$$

Proof Let $(\Omega, \mathcal{F}, \mathbb{P})$ denote the probability space on which the random variables φ_1 and φ_2 are defined. For any $\omega \in \Omega$, one of the following three cases holds:

1. $\varphi_1(\omega) \geq 0$. Then, $\varphi_1^2(\omega) \leq \mu^2 \leq \varphi_2^2(\omega)$ if and only if $\varphi_1(\omega) \leq \mu \leq \varphi_2(\omega)$;
2. $\varphi_1(\omega) < 0$ and $\varphi_2(\omega) \geq 0$. Now, $0 \leq \mu^2 \leq \max(\varphi_1^2(\omega), \varphi_2^2(\omega))$ if $0 \leq |\mu| \leq \max(|\varphi_1(\omega)|, |\varphi_2(\omega)|)$, which holds if $\varphi_1(\omega) \leq \mu \leq \varphi_2(\omega)$;
3. $\varphi_2(\omega) < 0$. In this case, $\varphi_2^2(\omega) \leq \mu^2 \leq \varphi_1^2(\omega)$ if and only if $-\varphi_2(\omega) \leq -\mu \leq -\varphi_1(\omega)$, which holds if and only if $\varphi_1(\omega) \leq \mu \leq \varphi_2(\omega)$.

As a final result,

$$\mathbb{P}(\{\omega \in \Omega \mid \mu^2 \in [\psi_1(\omega), \psi_2(\omega)]\}) \geq \mathbb{P}(\{\omega \in \Omega \mid \mu \in [\varphi_1(\omega), \varphi_2(\omega)]\}) \geq \gamma \qquad \blacksquare$$

Theorem 2.15 (Reciprocal) *If the stochastic interval $[\varphi_1, \varphi_2]$ is a γ Confidence Interval for $\mu \neq 0$, then the stochastic interval $[\psi_1, \psi_2]$ is a γ Confidence Interval for $\frac{1}{\mu}$, where*

$$\psi_1, \psi_2 = \begin{cases} \frac{1}{\varphi_2}, \frac{1}{\varphi_1} & \textit{if } \varphi_1 > 0 \\ \frac{1}{\varphi_2}, \infty & \textit{if } \varphi_1 = 0 \textit{ and } \varphi_2 > 0 \\ -\infty, \infty & \textit{if } \varphi_1 < 0 \textit{ and } \varphi_2 > 0 \\ -\infty, \frac{1}{\varphi_1} & \textit{if } \varphi_1 < 0 \textit{ and } \varphi_2 = 0 \\ \frac{1}{\varphi_2}, \frac{1}{\varphi_1} & \textit{if } \varphi_2 < 0 \end{cases}$$

Proof Let $(\Omega, \mathcal{F}, \mathbb{P})$ denote the probability space on which the random variables φ_1 and φ_2 are defined. For any $\omega \in \Omega$, one of the following five cases holds:

1. $\varphi_1(\omega) > 0$. Then, $\frac{1}{\varphi_2(\omega)} \leq \frac{1}{\mu} \leq \frac{1}{\varphi_1(\omega)}$ if and only if $\varphi_1(\omega) \leq \mu \leq \varphi_2(\omega)$;
2. $\varphi_1(\omega) = 0$ and $\varphi_2(\omega) > 0$. Now, $\frac{1}{\varphi_2(\omega)} \leq \frac{1}{\mu} \leq \infty$ if $\varphi_1(\omega) \leq \mu \leq \varphi_2(\omega)$ by the extended rules of arithmetic for the division operation on $\bar{\mathbb{R}}$;
3. $\varphi_1(\omega) < 0$ and $\varphi_2(\omega) > 0$. Then, $-\infty \leq \frac{1}{\mu} \leq \infty$ if $\varphi_1(\omega) \leq \mu \leq \varphi_2(\omega)$ by the extended rules of arithmetic for the division operation on $\bar{\mathbb{R}}$;
4. $\varphi_1(\omega) < 0$ and $\varphi_2(\omega) = 0$. Now, $-\infty \leq \frac{1}{\mu} \leq \frac{1}{\varphi_1(\omega)}$ if $\varphi_1(\omega) \leq \mu \leq \varphi_2(\omega)$ by the extended rules of arithmetic for the division operation on $\bar{\mathbb{R}}$;
5. $\varphi_2(\omega) < 0$. In this case, $\frac{1}{\varphi_2(\omega)} \leq \frac{1}{\mu} \leq \frac{1}{\varphi_1(\omega)}$ if and only if $\varphi_1(\omega) \leq \mu \leq \varphi_2(\omega)$.

Hence,

$$\mathbb{P}(\{\omega \in \Omega \mid \mu^2 \in [\psi_1(\omega), \psi_2(\omega)]\}) \geq \mathbb{P}(\{\omega \in \Omega \mid \mu \in [\varphi_1(\omega), \varphi_2(\omega)]\}) \geq \gamma \qquad \blacksquare$$

Binary Operators The binary operations defined on the set of Confidence Intervals are the addition, subtraction, multiplication and division operations. The addition and subtraction operations are defined according to theorem 2.16 and corollary 2.17.

Theorem 2.16 (Addition) *If the stochastic interval $[\varphi_1, \varphi_2]$ is a γ Confidence Interval for μ and the stochastic interval $[\phi_1, \phi_2]$ is a δ Confidence Interval for λ, then the stochastic interval $[\varphi_1 + \phi_1, \varphi_2 + \phi_2]$ is a $(\gamma + \delta) \dot{-} 1$*[10] *Confidence Interval for $\mu + \lambda$.*

Proof Let $(\Omega, \mathcal{F}, \mathbb{P})$ denote the probability space on which the random variables φ_1, φ_2, ϕ_1 and ϕ_2 are defined. For any $\omega \in \Omega$, it follows that $\varphi_1(\omega) + \phi_1(\omega) \leq \mu + \lambda \leq \varphi_2(\omega) + \phi_2(\omega)$ if both $\varphi_1(\omega) \leq \mu \leq \varphi_2(\omega)$ and $\phi_1(\omega) \leq \lambda \leq \phi_2(\omega)$. As a result,

$$\mathbb{P}(\{\omega \in \Omega \mid \mu + \lambda \in [\varphi_1(\omega) + \phi_1(\omega), \varphi_2(\omega) + \phi_2(\omega)]\}) \geq \\ \mathbb{P}(\{\omega \in \Omega \mid \mu \in [\varphi_1(\omega), \varphi_2(\omega)]\} \cap \{\omega \in \Omega \mid \lambda \in [\phi_1(\omega), \phi_2(\omega)]\})$$

Hence, by lemma A.1,

$$\mathbb{P}(\{\omega \in \Omega \mid \mu + \lambda \in [\varphi_1(\omega) + \phi_1(\omega), \varphi_2(\omega) + \phi_2(\omega)]\}) \geq \\ \mathbb{P}(\{\omega \in \Omega \mid \mu \in [\varphi_1(\omega), \varphi_2(\omega)]\}) + \mathbb{P}(\{\omega \in \Omega \mid \lambda \in [\phi_1(\omega), \phi_2(\omega)]\}) - 1$$

which is greater or equal to $(\gamma + \delta) \dot{-} 1$ $\blacksquare$

Corollary 2.17 (Subtraction) *If the stochastic interval $[\varphi_1, \varphi_2]$ is a γ Confidence Interval for μ and the stochastic interval $[\phi_1, \phi_2]$ is a δ Confidence Interval for λ, then the stochastic interval $[\varphi_1 - \phi_2, \varphi_2 - \phi_1]$ is a $(\gamma + \delta) \dot{-} 1$ Confidence Interval for $\mu - \lambda$.*

[10] The monus operation allows taking into account the case that the confidence level of a Confidence Interval obtained after performing the addition operation would be less than 0. Notice that Confidence Intervals with confidence level 0 are not useful for deciding whether estimation results are credible.

Proof The result follows from adding $[\varphi_1, \varphi_2]$ with the negation of $[\phi_1, \phi_2]$. ■

Finally, the multiplication and division operations are defined in accordance with theorem 2.18 and corollary 2.19 respectively.

Theorem 2.18 (Multiplication) *If the stochastic interval $[\varphi_1, \varphi_2]$ is a γ Confidence Interval for μ and the stochastic interval $[\phi_1, \phi_2]$ is a δ Confidence Interval for λ, then the stochastic interval $[\psi_1, \psi_2]$ is a $(\gamma + \delta) \dot{-} 1$ Confidence Interval for $\mu \cdot \lambda$ where*

$$\psi_1, \psi_2 = \begin{cases} \varphi_1 \cdot \phi_1, \varphi_2 \cdot \phi_2 & \textit{if } \varphi_1 \geq 0 \textit{ and } \phi_1 \geq 0 \\ \varphi_2 \cdot \phi_1, \varphi_2 \cdot \phi_2 & \textit{if } \varphi_1 \geq 0, \phi_1 < 0 \textit{ and } \phi_2 \geq 0 \\ \varphi_2 \cdot \phi_1, \varphi_1 \cdot \phi_2 & \textit{if } \varphi_1 \geq 0 \textit{ and } \phi_2 < 0 \\ \varphi_1 \cdot \phi_2, \varphi_2 \cdot \phi_2 & \textit{if } \varphi_1 < 0, \varphi_2 \geq 0 \textit{ and } \phi_1 \geq 0 \\ \min(\varphi_1 \cdot \phi_2, \varphi_2 \cdot \phi_1), & \textit{if } \varphi_1 < 0, \varphi_2 \geq 0, \phi_1 < 0 \textit{ and } \phi_2 \geq 0 \\ \quad \max(\varphi_1 \cdot \phi_1, \varphi_2 \cdot \phi_2) & \\ \varphi_2 \cdot \phi_1, \varphi_1 \cdot \phi_1 & \textit{if } \varphi_1 < 0, \varphi_2 \geq 0 \textit{ and } \phi_2 < 0 \\ \varphi_1 \cdot \phi_2, \varphi_2 \cdot \phi_1 & \textit{if } \varphi_2 < 0 \textit{ and } \phi_1 \geq 0 \\ \varphi_1 \cdot \phi_2, \varphi_1 \cdot \phi_1 & \textit{if } \varphi_2 < 0, \phi_1 < 0 \textit{ and } \phi_2 \geq 0 \\ \varphi_2 \cdot \phi_2, \varphi_1 \cdot \phi_1 & \textit{if } \varphi_2 < 0 \textit{ and } \phi_2 < 0 \end{cases}$$

Proof Let $(\Omega, \mathcal{F}, \mathbb{P})$ denote the probability space on which the random variables φ_1, φ_2, ϕ_1 and ϕ_2 are defined. For any $\omega \in \Omega$, one of the following nine cases holds:

1. $\varphi_1(\omega) \geq 0$ and $\phi_1(\omega) \geq 0$. Then, $\varphi_1(\omega) \cdot \phi_1(\omega) \leq \mu \cdot \lambda \leq \varphi_2(\omega) \cdot \phi_2(\omega)$ if and only if both $\varphi_1(\omega) \leq \mu \leq \varphi_2(\omega)$ and $\phi_1(\omega) \leq \lambda \leq \phi_2(\omega)$;

2. $\varphi_1(\omega) \geq 0$, $\phi_1(\omega) < 0$ and $\phi_2(\omega) \geq 0$. Now, $\varphi_2(\omega) \cdot \phi_1(\omega) \leq \mu \cdot \lambda \leq \varphi_2(\omega) \cdot \phi_2(\omega)$ if both $\varphi_1(\omega) \leq \mu \leq \varphi_2(\omega)$ and $\phi_1(\omega) \leq \lambda \leq \phi_2(\omega)$;

3. $\varphi_1(\omega) \geq 0$ and $\phi_2(\omega) < 0$. In this case, $\varphi_2(\omega) \cdot \phi_1(\omega) \leq \mu \cdot \lambda \leq \varphi_1(\omega) \cdot \phi_2(\omega)$ if both $\varphi_1(\omega) \leq \mu \leq \varphi_2(\omega)$ and $\phi_1(\omega) \leq \lambda \leq \phi_2(\omega)$;

4. $\varphi_1(\omega) < 0$, $\varphi_2(\omega) \geq 0$ and $\phi_1(\omega) \geq 0$. If so, $\varphi_1(\omega) \cdot \phi_2(\omega) \leq \mu \cdot \lambda \leq \varphi_2(\omega) \cdot \phi_2(\omega)$ in case both $\varphi_1(\omega) \leq \mu \leq \varphi_2(\omega)$ and $\phi_1(\omega) \leq \lambda \leq \phi_2(\omega)$;

5. $\varphi_1(\omega) < 0$, $\varphi_2(\omega) \geq 0$, $\phi_1(\omega) < 0$ and $\phi_2(\omega) \geq 0$. Now, $\min(\varphi_1(\omega) \cdot \phi_2(\omega), \varphi_2(\omega) \cdot \phi_1(\omega)) \leq \mu \cdot \lambda \leq \max(\varphi_1(\omega) \cdot \phi_1(\omega), \varphi_2(\omega) \cdot \phi_2(\omega))$ if both $\varphi_1(\omega) \leq \mu \leq \varphi_2(\omega)$ and $\phi_1(\omega) \leq \lambda \leq \phi_2(\omega)$;

6. $\varphi_1(\omega) < 0$, $\varphi_2(\omega) \geq 0$ and $\phi_2(\omega) < 0$. In this case, $\varphi_2(\omega) \cdot \phi_1(\omega) \leq \mu \cdot \lambda \leq \varphi_1(\omega) \cdot \phi_1(\omega)$ if both $\varphi_1(\omega) \leq \mu \leq \varphi_2(\omega)$ and $\phi_1(\omega) \leq \lambda \leq \phi_2(\omega)$;

7. $\varphi_2(\omega) < 0$ and $\phi_1(\omega) \geq 0$. If so, $\varphi_1(\omega) \cdot \phi_2(\omega) \leq \mu \cdot \lambda \leq \varphi_2(\omega) \cdot \phi_1(\omega)$ in case both $\varphi_1(\omega) \leq \mu \leq \varphi_2(\omega)$ and $\phi_1(\omega) \leq \lambda \leq \phi_2(\omega)$;

8. $\varphi_2(\omega) < 0$, $\phi_1(\omega) < 0$ and $\phi_2(\omega) \geq 0$. Now, $\varphi_1(\omega) \cdot \phi_2(\omega) \leq \mu \cdot \lambda \leq \varphi_1(\omega) \cdot \phi_1(\omega)$ if both $\varphi_1(\omega) \leq \mu \leq \varphi_2(\omega)$ and $\phi_1(\omega) \leq \lambda \leq \phi_2(\omega)$;

9. $\varphi_2(\omega) < 0$ and $\phi_2(\omega) < 0$. In this case, $\varphi_2(\omega) \cdot \phi_2(\omega) \leq \mu \cdot \lambda \leq \varphi_1(\omega) \cdot \phi_1(\omega)$ if and only if both $\varphi_1(\omega) \leq \mu \leq \varphi_2(\omega)$ and $\phi_1(\omega) \leq \lambda \leq \phi_2(\omega)$.

As a result,

$$\begin{aligned}&\mathbb{P}(\{\omega \in \Omega \mid \mu \cdot \lambda \in [\psi_1(\omega), \psi_2(\omega)]\}) \geq \\ &\qquad\qquad \mathbb{P}(\{\omega \in \Omega \mid \mu \in [\varphi_1(\omega), \varphi_2(\omega)]\} \cap \{\omega \in \Omega \mid \lambda \in [\phi_1(\omega), \phi_2(\omega)]\})\end{aligned}$$

Hence, by lemma A.1,

$$\begin{aligned}&\mathbb{P}(\{\omega \in \Omega \mid \mu \cdot \lambda \in [\psi_1(\omega), \psi_2(\omega)]\}) \geq \\ &\qquad \mathbb{P}(\{\omega \in \Omega \mid \mu \in [\varphi_1(\omega), \varphi_2(\omega)]\}) + \mathbb{P}(\{\omega \in \Omega \mid \lambda \in [\phi_1(\omega), \phi_2(\omega)]\}) - 1\end{aligned}$$

which is greater or equal to $(\gamma + \delta) \dot{-} 1$ ■

The square operation is preferred over the multiplication operation if the operands for the multiplication operation are the same. The reason is that the square operation does not reduce the confidence level, whereas the multiplication operation does.

Corollary 2.19 (Division) *If the stochastic interval $[\varphi_1, \varphi_2]$ is a γ Confidence Interval for μ and the stochastic interval $[\phi_1, \phi_2]$ is a δ Confidence Interval for $\lambda \neq 0$, then the stochastic interval $[\psi_1, \psi_2]$ is a $(\gamma + \delta) \dot{-} 1$ Confidence Interval for μ/λ where*

$$\psi_1, \psi_2 = \begin{cases} \frac{\varphi_1}{\phi_2}, \frac{\varphi_2}{\phi_1} & \textit{if } \varphi_1 \geq 0 \textit{ and } \phi_1 > 0 \\ \frac{\varphi_2}{\phi_2}, \infty & \textit{if } \varphi_1 \geq 0, \phi_1 = 0 \textit{ and } \phi_2 > 0 \\ -\infty, \infty & \textit{if } \varphi_1 \geq 0, \phi_1 < 0 \textit{ and } \phi_2 > 0 \\ -\infty, \frac{\varphi_1}{\phi_1} & \textit{if } \varphi_1 \geq 0, \phi_1 < 0 \textit{ and } \phi_2 = 0 \\ \frac{\varphi_2}{\phi_2}, \frac{\varphi_1}{\phi_1} & \textit{if } \varphi_1 \geq 0 \textit{ and } \phi_2 < 0 \\ \frac{\varphi_1}{\phi_1}, \frac{\varphi_2}{\phi_1} & \textit{if } \varphi_1 < 0, \varphi_2 \geq 0 \textit{ and } \phi_1 > 0 \\ -\infty, \infty & \textit{if } \varphi_1 < 0, \varphi_2 \geq 0, \phi_1 = 0 \textit{ and } \phi_2 > 0 \\ -\infty, \infty & \textit{if } \varphi_1 < 0, \varphi_2 \geq 0, \phi_1 < 0 \textit{ and } \phi_2 > 0 \\ -\infty, \infty & \textit{if } \varphi_1 < 0, \varphi_2 \geq 0, \phi_1 < 0 \textit{ and } \phi_2 = 0 \\ \frac{\varphi_2}{\phi_2}, \frac{\varphi_1}{\phi_2} & \textit{if } \varphi_1 < 0, \varphi_2 \geq 0 \textit{ and } \phi_2 < 0 \\ \frac{\varphi_1}{\phi_1}, \frac{\varphi_2}{\phi_2} & \textit{if } \varphi_2 < 0 \textit{ and } \phi_1 > 0 \\ -\infty, \frac{\varphi_2}{\phi_2} & \textit{if } \varphi_2 < 0, \phi_1 = 0 \textit{ and } \phi_2 > 0 \\ -\infty, \infty & \textit{if } \varphi_2 < 0, \phi_1 < 0 \textit{ and } \phi_2 = 0 \\ \frac{\varphi_2}{\phi_1}, \infty & \textit{if } \varphi_2 < 0, \phi_1 < 0 \textit{ and } \phi_2 = 0 \\ \frac{\varphi_2}{\phi_1}, \frac{\varphi_1}{\phi_2} & \textit{if } \varphi_2 < 0 \textit{ and } \phi_2 < 0 \end{cases}$$

Proof Using the extended rules of arithmetic for the multiplication operation on $\bar{\mathbb{R}}$, the result follows from multiplying $[\varphi_1, \varphi_2]$ with the reciprocal of $[\phi_1, \phi_2]$. ■

The set of Confidence Intervals together with the negation, square, reciprocal, addition, subtraction, multiplication and division operations as defined above is in this thesis referred to as the *algebra of Confidence Intervals*.

Estimating Common Complex Long-Run Averages Let $[\varphi_1, \varphi_2]$ be a Confidence Interval for a conditional long-run sample average μ_{cs} defined according to equation (2.24) on page 45. An important property for this stochastic interval is that both $\varphi_1 \xrightarrow{\text{a.s.}} \mu_{cs}$ and $\varphi_2 \xrightarrow{\text{a.s.}} \mu_{cs}$ (see also section 2.1.3). Of interest is whether the algebra of Confidence Intervals preserves this property of the convergence of the bounds.

Consider for example the addition of Confidence Interval $[\varphi_1, \varphi_2]$ for μ with Confidence Interval $[\phi_1, \phi_2]$ for λ. It is easy to see that $\varphi_1+\phi_1 \xrightarrow{\text{a.s.}} \mu+\lambda$ and $\varphi_2+\phi_2 \xrightarrow{\text{a.s.}} \mu+\lambda$ and hence, the addition preserves the property of almost sure convergence of the bounds. Without further proof it is stated that the other operations also preserve this property. The crux of such a proof regarding for example the reciprocal of Confidence Interval $[\varphi_1, \varphi_2]$ for μ is that either the first or last case in theorem 2.15 will eventually hold for long traces. It is easy to show that for these cases, both the upper and lower bound of the resulting Confidence Interval converge to $\frac{1}{\mu}$ and hence, the reciprocal operation preserves the property of the convergence of the bounds.

Theorem 2.20 *The operations defined on the set of Confidence Intervals preserve almost sure convergence of the bounds of a Confidence Interval to the involved long-run average.*

Now, let Confidence Interval $[\varphi_1, \varphi_2]$ for some complex long-run average μ be determined based on the algebra of Confidence Intervals. Then, for any $x, y \in \mathbb{R}$ with $x + y \neq 0$, it follows that $\frac{1}{x+y}(x \cdot \varphi_1 + y \cdot \varphi_2) \xrightarrow{\text{a.s.}} \mu$ since $\varphi_1 \xrightarrow{\text{a.s.}} \mu$ and $\varphi_2 \xrightarrow{\text{a.s.}} \mu$ by theorem 2.20. Therefore, any $\frac{1}{x+y}(x\cdot\varphi_1+y\cdot\varphi_2)$ with $x+y \neq 0$ is a strongly consistent (but generally biased) point estimator for μ. Instead of deriving a point estimate for a complex long-run average by combining point estimates of the constituent conditional long-run sample averages, the result can be derived from a confidence interval obtained for the complex long-run average based on application of the algebra of Confidence Intervals. To minimise simulation time before automatically terminating it (see also section 2.1.3), the point estimator that yields the smallest upper bound for the relative error should be used. Notice that this is point estimator $\frac{1}{2}(\varphi_1 + \varphi_2)$.

2.3.3 Related Research

Accuracy analysis of long-run average performance metrics has mainly focussed on deriving an unbiased point estimator and accompanying Confidence Interval for a quotient of expectations. This is because the elementary performance metric (long-run sample average) is of that form, see equation (2.8) on page 26. Although different point estimators (like the Jacknife estimator [87, 105]) and Confidence Intervals (such as bootstrap Confidence Intervals [38]) have been proposed, these are only applicable for estimating a single long-run sample average. Notice that since a long-run sample average is a quotient of expectations that can be estimated separately, the algebra of Confidence Intervals could even be used for analysing the accuracy of long-run sample averages. Nevertheless, this approach would give a less precise Confidence Interval than the Confidence Interval derived in section 2.1.3. Compared to the other approaches, the algebra of Confidence Intervals is however more generally applicable in the sense that it also provides a means for analysing the accuracy of complex combinations of long-run sample averages, like long-run time variances.

Traditionally, estimating a complex long-run average performance metric involves deriving a point estimator and a Confidence Interval. Although this is often a very complicated task, a straightforwardly derived point estimator and Confidence Interval exists for some complex long-run averages. In [125] for example, estimation of the subtraction of two long-run sample averages and the division of two long-run sample variances are discussed as well as estimation of long-run sample variances. Point estimation of the first two performance metrics is accomplished by combining the point estimates of the constituent performance metrics, while long-run sample variances are estimated based on the standard sample variance. Accuracy analysis of point estimates obtained with these point estimators can be accomplished by using the Confidence Intervals of [125]. These Confidence Intervals are derived based on the distribution to which the combined point estimators of these complex long-run averages converge. Although being more precise than the Confidence Intervals that would be the result of using the algebra of Confidence Intervals, the Confidence Intervals of [125] are only applicable for the indicated performance metric types. The algebra of Confidence Intervals, on the other hand, provides a means for deriving a Confidence Interval for any complex long-run average performance metric.

2.4 Conclusions

The classical Markov-chain based performance analysis techniques provide ample means for evaluating long-run sample averages. This chapter extended these techniques to enable both analytical computation and simulation-based estimation of various common forms of more complex long-run average performance metrics.

With the reduction technique, long-run sample averages can be evaluated, for which a certain condition on the states must be taken into account. The reduction technique bases the evaluation of such a conditional long-run sample average on defining a reduced Markov chain that includes only those states for which the condition holds. The reduction technique matches conditional long-run sample averages for the original Markov chain with corresponding long-run sample averages defined for the reduced Markov chain. As a result, it allows evaluating conditional long-run sample averages based on applying the classical performance analysis techniques to the reduced Markov chain. Using the reduction technique requires that the original Markov chain is ergodic and that it has a recurrent state for which the condition holds. In addition, the reward function based on which reward values are obtained must be proper. These rather mild conditions are the same for both analytical computation and simulation-based estimation of conditional long-run sample averages.

In case of analytical computation of a conditional long-run sample average with the reduction technique, two approaches are identified for determining the equilibrium distribution of the reduced Markov chain. This equilibrium distribution is needed by the classical performance analysis technique to compute the corresponding long-run sample average defined for the reduced Markov chain. The first approach is based on the relation that exists between the equilibrium probabilities of the original and reduced Markov chains. Compared to directly computing a conditional long-run sample average based on the original Markov chain, this approach does not yield an improvement in performance evaluation speed. The second approach involves explicit construction of the reduced Markov chain, which is based on the relation between the transition and initial probabilities of the original and reduced Markov chains. Although applying this approach for computing a conditional long-run sample average does not improve performance evaluation speed in general, there are cases for which performance evaluation speed does improve.

The reduction technique enables estimating conditional long-run sample averages by means of simulating the original Markov chain. This approach is based on updating intermediate estimation results for a long-run sample average only in those states for which the condition holds. An important advantage of applying the reduction technique over straightforwardly estimating conditional long-run sample averages is avoidance of the difficulty of deriving a suitable point estimator and Confidence Interval for the quotient of expectations that a conditional long-run sample average is. Straightforwardly estimating a conditional long-run sample average with such a point estimator and Confidence Interval would also require to update intermediate estimation results in all states visited during simulation. Hence, performance evaluation speed improves considerably when applying the reduction technique and therefore more accurate results can be obtained in the same simulation time.

Next to conditional long-run sample averages, a number of more complex forms of long-run average performance metrics are identified. They concern algebraic combinations of conditional long-run sample averages. Analytical computation of such complex long-run averages can be accomplished by first determining the constituent conditional long-run sample averages with the reduction technique and then combining their results. A similar approach can be applied in case of simulation-based estimation. After obtaining a point estimate for the constituent conditional long-run sample averages using the reduction technique, a point estimate for the complex long-run average can be derived. The accuracy of such a point estimate can be analysed based on the proposed algebra of Confidence Intervals. It provides a number of operations on Confidence Intervals that allow combining confidence intervals obtained for the constituent conditional long-run sample averages to determine a confidence interval for the complex long-run average. Some of the operations reduce the confidence level for the resulting Confidence Interval. Hence, if a certain confidence level is desired for a complex long-run average, it is necessary to estimate the constituent conditional long-run sample averages with a greater confidence level.

Chapter 3

Reflexive Performance Analysis with POOSL

Using Markov chain-based techniques for evaluating the performance of a system requires formalising its behaviour in terms of states and transitions. However, the complexity of industrial hardware/software systems often hinders directly writing down their states and transitions due to the state-space explosion problem [126]. A more convenient way of specifying behaviour, which matches closely the way designers reason about the working of systems, is offered by modelling languages. A modelling language provides primitives for specifying how the complex behaviour of a system is composed of the simple behaviours of its components. To use Markov chain-based performance analysis techniques in this case, the behaviour specified in the model must be mapped onto the states and transitions of a Markov chain.

Next to formalising the behaviour of a system with a Markov chain, the performance properties of interest need to be formalised as reward functions. When applying the model-checking approach for evaluating performance properties, defining a reward function involves specifying a reward value for each state separately. If the number of states is large, this way of assigning reward values is impractical. For an infinite number of states, specifying reward values for all states is in general even impossible. With reflexive performance analysis on the other hand, reward functions can be defined by adding variables to a model. By extending the behaviour of the model, reward values can be assigned to such variables in accordance with the occurrence of events that affect the performance property. To apply Markov chain-based performance analysis techniques in this case, the introduced variables must be matched with reward functions that assign reward values to the states of a Markov chain.

An important advantage of modelling languages with a formal semantics is that models constructed with them implicitly define a certain mathematical structure. An example of such a formal modelling language is the Parallel Object-Oriented Specification Language (POOSL) [145, 28]. The semantics of a POOSL model defines a timed probabilistic labelled transition system [28] that can be transformed into a Markov chain [180]. After constructing the Markov chain, reward values can be assigned to the states separately and then the performance properties of interest can be evaluated. As illustrated in [180], this model-checking approach enables analytical computation of the system's performance using classical Markov-chain based performance analysis techniques. In section 3.1, the results of [180] are used to found a framework for reflexive performance analysis with POOSL. This framework allows estimation of the system's performance by executing (simulating) the POOSL model. Appendix B includes an introductory overview of the syntax and semantics of POOSL, which are presumed to be understood before reading this chapter.

Although the framework for reflexive performance analysis with POOSL provides a sound basis for obtaining credible performance results, a practical problem arises when applying the simulation-based estimation techniques discussed in chapter 2. It concerns the necessity of identifying a recurrent state. Visiting such a recurrent state marks the beginning of a regenerative cycle of independent behaviour. Determining whether a state visited during simulation is equal to a previously visited state if often prohibitively complex. Section 3.2 elaborates on two approaches for defining a condition to enforce the beginning of a regenerative cycle. These approaches allow applying the simulation-based estimation techniques in the form of performance monitors. The implementation of such performance monitors by means of library classes for POOSL is discussed in section 3.3. Section 3.4 investigates the quality of the accuracy results that are obtained when using these library classes.

3.1 Framework for Reflexive Performance Analysis

3.1.1 Specifying Markov Chains

As discussed in more detail in appendix B, the formal semantics of a POOSL model $\mathcal{M}$ defines a unique *timed probabilistic labelled transition system* of the form:

$$\mathcal{M} = (\mathcal{C}, C_s, \mathcal{A}, \{\xrightarrow{a} \subseteq \mathcal{C} \times \mathcal{D}(\mathcal{C}) \mid a \in \mathcal{A}\}, \mathcal{T}, \{\xrightarrow{t} \subseteq \mathcal{C} \times \mathcal{C} \mid t \in \mathcal{T}^+\}) \tag{3.1}$$

where $\mathcal{C}$ denotes the set of configurations with initial configuration $C_s \in \mathcal{C}$, $\mathcal{A}$ is the set of actions, $\mathcal{T}$ is the time domain and finally $\{\xrightarrow{a} \subseteq \mathcal{C} \times \mathcal{D}(\mathcal{C}) \mid a \in \mathcal{A}\}$ and $\{\xrightarrow{t} \subseteq \mathcal{C} \times \mathcal{C} \mid t \in \mathcal{T}^+\}$ denote the sets of *action transitions* and *time transitions* respectively. Transition system $\mathcal{M}$ can be visualised as a graph. The nodes of this graph represent the configurations of $\mathcal{M}$. The initial configuration C_s is identified with a symbol $>$ directed towards the node. For any action transition relation $C \xrightarrow{a} \boldsymbol{\pi}$ with $\boldsymbol{\pi} \in \mathcal{D}(\mathcal{C})$, a directed double (multi) arrow is drawn from node C to all nodes C' to which the system can transit with positive probability $\boldsymbol{\pi}(C')$. The first part of the double (multi) arrow is a directed arrow labelled with action a and the second part is a fan-out of directed arrows labelled with the probabilities $\boldsymbol{\pi}(C')$ [28]. For any time transition relation $C \xrightarrow{t} C'$, a directed arrow is drawn from node C to node C', which is labelled with t. Since time transitions denote the maximal amount of time that the model is willing to wait before continuing performing action transitions, t is either a (positive) number or ∞. Example 3.1 illustrates visualisation of the transition system corresponding to the behaviour specified by one process method.

Example 3.1 *Generating bursts of (fixed-size) packets can be specified with the process method* `GenerateTraffic` *in figure 3.1. In line* 1*, a local variable* `p` *of type* `Packet` *is declared. The actual behaviour specification of* `GenerateTraffic` *starts in line* 2 *with the creation of a new instance of data class* `Packet`*. After creating the new packet, its identifier is set to* `PacketNumber`*. In line* 3*,* `PacketNumber` *is incremented to ensure that the identifier of the generated packet is unique. The braces in lines* 2 *and* 3 *specify that the enclosed expressions are to be executed as one indivisible action. The send statement in line* 4 *specifies sending* `p` *through port* `Out`*. The duration of sending* `p`*, which equals the size* `PacketSize` *of a packet divided by the (fixed) rate* `Bandwidth` *with which packets are sent, is modelled with the* **`delay`***-statement in line* 5*. Line* 6 *models the probability of inserting a period without traffic after sending* `p`*. To this end, a sample is drawn from the Bernoulli distribution* `InsertIdleTime`*. If sending another packet must be postponed, the* **`delay`***-statement in line* 7 *is executed. The exact time to wait depends on a sample drawn from the discrete uniform distribution* `IdleTimeDistribution`*. Infinite repetition of sending a packet and inserting an idle period is modelled with tail-recursion in line* 9*.*

```
1  GenerateTraffic()() |p: Packet|

2  {p := new(Packet) withNumber(PacketNumber);
3     PacketNumber := PacketNumber + 1};
4  Out!Packet(p);
5  delay(PacketSize / Bandwidth);
6  if (InsertIdleTime yieldsSuccess) then
7     delay(IdleTimeDistribution sample)
8  fi;
9  GenerateTraffic()().
```

Figure 3.1: Generating bursts of packets.

When abstracting from the actual values of `p` *and* `PacketNumber`*, the timed probabilistic labelled transition system defined by method* `GenerateTraffic` *can be visualised as shown in figure 3.2. The initial configuration* C_1 *reflects the behaviour specified by the expressions of lines* 2 *and* 3*. The internal action transition from* C_1 *to* C_2 *originates from creating a new packet and incrementing* `PacketNumber` *in lines* 2 *and* 3 *together (because of the braces) and is performed with probability* 1*. The send statement in line* 4 *specifies the send action transition from configuration* C_2 *to* C_3*. Remark that send action* `Out!Packet` *denotes a synchronisation with the environment. If the environment is not prepared to participate in performing a matching receive action, execution of* `Out!Packet` *is postponed until the environment is ready to receive the packet. Hence, the send statement also specifies the time transition from* C_2 *to* C_2 *labelled with* ∞*, which denotes the willingness to wait for an arbitrary amount of time before performing send action* `Out!Packet`*.*

The time transition from configuration C_3 *to* C_4 *originates from the* **`delay`***-statement in line 5 of figure 3.1. The label* $\frac{7}{2}$ *is obtained by assuming that* `PacketSize` *equals* 140 *data units and the rate* `Bandwidth` *of sending packets is* 40 *data units per time unit. The* **`if`***-statement in line* 6 *results in an internal action with branches to configurations* C_5 *and* C_6*. Assuming that the Bernoulli distribution* `InsertIdleTime` *yields success with probability* $\frac{1}{8}$*, the system transits to* C_5 *if no idle period has to be inserted and to* C_6 *otherwise. From configuration* C_5*, the fix action transition to* C_1 *reflects the tail-recursive call of method* `GenerateTraffic`*. On the other hand, the part of the transition system in figure 3.2 regarding configurations* C_6 *to* C_{18} *is the result of the* **`delay`***-statement in line 7. This* **`delay`***-statement illustrates how probabilistic timing behaviour can be specified in POOSL. Its execution first involves evaluating the amount of time to wait [28]. Suppose that*

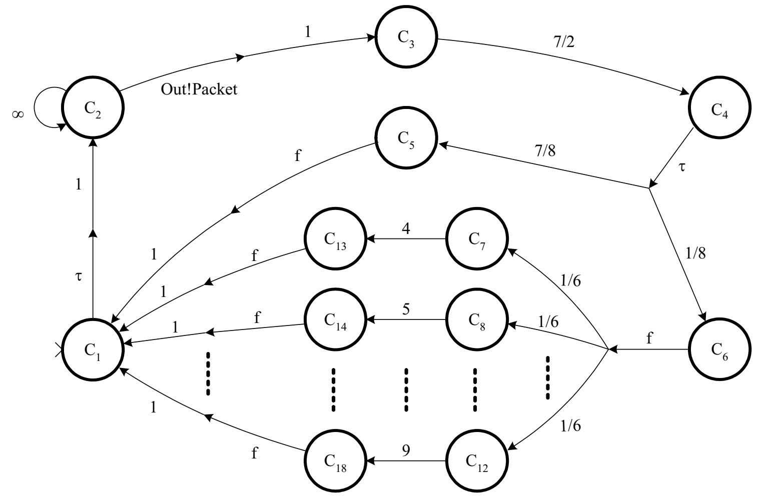


Figure 3.2: Visualisation of a timed probabilistic labelled transition system.

`IdleTimeDistribution` is a discrete uniform distribution assuming the values 4, 5, 6, 7, 8 and 9. Then, the resulting fix action has branches to 6 *different configurations, which are all entered with the same probability. From these configurations, the time can pass for an amount equal to the involved sample of `IdleTimeDistribution` and after that the system resides in the corresponding configuration C_{13} to C_{18}. Finally, the fix actions from these configurations to C_1 reflect the tail-recursive call of method `GenerateTraffic`.*

Composition of Transition Systems The transition system derived in example 3.1 reflects the behaviour of only a single activity. Based on using the **`par`**-statement, processes may however include a number of activities that operate concurrently. The behaviour of each concurrent activity can be studied in isolation by separately deriving its transition system (as in example 3.1). Such a transition system may express dependencies on behaviour of other activities within the process, originating from using guards on data objects shared with the other activities. Instead of studying the activities in isolation, the behaviour of the composition of all concurrent activities within a process can be studied as a whole. The composite transition system for the process is then defined in accordance with the semantical rules for the **`par`**-statement [28]. The configurations of this composite transition system are determined by the combination of the local configurations of the individual concurrent activities.

Similar to how processes may include a number of concurrent activities, a complete model or a cluster may include several concurrently operating processes. The behaviour of each process can be studied in isolation by deriving its transition system as explained above. The transition systems of the individual processes may express behavioural dependencies with other processes in the model or cluster due to the use of the statements for sending and receiving messages. Instead of studying the processes in isolation, the behaviour of the composition of all processes in a model or cluster can be studied. The composite transition system is then defined based on the semantical rules for parallel composition of processes, channel hiding and channel relabelling for POOSL [28], which are similar to those defined for CCS [122].

A transition system can be interpreted as being closed (self-contained) or open. Interpreting a transition system as *closed* excludes the possibility of interacting with the environment [57]. In this case, action transitions that depend on the environment are blocked, while all other action transitions are not blocked. Interpreting a transition system as *open* allows the possibility of interacting with the environment. When assuming that the environment is capable of participating in any such interactions at any time, no action transition is blocked. Interpreting for example the transition system of a single process as open allows assuming that its environment is always willing to participate in any send or receive action transition specified for it.

The tools for executing POOSL models (see also appendix B) interpret the transition system of a complete model as closed. As a consequence, analysing the performance of a complex hardware/software system in practice requires to include a model of both the system and the environment with which it interacts.

Transforming POOSL Models into Markov Chains The timed probabilistic labelled transition system defined by a POOSL model can be transformed into (time-homogeneous discrete-time) Markov chain in three steps [180] as follows:

Transformation Step 1: Assuming Maximal Progress

The transition system defined by a POOSL model may include configurations from which both communication action transitions and time transitions can be performed (like configuration C_2 in example 3.1). Since all the involved action transitions depend on the environment, interpreting the transition system as closed disables performing any of these blocked action transitions. On the other hand, when interpreting the transition system as open, an assumption must be made about the willingness of the environment to participate in the communication action transitions.

To study the behaviour of a concurrent activity or process in isolation and taking any interaction with its environment into account, it is practical to interpret the involved transition system as open and assume that the environment is always willing to participate in any communication action that the transition system wants to perform. This means that the activity or process will make *maximal progress*. Maximal progress implies the removal of the time transitions from configurations allowing both communication action transitions and time transitions. After applying the assumption of maximal progress, the remaining time transitions are interpreted slightly different. Instead of indicating the *maximal* amount of time that the model is willing to wait before continuing performing actions, time transitions now present the *exact* amount of time that must pass[1].

Transformation Step 2: Resolving Non-Determinism

A transition system may include configurations from which several (non-blocking) action transitions can be performed. Such non-determinism must be resolved in order to transform the transition system into a Markov chain. This is achieved by using an (external) *scheduler* [176] (also called rule or policy in the theory of Markov decision processes [45]) for determining the actual order in which the involved actions are performed. Notice that different schedulers may execute these actions in dif-

[1]It is noted that when interpreting a transition system as closed, time transitions also denote the exact amount of time that must pass. This common interpretation is necessary to actually execute models.

ferent orders. As a consequence, the actual sequence of visited configurations may differ for different schedulers and therefore different results could be obtained for a specific performance metric. Hence, in case non-determinism is not resolved, a performance metric gives rise to a collection of results. Only after resolving non-determinism, performance metrics are given by a single result.

When using a modelling language that supports non-determinism (see for example [23, 51, 71] and [91]), one is often not aware of the presence of non-determinism. Tools for executing models developed with such modelling language commonly resolve non-determinism implicitly, without knowing the exact policy for doing so. Although resolving non-determinism is a common aspect of discrete-event simulation, it is rarely recognised as a possible cause of unrealistic performance results.

As indicated in appendix B, the tools for executing POOSL models include a scheduler that resolves non-determinism explicitly based on a uniform distribution over the set of next possible action transitions. It is remarked that if no formal semantics would have been available, resolving non-determinism in such a fair way is difficult to guarantee. Future research includes an investigation on other approaches for resolving non-determinism and their effects on the obtained performance results. It is also possible to not resolve non-determinism. In this case, a POOSL model can be transformed into a Markov decision process. More research is needed for developing techniques that allow determining the complete collection of results for a performance metric when the behaviour is formalised as Markov decision process.

After resolving the non-determinism in a POOSL model, a new timed probabilistic labelled transition system is obtained. This transition system is of the form:

$$(\mathcal{C}, C_s, \mathcal{A}, \{\overset{a,p}{\Longrightarrow} \subseteq \mathcal{C} \times \mathcal{C} \mid a \in \mathcal{A}, p \in [0,1]\}, \mathcal{T}, \{\overset{t}{\Longrightarrow} \subseteq \mathcal{C} \times \mathcal{C} \mid t \in \mathcal{T}^{+}\}) \tag{3.2}$$

where $\mathcal{C}$, C_s, $\mathcal{A}$ and $\mathcal{T}$ denote again the set of configurations, the initial configuration, the set of actions and the time domain respectively. The sets $\{\overset{a,p}{\Longrightarrow} \subseteq \mathcal{C} \times \mathcal{C} \mid a \in \mathcal{A}, p \in [0,1]\}$ and $\{\overset{t}{\Longrightarrow} \subseteq \mathcal{C} \times \mathcal{C} \mid t \in \mathcal{T}^{+}\}$ indicate the action transitions and time transitions respectively *after* resolving non-determinism. When residing in configuration C, relation $C \overset{a,p}{\Longrightarrow} C'$ holds if configuration C' can be entered with probability p after performing action a. Observe that this implies that there is at most one action transition $\overset{a,p}{\Longrightarrow}$ leading from C to C'. The relations $\overset{a,p}{\Longrightarrow}$ are straightforwardly determined from the relations $\xrightarrow{a}$ of transition system (3.1) and the scheduler, see example 3.2.

Example 3.2 *The piece of POOSL code in figure 3.3 specifies the non-deterministic selection of three different behaviours. The corresponding (partial) transition system defined based on the semantics is depicted in the bottom left corner of figure 3.3. The* **`skip`***-statement in line 2 results in the internal action transition from A to B. Assuming that* `InsertIdleTime` *in line 4 is the same Bernoulli distribution as in example 3.1, the* **`if`***-statement defines an internal action transition from A with branches to configurations C and D. The method call in line 7 results in the fix action from A to E. After resolving non-determinism based on a uniform distribution over the next possible action transitions, the (partial) transition system depicted at the bottom right of figure 3.3 is obtained. Because three different actions can be performed when residing in A, the scheduler executes each of the corresponding actions with probability $\frac{1}{3}$. The resulting probability of residing in B, C, D or E after A therefore equals $\frac{1}{3}$ times the probability of performing the corresponding action.*

```
...
sel
   skip; ...
or
   if (InsertIdleTime yieldsSuccess) then ...
   else ... fi; ...
or
   Method()()
les;
...
```

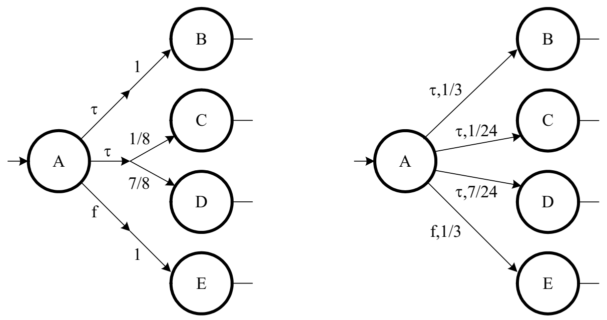


Figure 3.3: Resolving non-determinism in a probabilistic way.

When residing in configuration C, relation $C \overset{t}{\Longrightarrow} C'$ holds if configuration C' can be entered after waiting for the (positive) amount t of time. Because of assuming maximal progress, $C \overset{t}{\Longrightarrow} C'$ holds if $C \xrightarrow{t} C'$ and no action transitions can be performed from C. The latter is ensured by transformation step 1. Since time transitions are deterministic (see appendix B), they are considered to be taken with probability 1.

Transformation Step 3: Shifting Action and Time Information into States

The final transformation step boils down to shifting the action and time labels of the transitions into the nodes of the graph representing transition system (3.2). This is similar to the approach implicitly used in for example [129]. The result is a (time-homogeneous discrete-time) Markov chain, where information about the *actual* occurrence of actions and passage of time can be deduced from the states (from transition system (3.2), it can only be concluded that an action or time transition *can* occur). Hence, action and time information can be encoded as reward values, which is necessary for evaluating performance metrics depending on such information.

The state space $\mathcal{S}$ of the Markov chain obtained after shifting the action and time information is defined as a subset of the set $\mathcal{C} \times (\mathcal{A} \cup \mathcal{T}^+ \cup \{-\})$. In general, $\mathcal{C} \times (\mathcal{A} \cup \mathcal{T}^+ \cup \{-\})$ is uncountable due to the cardinality of time domain $\mathcal{T}$, which may be the non-negative real numbers. However, since $\mathcal{C}$ is countable, the sets $\{\overset{a,p}{\Longrightarrow} \subseteq \mathcal{C} \times \mathcal{C} \mid a \in \mathcal{A}, p \in [0,1]\}$ and $\{\overset{t}{\Longrightarrow} \subseteq \mathcal{C} \times \mathcal{C} \mid t \in \mathcal{T}^+\}$ are countable and hence, $\mathcal{S}$ is countable. If $|\mathcal{C}|$ is finite, then $|\mathcal{S}|$ equals the number of action and time transitions with different label/target-configuration combinations for transition system (3.2) plus 1.

The interpretation of a state in $\mathcal{S}$ of the form (C, a) with $a \in \mathcal{A}$ is that C is entered after having performed action a. A state (C, t) with $t \in \mathcal{T}^+$ is interpreted as entering C after having performed a time transition labelled with t, while $(C, -)$ denotes the entrance of C without having performed any action or time transition. Remark that only the initial configuration C_s of transition system (3.2) is entered without performing a transition. The initial distribution I of the obtained Markov chain is defined as

$$I(C,\alpha) = \begin{cases} 1 & \text{if } C = C_s \text{ and } \alpha = - \\ 0 & \text{otherwise} \end{cases}$$

Notice that the sum of all initial probabilities equals 1. For any $\alpha, \beta \in \mathcal{A} \cup \mathcal{T}^+ \cup \{-\}$, the probability $\mathcal{P}_{(C,\alpha),(C',\beta)}$ of transferring from state (C,α) to (C',β) is defined as

$$\mathcal{P}_{(C,\alpha),(C',\beta)} = \begin{cases} p & \text{if } \beta \in \mathcal{A} \text{ and } C \overset{\beta,p}{\Longrightarrow} C' \\ 1 & \text{if } \beta \in \mathcal{T}^+ \text{ and } C \overset{\beta}{\Longrightarrow} C' \\ 0 & \text{otherwise} \end{cases}$$

Because either action transitions or a time transition can be performed, it follows that for any $C \in \mathcal{C}$ and $\alpha \in \mathcal{A} \cup \mathcal{T}^+ \cup \{-\}$,

$$\sum_{C' \in \mathcal{C},\ \beta \in \mathcal{A} \cup \mathcal{T}^+ \cup \{-\}} \mathcal{P}_{(C,\alpha),(C',\beta)} = 1$$

Hence, the conditions for the existence of a Markov chain (see section 2.1) with state space $\mathcal{S}$, initial distribution I and transition matrix $\mathcal{P}$ are satisfied.

Example 3.3 *Consider transforming the transition system of example 3.1 into a Markov chain. By transformation step* 1*, the* ∞*-labelled time transition from* C_2 *to* C_2 *in figure 3.2 is removed. The resulting transition system does not include any non-determinism and hence, transformation step* 3 *can be performed immediately. When shifting the action and time information into the states, the Markov chain depicted in figure 3.4 is obtained. The state* $(C_1, -)$ *corresponds to the initial configuration* C_1 *before performing any action or time transitions. State* (C_1, f) *also originates from configuration* C_1*, but now indicates that a fix action was performed before entering it. State* $(C_3, \texttt{Out!Packet})$ *indicates entering* C_3 *in figure 3.2 after having performed the send action. And, as a final example,* $(C_{13}, 4)$ *originates from entering configuration* C_{13} *after the time passed for* 4 *time units.*

Let $\{X_i \mid i \geq 1\}$ denote the Markov chain that is obtained after transforming the transition system defined by the semantics of a POOSL model. An important issue for applying Markov-chain based performance analysis techniques is the question whether $\{X_i \mid i \geq 1\}$ is ergodic. Unfortunately, $\{X_i \mid i \geq 1\}$ is in general not ergodic. Examples of models that do not result in an ergodic Markov chain are those which include a non-deterministic selection between two infinitely repeating activities. This is because these models imply a state space where certain recurrent states cannot be reached from other recurrent states. Nevertheless, most models in practice do imply ergodic Markov chains [175]. More research is needed to identify the exact conditions for which POOSL models transform into ergodic Markov chains. In this thesis, it is assumed that the Markov chain defined based on a POOSL model of any realistic hardware/software system satisfies all conditions (including ergodicity) for applying the performance analysis techniques discussed in chapter 2.

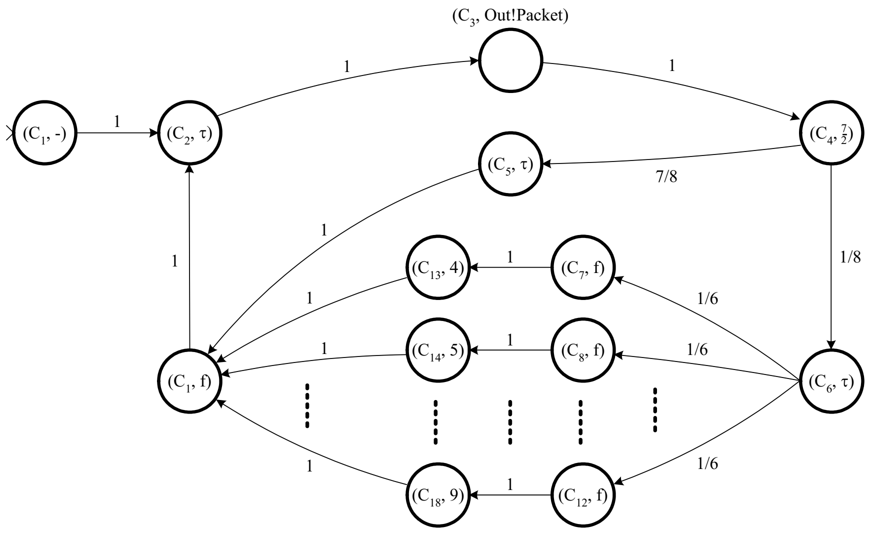


Figure 3.4: Markov chain corresponding to the transition system of figure 3.2.

After transforming a POOSL model into a Markov chain, reward functions can be defined separately to allow performance metrics to be computed analytically. This model-checking approach is exploited in [180]. Explicitly constructing the Markov chain can also be used for estimating performance metrics by simulating the Markov chain. Transforming a POOSL model into a Markov chain can however be prohibitively complex. An approach that is often more suitable for industrial hardware/software systems is *reflexive performance analysis*. It is based on extending a model with variables to define reward functions, see also section 3.1.2. When adding monitor behaviour that specifies the actual evaluation of performance metrics by performing operations on these variables, execution of the extended model allows estimation of performance metrics without the necessity to construct the Markov chain. Based on the mathematical relation between a POOSL model and a Markov chain, execution of the model corresponds to generating a trace from the underlying Markov chain. The number of action *and* time transitions execution of a POOSL model equals the current time-epoch of the underlying Markov chain minus 1.

Section 1.2.2 notes that extending a model with additional variables and behaviour to specify monitors results in polluting the representation of the system's behaviour, which may decrease the intuitiveness of the model. Notice that such extensions may however also affect the behaviour of the system itself if no precautions are taken. Section 4.3.2 elaborates on how POOSL models can be extended for reflexive performance analysis such that non-intrusive monitor behaviour is obtained and the intuitive understanding of the model is maintained.

Library Classes for Distributions The probabilistic interpretation of POOSL models originates from using data objects that are derived from instances of data class `RandomGenerator`. Such data objects are the basis for determining the transition probabilities of the underlying Markov chain. The precise value of these transition probabilities depends on the distribution denoted by the involved data object. The instance variables InsertIdleTime and IdleTimeDistribution in example 3.1 present for example a Bernoulli distribution and a discrete uniform distribution respectively. Data classes for such distributions can be derived from RandomGenerator in accordance with the approaches proposed in [105].

To assist the designer in constructing models that include probabilistic behaviour, a library of data classes for representing different types of distributions (Bernoulli, DiscreteUniform, Exponential, Normal and Uniform) has been developed. Figure 3.5 depicts a class diagram of this library in accordance with the UML profile for SHE, see section 4.2. Although the superclass Distribution can be used to create an instance of any of the distribution types included in the library (based on method ofType), such instances can also be created by using the corresponding library data class directly. After creating a distribution, method withParameter or withParameters initialises the parameter(s). Methods withParameter and withParameters call method initialise of the superclass Distribution to create an instance of data class RandomGenerator, which is used to derive random numbers for the distribution. To ensure that different sequences of random numbers are generated when having multiple instances of the same distribution type (see also

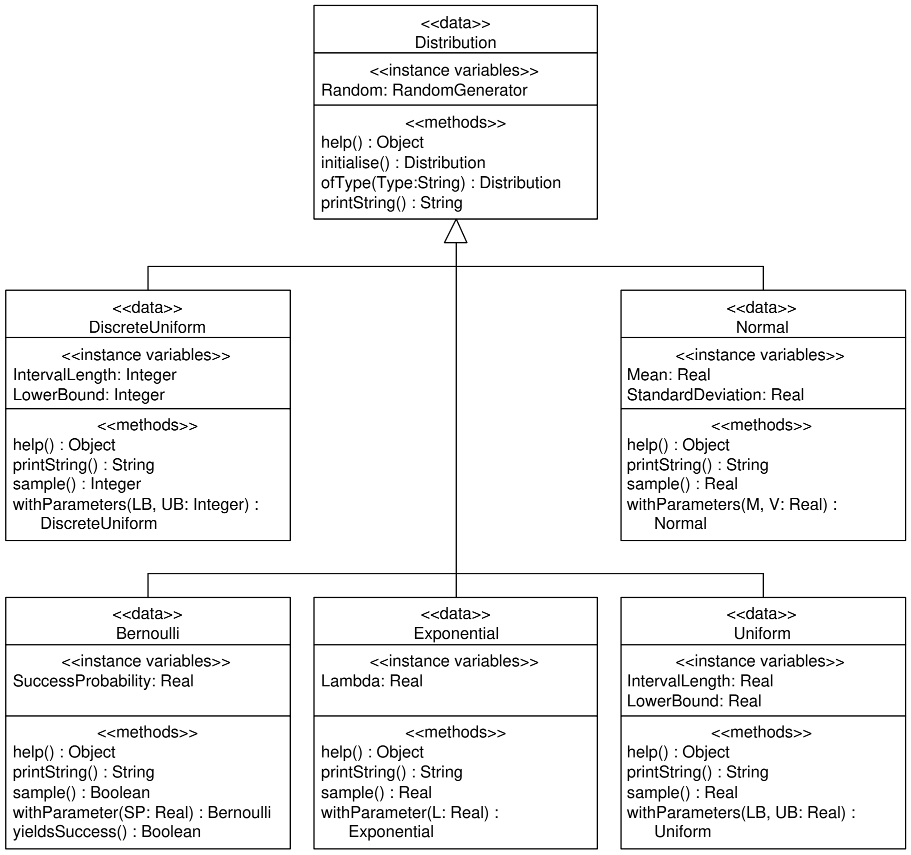


Figure 3.5: Library classes for distributions.

appendix B), the seeds of the different instances of `RandomGenerator` are randomised. For obtaining a random number that is in accordance with a certain distribution, method `sample` can be used[2]. The different `sample` methods actually implement the approaches proposed in [105]. More information about the use of the library classes is given in the `help` methods, which do not perform any operations. A method `printString` is available for each of the classes, which allows displaying information about the type and parameter(s) of a distribution when inspecting it with the SHESim tool (see also appendix B).

Example 3.4 *Assume that method `GenerateTraffic` in example 3.1 is of process class `Source`. The Bernoulli distribution `InsertIdleTime` and discrete uniform distribution `IdleTimeDistribution` are instance variables of class `Source`. These distributions can be initialised using the library of data classes by the initial method for `Source` as depicted in figure 3.6. In lines 2 - 3, a new instance of data class `Bernoulli` is created and the parameter of the Bernoulli distribution is set to $\frac{1}{8}$, after which it is assigned to variable `InsertIdleTime`. Lines 4 - 5 indirectly initialise `IdleTimeDistribution` as a discrete uniform distribution over interval $[4, 9]$ by using method `ofType` of superclass `Distribution`. After initialising `InsertIdleTime` and `IdleTimeDistribution`, generating bursts of packets starts by calling method `GenerateTraffic`.*

```
1  Init()()

2  InsertIdleTime := new(Bernoulli)
3    withParameter(1/8);
4  IdleTimeDistribution := new(Distribution)
5    ofType("DiscreteUniform") withParameters(4,9);
6  GenerateTraffic()().
```

Figure 3.6: Different ways of initialising distributions.

3.1.2 Defining Reward Functions

The framework for reflexive performance analysis with POOSL involves extending a model with additional variables to define reward functions for the implicitly defined Markov chain. Let a model $\mathcal{M}$ define the ergodic Markov chain $\{X_i \mid i \geq 1\}$ with state space $\mathcal{S} \subseteq \mathcal{C} \times (\mathcal{A} \cup \mathcal{T}^+ \cup \{-\})$, where $\mathcal{C}$ is the set of configurations, $\mathcal{A}$ the set of actions and $\mathcal{T}$ the time domain of $\mathcal{M}$. Reward functions for $\{X_i \mid i \geq 1\}$ are to be defined as complete functions from $\mathcal{S}$ to $\mathbb{R}$, see section 2.1.1. As a result of relation $\mathcal{S} \subseteq \mathcal{C} \times (\mathcal{A} \cup \mathcal{T}^+ \cup \{-\})$, reward functions may be defined in terms of the (contents of) configurations or in terms of the occurrence of action and time transitions.

Each configuration $(\mathcal{B}, \mathcal{I})$ in $\mathcal{C}$ includes the specification of *behaviour* $\mathcal{B}$ that is to be executed in the context of *information* $\mathcal{I}$, see also appendix B. $\mathcal{I}$ captures the *variables context* of model $\mathcal{M}$ and the values that are assigned to the variables. When adding a variable `R` to $\mathcal{M}$, the variables context is extended with `R` and hence, the information included in all configurations is extended. Since $\mathcal{S} \subseteq \mathcal{C} \times (\mathcal{A} \cup \mathcal{T}^+ \cup \{-\})$, a complete function can be defined on $\mathcal{S}$, which retrieves the values of `R` from the variables

[2]For data class `Bernoulli`, method `yieldsSuccess` is equivalent to method `sample`.

context and assigns them as reward values to the states. In case variable `R` assumes values in $\mathbb{R}$, such a function is an implicitly defined reward function for $\{X_i \mid i \geq 1\}$.

Recall that applying the Markov-chain based performance analysis techniques discussed in chapter 2 requires reward functions to be proper. It is reasonable to assume that this is indeed the case for reward functions implicitly defined by adding variables to a model based on the remarks succeeding definition 2.8 on page 29 and the consideration that in practice models never involve assignments of ∞ to variables.

Although adding a variable `R` to model $\mathcal{M}$ implies a reward function r for $\{X_i \mid i \geq 1\}$, specification of the exact reward values that are to be assigned to the states in $\mathcal{S}$ requires explicitly assigning elements of $\mathbb{R}$ to `R`. Commonly, such an assignment is specified based on the occurrence of a certain event that affects the performance metric. Instead of defining long-run average performance metrics as (combinations of) long-run sample averages, it is useful to define them as (combinations of) *conditional* long-run sample averages, where the condition is an explicit assignment to any of the variables. This approach allows to estimate long-run average performance metrics without the necessity to take the corresponding reward values into account for all states, see section 2.2.4. The semantics of assignments (which are expressions) imply (internal or fix) action transitions [28]. To take the condition of an explicit assignment to `R` into account, a conditional reward function can be defined, which equals 1 if action transitions are performed that include an explicit assignment to `R` and 0 otherwise. It is assumed that for each reward function defined by adding a variable, such an accompanying conditional reward function is defined *implicitly*.

Special Reward Functions Section 2.3.1 introduced reward functions $\top$ and Δ to enable expressing conditional long-run time averages and variances. Let a model $\mathcal{M}$ define the ergodic Markov chain $\{X_i \mid i \geq 1\}$ with state space $\mathcal{S} \subseteq \mathcal{C} \times (\mathcal{A} \cup \mathcal{T}^+ \cup \{-\})$, where $\mathcal{C}$, $\mathcal{A}$ and $\mathcal{T}$ are the set of configurations, the set of actions and the time domain for $\mathcal{M}$ respectively. Now, reward function $\top$, which assigns the progress in model time to each state of $\{X_i \mid i \geq 1\}$, is *implicitly* defined as

$$\top(C, \alpha) = \begin{cases} \alpha & \text{if } \alpha \in \mathcal{T}^+ \\ 0 & \text{otherwise} \end{cases}$$

for all $C \in \mathcal{C}$ and $\alpha \in \mathcal{A}$. Notice that $\top$ is defined based on the *actual* occurrence of time transitions. Before defining reward function Δ, recall that POOSL offers the statement **`currentTime`**. It returns the current model time of an executing model (see appendix B). Hence, **`currentTime`** can be considered to be a reward function $currentTime : \mathcal{S} \to \mathbb{R}$ defined as

$$currentTime(X_n) = \sum_{i=1}^{n} \top(X_i)$$

for any time-epoch n of the Markov chain $\{X_i \mid i \geq 1\}$. Now, let c be the conditional reward function for a reward function r that is defined based on adding a variable to $\mathcal{M}$. Then, reward function Δ defined by equation (2.29) on page 50, which denotes the duration of each value of r, can be written as

$$\Delta(X_i) = \begin{cases} 0 & \text{if } c(X_j) = 0 \text{ for all } j = 1, \ldots, i \\ 0 & \text{if } c(X_i) = 1 \\ currentTime(X_i) - & \text{if } c(X_j) = 1 \text{ for } j < i \text{ and} \\ \quad currentTime(X_j) & \quad c(X_k) = 0 \text{ for all } k = j+1, \ldots, i \end{cases} \tag{3.3}$$

by using the property that if action transitions are performed, the model time does not increase. Hence, the conditional long-run time average and variance of r can be evaluated by adding a variable to model $\mathcal{M}$ and using statement **`currentTime`**. Sections 3.3.1 and 4.3.2 discuss in more detail how this can be accomplished.

3.1.3 Related Research

Specifying Markov Chains The benefits of model-based approaches for defining Markov chains have been emphasised by other authors. Several formalisms have been developed and exploited, often focussing on the specification of continuous-time Markov chains [39, 175]. An example are queueing networks [102, 5], which require the behaviour of a system to be described in terms of service facilities that are interconnected by (store-and-forward) queues. Depending on how such queues and service facilities are interconnected and on the properties of the behaviour of both the environment and the service facilities, a queueing model implies a special Markov chain. This Markov chain is known as the birth-death process [5]. Examples of assumptions that make queueing networks mathematically tractable are the use of exponentially distributed inter-arrival times for the items that need to be processed by the service facilities as well as exponentially distributed service times. Although queueing networks are very useful for analysing congestion problems when accessing shared resources, applying these assumptions for modelling complete industrial hardware/software systems might lead to inadequate performance models.

In [124], stochastic Petri nets are used as a basis for performance analysis. Petri nets assume the behaviour of a system to be modelled in terms of places and transitions. Places may contain tokens and depending on the availability of tokens, transitions are performed. The numbers of tokens in all places reflect the marking (state) of the system. [124] extends this basic formalism by associating exponentially distributed random variables to all transitions in order to express their duration. Instead of such a tight coupling of time to actions, other types of Petri nets distinguish action and time transitions in a similar way as POOSL does. An example are generalised stochastic Petri nets [114], where exponentially distributed random variables are associated to time transitions, whereas immediate transitions take no time. Originally, generalised stochastic Petri nets lacked a mechanism for specifying the behaviour of a system in a compositional way, similarly as was the case for Markov chains. The ideas for extending Petri nets presented in [90] now do allow a hierarchical composition of generalised stochastic Petri nets, see [33]. Nevertheless, the imperative of using only exponential distributions in such Petri nets may be too restrictive for adequately performance modelling a complete industrial hardware/software system.

The difficulty of directly specifying Markov chains also triggered researchers in the area of process algebras [32, 96]. Process algebras are mathematical theories that allow modelling hardware/software systems in a compositional way and provide a means for reasoning about the (structural and behavioural) properties of a system based on equivalence relations. Traditional process algebras like CCS [122], CSP [77], ACP [10] and LOTOS [29] have been extended in many different ways to enable expressing time or stochasticity, see for example [96, 70, 181]. Restricting ourselves to process algebras that combine the modelling concept of asynchronous concurrency with synchronous message passing (as in POOSL), only a few offer both time and stochasticity. In [76], the stochastic process algebra PEPA is introduced, which associates exponentially distributed random variables to actions for denoting their duration. The stochastic process algebras EMPA [21] and TIPP [65] are also based on the approach that actions take exponentially distributed time. PEPA and EMPA models can be transformed either into stochastic Petri nets [147] and generalised stochastic Petri nets [20] respectively or directly into continuous-time Markov chains. Also TIPP models can be transformed into continuous-time Markov chains. Although improving the compositional way in which Markov chains can be specified, the mentioned stochastic process algebras still have insufficient expressive power to adequately model industrial hardware/software systems. This insufficiency originates amongst others from using only exponential distributions, the impossibility of modelling the data that is to be processed by a system, the lack of behavioural constructs and the restricted capabilities of expressing non-determinism[3]. Furthermore, designers often consider a mathematical syntax such as the ones of PEPA, EMPA and TIPP to be less intuitive than the lightweight notations provided by modelling languages like UML [149], SystemC [68] and POOSL.

To improve the expressive power of process algebras, recent research in this area concentrates on integrating the possibility of using various types of distributions to specify probabilistic timing behaviour. In [19] for example, EMPA is extended such that the duration of actions had a general distribution. Similarly, the process algebras[4] ♠ [96] and SPADES [71] were introduced as formalisms supporting the application of Markov chain-based performance analysis in the context of discrete event simulation. The used underlying mathematical structure for these process algebras is that of generalised semi-Markov processes [157]. The formal definition of a generalised semi-Markov process is in terms of a discrete-time Markov chain where the transitions are triggered by the occurrence of events. Associated with each event is a clock (timer) that records the remaining time until the event is scheduled to trigger a state transition. A number of events may occur at a certain state, but what event actually occurs depends on the speed at which the timers expire. The timers can be set in accordance with any distribution, thereby improving the expressive power.

To define a generalised semi-Markov chain, the process algebra SPADES in [71] extends CCS and separates action and time transitions under the assumption of action urgency. Also POOSL is based on an extension of CCS, separates action and time transitions and adopts action urgency (see also appendix B). SPADES [71] is however less expressive than POOSL, which emerges amongst others from the fact that designers have to resolve non-determinism in models constructed with SPADES by themselves. Furthermore, POOSL offers two levels of concurrency (concurrent activities and processes), whereas SPADES offers only one. SPADES also lacks the possi-

[3]For example, PEPA, EMPA and TIPP do not allow complete models to contain non-determinism [4].

[4]Although [96] and [71] use the same acronym (SPADES), different formalisms are involved.

bility of modelling the data processed by a system and it provides no primitives for interruption and abortion of behaviour. Finally, SPADES has a mathematical syntax, which is more difficult to interpret compared to the lightweight notation of POOSL.

Specifying Performance Properties Opposed to the numerous formalisms for denoting correctness properties (examples are LTL [112], CTL [49], FTL [8] and PSL [2]), only a few formalisms exist for specifying performance metrics. Most of these formalisms are developed in the context of continuous-time Markov chains. In [9, 11] for example, the temporal logic CSL is introduced, which allows the specification of various types of performance properties. In [41] and [17], formalisms are provided that enable specifying reward functions for Markov chains derived from PEPA and EMPA models respectively. The formalism of experiments in [4] can be used for specifying long-run average performance metrics in the context of Markov decision processes [45]. To define reward function Δ in equation (2.29), the formalism of temporal rewards proposed in [180] is used. Temporal rewards allow expressing various types of long-run average performance metrics for discrete-time Markov chains. A disadvantage of using the mentioned formalisms is the necessity to first construct the involved Markov chain or Markov decision process. Then, the performance metrics of interest are to be specified, usually by means of mathematical formulae that are often difficult to interpret. Next to suffering from the state-space explosion problem [126] when constructing the Markov chain, these formalisms cannot express many of the common performance metrics. An example is the average latency of transferring data through a system, which requires to associate the time at which a specific data item left the system with the time at which that data item entered the system.

The proposed reflexive approach of specifying reward functions is similar to the traditional way designers tend to extend a model with variables and behaviour for validation purposes. Validation is by definition an informal process (i.e., not mathematically founded) that is concerned with checking whether the developed formalisation of the system's behaviour complies with the informally specified ideas for realising its functionality. On the other hand, extending a model with the application of Markov-chain based performance analysis techniques in mind requires mathematically relating the variables added to a model with reward functions. Although reflexive performance analysis with POOSL follows the familiar approach for extending models, the model and its extension are implicity related with a Markov chain and reward functions to enable obtaining credible performance results.

3.2 Approximation of Recurrence Condition

The relation between POOSL models and their extensions with Markov chains and reward functions provides a sound basis for evaluating the performance of industrial hardware/software systems. As discussed in section 3.1, such performance analysis is accomplished by executing the (extended) POOSL model and relies on the simulation-based estimation techniques for evaluating conditional long-run averages presented in chapter 2. Direct application of these techniques, which is referred to as the *technique of regenerative cycles* [43], requires the specification of a relevant recurrent state (i.e., a recurrent state in which reward values have been assigned to the variables regarding the involved performance metric). Visiting such a relevant recurrent state marks the beginning of a regenerative cycle of independent behaviour for the Markov chain defined by a POOSL model *and* the accompanying reduced Markov chain. Although any relevant recurrent state can serve as a basis for applying the technique of regenerative cycles, the states and transitions of these implicitly defined (but not constructed) Markov chains are unknown prior to simulating the model. Hence, it is also unknown whether a state is transient or recurrent.

Since it is not possible to identify a recurrent state in advance, one would like to identify a recurrent state during simulation. Recall that by the ergodicity of the Markov chain implied by a POOSL model, only states that are visited infinitely often with probability 1 are of interest (see also sections 2.1.2 and 3.1.1). Notice that this property *cannot* be determined with a (finite number of) finite simulation(s). Hence, it is only possible to assume that a state, which is revisited during a simulation, is a recurrent state. Detecting the revisiting of a state during simulation can be accomplished with the following straightforward approach. After each simulation step, it is checked whether the newly entered state is relevant by observing whether a reward value has been assigned to the variable that implies the reward function of interest. If so, compare the newly entered relevant state with all previously visited relevant recurrent states. If a match is found, the state is revisited. Unfortunately, the sketched approach is fairly impractical for industrial hardware/software systems because the number of states that have to be compared is extremely large and the states themselves are very complex (large variables context for example). Consequently, detecting the revisiting of a state during simulation is often too time-consuming.

Instead of trying to identify the properties that mark the beginning of a regenerative cycle, it is more practical to predefine some condition for *enforcing* the start of a new cycle. Such a *recurrence condition* concerns a relatively simple property that is considered to form a suitable basis for obtaining regenerative cycles of (nearly) independent behaviour. This section presents two approaches for defining a recurrence condition. The first exploits the re-occurrence of a certain known point in the local behaviour of a component. The second approach is based on approximating the independence of regenerative cycles based on a sufficiently large fixed cycle length.

3.2.1 Exploiting Local Recurrence

Performance metrics are often defined in relation to a specific component of a system. Examples are the throughput of some communication resource (like an on-chip bus), the expected occupancy of a storage resource (such as a queue for buffering data) and the expected time needed to process data on some processor. The actual result for such a performance metric depends on the behaviour exposed by the component as a reaction to its local state combined with the interaction with its environment.

Consider a POOSL model $\mathcal{M}$ that defines the ergodic Markov chain $\{X_i \mid i \geq 1\}$ with state space $\mathcal{S}$. Let $\mathcal{M}$ include a variable `R` implying reward function r and accompanying conditional reward function c for $\{X_i \mid i \geq 1\}$ as discussed in section 3.1.2. Moreover, assume the evaluation of some conditional long-run average performance metric M expressed in terms of r and c, which is defined in relation to a specific component. To enable applying the simulation-based estimation techniques discussed in chapter 2, a recurrent state that is relevant with respect to c must be specified. Whether a state in $\mathcal{S}$ is relevant for evaluating M merely depends on the actual assignment of a reward value to R. Such an assignment is specified based on the occurrence of specific events that affect the behaviour of the component. As a result, the reduced Markov chain $\{X_i^{\upharpoonright c} \mid i \geq 1\}$ will reflect behaviour of only this specific component. On the other hand, whether a state of $\{X_i^{\upharpoonright c} \mid i \geq 1\}$ is recurrent depends on the behaviour and variables context of the complete model $\mathcal{M}$.

A practical approach for enforcing the beginning of regenerative cycles is to define a recurrence condition based on a certain local property. Sometimes, the assignment of a specific reward value v to R is known to re-occur due to specific events that affect the behaviour of the component. If the assignment of v to R is known to be reflected in only one recurrent state[5], this specific assignment can serve as recurrence condition. Sometimes, such recurrence condition can be recognised easily.

Example 3.5 *Consider a system that includes several buffers with FIFO policy for temporarily storing data items. Of interest is the evaluation of the average occupancy of one particular buffer* B*, which has a capacity of N. Let $\{X_i \mid i \geq 1\}$ denote the Markov chain defined by a POOSL model $\mathcal{M}$ of the system and that the variable* R *stores the current occupancy of* B*. Similar as in example 2.11 on page 52, the average occupancy of* B *can be expressed as the conditional long-run time average of the reward function implied by* R*, where the condition is an assignment to* R*. Assume that the inter-arrival time of data items to store in* B *and the duration of storing data items in* B *are exponentially distributed. If, in this case, the storing of data items into* B *and the retrieving of data items from* B *is performed by two independent activities, then there is exactly one recurrent state in the reduced Markov chain[6] of $\{X_i \mid i \geq 1\}$ for each possible occupancy of* B*. Hence, detecting the beginning of a regenerative cycle for $\{X_i \mid i \geq 1\}$ can be based on the property of* B *having a specific occupancy o. Notice that checking whether o is assigned to* R *is easy to perform during simulation.*

Defining a recurrence condition based on local properties generally results in more updates of intermediate estimation results for the same simulation time compared to when requiring that the behaviour and variables context of the complete model is taken into account. Hence, the proposed approach may considerably reduce the time until automatically terminating a simulation *without* harming the accuracy of the obtained estimation results. Nevertheless, the requirement of having specific knowledge about the behaviour of a component hinders its applicability.

Rare Events A performance metric may express the probability of the occurrence of a certain event. Such performance metrics are useful if they are related to rare events. A *rare event* is an event that occurs only occasionally, but which may strongly affect the behaviour of a system (many special operations can for example be needed to deal with such an event) and therefore its performance. Commonly, designers tend to minimise the occurrence of rare events [156] that have a negative effect on the system's performance. An example is the loss of data due to a full storage resource,

[5]Notice that there may exist several states reflecting the assignment of v to R. Without proof, it is stated that if the Markov chain is finite, then the mentioned requirement can be relaxed in case those states are recurrent and equivalent in the sense of being ordinarily lumpable [34, 162] based on equal reward values.

[6]This Markov chain represents a $M/M/1/N$ queue [5] with each state identifiable by the value of R.

where it is desirable to reduce the probability of loosing data as much as possible but without increasing the storage resource capacity too much.

Remark that the probability on a specific event equals the sum of the equilibrium probabilities of the states in which this event occurs. Simulation-based estimation can be accomplished by defining the probability of the event as a long-run average performance metric. For example, the above loss probability can be expressed[7] as the conditional long-run sample average of the reward function that is implied by a variable to which reward value 1 is assigned if data is lost and 0 otherwise. The condition for this performance metric is the reception of data that should be stored.

In case of estimating a performance metric for a specific component, which depends on known re-occurring events, the knowledge about the existence of each of these events may be used to define a recurrence condition. However, if one of the events is rare, then special precautions are needed to ensure a credible estimation result. The following example illustrates the importance of properly defining a recurrence condition when an estimation result depends on a rarely re-occurring event.

Example 3.6 *Let figure 3.7 represent the Markov chain that is obtained after reducing the Markov chain defined by some POOSL model. Consider estimating the long-run sample average μ of a reward function r, for which the reward values in the relevant states are as indicated. Using the approach shown in example 2.2 on page 26, it can be shown that μ is approximately equal to $\frac{1002}{3}$. When choosing the visit to state A or B as recurrence condition, intermediate estimation results are updated within two simulation steps with a probability that almost equals 1. As a consequence, it is very likely that intermediate estimation results for μ converge very rapidly, without taking the reward value for C into account. The simulation may therefore be terminated too soon by on concluding that a result of approximately $\frac{2}{3}$ for μ is accurate. On the other hand, when using the visit to C as recurrence condition, it is ensured that the reward values for all states are taken into account in accordance with the weights defined by the equilibrium distribution. In this case, a simulation will continue to run until it is (correctly) decided that a result of approximately $\frac{1002}{3}$ for μ is accurate.*

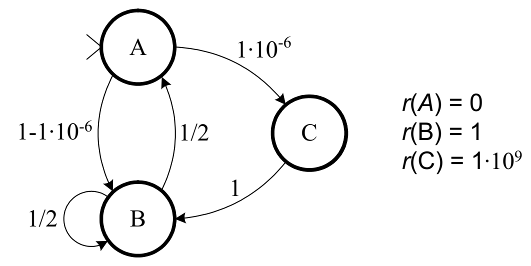


Figure 3.7: Markov chain with a rarely visited recurrent state.

Based on the observations in example 3.6, it is concluded that the occurrence of a known rare event can be a suitable recurrence condition for estimating performance metrics that depend on that event. Hence, the actual loss of data due to a full storage resource is a suitable recurrence condition for estimating the data loss probability.

[7]A proof for expressing the probability on an event in the this way can be based on theorem 2.1.

3.2.2 Batch-Means Approach

A popular statistical technique for estimating the expected value of the random variables that defined a covariance-stationary stochastic process (see appendix A) is the technique of (non-overlapping) *batch means* [106, 105, 3]. This section investigates the possibility of applying this classic batch-means technique for estimating conditional long-run sample averages defined for the Markov chain underlying a POOSL model.

Classic Batch Means Consider the covariance-stationary stochastic process $\{R_i \mid i \geq 1\}$ with $\mathbb{E}[R_i] = \mu$ for all $i \geq 1$. The *batch-means technique* involves grouping a prefix of $R_1, R_2, \ldots$ obtained after simulation of n time-epochs in k batches of size m (assume that $n = km$). The k^{th} so-called *batch mean*, denoted by B^k, is defined as

$$B^k = \frac{1}{m} \sum_{i=(k-1)m+1}^{km} R_i$$

The expected value $\mathbb{E}[B]$ of the discrete random variables B^k equals $\mathbb{E}[B] = \mu$, which can be estimated using the strongly consistent (and unbiased) point estimator

$$B(n) = \frac{1}{n} \sum_{i=1}^{n} B^k$$

Based on this result, the batch-means technique involves using $B(n)$ to estimate μ.

To determine the accuracy of a point estimate $\bar{\mu}$ obtained with $B(n)$, the batch-means technique gives rise to deriving a Confidence Interval based on the following reasoning. In case m is sufficiently large, it can be shown that (for relatively mild conditions) the random variables B^k are approximately uncorrelated [106]. Now, suppose that m can be chosen large enough such that the batch means are jointly normally distributed [132] (see also appendix B). Then, the random variables B^k are independent and normally distributed [105]. Since $\{R_i \mid i \geq 1\}$ is covariance-stationary, the batch means can actually be treated as independent random variables with identical normal distribution $\mathrm{N}(\mu, \sigma^2)$, where σ^2 denotes $\mathrm{var}[B]$. Based on these observations, a Confidence Interval can be derived for μ as follows. For $k \geq 1$, define the normally distributed discrete random variables H^k as $H^k = B^k - \mu$. Notice that their expected value $\mathbb{E}[H]$ is equal to 0. $\mathbb{E}[H]$ can be estimated using the (strongly consistent and unbiased) point estimator

$$H(n) = \frac{1}{n} \sum_{k=1}^{n} H^k = \frac{1}{n} \sum_{k=1}^{n} B^k - \mu = B(n) - \mu$$

Furthermore, the discrete random variables H^k are in $\mathcal{L}^2$ because $B^k \in \mathcal{L}^2$ for all $k \geq 1$ as a result of $\{R_i \mid i \geq 1\}$ being covariance-stationary. Notice that $\mathrm{var}[H] = \mathrm{var}[B] = \sigma^2$. If $\sigma^2 > 0$, then the assumption of the batch means being independent allows the application of the central limit theorem, which gives that

$$\frac{\sqrt{n}}{\sigma} H(n) \xrightarrow{\text{d.}} \mathrm{N}(0, 1)$$

Hence, for sufficiently large n and every $\kappa \in \mathbb{R}$,

$$\mathbb{P}\left(-\kappa \leq \frac{\sqrt{n}}{\sigma}H(n) \leq \kappa\right) = \mathbb{P}\left(B(n) - \frac{\kappa\sigma}{\sqrt{n}} \leq \mu \leq B(n) + \frac{\kappa\sigma}{\sqrt{n}}\right)$$

is approximately $2\mathfrak{N}(\kappa) - 1$. To use this result for defining a Confidence Interval, the variance σ^2 must be known. It can be estimated with the (unbiased) point estimator

$$\hat{\sigma}^2 = \frac{1}{n-1}\sum_{k=1}^{n}(B^k - \mu)^2$$

in which μ can be replaced by $B(n)$ since $B(n) \xrightarrow{\text{a.s.}} \mu$. Hence, the stochastic interval

$$\left[B(n) - \frac{\kappa\hat{\sigma}}{\sqrt{n}}, B(n) + \frac{\kappa\hat{\sigma}}{\sqrt{n}}\right] \quad \text{where} \quad \hat{\sigma}^2 = \frac{1}{n-1}\sum_{k=1}^{n}(B^k - B(n))^2 \tag{3.4}$$

is an (approximate) $2\mathfrak{N}(\kappa)-1$ Confidence Interval for μ that can be used for analysing the accuracy of a point estimate $\bar{\mu}$ obtained with $B(n)$. Remark that interval estimation of μ with the batch-means technique heavily relies on the assumption that m is sufficiently large. A major cause of error is therefore to take m too small, which may result in a high correlation between the batch means and a severely biased $\hat{\sigma}^2$ [105].

Batch-Means Estimation for Markov Chains Let the proper reward function r be defined for the ergodic Markov chain $\{X_i \mid i \geq 1\}$ such that the long-run sample average of r equals μ_s. It will be shown that (for appropriate assumptions and conditions) estimating μ_s with the batch-means technique using batch size m is comparable to applying the technique of regenerative cycles with a *fixed* cycle length of m. To this end, assume temporarily that $\{X_i \mid i \geq 1\}$ is covariance-stationary.

For $k \geq 1$, define t^k as the time-epoch at which a cycle with a fixed length of m starts, where t^1 is set to 1. Denoting the sum of reward values earned during the k^{th} cycle with Y^k and their length with L^k (instead of $Y^k_{S_r}$ and $L^k_{S_r}$ respectively as in section 2.1.3 where a cycle is through a recurrent state S_r of $\{X_i \mid i \geq 1\}$), it follows that

$$Y^k = \sum_{i=t^k}^{t^{k+1}-1} r(X_i) = \sum_{i=(k-1)m+1}^{km} r(X_i) \quad \text{and} \quad L^k = t^{k+1} - t^k = m$$

Notice that the expected value $\mathbb{E}[Y]$ of the random variables Y^k is equal to $m \cdot \mu_s$, while the expected value $\mathbb{E}[L]$ obviously is m. Hence, the classical point estimator

$$\hat{\mu}_s = \frac{Y(n)}{L(n)} \quad \text{with} \quad Y(n) = \frac{1}{n}\sum_{k=1}^{n} Y^k \quad \text{and} \quad L(n) = \frac{1}{n}\sum_{k=1}^{n} L^k = m$$

is also a strongly consistent point estimator for μ_s when the cycle length is fixed. Notice that $\hat{\mu}_s$ can now be rewritten as follows

$$\hat{\mu}_s = \frac{Y(n)}{m} = B(n)$$

where $B(n)$ is the point estimator for estimating μ_s with the batch-means technique.

A Confidence Interval for analysing the accuracy for a point estimate $\bar{\mu}_s$ obtained with $\hat{\mu}_s$ is determined based on the random variables $Z^k = Y^k - \mu_s \cdot L^k$ defined for $k \geq 1$ with expected value $\mathbb{E}[Z] = 0$. Similarly as for the classic batch-means technique, it is now assumed that m is sufficiently large such that the random variables Z^k can be treated as independent and normally distributed with $\mathrm{N}(0, \tau^2)$, where $\tau^2 = \mathrm{var}[Z]$. The resulting (approximate) $2\mathfrak{N}(\kappa) - 1$ Confidence Interval obtained after applying the central limit theorem (assuming $\tau^2 > 0$) is (see also section 2.1.3)

$$\left[\hat{\mu}_s - \frac{\kappa\hat{\tau}}{\sqrt{n}} \cdot \frac{1}{L(n)}, \hat{\mu}_s + \frac{\kappa\hat{\tau}}{\sqrt{n}} \cdot \frac{1}{L(n)}\right] \quad \text{where} \quad \hat{\tau}^2 = \frac{1}{n-1}\sum_{k=1}^{n}(Y^k - \hat{\mu}_s \cdot L^k)^2 \tag{3.5}$$

Recall that the Confidence Interval for the classic batch-means technique in equation (3.4) was determined based on random variables $H^k = B^k - \mu_s$ with $B^k = \frac{1}{m}Y^k$ and $\mathrm{var}[H] = \sigma^2$. Remark that $Z^k = Y^k - \mu_s \cdot L^k = m \cdot (\frac{1}{m}Y^k - \mu_s) = m \cdot H^k$ and hence, $\tau^2 = m^2\sigma^2$. As a result, the Confidence Interval in equation (3.5) can be rewritten as

$$\left[\frac{Y(n)}{m} - \frac{\kappa\hat{\tau}}{\sqrt{n}} \cdot \frac{1}{m}, \frac{Y(n)}{m} + \frac{\kappa\hat{\tau}}{\sqrt{n}} \cdot \frac{1}{m}\right] \quad \text{and hence as} \quad \left[B(n) - \frac{\kappa\hat{\sigma}}{\sqrt{n}}, B(n) + \frac{\kappa\hat{\sigma}}{\sqrt{n}}\right]$$

which is the Confidence Interval derived for the batch-means technique in equation (3.4). Hence, fixing the cycle length for the technique of regenerative cycles is comparable to using the batch-means technique. Finalising a batch of reward values is therefore a suitable recurrence condition if the involved assumptions are satisfied.

Next to the assumption of a sufficiently large batch size, estimating a long-run sample average μ_s defined for an ergodic Markov chain $\{X_i \mid i \geq 1\}$ using the batch-means technique involves the assumption that $\{X_i \mid i \geq 1\}$ is covariance-stationary. In general, $\{X_i \mid i \geq 1\}$ is *not* covariance-stationary since a particular simulation of $\{X_i \mid i \geq 1\}$ usually starts from a specific state with probability 1 [105, 121] (this is always the case for the Markov chain derived from POOSL models). However, for some simulations, $X_t, X_{t+1}, \ldots$ will become approximately covariance-stationary from a certain time-epoch t if this t is sufficiently large [105]. The period until t is then called the *initial transient* or *warm-up period* of the simulation [135]. Several approaches have been proposed to detect the end of the warm-up period during simulation, including Welch's procedure [182] and the algorithm by Schruben in [152]. A thorough discussion of the theoretical details behind these procedures and on approaches for choosing a sufficiently large batch size is beyond the scope of this thesis.

3.2.3 Related Research

Knowledge about the re-occurrence of certain local behaviour is not often recognised as a means for improving the applicability of the technique of regenerative cycles. On the other hand, knowledge about the behaviour of the system is known to be substantial for applying variance reduction techniques [105, 101]. These techniques improve the statistical efficiency of the used point estimator and hence, they allow to achieve a predefined accuracy within less simulation time [27]. Especially the variance reduction technique of importance sampling [40, 61] is commonly applied for quickly estimating the probability on a rare event, see for example [161, 72] and [93]. Importance sampling relies on simulating a Markov chain with a different probability measure, which emphasises the occurrence of the event, and then appropriately unbiasing the obtained estimation result [93]. Future research includes an investigation on the possibility of using the knowledge required for exploiting local recurrence as a basis for applying the technique of importance sampling as well.

The difficulties of straightforwardly applying the technique of regenerative cycles has initiated research on several other approaches next to the classic batch-means technique. Some variants of the classic batch-means technique have been proposed, including overlapped batch means [119, 3], weighted batch means [25] and replicated batch means [6]. Another approach is that of replication/deletion [3, 105], which assumes performing a number of independent simulation runs (using other seeds for the random number generators) to derive a Confidence Interval. Not only these approaches, but also the autoregressive method [52] and the method of spectrum analysis [73] involve the same assumption as the batch-means technique regarding the covariance-stationarity of the underlying stochastic process. Similarly as for the classic batch-means technique, all these approaches also rely on certain additional assumptions [105, 3]. The standardised time series method [153] requires even stronger conditions to be satisfied for the underlying stochastic process. Due to these conditions, all these approaches suffer from the problem of detecting the end of the warm-up period [105, 3]. Extensions to the classical batch-means technique have been proposed as well. They intend to minimise the correlation between the batch means. An example of such an approach is discussed in [50], where the correlation between the batches is estimated during simulation and depending on the results, a minimal batch size for the final estimation of the performance metric is determined.

3.3 Performance Monitors

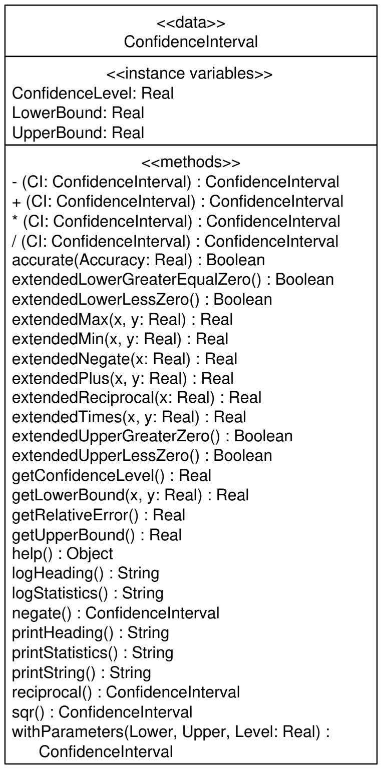
To provide user-friendly support for analysing long-run average performance metrics with POOSL, the simulation-based estimation techniques presented in chapter 2 have been implemented in a library of data classes. This library provides templates for monitors that shield the designer from the mathematical details of the techniques and rely on the results of sections 3.1 and 3.2 to enable applying these techniques.

3.3.1 Library Classes for Accuracy Analysis

To ease the accuracy analysis of common complex (conditional) long-run average performance metrics (see section 2.3.1) with POOSL, a library of six data classes has been developed. The classes named `LongRunSampleAverage`, `LongRunSample-Variance`, `LongRunTimeAverage` and `LongRunTimeVariance` present the actual performance monitor classes, while data class `PerformanceMonitor` is their superclass. It includes instance variables denoting common aspects like the confidence level and the obtained confidence interval. Such confidence intervals are represented by instances of data class `ConfidenceInterval`. Before elaborating on the actual performance monitor classes, the latter class is discussed in more detail.

Implementing the Algebra of Confidence Intervals Figure 3.8 shows a class diagram of `ConfidenceInterval` (see also section 4.2). Instance variables `Lower-`

Bound and UpperBound are used for storing the lower and upper bound respectively of the represented confidence interval. Recall that the bounds of a confidence interval are in $\bar{\mathbb{R}}$. Since only the lower bound can be $-\infty$ and only the upper bound can be ∞, it is possible to represent both $-\infty$ and ∞ with the elementary data object **nil**[8]. Hence, LowerBound and UpperBound can be instances of Real, while any operations on them are implemented in the methods of ConfidenceInterval (see also table 3.2 below). The instance variable ConfidenceLevel stores the confidence level of the represented confidence interval. Method withParameters initialises the LowerBound, UpperBound and ConfidenceLevel of a new instance.



```
<<data>>
ConfidenceInterval

<<instance variables>>
ConfidenceLevel: Real
LowerBound: Real
UpperBound: Real

<<methods>>
- (CI: ConfidenceInterval) : ConfidenceInterval
+ (CI: ConfidenceInterval) : ConfidenceInterval
* (CI: ConfidenceInterval) : ConfidenceInterval
/ (CI: ConfidenceInterval) : ConfidenceInterval
accurate(Accuracy: Real) : Boolean
extendedLowerGreaterEqualZero() : Boolean
extendedLowerLessZero() : Boolean
extendedMax(x, y: Real) : Real
extendedMin(x, y: Real) : Real
extendedNegate(x: Real) : Real
extendedPlus(x, y: Real) : Real
extendedReciprocal(x: Real) : Real
extendedTimes(x, y: Real) : Real
extendedUpperGreaterZero() : Boolean
extendedUpperLessZero() : Boolean
getConfidenceLevel() : Real
getLowerBound(x, y: Real) : Real
getRelativeError() : Real
getUpperBound() : Real
help() : Object
logHeading() : String
logStatistics() : String
negate() : ConfidenceInterval
printHeading() : String
printStatistics() : String
printString() : String
reciprocal() : ConfidenceInterval
sqr() : ConfidenceInterval
withParameters(Lower, Upper, Level: Real) :
    ConfidenceInterval
```

Figure 3.8: Data class ConfidenceInterval.

Recall that a confidence interval may be obtained based on a Confidence Interval that is derived from other Confidence Intervals using operations of the algebra of Confidence Intervals (see section 2.3.2). Class ConfidenceInterval implements the negation, square, reciprocal, addition, subtraction, multiplication and division operations of this algebra in the methods negate, sqr, reciprocal, +, −, * and / respectively. The implementation of the subtraction and division operations is based on the proofs given for corollaries 2.17 and 2.19. To implement the square, reciprocal and multiplication operations, it is necessary to determine which cases in theorems 2.14, 2.15 and 2.18 respectively are actually involved. For this purpose, the methods shown in table 3.1 are introduced. These methods enable to check the indicated

[8]An additional requirement is that the results returned by the methods in table 3.2 are used in an unambiguous way. This is ensured by the implementation of the operations defined on Confidence Intervals.

Method	Property Checked
`extendedLowerGreaterEqualZero`	`LowerBound` ≥ 0.0
`extendedLowerLessZero`	`LowerBound` < 0.0
`extendedUpperGreaterZero`	`UpperBound` > 0.0
`extendedUpperLessZero`	`UpperBound` < 0.0

Table 3.1: Checking properties of lower and upper bounds.

property, taking into account that `LowerBound` and `UpperBound` may be **`nil`** and following the property that $-\infty < x < \infty$ for all $x \in \mathbb{R}$ (see also appendix A). It is not difficult to demonstrate that methods `extendedLowerGreaterEqualZero` and `extendedUpperLessZero` provide sufficient means for determining which case in theorems 2.14 and 2.18 (see pages 53 and 55 respectively) is actually involved. To show that the methods in table 3.1 are also sufficient for implementing the reciprocal operation, consider the methods listed in table 3.2. These methods are defined for computing the bounds of the confidence interval resulting from the different operations and basically implement the extended rules of arithmetic for $\bar{\mathbb{R}}$ in [95]. For the reciprocal operation, the new bounds for the first and last two cases in theorem 2.15 (see page 53) can be computed with `extendedReciprocal`, which returns **`nil`** if its parameter is 0. Hence, if `extendedLowerLessZero` and `extended-UpperGreaterZero` are not both **`true`**, then `extendedReciprocal` can be used.

Method	Result	Used for Computing
`extendedNegate(x)`	$-x$	Lower and Upper Bounds
`extendedReciprocal(x)`	$\frac{1}{x}$	Lower and Upper Bounds
`extendedMax(x,y)`	$\max(x, y)$	Upper Bounds Only
`extendedMin(x,y)`	$\min(x, y)$	Lower Bounds Only
`extendedPlus(x,y)`	$x + y$	Lower and Upper Bounds
`extendedTimes(x,y)`	$x * y$	Lower and Upper Bounds

Table 3.2: Computing new lower and upper bounds.

Figure 3.9 shows the use of the methods in tables 3.1 and 3.2 to implement the reciprocal operation in method `reciprocal`. The expression in line 2 denotes a check on the third case in theorem 2.15. For this case, the lower and upper bound of the resulting confidence interval are $-\infty$ and ∞ respectively. In the other cases, the new bounds can be computed with `extendedReciprocal` as explained above. To de-

```
1   reciprocal() : ConfidenceInterval |Lower, Upper: Real|

2   if (self extendedLowerLessZero) & (self extendedUpperGreaterZero) then
3      Lower := nil; Upper := nil
4   else
5      Lower := self extendedReciprocal(UpperBound);
6      Upper := self extendedReciprocal(LowerBound)
7   fi;
8   return(new(ConfidenceInterval) withParameters(Lower, Upper, ConfidenceLevel)).
```

Figure 3.9: Implementation of the `reciprocal` method.

```
+(CI: ConfidenceInterval) : ConfidenceInterval |Lower, Upper, Level: Real|

Lower := self extendedPlus(LowerBound, CI getLowerBound);
Upper := self extendedPlus(UpperBound, CI getUpperBound);
Level := ConfidenceLevel + CI getConfidenceLevel - 1;
return(new(ConfidenceInterval) withParameters(Lower, Upper, Level)).
```

Figure 3.10: Implementation of the + method.

termine the new bounds and confidence level in case a binary operation is involved, methods `getLowerBound`, `getUpperBound` and `getConfidenceLevel` are defined. Their use is illustrated in figure 3.10, which shows the implementation of +.

Data class `ConfidenceInterval` includes the methods `getRelativeError` and `accurate`. The implementation of these methods is shown in figure 3.11. Assuming that the actual value of the long-run average being estimated is in the represented confidence interval, `getRelativeError` returns the upper bound for the relative error of a point estimate for this long-run average as defined by equation (2.13) on page 29. In case the upper bound is ∞, **`nil`** is returned. Method `accurate` determines whether the estimation result for the monitored long-run average is accurate according to an accuracy bound `Accuracy`, which is a parameter of `accurate`.

```
getRelativeError() : Real |Lower, Upper: Real|

if (LowerBound = nil) | (UpperBound = nil) then
   return(nil)
else if LowerBound > 0.0 then
   return((UpperBound - LowerBound) / (2 * LowerBound))
else if UpperBound < 0.0 then
   return((LowerBound - UpperBound) / (2 * UpperBound))
else
   return(nil)
fi fi fi.
```

```
accurate(Accuracy: Real) : Boolean |RelativeError: Real|

RelativeError := self getRelativeError;
if RelativeError = nil then return(false)
else return(RelativeError <= 1 - Accuracy) fi.
```

Figure 3.11: Implementation of `getRelativeError` and `accurate`.

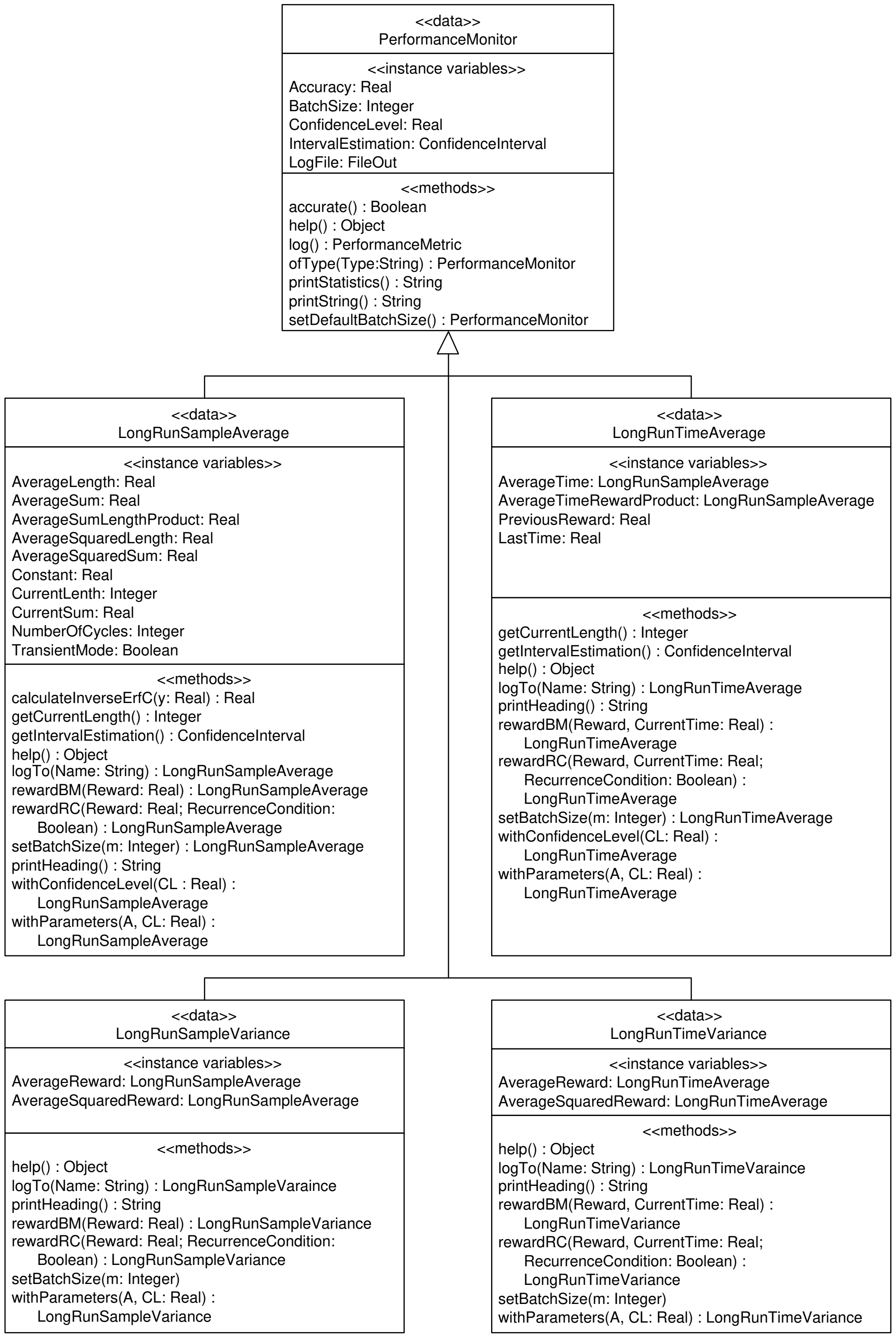
To provide information about a long-run average during validation with the SHESim tool, a `printString` method is defined (see also appendix B). It uses the strings constructed by `printHeading` and `printStatistics`. `printHeading` gives a heading for the actual estimation results returned by `printStatistics`. Next to the confidence interval itself and its confidence level, `printStatistics` returns the relative error as computed by `getRelativeError` and a point estimate for the monitored long-run average. This point estimate is determined as the center of the represented confidence interval (if any of the bounds is **`nil`**, then the point estimate is not specified), which is consistent with the remarks at the end of section 2.3.2. The methods `logHeading` and `logStatistics` are similar to `printHeading` and `printStatistics` respectively, but they produce a slightly different layout and are intended for logging the estimation results to file. Method `help` does not specify any behaviour but provides some information on the use of `ConfidenceInterval`.

Estimating Common Complex Long-Run Averages The performance monitors for common long-run averages are implemented in the classes depicted in the class diagram of figure 3.12, which is in accordance with the UML profile for SHE (see section 4.2). Instance variables `Accuracy`, `ConfidenceLevel` and `IntervalEstimation` of superclass `PerformanceMonitor` store respectively the desired accuracy bound, the desired confidence level and the obtained confidence interval. The confidence interval can be determined with the technique of regenerative cycles by exploiting local recurrence or by fixing the length of cycles in accordance with the batch-means technique. For the latter case, the instance variable `BatchSize` stores the fixed cycle size. Finally, `LogFile` refers to the file in which the estimation results for the monitored long-run average are to be logged.

Although superclass `PerformanceMonitor` can be used to create an instance of any of the monitor types included in the library (based on method `ofType`), such instances can also be created using the corresponding monitor data class directly. After creating a monitor, the method `withParameters` of the corresponding subclass must be used to initialise it with the desired accuracy bound and confidence level. Before actually doing so, it is first checked whether the accuracy bound and confidence level are in $(0, 1)$ and $[0, 1)$ respectively. Notice that an accuracy bound less then or equal to 0 makes no sense and that an accuracy bound of 1 is in general only attainable for infinite simulation runs. Moreover, confidence levels must be in $[0, 1]$ according to definition A.1, while a confidence level of 1 is in general only attainable for infinite simulation runs. If the conditions are not satisfied, an error is generated by calling method `error` (see appendix B). To actually check (during simulation) whether the estimation results for the monitored long-run average are accurate, the method `accurate` can be used. To this end, it simply calls method `accurate` of data class `ConfidenceLevel` on the instance variable `IntervalEstimation`.

When initialising a monitor, its instance variable `BatchSize` is set to a default value by calling method `setDefaultBatchSize` of superclass `PerformanceMonitor`. In case a different batch size is desired, the method `setBatchSize` of the corresponding subclass can be used. Notice that for classes `LongRunSampleVariance`, `LongRunTimeAverage` and `LongRunTimeVariance`, the method `SetBatchSize` sets the batch size for the constituent long-run sample averages to the same value. To initialise the file `LogFile` in which the estimation results must be logged, the method `logTo` should be used. Actually logging (intermediate) estimation results to this file is accomplished by calling method `log`. For validation purposes, a `printString` method (see also appendix B) is defined for `PerformanceMonitor`, which relies on the methods `printHeading` of the corresponding monitor class and `printStatistics`. The implementation of `printStatistics` relies on calling method `printStatistics` of class `ConfidenceInterval` on `IntervalEstimation`. Finally, each of the monitor data classes in the library includes a method `help`, which does not specify any behaviour but provides information on their use.

Now, consider subclass `LongRunSampleAverage` in more detail. Its instance variables `CurrentLength`, `CurrentSum` and `NumberOfCycles` store respectively the



<<data>>
PerformanceMonitor
<<instance variables>>
Accuracy: Real
BatchSize: Integer
ConfidenceLevel: Real
IntervalEstimation: ConfidenceInterval
LogFile: FileOut
<<methods>>
accurate() : Boolean
help() : Object
log() : PerformanceMetric
ofType(Type:String) : PerformanceMonitor
printStatistics() : String
printString() : String
setDefaultBatchSize() : PerformanceMonitor
<<data>>
LongRunSampleAverage
<<instance variables>>
AverageLength: Real
AverageSum: Real
AverageSumLengthProduct: Real
AverageSquaredLength: Real
AverageSquaredSum: Real
Constant: Real
CurrentLenth: Integer
CurrentSum: Real
NumberOfCycles: Integer
TransientMode: Boolean
<<methods>>
calculateInverseErfC(y: Real) : Real
getCurrentLength() : Integer
getIntervalEstimation() : ConfidenceInterval
help() : Object
logTo(Name: String) : LongRunSampleAverage
rewardBM(Reward: Real) : LongRunSampleAverage
rewardRC(Reward: Real; RecurrenceCondition: Boolean) : LongRunSampleAverage
setBatchSize(m: Integer) : LongRunSampleAverage
printHeading() : String
withConfidenceLevel(CL : Real) : LongRunSampleAverage
withParameters(A, CL: Real) : LongRunSampleAverage
<<data>>
LongRunTimeAverage
<<instance variables>>
AverageTime: LongRunSampleAverage
AverageTimeRewardProduct: LongRunSampleAverage
PreviousReward: Real
LastTime: Real
<<methods>>
getCurrentLength() : Integer
getIntervalEstimation() : ConfidenceInterval
help() : Object
logTo(Name: String) : LongRunTimeAverage
printHeading() : String
rewardBM(Reward, CurrentTime: Real) : LongRunTimeAverage
rewardRC(Reward, CurrentTime: Real; RecurrenceCondition: Boolean) : LongRunTimeAverage
setBatchSize(m: Integer) : LongRunTimeAverage
withConfidenceLevel(CL: Real) : LongRunTimeAverage
withParameters(A, CL: Real) : LongRunTimeAverage
<<data>>
LongRunSampleVariance
<<instance variables>>
AverageReward: LongRunSampleAverage
AverageSquaredReward: LongRunSampleAverage
<<methods>>
help() : Object
logTo(Name: String) : LongRunSampleVaraince
printHeading() : String
rewardBM(Reward: Real) : LongRunSampleVariance
rewardRC(Reward: Real; RecurrenceCondition: Boolean) : LongRunSampleVariance
setBatchSize(m: Integer)
withParameters(A, CL: Real) : LongRunSampleVariance
<<data>>
LongRunTimeVariance
<<instance variables>>
AverageReward: LongRunTimeAverage
AverageSquaredReward: LongRunTimeAverage
<<methods>>
help() : Object
logTo(Name: String) : LongRunTimeVaraince
printHeading() : String
rewardBM(Reward, CurrentTime: Real) : LongRunTimeVariance
rewardRC(Reward, CurrentTime: Real; RecurrenceCondition: Boolean) : LongRunTimeVariance
setBatchSize(m: Integer)
withParameters(A, CL: Real) : LongRunTimeVariance

Figure 3.12: Performance monitors for common long-run averages.

length of the current cycle through some predefined recurrent state, the sum of reward values obtained in the current cycle and the number of cycles that have been completed. `TransientMode` indicates whether the recurrent state has not been yet visited for the first time. Instance variable `AverageLength` is introduced for storing the point estimate for the expected cycle length, which is obtained according to the second point estimator in equation (2.9) on page 27. Similarly, `AverageSum` stores the point estimate for the expected sum of the reward values earned during a cycle, which is obtained through the first point estimator in equation (2.9). Recall that the quotient of the point estimates stored in `AvrageSum` and `AverageLength` gives the point estimate for the monitored long-run sample average, see equation (2.10). To derive a confidence interval for this point estimate by equation (2.12) on page 28, the result for the point estimator in equation (2.11) must be determined. To ease computing this point estimate, the results for the three constituent point estimators recognisable in the last line of equation (2.11) are computed and stored separately in the instance variables `AverageSquaredLength`, `AverageSumLengthProduct` and `AverageSquaredSum` respectively. Instance variable `Constant` is introduced for storing the value of κ if the desired confidence level is $2\Re(\kappa) - 1$.

The method `withParameters` of class `LongRunSampleAverage` actually only initialises `Accuracy` and then calls method `withConfidenceLevel` to initialise the other instance variables (except for `logFile`) to 0. The method `withConfidence-Level` initialises `Constant` based on a standardised implementation of the inverse complementary error function (often denoted with erfc^{-1}) in the method `calcula-teInverseErfC`. The complementary error function erfc [1] is defined as

$$\text{erfc}(z) = \frac{2}{\sqrt{\pi}} \int_z^\infty e^{-x^2} dx$$

and hence, it can be derived that

$$\Re(\kappa) = 1 - \frac{1}{2}\text{erfc}(\frac{\kappa}{\sqrt{2}})$$

As a result, for a confidence level $\gamma = 2\Re(\kappa) - 1$, it follows that $\kappa = \sqrt{2} \cdot \text{erfc}^{-1}(1 - \gamma)$.

The actual use of monitor class `LongRunSampleAverage` is as follows. To evaluate a long-run sample average with the technique of regenerative cycles, method `rewardRC` is provided. Figure 3.13 shows how `rewardRC` implements the interval estimation of a long-run sample average using equation (2.12). In line 3, it is checked whether a new cycle through some predefined recurrent state, which is identifiable by the `RecurrenceCondition` being **`true`**, has started. If so and if `TransientMode` was still **`true`**, then this recurrent state is visited for the first time and hence, setting `TransientMode` to **`false`** in line 5 indicates that only recurrent states will be visited in the sequel. In case the predefined recurrent state was already visited before, lines 7 - 28 update the confidence interval. To this end, lines 9 - 17 first update the point estimates obtained with the point estimators that constitute equation (2.12). The actual update of `IntervalEstimation` in lines 19 - 27 can only be performed if two or more cycles have been completed since only then `StandardDeviation` exists. Notice the use of the primitive method `asReal` to convert the `Integer` stored in `NumberOfCycles` to a `Real` such that its square root can be determined. Finally, in lines 30 and 31, the reward value obtained for the

```
rewardRC(Reward: Real, RecurrenceCondition: Boolean) : LongRunSampleAverage
|PointEstimation, StandardDeviation, Factor, HalfWidth: Real|

if RecurrenceCondition then
   if TransientMode then
      TransientMode := false
   else
      NumberOfCycles := NumberOfCycles + 1;
      Factor := (NumberOfCycles - 1) / NumberOfCycles;
      AverageSum := (Factor * AverageSum) + (CurrentSum / NumberOfCycles);
      AverageLength := (Factor * AverageLength) +
         (CurrentLength / NumberOfCycles);
      AverageSquaredSum := (Factor * AverageSquaredSum) +
         (CurrentSum sqr / NumberOfCycles);
      AverageSquaredLength := (Factor * AverageSquaredLength) +
         (CurrentLength sqr / NumberOfCycles);
      AverageSumLengthProduct := (Factor * AverageSumLengthProduct) +
         ((CurrentSum * CurrentLength) / NumberOfCycles);
      if NumberOfCycles > 1 then
         PointEstimation := AverageSum / AverageLength;
         StandardDeviation := ((1 / Factor) * (AverageSquaredSum -
            (2 * PointEstimation * AverageSumLengthProduct) +
            (PointEstimation sqr * AverageSquaredLength))) sqrt;
         HalfWidth := (Constant * StandardDeviation) /
            (AverageLength * (NumberOfCycles asReal sqrt));
         IntervalEstimation := new(ConfidenceInterval)
            withParameters(PointEstimation - HalfWidth,
            PointEstimation + HalfWidth, ConfidenceLevel)
      fi
   fi;
   CurrentSum := Reward;
   CurrentLength := 1
else
   if TransientMode not then
      CurrentSum := CurrentSum + Reward;
      CurrentLength := CurrentLength + 1
   fi
fi;
return(self).
```

```
rewardBM(Reward: Real) : LongRunSampleAverage

self rewardRC(Reward, (CurrentBatchLength = 0) |
   (CurrentBatchLength = BatchSize));
return(self).
```

Figure 3.13: Registering reward values for estimating long-run sample averages.

newly started cycle is stored in `CurrentSum` and the length of the cycle is reset to 1. In case the `RecurrenceCondition` was **`false`**, the sum of rewards and the length of the current cycle must be updated (but only if `TransientMode` is **`false`**).

Instead of evaluating a long-run sample average with the technique of regenerative cycles, method `rewardBM` of `LongRunSampleAverage` can be used to estimate it with the batch-means technique. The implementation of `rewardBM` relies on calling `rewardRC`, see figure 3.13. The recurrence condition parameter is set to the result of `CurrentLength = 0 | CurrentLength = BatchSize`. This condition holds each time a cycle of length `BatchSize` is completed and at the moment of registering the first reward value. Hence, the effect of the warm-up period is *not* taken into account. Future research includes an investigation on implementing the detection of the end of the warm-up period as proposed in for example [182] or [152].

Recall from section 2.3.1 that a long-run sample variance can be rewritten as the long-run sample average of squared reward values minus the square of the long-run sample average of reward values. To evaluate long-run sample variances based on this result, monitor class `LongRunSampleVariance` includes the instance variables `AverageSquaredReward` and `AverageReward`. They store estimation results obtained for the long-run sample average of squared reward values and the long-run sample average of reward values respectively. Method `withParameters` of `LongRunSampleVariance` initialises `AverageSquaredReward` and `Average-Reward` as monitors of class `LongRunSampleAverage`. The confidence level for both these constituent long-run sample averages is set (using method `withConfi-denceInterval` of `LongRunSampleAverage`) to $\frac{1}{2}(CL + 1)$, where CL is the desired confidence level for estimating the long-run sample variance. The reason for increasing the confidence levels of the constituent long-run sample averages is to (evenly) compensate for the effect of the subtraction operator of the algebra of Confidence Intervals on the confidence level. This operator forms the basis of determining `IntervalEstimation` in the methods `withParameters` and `rewardRC`.

Figure 3.14 depicts how method `rewardRC` of `LongRunSampleVariance` implements the interval estimation of long-run sample variances with the technique of regenerative cycles. After updating the point estimates for the constituent long-run sample averages in lines 2 and 3, it is checked whether the confidence interval stored in `IntervalEstimation` must be updated. If so, this update is performed in lines 5, 6 by methods `-` and `sqr` of class `ConfidenceInterval` conform the definition of long-run sample variances. Remark the use of method `getIntervalEstima-tion` of `LongRunSampleAverage` to obtain the confidence intervals for the constituent long-run sample averages. Figure 3.14 also shows how the implementation of method `rewardBM` (which allows the estimation of a long-run sample variance with the batch-means technique) relies on method `rewardRC` in a similar way as for monitor class `LongRunSampleAverage`. Notice the use of method `getCurrent-Length` of class `LongRunSampleAverage` to obtain the length of the current batch.

```
1    rewardRC(Reward: Real, RecurrenceCondition: Boolean) : LongRunSampleVariance

2    AverageReward rewardRC(Reward, RecurrenceCondition);
3    AverageSquaredReward rewardRC(Reward sqr, RecurrenceCondition);
4    if RecurrenceCondition then
5       IntervalEstimation := AverageSquaredReward getIntervalEstimation -
6          (AverageReward getIntervalEstimation sqr)
7    fi;
8    return(self).
```

```
9    rewardBM(Reward: Real) : LongRunSampleVariance

10   self rewardRC(Reward, (AverageReward getCurrentLength = 0) |
11      (AverageReward getCurrentLength = BatchSize));
12   return(self).
```

Figure 3.14: Registering reward values for estimating long-run sample variances.

Now, consider class `LongRunTimeAverage`. Recall from section 2.3.1 that a long-run time average can be written as a quotient of the long-run sample average of previous reward values times their durations and the long-run sample average of previous durations. The instance variables `AverageTimeRewardProduct` and `AverageTime` are introduced to store estimation results for these constituent long-run sample averages. Initialisation of these monitors by `withParameters` relies on method `withConfidenceInterval`. Instead of explicitly implementing the previous-reward operator Θ to enable obtaining the previous reward value and its duration, instance variables `PreviousReward` and `LastTime` are defined. `PreviousReward` stores the previously registered reward value, while `LastTime` is used for storing the most recent model time at which a reward value was registered. Recall that equation (3.3) on page 73 allows to determine the previous duration based on statement **`currentTime`**. Hence, the model time returned by **`currentTime`** must be a parameter for the methods that update the intermediate estimation results.

```
1    rewardRC(Reward, CurrentTime: Real, RecurrenceCondition: Boolean) :
2      LongRunTimeAverage

3    if LastTime != nil then
4      AverageRewardDurationProduct rewardRC(PreviousReward *
5        (CurrentTime - LastTime), RecurrenceCondition);
6      AverageDuration rewardRC(CurrentTime - LastTime, RecurrenceCondition)
7    fi;
8    PreviousReward := Reward;
9    LastTime := CurrentTime;
10   if RecurrenceCondition then
11     IntervalEstimation := (AverageRewardDurationProduct getIntervalEstimation) /
12       (AverageDuration getIntervalEstimation)
13   fi;
14   return(self).
```

```
15   rewardBM(Reward, CurrentTime: Real) : LongRunTimeAverage

16   self rewardRC(Reward, CurrentTime, (AverageTime getCurrentLength = 0) |
17      (AverageTime getCurrentLength = BatchSize));
18   return(self).
```

Figure 3.15: Registering reward values for estimating long-run time averages.

Figure 3.15 presents how method `rewardRC` enables to estimate long-run time averages with the technique of regenerative cycles. When calling `rewardRC`, parameter `CurrentTime` must be set to the model time as returned by **`currentTime`**. As a result, the previous duration can be determined by `CurrentTime - LastTime`, see also equation (3.3). However, if no reward value was registered before, `LastTime` is undefined (**`nil`**). Hence, an additional check is needed before updating the estimation results for the constituent long-run sample averages. This check is performed in line 3. If `LastTime` is not **`nil`**, then `AverageRewardDurationProduct` and `AverageDuration` can be updated, see lines 4 - 6. Then, the new reward value and time at which this reward value is obtained are stored in lines 8 and 9. Finally, if the recurrence condition holds, the confidence interval for the long-run time average is updated (lines 11, 12). This update is based on method `/` of class `ConfidenceInterval` conform the definition of long-run time averages. Figure 3.15 additionally shows how method `rewardBM` allows interval estimation of long-run time averages with the batch-means technique in a similar way as for the other monitor classes.

The estimation of long-run time variances is accomplished by using an instance of class `LongRunTimeVariance`. Recall that a long-run time variance can be rewritten as the long-run time average of the squared reward values minus the square of the long-run time average of reward values. To store the estimation results for these constituent long-run time averages, instance variables `AverageSquaredReward` and `AverageReward` are introduced. Figure 3.16 shows the implementation of methods `rewardRC` and `rewardBM`, which enable interval estimation of long-run time variances with the technique of regenerative cycles and the batch-means technique respectively. Notice that `rewardRC` relies on using the `-` and `sqr` method of class `ConfidenceInterval` conform the definition of long-run time variances.

```
rewardRC(Reward, CurrentTime: Real, RecurrenceCondition: Boolean) :
   LongRunTimeVariance

AverageReward rewardRC(Reward, CurrentTime, RecurrenceCondition);
AverageSquaredReward rewardRC(Reward sqr, CurrentTime, RecurrenceCondition);
if RecurrenceCondition then
   IntervalEstimation := (AverageSquaredReward getIntervalEstimation) -
      ((AverageReward getIntervalEstimation) sqr)
fi;
return(self).
```

```
rewardBM(Reward, CurrentTime: Real) : LongRunTimeVariance

self rewardRC(Reward, CurrentTime, (AverageReward getCurrentLength = 0) |
   (AverageReward getCurrentLength = BatchSize));
return(self).
```

Figure 3.16: Registering reward values for estimating long-run time variances.

3.3.2 On Tool Extensions

Superclass `PerformanceMonitor` of the monitor data classes discussed in section 3.3.1 includes a method `accurate` that returns a Boolean indicating whether the estimation result for the monitored long-run average is accurate. Often, a POOSL model is extended with several monitors. If the estimation results for *all* monitored performance metrics are accurate, then the simulation can be terminated. Currently, the tools for executing POOSL models do not offer sufficient features for a user-friendly way of terminating a simulation. As a consequence, checking whether the estimation results for all monitored performance metrics are accurate requires extending the original POOSL model with some activity that registers the accuracy status of all monitored performance metrics (see also section 4.3.2). The only way currently available for actually terminating a simulation is to enforce deadlock. This can be accomplished by aborting the behaviour of all processes. Future research includes an investigation on how the accuracy status of the monitored performance metrics can be used as a more direct means for the tools to terminate a simulation.

3.4 Quality Assessment

A suitable approach for evaluating the quality of estimation results obtained with the library classes for accuracy analysis presented in section 3.3.1 is to analyse the coverage of the underlying Confidence Intervals [118]. The *coverage* of a Confidence Interval is defined as the frequency with which confidence intervals obtained for a certain performance metric contain the true value of the performance metric. Notice that the coverage of a Confidence Interval should converge to at least its confidence level when the number of obtained confidence intervals becomes large.

Coverage analysis is limited to systems for which performance metrics can also be computed analytically [118]. The well-known M/M/1 queueing system [5, 102] forms a suitable basis for determining the coverage of the Confidence Intervals underlying all four types of performance metrics for which the library classes provide monitors. The performance modelling of the M/M/1 queueing system is discussed in the next section, while the coverage results are presented in section 3.4.2.

3.4.1 Experiment

The M/M/1 queueing system consists of an unbounded queue for storing received tokens and a single server to process these tokens. Assume that the parameter for the exponentially distributed inter-arrival time of the tokens is λ and that the parameter for the exponentially distributed processing time is μ. Then, for $\lambda < \mu$, the M/M/1 queueing system implies an ergodic (embedded) Markov chain of which all states are recurrent [102]. Now, let N denote the number of tokens in the system, i.e., the number of tokens in the unbounded queue plus the one being processed by the server. Two important performance metrics for the M/M/1 queueing system are $\mathbb{E}[N]$ and var$[N]$, which are given by [5, 102]

$$\mathbb{E}[N] = \frac{\lambda}{\mu - \lambda} = \frac{\rho}{1-\rho} \quad \text{and} \quad \mathrm{var}[N] = \frac{\mu\lambda}{(\mu-\lambda)^2} = \frac{\rho}{(1-\rho)^2}$$

where $\rho = \frac{\lambda}{\mu}$ is called the load. Two other important performance metrics for the M/M/1 queueing system are related to the sojourn time w for tokens. The sojourn time of a token is the time between entering the queue and leaving the system after being processed by the server and hence, it equals the time that the token spends waiting in the queue plus its processing time. Due to the exponentially distributed inter-arrival and processing times, the sojourn time is again exponentially distributed and the parameter of this exponential distribution is $\mu(1-\rho)$ [31]. Hence, the expected value $\mathbb{E}[w]$ and variance var$[w]$ of the sojourn time are given by

$$\mathbb{E}[w] = \frac{1}{\mu - \lambda} = \frac{1}{\mu(1-\rho)} \quad \text{and} \quad \mathrm{var}[w] = \frac{1}{(\mu-\lambda)^2} = \frac{1}{\mu^2(1-\rho)^2}$$

respectively. Table 3.3 shows some numerical performance results for different loads.

To analyse the coverage of the Confidence Intervals underlying the POOSL library classes for accuracy analysis, a model of the M/M/1 queueing system has been developed. A screen dump of this model in the SHESim tool is depicted in figure 3.17.

λ	μ	ρ	$\mathbb{E}[N]$	var$[N]$	$\mathbb{E}[w]$	var$[w]$
1	10	0.1	1/9	10/81	1/9	1/81
1	5	0.2	1/4	5/16	1/4	1/16
3	10	0.3	3/7	30/49	1/7	1/49
2	5	0.4	2/3	10/9	1/3	1/9
1	2	0.5	1	2	1	1
3	5	0.6	3/2	15/4	1/2	1/4
7	10	0.7	7/3	70/9	1/3	1/9
4	5	0.8	4	20	1	1
9	10	0.9	9	90	1	1

Table 3.3: True performance of the M/M/1 queueing system for different λ and μ.

The model includes the processes `Sender`, `Buffer` and `Receiver`. After creating a new token, the `Sender` waits with actually sending it to the `Buffer` until having delayed for a time that is exponentially distributed with parameter λ. To this end, the `Sender` includes an instance variable of data class `Exponential` from the library for modelling distributions, see section 3.1.1. The `Buffer` process, which models the M/M/1 queueing system, contains an instance variable of a container data class (see also section 4.3.1) `Queue`, which represents an unbounded queue. The behaviour of `Buffer` is defined by two independent concurrent activities as proposed in example 3.5 on page 77. One activity is concerned with receiving tokens from the `Sender` and putting them in the `Queue`. Whenever there is a token in the `Queue`, the other activity models the processing of it by delaying for an exponentially distributed time with parameter μ. Only after having delayed, this activity removes the token from the `Queue` and sends it to the `Receiver`. To model the processing time, `Buffer` also contains an instance variable of class `Exponential`. The actual sending and receiving of tokens by the concurrent activities is performed instantaneously.

Because of keeping the tokens in the `Queue` while they are being processed, the occupancy of the `Queue` equals the number of tokens in the system. Hence, the time-average `Queue` occupancy and the time-variance in the `Queue` occupancy con-

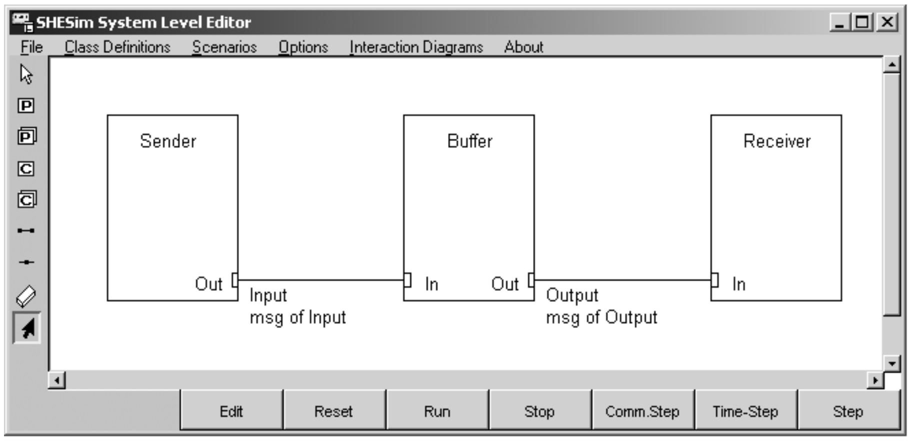


Figure 3.17: Model of the M/M/1 queueing system in SHESim.

verge almost surely to $\mathbb{E}[N]$ and var$[N]$ respectively. Observe that these performance metrics concern a conditional long-run time average and variance respectively, where the condition for both is putting or removing a token from the `Queue` (see also example 2.11 on page 52). Moreover, the sample-average duration of being in the `Queue` and the sample-variance in this duration converges almost surely to $\mathbb{E}[w]$ and var$[w]$ respectively. These performance metrics concern a conditional long-run sample average and variance respectively, where the condition for both is the completion of the processing of a token (see also example 2.10 on page 50). Remark that an alternative for this condition is the removal of a token from the `Queue`.

To estimate the identified performance metrics, the POOSL model is extended with eight performance monitors. Each of the four performance metrics is estimated with both the technique of regenerative cycles and the batch-means technique. To apply the technique of regenerative cycles, the situation of the `Queue` being empty is used as recurrence condition for all performance metrics. Notice that this recurrence condition is appropriate for estimating $\mathbb{E}[N]$ and var$[N]$ (see also example 3.5). However, using the emptiness of the `Queue` as recurrence condition for estimating $\mathbb{E}[w]$ and var$[w]$ is actually *not* advisable. This is because the emptiness of the `Queue` does not represent the completion of processing a token that took a *specific time* and hence, the state in which `Queue` is empty does not necessarily imply a relevant *recurrent* state for the implied reduced Markov chain (i.e., the Markov chain that is obtained after reducing the Markov chain defined by the POOSL model with respect to the condition of completing the processing of a token). Nevertheless, the emptiness of the `Queue` is still used as recurrence condition because it is not evident which processing time is suitable for defining a recurrence condition. For the batch-means technique, a default batch size as provided by the monitor classes is used.

To obtain a number of confidence intervals for each of the eight performance monitors (with which the coverage of their underlying Confidence Intervals can be estimated), some further extensions to the model are needed. These extensions allow to reset the M/M/1 queueing system by re-initialising the `Queue`, all performance monitors and both instances of `Exponential`. Recall that initialising an instance of `Exponential` involves the creation of a random number generator. Since the library classes for distributions rely on `randomiseSeed` to initialise such random number generators, the simulation subruns will be *independent* (see also appendix B).

The actual coverage analysis is performed for each of the combinations of λ and μ indicated in table 3.3 as follows. Each coverage metric can be considered as a conditional long-run sample average, where the reward value is 1 if the true performance value is indeed in the obtained confidence interval and 0 if it is not. The condition for this metric is the completion of a simulation subrun. Since the simulation subruns are independent, this condition may also serve as recurrence condition for applying the technique of regenerative cycles. Hence, an additional eight performance monitors of class `LongRunSampleAverage` are introduced, which will be referred to as the *coverage monitors*. A reward value (0 or 1) is registered for a coverage monitor with method `rewardRC` at the moment that the estimation results for the accompanying performance metric are being considered accurate for the *first time*[9]. If *all* estimation results are considered to be accurate, then the current simulation subrun is terminated and the M/M/1 queueing system is reset as described above. The cri-

[9]If a model contains only one monitor, this is would be the criterium for terminating the simulation.

terium for terminating a simulation subrun is that all eight performance monitors have an accuracy of at least 0.95 (for a confidence level of 0.95). Each time a simulation subrun is terminated, the intermediate coverage results are logged to file as well. When a minimum of 500 simulation subruns have been performed[10] and *all* coverage results are considered to be accurate, then the coverage analysis is completed. The decision of whether the coverage results are accurate is based on initialising all eight coverage monitors with a desired confidence level and accuracy bound of 0.95.

3.4.2 Results

As indicated in section 3.4.1, the coverage analysis for each load in table 3.3 is completed when a minimum of 500 simulation subruns have been performed and the coverage results for all eight performance monitors become accurate. However, for the loads of 0.8 and 0.9, this approach turned out to imply exceptionally long execution times (over 12 days on a Mobile Intel Pentium 4 M running at 1.8 GHz). Therefore, the coverage analysis for these loads was terminated prematurely. Table 3.4 shows the number of simulation subruns that have been performed for each load. The coverage results that were obtained after performing the indicated numbers of simulation subruns have been documented in eight graphs, which are discussed below. These graphs plot the coverage results against the different loads. Vertical error bars are used to indicate the confidence intervals accompanying the coverage results.

ρ	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Number of subruns	836	775	802	665	735	789	818	500	250

Table 3.4: Actual numbers of simulation subruns that were performed.

Figures 3.18 and 3.19 show the coverage results that were obtained when estimating $\mathbb{E}[N]$ and var$[N]$ respectively with the technique of regenerative cycles, where an empty `Queue` served as recurrence condition. Recall that estimating $\mathbb{E}[N]$ and var$[N]$ with this approach is regarded as very appropriate and hence, good coverage results are expected. Indeed, the coverage results for $\mathbb{E}[N]$ are larger than the initialised confidence level of 0.95 for all loads. Moreover, the coverage results for var$[N]$ are 1 for all loads. So, all confidence intervals obtained for var$[N]$ contained its true value. It can be concluded that estimating $\mathbb{E}[N]$ and var$[N]$ using the Confidence Intervals underlying the monitor classes `LongRunTimeAvarage` and `LongRunTimeVariance` respectively as described here yields excellent coverage results.

The coverage results that were obtained for the case of estimating $\mathbb{E}[w]$ and var$[w]$ with the technique of regenerative cycles are depicted in figures 3.20 and 3.21 respectively. Recall that the used recurrence condition (emptiness of the `Queue`) is not considered to be ideal for estimating $\mathbb{E}[w]$ and var$[w]$. Hence, using this approach may result in unrealistic estimation results even though these results are regarded as accurate when terminating a simulation subrun. Obviously, the coverage results may suffer from improperly estimating $\mathbb{E}[w]$ and var$[w]$. As shown in figure 3.20, the coverage results for $\mathbb{E}[w]$ are indeed less than the initialised confidence level of

[10]It is necessary to perform a minimum number of simulation subruns to ensure that the coverage analysis is not terminated prematurely when all estimation results happen to be accurate immediately.

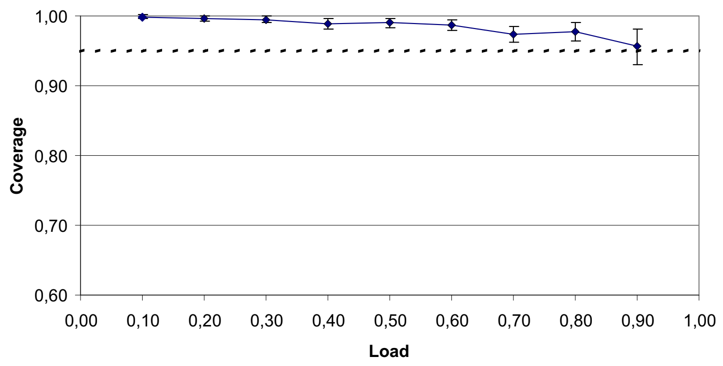


Figure 3.18: Coverage results when estimating $\mathbb{E}[N]$ using `rewardRC`.

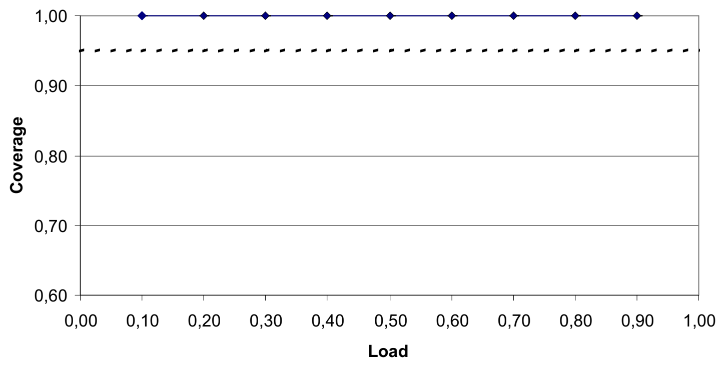


Figure 3.19: Coverage results when estimating var$[N]$ using `rewardRC`.

0.95 for all loads. Considering that the Confidence Interval used in class `LongRun-SampleVariance` is derived from the one use in `LongRunSampleAverage`, it is rather surprising that the coverage results for var$[w]$ are 1 for all loads (see figure 3.21). This suggests that using the presumably inappropriate recurrence condition of an empty `Queue` may not be so bad after all for estimating $\mathbb{E}[w]$. More research is needed to identify the exact reason for obtaining excellent coverage results when estimating var$[w]$ using class `LongRunSampleVariance` as described here.

Figures 3.22 and 3.23 depict the coverage results that were obtained when estimating $\mathbb{E}[N]$ and var$[N]$ respectively with the batch-means technique (using a default batch size of 10000). Recall that the implementation of the batch-means technique (see section 3.3.1) does not take the warm-up period of a simulation subrun into account. As a consequence, the estimation results for $\mathbb{E}[N]$ and var$[N]$ may be unrealistic even though they are regarded as being accurate when terminating a simulation subrun. Clearly, the coverage results for $\mathbb{E}[N]$ and var$[N]$ may suffer from this deficiency. As shown in figure 3.22, the coverage results for $\mathbb{E}[N]$ are less than the initialised confidence level of 0.95 for all loads. Repeating the estimation of $\mathbb{E}[N]$ with much larger batch sizes than the default did not improve the coverage results significantly, thereby ruling out the possible cause of using a too small batch size. Based on the commonly high influence of the warm-up period, much better coverage results are

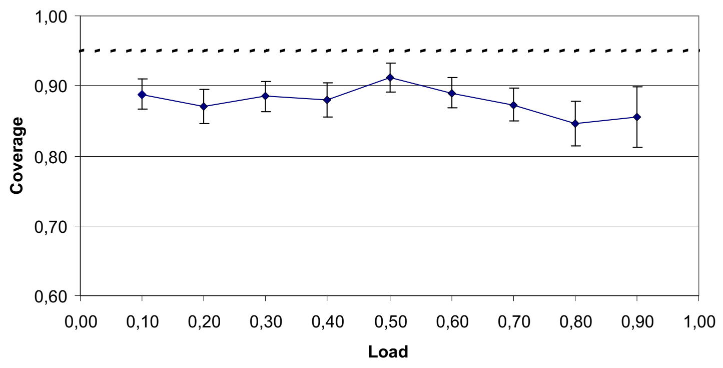


Figure 3.20: Coverage results when estimating $\mathbb{E}[w]$ using `rewardRC`.

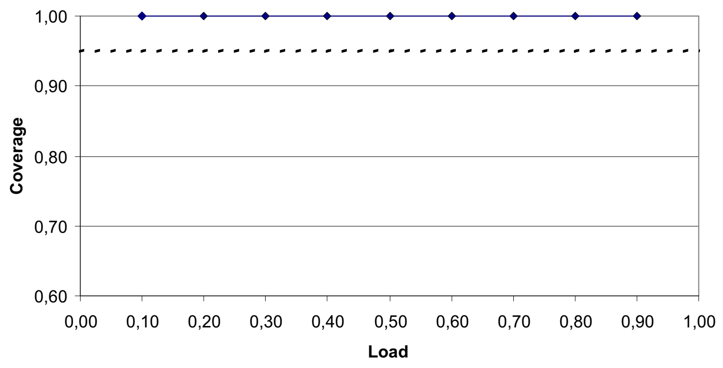


Figure 3.21: Coverage results when estimating var$[w]$ using `rewardRC`.

expected in case this warm-up period would have been taken into account according to for example Welch's procedure [182] or the algorithm by Schruben in [152]. Somewhat surprising is that the coverage results for estimating var$[N]$ are larger than 0.95 for all loads except 0.1 and 0.2 when considering that the Confidence Interval using in monitor class `LongRunTimeVariance` is derived from the one used in class `LongRunTimeAverage`. This suggests that the influence of the warm-up period may not be too high after all for these cases. More research is needed to identify the exact reason for obtaining reasonable coverage results when estimating var$[N]$ using performance monitor class `LongRunTimeVariance` as described here.

The coverage results that were obtained for estimating $\mathbb{E}[w]$ and var$[w]$ with the batch-means technique (using a default batch size of 10000) are presented in figures 3.24 and 3.25 respectively. The coverage results for $\mathbb{E}[w]$ are less than 0.80 for all loads, which is rather bad. In addition, the coverage results for var$[w]$ are only larger than 0.95 for the highest loads. The possible cause of using a too small batch size is ruled out by having repeated the experiment using much larger batch sizes than the default, which did not improve the coverage results significantly. A likely cause for getting bad coverage results is the fact that the current implementation of the batch-means technique does not take the warm-up period into account.

Now, a comparison can be made between the coverage results obtained for a perfor-

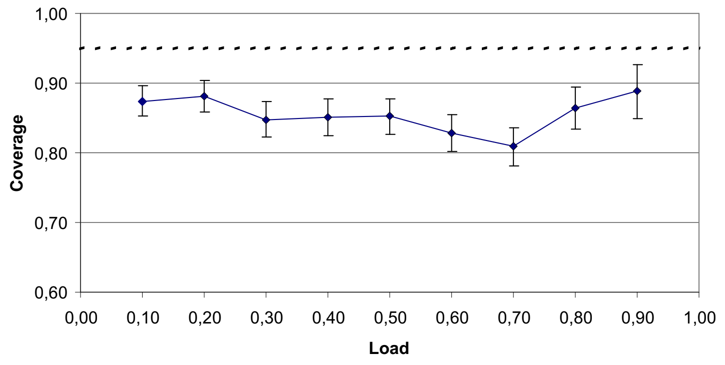


Figure 3.22: Coverage results when estimating $\mathbb{E}[N]$ using `rewardBM`.

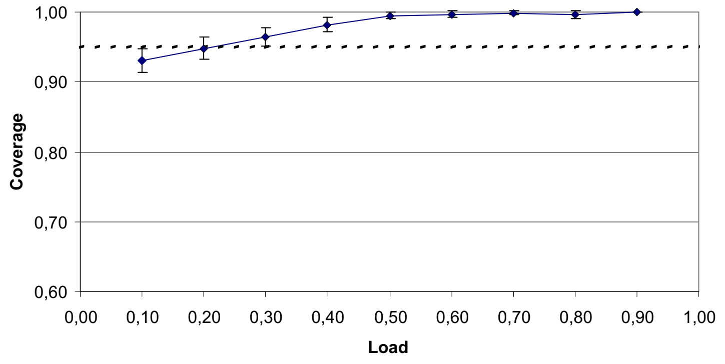


Figure 3.23: Coverage results when estimating $\mathrm{var}[N]$ using `rewardBM`.

mance metric when using the technique of regenerative cycles and those obtained for that metric when using the batch-means technique. For all performance metrics, it can be seen that for the performed experiment and the current implementation of the performance monitor classes, using the technique of regenerative cycles is favorable.

3.5 Conclusions

The modelling language POOSL enables to specify the behaviour of a system in a way that is more convenient than directly writing down the states and transitions of a Markov chain. After explicitly transforming the timed probabilistic labelled transition system defined by the semantics of a POOSL model into a Markov chain and separately specifying reward functions, Markov chain-based performance analysis techniques can be applied. This existing model-checking approach is supplemented in this chapter with a framework for reflexive performance analysis. Relying on the mathematical relation between POOSL models and their extensions with Markov chains and reward functions, the framework provides a sound basis for simulation-based estimation of performance metrics by executing the extended POOSL model.

The labelled transition system defined by a POOSL model distinguishes action tran-

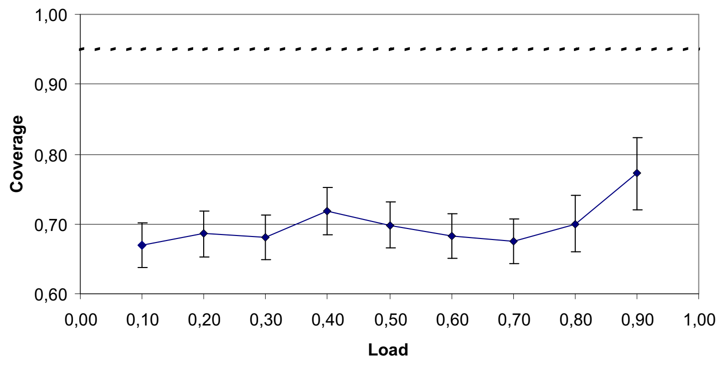


Figure 3.24: Coverage results when estimating $\mathbb{E}[w]$ using `rewardBM`.

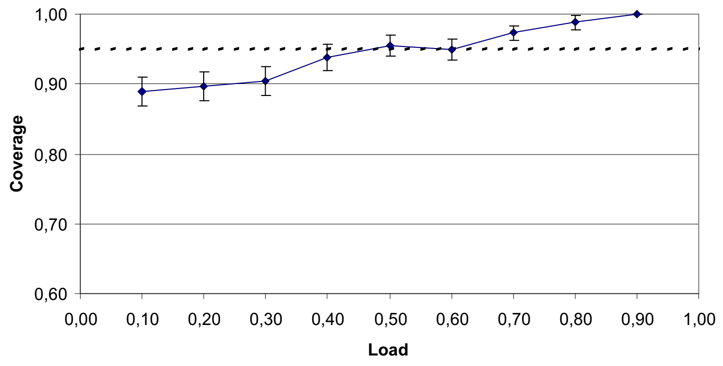


Figure 3.25: Coverage results when estimating $\mathrm{var}[w]$ using `rewardBM`.

sitions and time transitions. Action transitions have a probabilistic interpretation, which is reflected in the transition probabilities of the implicitly defined Markov chain. The precise values of these transition probabilities are determined by the distributions denoted by data objects that are derived from instances of class `Random-Generator`. A library of data classes for representing different types of distributions has been developed to assist the designer in expressing probabilistic behaviour.

The actual transformation of a POOSL model into a Markov chain involves three steps: assuming maximal progress, resolving non-determinism and shifting action and time information into states. The second step is concerned with resolving non-deterministic choices that exists between next possible action transitions. Resolving non-determinism is a common aspect of discrete-event simulation and is usually performed by some scheduler. Different schedulers may have different policies for choosing between the next possible action transitions and hence, different performance results may be obtained. Not resolving non-determinism therefore gives rise to a collection of results for a performance metric. Only after resolving non-determinism, a single result is obtained for the performance metric. To ensure obtaining realistic performance results with POOSL, the tools for executing POOSL models resolve non-determinism in a fair way by using a uniform distribution over the next possible action transitions.

Extending a POOSL model with variables and behaviour to evaluate performance properties yields the definition of reward functions for the underlying Markov chain. Specification of the exact reward values for the states of this Markov chain involves the explicit assignment of these values to the corresponding variables in the model. Such assignment is usually specified on basis of the occurrence of a certain event. Hence, for each variable there is an implicitly defined conditional reward function that indicates whether an assignment to the variable occurred. Based on the reduction technique, long-run average performance metrics can therefore be estimated by using the classical Markov-chain based performance analysis techniques and the algebra of Confidence Intervals. The estimation of long-run average performance metrics that depend on the duration of reward values can rely on the use of statement **`currentTime`** to determine this duration.

Straightforwardly applying the classical Markov-chain based analysis techniques in practice suffers from the difficulty of identifying a (relevant) recurrent state of the Markov chain. Two approaches are proposed for predefining a recurrence condition that enforces the beginning of a cycle of (nearly) independent behaviour. The first approach utilises that performance metrics are commonly defined in relation to a specific component of the system. In case the assignment of a specific reward value for such a performance metric is known to re-occur, this assignment may serve as recurrence condition for the technique of regenerative cycles. Although this approach is theoretically sound, the requirement of having knowledge about the behaviour of the system hinders its applicability. The second approach is referred to as the batch-means technique. It approximates the independence of regenerative cycles based on a sufficiently large fixed cycle length. Next to the difficulty of determining a suitable cycle length (batch size), the batch-means technique suffers from the fact that the Markov chain underlying a POOSL model is in general not covariance-stationary during the warm-up period of the simulation. Despite these theoretical deficiencies, the batch-means technique is generally applicable.

To assist the designer in constructing monitors for estimating complex long-run average performance metrics, a library of POOSL classes has been developed. These performance monitor classes enable accuracy analysis of four types of long-run averages based on implementing the algebra of Confidence Intervals. The estimation of such performance metrics can be performed by using either the technique of regenerative cycles with a certain local recurrence condition or the batch-means technique. The quality of the estimation results obtained with the performance monitor classes has been assessed with an experiment. The experiment involved analysing the coverage of the Confidence Intervals underlying the four long-run average performance metric types. The experiment showed that for the current implementation of the performance monitor classes, the technique of regenerative cycles is preferable over using the batch-means technique. The coverage results obtained with the batch-means approach may however improve considerable when adding a technique for disregarding reward values obtained during the warm-up period to the monitor classes.

Chapter 4

Extending the SHE Method for Performance Modelling

Mathematically founding performance analysis with a modelling language such as the Parallel Object-Oriented Specification Language (POOSL) supports obtaining credible performance results. The use of such an expressive modelling language facilitates the construction of intuitive models that abstract from many implementation details. However, due to the expressive power, it is sometimes unclear how the available primitives can be combined such that the behaviour of a system is modelled adequately. To assist the designer in constructing models with a certain modelling language, design methods provide guidelines. These guidelines include templates for adequately modelling typical aspects of hardware/software systems.

The modelling language POOSL was developed as a part of the system-level design method called Software/Hardware Engineering (SHE) [145]. Originally, SHE mainly provided guidelines for capturing the concepts and requirements for a system based on object-oriented analysis techniques and several diagram types. In [57], mathematical techniques for the verification of correctness properties and a few guidelines for their application in the context of POOSL are discussed. Furthermore, [57, 58] and [28] introduced tools for the construction, validation and execution of POOSL models. This chapter focuses on extending the SHE method with guidelines that assist in constructing adequate performance models. To this end, the next section first introduces a framework for structuring the exploration of design alternatives with SHE. This framework allows integrating the performance modelling guidelines with the existing techniques, guidelines and tools.

An important aspect of the SHE method is to derive POOSL models from several diagrams that capture the concepts and requirements for a system. An emerging standard modelling language for expressing this information is the Unified Modelling Language (UML) [149]. To allow replacing the original diagrams, section 4.2 discusses a profile for stereotyping UML diagrams in the context of SHE. Such stereotyped UML diagrams accelerate the construction of POOSL models after applying the object-oriented analysis techniques of [145, 185]. To this end, section 4.2 also discusses deriving POOSL models from the stereotyped UML diagrams.

Section 4.3 introduces some generally applicable modelling patterns that allow the construction of adequate performance models with POOSL. In addition, guidelines are provided for extending a POOSL model with monitors, which ensure preserving the intuitive understanding of the working of a system from the model. Finally, section 4.3 presents some guidelines for validating POOSL models against the stereotyped UML diagrams that capture the concepts and requirements for a system. Although being centered around the use of POOSL, most of the provided guidelines are expected to be useful in the context of other modelling languages as well.

4.1 Exploring Design Alternatives with SHE

As discussed in section 1.1, designing hardware/software systems is usually subdivided in a number of design phases. Each phase intends to refine the design by incrementing the amount of detail with which the question of how to realise the desired functionality such that the non-functional properties are satisfied is answered. In general, several options can be proposed for doing so and hence, each phase entails exploring such design alternatives. This section elaborates on developing design alternatives and evaluating their properties during system-level design with SHE.

4.1.1 Formulation, Formalisation and Evaluation

The proposed framework for developing design alternatives and investigating their properties with SHE is depicted in figure 4.1. It involves the stages of formulation, formalisation and evaluation. These three stages are discussed in more detail below.

Formulation The development of a feasible design starts with brainstorm sessions and/or discussions on possible concepts for realising the desired system. Such concepts are proposed based on their potential of satisfying the functional and non-functional requirements. Design experience and design decisions taken in the past affect the outcome of the discussions. For example, by immediately rejecting concepts that are thought not to lead to satisfying the requirements. On the other hand, designers may come up with alternative concepts that are believed to form the basis of a feasible design. The outcome of the discussions is an agreement on which ones of these design alternatives should be investigated further.

Since brainstorm sessions and discussions have a rather unstructured and undocumented character, a process of formulation is needed. *Formulation* concerns documenting proposed design alternatives and accompanying requirements to empower more detailed discussions on them during later stages of the design. Proposed concepts for realising the desired functionality are formulated in concept models. A *concept model* documents the desired behaviour of a system using diagrams and texts. It may include for example a sketch of a system architecture together with a specification of the order in which certain information must be exchanged between components. Proposing concepts for realising the desired functionality may imply additional requirements for the system or its components. An example of such a requirement is a maximum on the process time of a specific component to ensure satisfying some overall latency requirement. The requirements that have to be satisfied by a design are documented as *desired properties*, again using diagrams and texts. Notice that the desired properties basically reflect the design issues that are to be addressed. The result of the formulation stage is a structured but informal documentation of concepts and accompanying requirements as concept models and accompanying desired properties, which present the deliverable at milestone A.

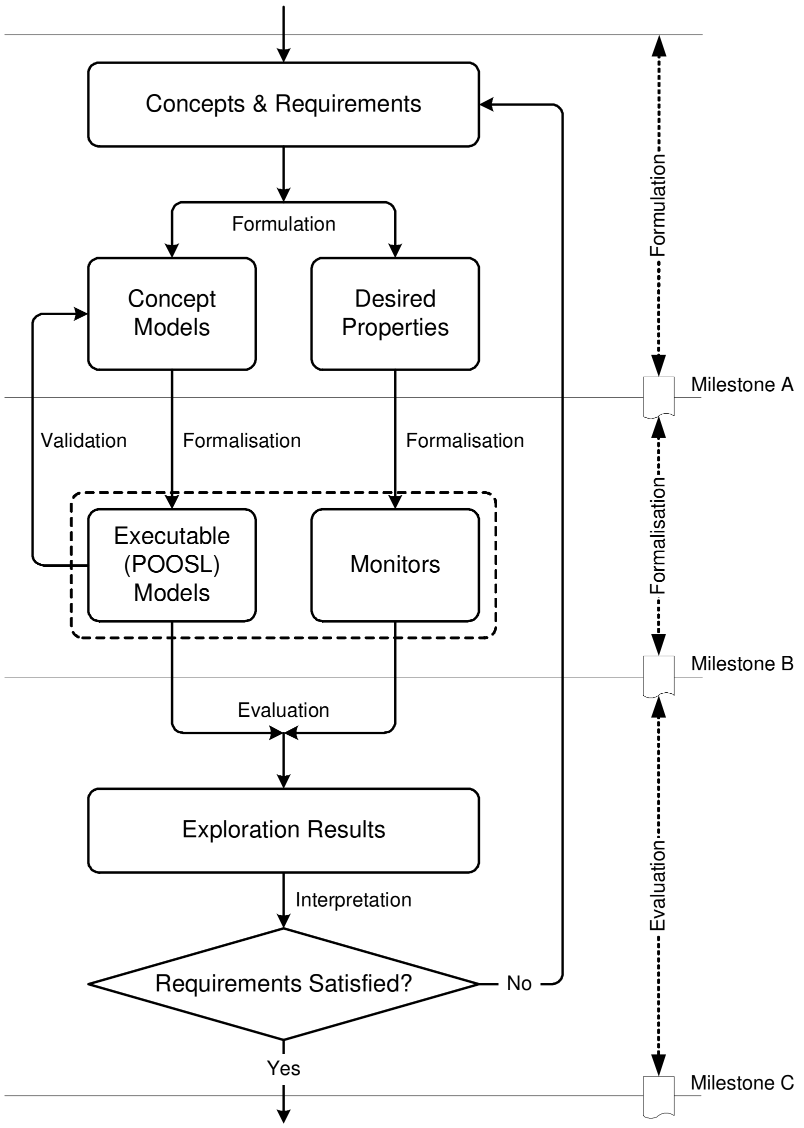


Figure 4.1: Framework for exploring design alternatives with the SHE method.

The SHE method provides support for the formulation stage in the form of numerous guidelines for applying object-oriented analysis techniques, see [145, 185]. Although any form of diagram or any language (plain English for example) can in principle be used to denote concept models, it is advised to apply the UML profile for SHE discussed in section 4.2. Using UML diagrams that are stereotyped in accordance with this profile accelerates deriving POOSL models in the formalisation stage (see below). Notice that it is still required to include a textual explanation of how the UML diagrams should be interpreted. Similarly as for concept models, the accompanying desired properties can be explained in any language. It is however recommended to annotate the UML diagrams regarding the involved concept model. Such annotations can express both functional and non-functional desired properties. Future research includes an investigation on using the recently developed profile for specifying schedulability, performance and time in [130, 154] to annotate UML diagrams.

Formalisation Concept models developed as a result of the formulation stage can only be used for (non-automated) verbal reasoning about the properties of proposed design alternatives. Verbal reasoning is however less suited for properly analysing the properties of interest due to the difficulty of taking the effects of all relevant aspects into account. To enable an automated and more credible evaluation, concept models have to be *formalised* into *(formal) executable models* that are amenable to the automated application of mathematical analysis techniques. SHE provides the modelling language POOSL to construct such executable models. A smooth formalisation of concept models into POOSL models is supported by the UML profile for SHE. Section 4.3.1 presents several modelling patterns for the construction of POOSL models, which adequately capture essential performance-related aspects.

Concept models are expressed using a (modelling) language with an informally defined semantics, whereas executable models are required to be expressed with a formal modelling language. Hence, deriving POOSL models from concept models can not be automated based on mathematical techniques. To ensure that a POOSL model appropriately captures the concepts for realising the desired functionality, a process of *validation* is needed. Section 4.3.3 presents some guidelines that assist the designer in validating POOSL models against concept models documented with the UML profile for SHE.

Next to formalising the concept model, the desired properties have to be formalised into mathematical formulae that are tractable by the provided analysis techniques. Such formulae lead to the definition of *monitors*. The SHE method supports analytical computation[1] as well as simulation-based estimation of both correctness and performance properties. Computation of correctness properties is based on defining monitors separately from POOSL models as presented in [56, 57]. Such a model-checking approach is also applied in case of computing performance properties according to [180]. Simulation-based estimation of performance properties is based on the framework for reflexive performance analysis discussed in section 3.1, which involves explicitly extending a POOSL model with monitors. In this case, monitors for analysing common long-run average performance metrics can be based on using the library classes presented in section 3.3.1. Section 4.3.2 discusses how POOSL models can be extended with monitors without decreasing their intuitiveness.

The result of the formalisation stage is a structured and formal documentation of the

[1]Remark that computation of correctness and performance properties according to [56, 57] and [180] respectively first requires to construct the appropriate mathematical structure from the POOSL model. In case of computing performance properties, this mathematical structure is the implicitly defined Markov chain, see also section 3.1. Furthermore, the correctness or performance properties of interest must first be expressed in the temporal logic of [57] or the formalism of temporal rewards in [180] respectively.

proposed concepts and accompanying requirements as executable (POOSL) models and monitors, which presents the deliverable at milestone B (see figure 4.1).

Evaluation The final stage is concerned with *evaluating* the actual properties of the proposed design alternatives and making design decisions. Correctness properties are evaluated based on the techniques discussed in [57], while long-run average performance metrics are evaluated according to the techniques presented in chapter 2.

Based on the *exploration results*, it can be concluded which ones of the proposed design alternatives satisfy the requirements. Design alternatives satisfying all requirements are designated to be feasible, others are infeasible. The deliverable at milestone C documents the evaluation results of all design alternatives and the foundation for judging the designs to be feasible or infeasible. For infeasible designs, the deliverable at milestone C clearly pinpoints why the requirements are not satisfied. In case no feasible design is found, improvements for the original concepts or new concepts for realising the desired functionality have to be developed[2]. The information on why the requirements were not satisfied may then help in improving the original concepts or developing new concepts. Notice that all three stages have to be repeated in this case to investigate whether the proposed changes indeed result in satisfying the requirements. On the other hand, if there are feasible designs, it can be concluded which alternative is favorable. This alternative can then be selected as the one that will be used to realise the system. This design decision and why the specific alternative is favourable is documented in the deliverable at milestone C. The information on the selected alternative documented in all three deliverables suffices for refining the design of the system towards a realisation.

Tool Support The SHE method is not yet accompanied with a full tool suite. Currently, SHE relies on the use of commercial tools for drawing stereotyped UML diagrams during the formulation stage[3]. The construction of POOSL models based on such UML diagrams during the formalisation stage is performed by hand. Constructing and validating POOSL models is supported with the graphical SHESim tool [58]. SHESim is also used when explicitly extending a POOSL model with monitors. In the evaluation stage, the textual Rotalumis tool [28] for high-speed execution of POOSL models can be used. Both SHESim and Rotalumis execute a POOSL model unambiguously based on a mathematical structure that is derived from the formal semantics of the model. Appendix B discusses several aspects of the SHESim and Rotalumis tools in some more detail. Future research includes an investigation on a tool that integrates support for all three stages of the SHE method.

4.1.2 Discussion

The flow of steps performed during the exploration of design alternatives as depicted in figure 4.1 is an idealisation of what may actually happen in practice:

- Formulating the concepts and accompanying requirements may involve several iterations. The brainstorm sessions and discussions held during the for-

[2]Sometimes, it is possible to adapt the requirements such that the original concepts can still be used.

[3]The UML diagrams in this thesis are drawn with a drawing tool and not with a UML tool.

mulation stage lead to proposing concepts for realising the desired system. Actually documenting such concepts and their accompanying requirements may trigger the need to elaborate on some more details about their working and expected effects. This is the case, for example, when discovering unclarities, inconsistencies or missing information in the initial specification of the system that is to be designed. Designers may start additional discussions to reveal such information before finalising the formulation of the concepts and accompanying requirements as concept models and desired properties.

- The formalisation of a concept model into a POOSL model and their validation may involve some iterations because POOSL models are usually developed in an incremental way. After constructing an initial version of a model, its validation may expose several aspects revealing that the concept model is not represented appropriately. These aspects trigger the development of an improved version of the POOSL model, which must then be validated again.

- Similar to validating POOSL models against concept models, formalising the desired properties into mathematical formulae involves validating whether these formulae appropriately represent the desired properties. Moreover, it must be checked whether the resulting monitors gather information about the desired properties in an appropriate way. In case of having the intention to apply the library classes for evaluating long-run averages presented in section 3.3.1, it is necessary to ensure that a property of interest is indeed one of the long-run average types for which the library provides monitors.

- The framework for exploring design alternatives as it is depicted in figure 4.1 may suggest that the formulation and formalisation of concepts into POOSL models can be performed in parallel with the formulation and formalisation of requirements into monitors. It is however not recommended to do so because these processes dependent on each other. Moreover, the formulation and formalisation of the concepts is usually much more difficult than formulating and formalising the requirements. Hence, formulation of the requirements into desired properties is to be performed after formulating the concepts into concept models. Similarly, the formalisation of desired properties into monitors must be performed after formalising a concept model into a POOSL model. Remark that the formulation and formalisation stages for different design alternatives can of course be performed in parallel, although it is sometimes possible to capture several design alternatives in one parameterised model.

- During the construction of a POOSL model, it is possible to already evaluate certain properties before actually entering the evaluation stage. When doing so and if the prematurely obtained evaluation results are unrealistic, then the model either does not capture the concepts for realising the desired functionality appropriately or inadequate abstractions are made. It is advised to evaluate some easy-to-check properties already during the formalisation stage to help improving the model (see also section 4.3.3). An example of an easy-to-check property is that a certain throughput does not exceed the available bandwidth.

- Evaluating easy-to-check properties during the formalisation stage may already reveal deficiencies in the originally proposed concepts. Such deficiencies may lead to immediate improvements for a POOSL model without adapting the concept model. In case such improvements are made, the deliverable at milestone B must contain the detected deficiencies and the applied solutions.

Separation of Application and Architecture Various methods for exploring design alternatives during system-level design clearly separate the application from the architecture in accordance with the Y-chart approach presented in [98, 99]. Two examples of such methods are the design space exploration framework in [174] and the Artemis framework [137]. This thesis does not make the conceptual choice of whether to separate application from architecture because both approaches are supported. Separating application from architecture is considered to be a set of additional guidelines for performing the actual formulation of concepts into concept models and formalisation of concept models into POOSL models. In many of the case studies performed with the SHE method, the application and architecture are simply intertwined in a single model, see for example [171, 165, 172, 80, 82] and chapter 5. In [184, 183], several guidelines are presented for clearly separating application and architecture when developing performance models with POOSL.

4.2 UML Profile for SHE

Documenting the proposed concepts for realising the desired functionality and the accompanying requirements involves the use of diagrams. To assist the designer in this formulation process, [145] defined 5 diagram types for the SHE method, which perfectly match the use of POOSL when constructing executable models. Around the same time, UML was introduced [149]. The first versions of UML provided diagram types that are particularly suitable for documenting software systems. The UML community worked on making it suitable for system design as well, which resulted in the development of UML 2.0 [26]. This latest version of UML extends the original UML with primitives that enabled defining a total of 12 diagram types[4]. Four of these diagram types represent views on the static structure of a system; five reflect views on the dynamic behaviour of a system and three express ways to organise and manage (the software of) systems. Several UML diagram types of especially the first two categories are very similar to the original SHE diagram types. This section investigates the replacement of the SHE diagram types with UML diagram types.

Models expressed in UML are in principle not executable due to the incompleteness of the behavioral semantics of UML. Although UML defines actions as the basic unit of behaviour, which can be arranged into activities, their exact semantics is not defined. The UML community has made an effort to define an (informal) action semantics [69] but it still leaves too much semantical freedom for executing models. To extend or restrict the meaning of its primitives, UML offers the mechanism of *profiles*, which are based on *constraints*, *tagged values* and *stereotypes* [149, 26]. Profiles can therefore be used to adapt UML in accordance with a host language like SDL or SystemC for making UML models executable. Using UML 2.0 as starting point, this section presents the main ideas for developing a UML profile for the SHE method (where the host language is POOSL), which were initially introduced in [167].

[4]This thesis adopts the diagram types as defined by the Object Management Group (OMG).

4.2.1 Data, Process and Cluster Classes

Similar to a suggestion in [130], SHE distinguishes two types of resources for hardware/software systems. *Active resources* are components of a system that may take the initiative to perform certain functionality without involving any other resource. Active resources can often be arranged according to a hierarchical structure. For example, a system may incorporate several concurrently operating interconnected components that are composed of smaller concurrent units, etcetera. *Passive resources*, on the other hand, present information or data that is generated, exchanged, interpreted, modified or consumed by other resources. Figure 4.2 depicts[5] examples of the classes that the SHE method distinguishes to specify different resource types.

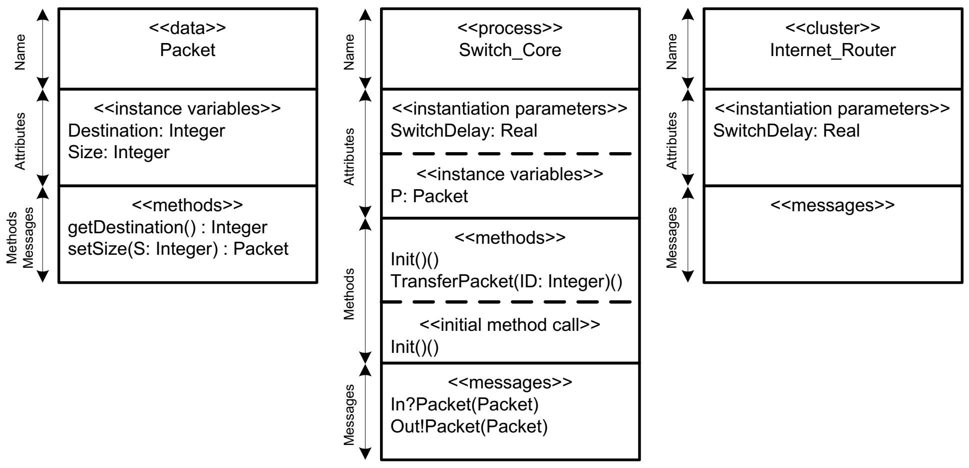


Figure 4.2: Illustration of the class symbols for SHE.

Data Classes SHE offers *data objects*, which are instances of *data classes*, for modelling passive resources. Figure 4.2 shows that the class symbol for data classes consists of three compartments. The stereotype `<<data>>` in the name compartment denotes that a class is a data class. The attributes of a data class are specified as *instance variables*, which is indicated by stereotyping the corresponding compartment with `<<instance variables>>`. The behavior of data objects is described with *(data)*[6] *methods*. Execution of such a method is initiated by sending an equally named message to the data object. Upon completion of executing the method, the result of some calculation or the data object itself is returned. The class symbol includes the headers of all methods (with the declaration of input parameters and the result type) in the methods/messages[7] compartment, which is stereotyped with `<<methods>>`. The header of the *primitive* methods for the predefined data classes `Object`, `Boolean`, `Integer`, `Real`, `String`, `Nil`, `Array`, `FileIn`, `FileOut` and `RandomGenerator` (see appendix B) are preceded by stereotype `<<primitive>>`.

[5]The labelled arrows are not a part of the actual class symbols but indicate the compartment names.
[6]Prefixing methods by the word data or process is omitted when this is clear from the context.
[7]Notice that UML uses the term *operations* compartment. A different terminology is introduced for SHE in order to deal with the differences between methods and messages for process (and cluster) classes.

Data classes must inherit from a *single* other data class (except for data class `Object` from which all other data classes eventually inherit). In *class diagrams*[8], such a relation is drawn as a generalisation/specialisation relation using a triangle at the border of the superclass. Based on the stereotypes defined for the class symbol and the inheritance relation, POOSL skeleton code can be derived easily for any data class (this may be automated by a future tool). The remainder of formalising a data class with POOSL then concerns defining the actual behaviour of all (non-primitive) methods.

Process Classes To model the basic (non-composite) active resources of hardware/software systems, SHE provides *process objects* or *processes*. The precise difference between basic and composite active resources depends on the modelling approach. Sketches of possible system architectures drawn during brainstorm sessions and discussions on concepts for realising the system reflect the (hierarchical) way in which a system is composed of components. A suitable modelling approach is often to define process classes for those active resource that are drawn as non-subdivided blocks in such sketches. Notice that these components may still expose concurrent behaviour.

As shown in figure 4.2, the name compartment of the class symbol for process classes is stereotyped with `<<process>>`. The attributes of process classes include *instance variables* and *instantiation parameters*. The difference is that instantiation parameters allow parameterising the behavior of a process at instantiation. The attributes compartment in the class symbol separates the instance variables from the instantiation parameters with a dashed line and stereotypes the sub-compartments with `<<instantiation parameters>>` and `<<instance variables>>`.

The behaviour of processes is described with *(process) methods*, which may include the specification of sending or receiving messages (see also appendix B). The start behavior of a process is defined by the *initial method call*. The methods compartment of the class symbol includes the specification of both the methods and the initial method call. They are separated from each other by a dashed line and the sub-compartments are stereotyped with `<<methods>>` and `<<initial method call>>` respectively. The `<<methods>>` sub-compartment includes the headers of all methods (with the declaration of input and output parameters). The `<<initial method call>>` sub-compartment specifies the *actual* call of one of those methods.

To describe the services available from classes, UML defines two kinds of interfaces. *Provided interfaces* describe services offered by a class, while *required interfaces* denote services that must be provided by other classes in order to operate properly [26]. A set of provided and required interfaces can be grouped into a *port*. In the UML profile for SHE, the UML concept of ports is matched to the concept of the ports through which processes (or clusters) may communicate messages[9]. Correspondingly, required interfaces are related to messages that processes (or clusters) can *send*, while provided interfaces are related to messages that can be *received*. An important difference between interfaces and messages is that messages are not directly coupled to process methods (services or operations), see also appendix B. A separate compartment in the class symbol, which is stereotyped with `<<messages>>`, specifies

[8]Class diagrams are sometimes also called *static structure diagrams*.

[9]Consequently, services provided or required by data classes are not expressible with ports and interfaces. Such information can however be expressed by for example relations, dependencies or associations.

all messages that can be sent or received. Such a specification includes the actually[10] involved port, the symbol ! or ? for message send (required interface) and message receive (provided interface) respectively, the message name and parameters.

Process classes may have an inheritance relation with a *single* other process class. In class diagrams, this is drawn as a generalisation/specialisation relation. Based on the stereotypes defined for the class symbol and the inheritance relation, POOSL skeleton code can be derived easily for any process class (this may be automated by a future tool). Notice that the information in the messages compartment of the class symbol suffices for deriving both the *port interface* and *message interface* in POOSL (see also appendix B). Hence, the remainder of formalising a process class concerns defining the actual behaviour of the methods in the `<<methods>>` sub-compartment.

Cluster Classes To model composite active resources, SHE provides clusters. *Clusters* are instances of *cluster classes* and group a set of processes and clusters (of other cluster classes). Similar ways of grouping are also allowed in UML, but an important restriction is that a cluster does not extend the behaviour of its constituents[11]. In addition, cluster classes cannot inherit from any other class. The need for defining a cluster class emerges for example from aggregation or composition relations in class diagrams between classes that represent active resources. Figure 4.2 shows the use of stereotype `<<cluster>>` to denote that a class is a cluster class. The attributes of cluster classes include only instantiation parameters, which are indicated in the corresponding compartment with stereotype `<<instantiation parameters>>`. Messages of encapsulated processes (or clusters) that may pass the boundary of an instance of the cluster class through its ports are specified in the messages compartment, which is stereotyped with `<<messages>>`. The information in the message compartment suffices for deriving both the port interface and message interface in POOSL. The remainder of defining a cluster class involves specifying the composition of processes and clusters using instance structure diagrams, see section 4.2.2.

4.2.2 Behaviour of Objects

Data Methods Data methods basically reflect algorithmic operations on the passive resources modelled by the data class. UML and SHE provide only little support for specifying the effect of such algorithmic operations. The need for a data method emerges for example from relations with data classes in class diagrams other than generalisation/specialisation or aggregation. Although proper naming of a method and its input parameters should give an idea of what the method is about, it is advised to use *notes* for describing the *pre* and *post conditions* of a method. Such information should at least be documented in the deliverable at milestone A of the SHE method. Notice that this deliverable should only contain the effect of a method and not how the method's behaviour is to be formalised with expressions in POOSL.

Process Methods UML 2.0 allows expressing information on the behaviour of active resources in 9 different diagram types: *class diagrams*, *object diagrams*, *component diagrams*, *deployment diagrams*, *use case diagrams*, *sequence diagrams*, *activity diagrams*,

[10]POOSL supports dynamic port naming, see also appendix B.

[11]Consequently, the class symbol for cluster classes do not include a methods compartment.

collaboration diagrams and *statechart diagrams*[12]. The use of class, object, component or deployment diagrams is however limited to representing a certain static view on the behaviour. The (dynamic) flow of behaviour to perform can only be captured by diagrams of the other 5 types. Guidelines on when to use which diagram type for what purpose can be found in many text books on UML such as [149, 63, 143] but also in SHE-related publications like [144]. If certain information is expressed in different diagram types, then the involved diagrams are presumed to be consistent. Commonly, UML tools are capable of performing the necessary consistency checks.

As discussed in section 3.1.1, the tools for executing POOSL models interpret such models as being closed. Consequently, the SHE method requires modelling the environment of a system as well [145]. UML provides *actors* to specify the interaction of a system with its environment in use case diagrams. The concept of actors is however not adopted for the SHE method since actors are basically *active resources* (of the environment), which can therefore be modelled with processes and clusters. SHE does provide a similar concept as *use cases*, which is called *scenarios* [145, 58]. Different sets of (consistent) diagrams may be used to formulate the behaviour that is to be exposed for different scenarios. Constructing a POOSL model from these diagram sets involves unifying the different behaviours of a passive or active resource that take part in various scenarios into a single data object, process or cluster.

An important aspect of active resources in hardware/software systems is that they exchange information (passive resources). The SHE method especially adopts the use of sequence diagrams and collaboration diagrams as a means to formulate the communication of messages between processes and clusters. An important restriction is however that communicating objects (processes and clusters) cannot be created or destructed. Next to parameterising messages with data objects, it is allowed to annotate messages with a note specifying a condition that must be satisfied in order to perform the actual communication. Such notes are stereotyped with `<<condition>>` and imply the use of a condition for the receive statement in the resulting POOSL model. Messages can also be annotated with the time at which they should be exchanged. Finally, [145] defined various message types together with special symbols. These are added[13] to the UML profile for SHE. Since POOSL only includes a statement for synchronous message passing (see appendix B), modelling patterns are defined for formalising the other message types, see [144].

Statechart diagrams and activity diagrams are very suitable for formulating the dynamic behaviour of processes. Basically, SHE supports these diagrams in the way they are defined originally, but with some restrictions and extensions. The interpretation of the diagram elements in the context of SHE can best be explained by relating them to POOSL, which also involves clarifying some terminological mismatches:

- Actions in UML do not match with actions in POOSL. What is called an *action* in UML matches *only* with a non-communication statement that concerns atomically evaluated expressions (The syntax of such statements is given by the first line of table B.2 in appendix B). These statements can only imply internal or fix action transitions in the timed probabilistic transition system defined

[12]These are all the diagram types in the first two categories indicated at the beginning of section 4.2.

[13]Instead of adding the symbols introduced in [145] to the UML profile as proposed here, one could also define a stereotype for each message type.

by an activity or process (see also section 3.1.1). *Activities* in UML are matched with activities in POOSL and concern a sequence of statements. Activities may relate to any sequence of action *and* time transitions in the transition system. Hence, *activity states* in UML match with process statements in POOSL.

- States in UML are related to the state of a process during the execution of a POOSL model. Hence, a *state* refers to a configuration in the timed probabilistic labelled transition system defined by an activity or process. Consequently, the *initial* state in UML diagrams refers to the initial configuration for such a transition system. A *final* state in UML diagrams refers to a configuration in the transition system from which no transitions to other configurations than itself can be performed. Hence, it is allowed to have multiple final states for a single activity or process. A *composite* state can be formalised by defining a method that abstracts from the statements grouped by that method. *Concurrent* composite states UML diagrams can be formalised by using the **`par`**-statement.

- *Transitions* in UML are similar to the transitions in the timed probabilistic labelled transition system of an activity or process. Transitions are either action transitions or time transitions. A restriction is that if one specifies transitions of both types to leave from a state, the action transitions must reflect the sending or receiving of messages (see also section 3.1.1). Time transitions model the passage of time and are annotated with a positive real number preceded by stereotype `<<delay>>`. Time transitions imply a **`delay`**-statement in POOSL. In case only action transitions leave from a state[14], they can be annotated with a real number between 0 and 1 preceded by stereotype `<<probability>>` specifying the probabilities with which the transitions are performed. Notice that the sum of these probabilities must equal 1. If, in case of multiple outgoing action transitions, no probabilities are specified, then the choice of which transition will be performed depends either on the *event* triggering the transition or it is chosen non-deterministically. Examples of possible events are the sending or receiving of a message. In UML, transitions may also be triggered based on *guard conditions*, which are formalised in POOSL using guarded commands.

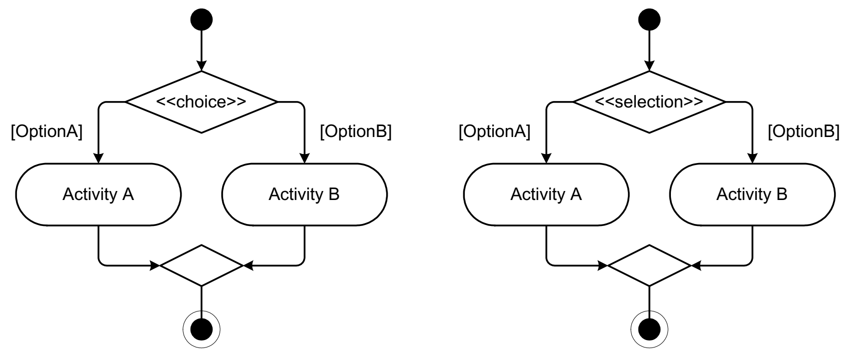
- UML enables to specify that activities, which may be formalised with separate process methods in POOSL, perform their behaviour concurrently. This involves the use of *fork transitions* in UML, which can be formalised with the **`par`**-statement if it concerns the creation of concurrent activities[15]. In this case, *joint transitions* match the termination of a **`par`**-statement. However, if fork and joint transitions denote synchronisation moments between objects (processes or clusters) in different *swimlines*, then their formulation requires using message send and receive statements (they can be conditional, in which case they are to be annotated with stereotype `<<condition>>`).

- UML provides *branches* to denote choices between performing transitions to alternative activities or states. For the SHE method, such branches are categorised in (probabilistic) choices or (non-deterministic) selections. *Choices* are decisions between alternative activities and lead to the use of an **`if`**-statement in POOSL. In such case, the branch is stereotyped with `<<choice>>` and the

[14]It is not necessary in SHE to define guard conditions in case of multiple outgoing action transitions.
[15]Instead of concurrent activities, which was introduced for SHE, UML uses the term *concurrent threads*.

involved transitions may be annotated with a probability by using stereotype `<<probability>>`, similarly as above. Branches denoting a selection are stereotyped with `<<selection>>` and imply the use of a **`sel`**-statement. If a branch is a selection, then the choice of which alternative transition is performed is made non-deterministically. Notice that *merges* in UML now match with the termination of either an **`if`** or **`sel`**-statement.

Example 4.1 *To formalise deterministic choices between (mutual-exclusive) alternative transitions, both the* **`if`***-statement and the* **`sel`***-statement can be used. For both cases, the same guard conditions must be specified in for example activity diagrams, see figure 4.3. Notice however that using the* **`if`** *may result in a different behaviour then when using the* **`sel`***-statement because of their semantical differences:* **`if`***-statements are evaluated only once, whereas the guarded commands in the* **`sel`***-statement are evaluated until one of them evaluates to* **`true`***. Using the stereotypes* `<<choice>>` *and* `<<selection>>` *distinguishes these interpretations.*



```
if OptionA then
   ActivityA()()
else if OptionB then
   ActivityB()()
fi fi.
```

```
sel
   [OptionA] ActivityA()()
or
   [OptionB] ActivityB()()
les.
```

Figure 4.3: Two ways to specify deterministic choices.

- SHE allows interrupting or aborting behaviour based on using the **`interrupt`** and **`abort`**-statements respectively in POOSL. Such preemptions can be specified in UML diagrams as *abnormal exits*. Abnormal exits causing the interruption or abortion of the behaviour reflected by an activity state or activity are stereotyped with `<<interrupt>>` and `<<abort>>` respectively.

- An action transition can represent empty behaviour by stereotyping it with `<<skip>>`. Such an action transition implies using the **`skip`**-statement.

Despite the above guidelines for using sequence, collaboration, statechart and activity diagrams in the formulation stage, deriving a POOSL model from the information in such diagrams is not a straightforward task. This is due to the fact that the elements of the diagrams cannot always be mapped in a one-to-one way on (a template of) POOSL statements. To illustrate this, consider the following simple example.

Example 4.2 *Statechart and activity diagrams may formulate (possibly non-terminating) repetitive behaviour. Such loops can be formalised in POOSL using the* **`while`**-*statement, tail-recursive method calls*[16] *or by methods that call each other in a tail-recursive way. Which formalisation approach should be chosen depends on the actual interpretation of a loop.*

Notice that the different ways in which stereotyped diagrams can be formalised with POOSL complicates automating the generation of POOSL code for process methods. As a consequence, validating a POOSL model against the information specified in sequence, activity, collaboration and statechart diagrams is of great importance.

Behaviour and System Specifications Originally, SHE introduced *instance structure diagrams* [145] to formulate (and formalise) the static structure of how processes and clusters are interconnected by channels (see also appendix B). Such channels are similar to *connectors* in UML. The need for channels originates from the possibility of communicating messages between active resources in any scenario. For clusters, SHE adopts both the *white* and *black box* views as defined in UML on instance structure diagrams, while only the black box view is supported for processes.

Various aspects of the information in instance structure diagrams are also reflected in class, object, component, deployment, collaboration and sequence diagrams. Class and object diagrams denote aggregation or composition relations between active resources, which are reflected in the use of clusters in instance structure diagrams. All the mentioned diagram types include information on the exchange of messages between active resources, which results in introducing channels. Instance structure diagrams are a unification of such information for active resources (instances of process and cluster classes) in all scenarios. Nevertheless, instance structure diagrams cannot be derived straightforwardly from class, object, component, deployment, collaboration and sequence diagrams because channels can be introduced in several ways in order to formalise the possibility of communicating messages. For example, one can use a separate channel for each message communicated between active resources or a single channel can be introduced for exchanging all those messages.

4.3 Performance Modelling with POOSL

Together with the object-oriented analysis techniques in [145, 185], the UML profile for SHE assists the designer in efficiently completing the formulation stage. Although this profile also provides some assistance in deriving POOSL models during the formalisation stage, guidelines for constructing adequate models for the specific purpose of performance analysis are missing. This section focuses on how adequate performance models can be constructed with POOSL and how such models can be extended with performance monitors without decreasing their intuitiveness. Moreover, validating POOSL models against stereotyped UML diagrams is discussed.

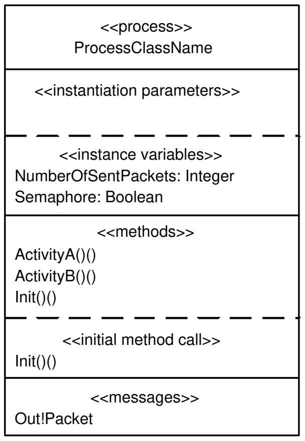
[16] A tail-recursive method call is only allowed if the involved method has no output parameters.

4.3.1 Modelling Patterns

During system-level design, developing abstract models that adequately represent implementation details, which are essential in the context of performance analysis, is difficult. This is due to the fact that not all of these details can be known in advance. Design experience and knowledge about the implementation of existing hardware/software systems are important ingredients for developing adequate models. *Modelling patterns* capture such knowledge in order to accelerate the exploration of design alternatives for future systems [177].

Modelling Pattern 1: Synchronising Concurrent Activities Modelling languages offer different ways to express concurrent behaviour [22, 172]. With the modelling concept of *synchronous concurrency*, concurrent behaviour is performed in a lock-step fashion, starting at synchronisation moments indicated by some master controller (a clock for example). Hence, the execution of functions is tightly coupled to the time. On the other hand, when using the modelling concept of *asynchronous concurrency*, concurrent behaviour is performed independently from each other (at their own speed). As opposed to synchronous concurrency, there is no master controller indicating moments at which all concurrent behaviours synchronise. Recall that concurrent activities and processes in POOSL behave asynchronously concurrent.

An important consequence of asynchronous concurrency is the difficulty of synchronising concurrent behaviour. Since there is no master controller that indicates such synchronisation moments, synchronisation has to be modelled explicitly by using communication. As argued in [58] and [166], a modelling language based on asynchronous concurrency should include primitives for *synchronous communication*. Synchronous communication can be accomplished by blocking the behaviour initiating



```
Init()()

NumberOfSentPackets := 0;
par ActivityA()() and ActivityB()() rap.
```

```
ActivityA()() |p: Packet|

...
p := new(Packet);
Out!Packet(p){Semaphore := true};
...
ActivityA()().
```

```
ActivityB()()

...
[Semaphore]
   NumberOfSentPackets := NumberOfSentPackets + 1;
...
Semaphore := false;
...

ActivityB()().
```

Figure 4.4: Example of synchronising concurrent activities.

the communication until some partner is ready to communicate. POOSL offers synchronous communication between processes based on message send and receive statements. Synchronisation of concurrent activities within processes can be accomplished by using guarded commands combined with the sharing of data objects as illustrated in figure 4.4. The concurrent activities specified by methods `ActivityA` and `ActivityB` are synchronised using the shared data object `Semaphore`. Incrementing the `NumberOfSentPackets` by `ActivityB` is postponed until `ActivityA` sends a packet as specified by the send statement (which blocks until another process is ready to communicate based on a matching receive statement).

Modelling Pattern 2: Data Abstraction An essential aspect of constructing models during system-level design is the abstraction from data. *Data abstraction* concerns using a coarser granularity of data units in a model of a system compared to the elementary data unit used in a realisation of that system (often being a bit or byte). For example, Internet packets actually consist of a number of bytes, which are transmitted individually over some communication medium. Abstraction of the Internet packets concerns considering such packets as elementary data elements, which have certain properties like size (expressed in bytes), destination and identifier. The data class `Packet` in figure 4.5 can be used as an abstraction of Internet packets.

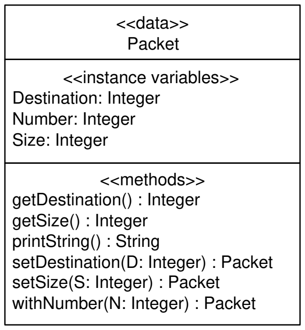


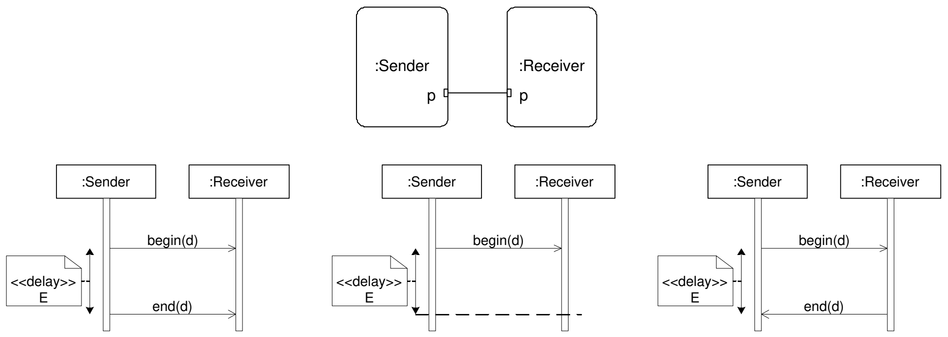
Figure 4.5: Example data class for modelling Internet packets.

The essential advantage of abstraction is the possibility of discarding implementation details like the way in which the content of an Internet packet is coded as a sequence of bytes. However, abstraction may imply difficulties in ensuring the adequacy of models. Due to using a coarse granularity of data units, the duration of generating, exchanging, interpreting, modifying or consuming data is also of a coarser granularity then for the realised system. Such timing information is often important for evaluating performance properties. Therefore, modelling patterns are needed that assist in developing abstract performance models that still are adequate.

Algorithmic operations on data are formalised in POOSL with expressions. The formal semantics of an expression specifies an *action* transition for the timed probabilistic labelled transition system that is defined by the corresponding activity or process (see also section 3.1.1). As a consequence, in case the actual duration of executing an algorithmic operation is relevant for obtaining credible performance results, this duration must be modelled explicitly. This can be achieved by using a **`delay`**-statement after the expression that formalises the algorithmic operation[17].

[17]Notice that it is required to have an accurate estimation of the duration available to enable obtaining

Adequately modelling the communication of data between processes is slightly more complicated. Let `d` denote some data object. Communicating `d` between two processes is formalised in POOSL using the message send and receive statements. The formal semantics of these statements specifies communication *action* transitions for the transition systems of the processes. To take the duration *`E`* of communicating `d` into account, waiting before performing other behaviour has to be formalised explicitly. Figure 4.6 shows three different approaches to accomplish this. Approaches A and C use communication actions to synchronise the behaviour of the `Sender` and `Receiver` processes both at the beginning and ending of communicating `d`. In approach B, the `Sender` and `Receiver` are synchronised at the beginning of the communication, while the termination of the **`delay`**-statements marks finalising it. For approaches A and C, only one of the processes must know *`E`*, whereas for approach B, both processes have to know *`E`*. Notice that *`E`* may have to be derived from the properties of `d` and the bandwidth of the involved communication medium.



Approach A		Approach B		Approach C	
Sender	Receiver	Sender	Receiver	Sender	Receiver

`...`
`p!begin(d);`
`delay``(E);`
`p!end(d);`
`...`
	`...`
	`p?begin(d);`
	`p?end(d);`
	`...`
		`...`
		`p!begin(d);`
		`delay``(E);`
		`...`
			`...`
			`p?begin(d);`
			`delay``(E);`
			`...`
				`...`
				`p!begin(d);`
				`p?end(d);`
				`...`
					`...`
					`p?begin(d);`
					`delay``(E);`
					`p!end(d);`
					`...`

Figure 4.6: Patterns for modelling the communication of data.

To evaluate the time-average occupancy of a storage resource, the duration of storing data in that resource must be taken into account. Consider a store-and-forward buffer for temporarily storing (fixed-size) packets. Figure 4.7 illustrates how the occupancy of a store-and-forward buffer in hardware may change in time, when transferring packets on a byte-by-byte basis. At t_a, the head of a packet (its first byte) has arrived and after some time $t_s - t_a$, the packet is received completely (including its last byte). From t_s on, the packet may be removed again from the buffer to forward it to another component. Assume that sending the head of the packet is started at $t_r \geq t_s$ and that it is completely sent at time t_e. Then $t_r - t_e$ equals the duration of sending the packet. Notice that $t_r - t_e$ does not necessarily equal $t_s - t_a$ if the packets are put into/removed from the buffer at different speeds.

credible performance results. Usually, such timing values are not available during system-level design. To cope with this general issue, designers often use timing values that are expected to be realistic.

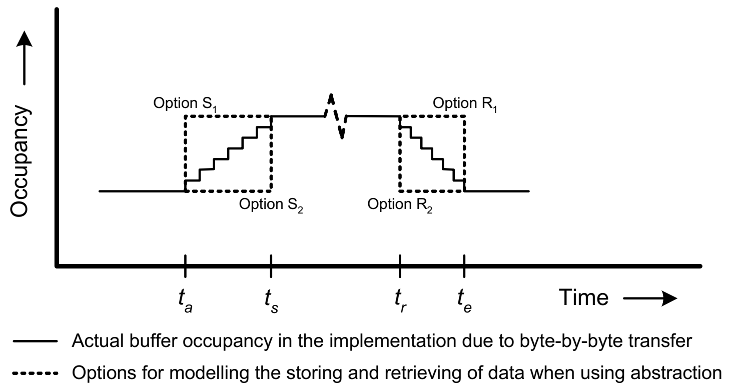
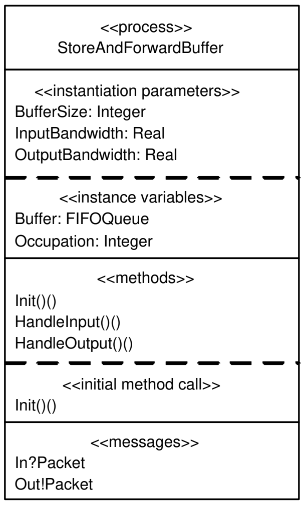

Occupancy
Option S1
Option R1
Option S2
Option R2
ta
ts
tr
te
Time
Actual buffer occupancy in the implementation due to byte-by-byte transfer
Options for modelling the storing and retrieving of data when using abstraction

Figure 4.7: Changes in buffer occupancy during storage of a packet.

The store-and-forward buffer can be modelled with a process that has two concurrent activities changing the properties (including the occupancy) of a shared data object `Buffer` (a FIFO queue). An *input handler* activity stores the received packets into the `Buffer`, while sending the packets after removing them from the `Buffer` is performed by an *output handler* activity. To obtain an adequate model for evaluating the time-average buffer occupancy, the output handler should only start sending a `Packet` after it was stored completely. This availability of a complete `Packet` in the `Buffer` can only be indicated by the input handler. Hence, the input and output handler activities should synchronise on this availability, which can be achieved by using the modelling pattern for synchronising concurrent activities, see figure 4.4.

The issue of adequately modelling the store-and-forward buffer when abstracting from the individual bytes of packets is the question at what moment during period $[t_a, t_e]$ a `Packet` should be stored into and removed from the `Buffer`. Figure 4.7 shows that there are two options (S_1 and S_2) for storing a `Packet` and two options (R_1 and R_2) for retrieving that `Packet`. One may choose to disregard the storage of parts of the packet during $[t_a, t_s]$ and $[t_r, t_e]$ (i.e., taking options S_2 and R_2). When doing so, then in case $t_s = t_r$, synchronising the input and output handlers at t_s yields the model to remove the `Packet` from the `Buffer` just after having it stored and hence, a unrealistic time-average occupancy would be obtained. If this result would be used to found a decision on dimensioning the buffer capacity, one may end up with a much higher probability of loosing data in the final implementation than was expected. To overcome this problem, it is often useful to develop conservative performance models. *Conservative* performance models give rise to slightly worse performance results compared to the true performance, whereas *optimistic* performance models lead to slightly better performance results than the true performance.

To develop a conservative performance model of the store-and-forward buffer, the use of storage resources during $[t_a, t_s]$ and $[t_r, t_e]$ should be taken into account. It should therefore be assumed that the buffer is occupied with the *complete* packet during the whole period $[t_a, t_e]$ (i.e., taking options S_1 and S_2 in figure 4.7). This modelling approach can be interpreted as follows. At t_a, the necessary amount of storage resources for a complete packet is reserved (if there are still enough storage resources available, otherwise the packet should be discarded), while actually putting the packet in the `Buffer` at t_s. Putting the packet in the `Buffer` may trigger



```
1   Init()()

2   Occupation := 0; Buffer := new(FIFOQueue);
3   par HandleInput()() and HandleOutput()() rap.
```

```
4   HandleInput()() |p:Packet, NotFull: Boolean|

5   In?Packet(p)
6      {NotFull := (Occupation < BufferSize);
7         if NotFull then
8            Occupaction := Occupation + 1 fi};
9   delay(p getSize / InputBandwidth);
10  if NotFull then Buffer put(p) fi;
11  HandleInput()().
```

```
12  HandleOutput()() |p: Packet|

13  [Occupation > 0] p := Buffer inspect;
14  Out!Packet(p);
15  delay(p getSize / OutputBandwidth);
16  {Buffer remove; Occupation := Occupation - 1};
17  HandleOutput()().
```

Figure 4.8: Handling the input and output of a store-and-forward buffer.

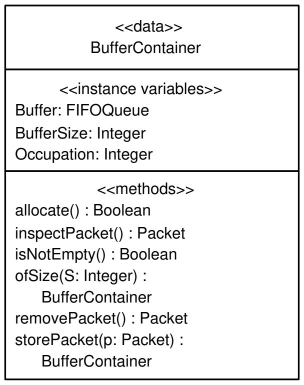
the output handler to start removing and forwarding it to another component. While doing so, the output handler does not deallocate the reserved storage resources. It does deallocate the storage resources at t_e since then the packet is sent completely.

Figure 4.8 shows how the proposed modelling approach can be formalised with POOSL. `BufferSize` denotes the maximum number of packets that can be stored in the buffer. After creating the `Buffer` and initialising its `Occupation`, the method `Init` creates the concurrent input handler and output handler activities, which are defined by methods `HandleInput` and `HandleOutput` respectively. Consider the input handler first. When receiving a packet `p`, it checks whether the `Buffer` is not yet full in line 6. Modelling approach B in figure 4.6 is used to model the duration of completely receiving `p`. This duration is derived from the size of p and the (fixed) bandwidth `InputBandwidth` of the communication medium over which `p` is received. After termination of the **`delay`**-statement in line 9, `p` is stored in the `Buffer` and `Occupation` is incremented (but only in case the `Buffer` was not yet full at the time of receiving the head of `p`). Tail-recursion is used to model the reception of the next packet. Now, consider the output handler. It synchronises with the input handler by means of the shared data object `Occupation`. Only in case `Occupation` is larger than 0, which denotes that at least one complete packet is available in the buffer, such synchronisation occurs. In case the guard condition holds, the first packet to be removed from the `Buffer` is assigned to `p` by a data method `inspect` and is then forwarded. Approach B in figure 4.6 is again used to model waiting until `p` is sent completely. The duration of sending `p` is derived from the size of p and the (fixed) bandwidth `OutputBandwidth` of the communication medium over which `p` is sent. Only after completely sending `p`, which is represented by the termination of the **`delay`**-statement in line 15, packet `p` is actually removed from the `Buffer` and `Occupation` is decremented in line 16.

Recall that the behaviour of concurrent activities is executed in an interleaved (non-deterministic) fashion (see also appendix B). To specify the necessary atomic execution of the expressions after the receive statement and in the statement of lines $5 - 8$ and also the statement in line 16, braces are used. They ensure that the other concurrent activity does not change the shared data objects `Buffer` and `Occupation` during the execution of these sequences of expressions.

Modelling Pattern 3: Container Data Classes As indicated in section 4.2.1, aggregation or composition relations between active resources result in defining cluster classes. Sometimes, aggregation and composition relations can also be identified for passive resources. Such relations imply data classes that model complex data structures. A *container* data class is a data class that models a number of the data structures exclusively used by instances of a single process class. Using *container* (data) objects (instances of container data classes) enables to write very compact code for process methods. It is advised to use container objects if it improves the intuitiveness of the process part of a model. The reason is that the process part reflects the most complex aspects of the model's behaviour and is therefore the most difficult part to interpret.

The use of container objects is illustrated for the store-and-forward buffer example of figure 4.8. The `StoreAndForwardBuffer` process class has an instance variable `Buffer` of data class `FIFOQueue` and a separate instance variable `Occupation` denoting the current occupancy of the `Buffer`. These aspects of the store-and-forward buffer can be modelled with an instance of the container data class in figure 4.9.



```
1    ofSize(S: Integer) : BufferContainer

2    Buffer := new(FIFOQueue);
3    BufferSize = S; Occupation := 0; return(self).
```

```
4    storePacket(p: Packet) : BufferContainer

5    Buffer put(p); return(self).
```

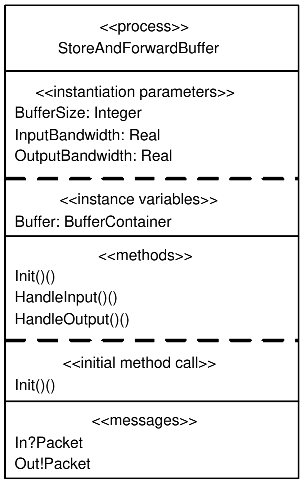
```
6    inspectPacket() : Packet

7    return(Buffer inspect).
```

```
8    allocate() : Boolean |NotFull: Boolean|

9    NotFull := (Occupation < BufferSize);
10   if NotFull then Occupaction := Occupation + 1 fi; return(NotFull).
```

```
12   removePacket() : Packet

13   Occupation := Occupation - 1; return(Buffer remove).
```

Figure 4.9: Container data class for modelling a store-and-forward buffer.

When using the container data class of figure 4.9, the model of the store-and-forward buffer in figure 4.8 can be replaced by the model in figure 4.10. Although defining container data class `BufferContainer` involves writing more code, the code of the new process class `StoreAndForwardBuffer` is more succinct compared to that in figure 4.8. Especially in case the processes model very complex behaviour, such succinctness is useful for understanding the working of the system from the model.



```
Init()()

Buffer := new(BufferContainer)
   ofSize(BufferSize);
par HandleInput()() and HandleOutput()() rap.
```

```
HandleInput()() |p:Packet, NotFull: Boolean|

In?Packet(p){NotFull := Buffer allocate};
delay(p getSize / InputBandwidth);
if NotFull then Buffer storePacket(p) fi;
HandleInput()().
```

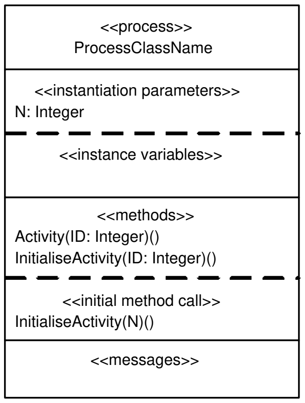
```
HandleOutput()() |p: Packet|

[Buffer isNotEmpty] p := Buffer inspectPacket;
Out!Packet(p);
delay(p getSize / OutputBandwidth);
Buffer removePacket;
HandleOutput()().
```

Figure 4.10: Succinctly modelling a store-and-forward buffer.

Modelling Pattern 4: Parameterising Instance Structures Hardware/software often include several active resources of the same kind. An example is a memory system that consists of a number of memory banks. In case it must be investigated how many memory banks should be used to satisfy certain performance requirements, it is useful to take the number of memory banks as a parameter of the system. Instance structure diagrams in POOSL statically interconnect processes and cluster. It is not possible to create processes, clusters or channels during the execution of a POOSL model. Consequently, modelling memory banks with (for example) a proces class requires to instantiate processes for all memory banks individually in the SHESim tool. This approach is very time-consuming if the number of memory banks is large.

The SHESim tool offers another way of parameterising instance structures. It allows instantiating multiple instances of a process or cluster class at once. Such a *multiple* is a single graphical representation for a fixed number of instances of a process or cluster class specified by the designer when instantiating it. Multiples in the SHESim tool are similar to *multi-objects* in UML. Notice that multiples do not extend the syntax or semantics of POOSL. As a consequence, the number of processes or clusters represented by a multiple cannot be determined with an expression (see also appendix B). Moreover, the number of processes or clusters represented by a multiple cannot change during the execution of a model. Hence, if a multiple should represent a different number of active resources in order to investigate design alternatives, that number must be explicitly changed in the instance structure diagram.

A more flexible way of parameterising instance structures is based on creating a number of *similar* concurrent activities. In figure 4.11, the initial method call dynamically creates `N` concurrent activities that are specified by method `Activity`. In line 3 of method `InitialiseActivity`, the activity with identity `ID` is created, while the **`if`**-statement checks whether another activity must be created. A disadvantage of this modelling approach is that the concurrent activities share the ports of the embedding process. To ensure obtaining an intuitive model in this case, the identity `ID` of the activity can be added as parameter to communicated messages.



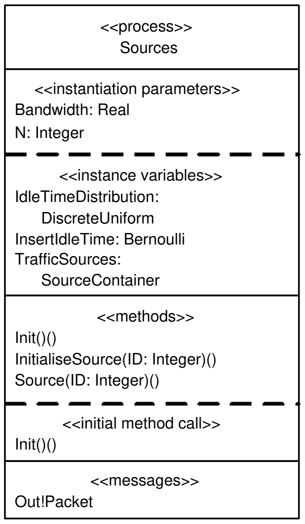
```
1  InitialiseActivity(ID: Integer)()

2  par
3     Activity(ID)()
4  and
5     if (ID > 1) then
6        InitialiseActivity(ID - 1)()
7     fi
8  rap.
```

Figure 4.11: Initialising `N` similar concurrent activities.

Example 4.3 *Recall the buffer system of example 1.1 on page 5. To investigate the feasibility of alternative combinations of output bandwidth factor M and buffer size S for different numbers (N) of Internet-traffic sources, it is useful to apply the modelling pattern of figure 4.11 to model the Internet-traffic sources. To this end, introduce a process class named `Sources` with a method `InitialiseSource` that is similar to `InitialiseActivity` in figure 4.11 and with a method `GenerateTraffic` that replaces the method `Activity`. For method `GenerateTraffic`, a modified version of the method in example 3.1 on page 62 is used in this example*[18]*. The resulting model is shown in figure 4.12.*

The modification of `GenerateTraffic` includes an input parameter for identifying the source activity and a parameter for sending packets via the shared port `Out` in line 17*. Instead of having an instance variable `PacketNumber` for process class `Sources` as in example 3.1 on page 62, the container object `TrafficSources` of a data class `SourcesContainer` (not shown) is used. Method `generatePacket` of this data class returns a new instance of data class `Packet` (see figure 4.5) and increments the number of generated packets, which is an instance variable of `SourcesContainer` in this example. To obtain variable-sized packets with different destinations for each source activity, the `Size` and `Destination` of a `Packet` generated by `generatePacket` are set according to certain probability distributions. Notice that all source activities share the `InsertIdleTime` and `IdleTimeDistribution` distributions (see also example 3.4), which ensures independency between the parallel sequences of random numbers (see also appendix B).*

[18]Remark that the methods `Init` and `GenerateTraffic` in figure 4.12 do not yield an appropriate generation of Internet-traffic. A more adequate model of Internet-traffic sources is discussed in [169].



```
Init()()

TrafficSources := new(SourcesContainer)
   forNumberOfSources(N);
InsertIdleTime := new(Bernoulli)
   withParameter(1/8);
IdleTimeDistribution := new(DiscreteUniform)
   withParameters(4, 9);
InitialiseSource(N)().
```

```
InitialiseSource(ID: Integer)()

par
   GenerateTraffic(ID)();
and
   if (ID > 1) then InitialiseSource(ID - 1)() fi
rap.
```

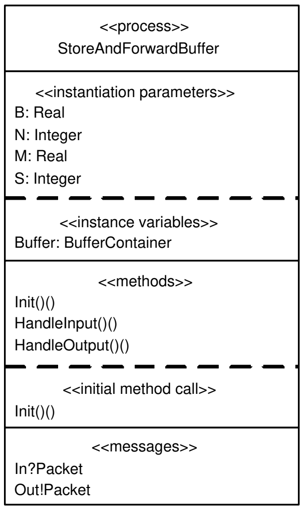
```
GenerateTraffic(ID: Integer)() |p: Packet|

p := TrafficSources generatePacket;
Out!Packet(ID, p)
delay(p getSize / Bandwidth);
if (InsertIdleTime yieldsSuccess) then
   delay(IdleTimeDistribution sample)
fi;
GenerateTraffic(ID)().
```

Figure 4.12: Modelling the Internet-traffic sources of example 1.1.

Modelling Pattern 5: Abstraction from Arbitration Mechanisms Hardware/software systems often include components that share resources such as processors, memories or communication media. Implementing the sharing of resources requires the use of some arbitration mechanism that determines in what order the different components may access a shared resource and for how long. This is usually accomplished by assigning a number of *access time-slots* to a component, which allows it to communicate a certain amount of data to the shared resource. The effect is that the total bandwidth for accessing the shared resource is subdivided in bandwidths for the individual components. These individual bandwidths are usually proportional to certain weight factors that determine the bandwidth shares.

During system-level design, the details of how and in which order access time-slots are assigned by an arbitration mechanism can be less important. Such implementation details might even be unknown at the time of developing the system concepts. In these cases, it is merely essential that some arbitration mechanism is needed but not how the sharing of bandwidth is realised. An adequate model that abstracts from the implementation details of arbitration mechanisms can be based on the ideal (but unimplementable) scheduling policy of *generalised processor sharing* explained in [97]. Generalised processor sharing serves the accesses to a shared resource as if there were separate logical connections with the appropriate bandwidth shares. By exploiting asynchronous concurrency, generalised processor sharing can be modelled in POOSL by receiving the data from all components with (independent) concurrent activities [172]. Each component sends its data to the shared resource at a speed matching the bandwidth share that would have been the result of a suitable arbitration mechanism. Notice that with this approach, the model of the shared resource may receive data from more than one component at the same time.

Example 4.4 *Consider again the buffer system of example 1.1 on page 5 and assume that the weight factors for all individual connections is $\frac{1}{N}$. Modelling the N Internet-traffic sources can be based on the model in example 4.3 by initialising instance parameter* `Bandwidth` *with $\frac{B}{N}$. Figure 4.13 shows how the input and output handlers of the store-and-forward buffer in figure 4.10 are adapted in order to model the buffer of example 1.1. Of interest are the modifications to the input handler, which allow the model of the buffer to concurrently receive data from different sources. After receiving a packet* `p` *from the concurrent source activity in* `Sources` *with identity* `SourceID`*, the input handler creates an activity for actually storing packet* `p` *(see lines 9 and 10 in figure 4.13). In addition, an activity is created for handling another packet, which may originate from a source activity with an identity different from* `SourceID`*. Next to abstracting from the arbitration mechanism that is needed for implementing the sharing of the buffer, the model dynamically creates concurrent activities only in case they are really needed. This in contrast to the modelling pattern in figure 4.11. Notice that the creation of concurrent activities is limited by the blocking receive statement.*



```
1    Init()()

2    Buffer := new(BufferContainer) ofSize(S);
3    par HandleInput()() and HandleOutput()() rap.
```

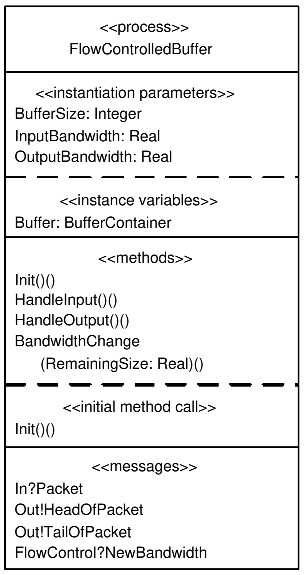
```
4    HandleInput()()
     |SourceID: Integer, p:Packet, NotFull: Boolean|
5
6    In?Packet(SourceID, p)
7       {NotFull := Buffer allocate};
8    par
9       delay((p getSize * N) / B);
10      if NotFull then Buffer storePacket(p) fi
11   and
12      HandleInput()()
13   rap.
```

```
14   HandleOutput()() |p: Packet|

15   [Buffer isNotEmpty] p := Buffer inspectPacket;
16   Out!Packet(p);
17   delay(p getSize / (M * B));
18   Buffer removePacket;
19   HandleOutput()().
```

Figure 4.13: Modelling the buffer of example 1.1.

Modelling Pattern 6: Dynamically Changing Bandwidth Components of hardware/software systems may communicate data in accordance with a dynamically changing bandwidth. This is for example the case if the arbitration mechanism for accessing a shared resource assigns other bandwidth shares to the components that access it during the operation of the system. Abstraction from the arbitration mechanism can still be done in a similar way as explained above. However, the effect of the changes in the bandwidth for the individual connections must now be taken into account. The difficulty of modelling this effect in POOSL lies in the fact that the amount of time to wait before continuing performing other behaviour as specified in a **delay**-statement cannot change during the execution of that **delay**-statement. In case the moment of the bandwidth change is known, then time until the bandwidth change can be computed and used in the **delay**-statement. Hence, at each bandwidth change, the remainder of the time to wait can be recomputed in accordance with the new bandwidth. On the other hand, if the moment of the bandwidth change is not known, another approach should be used. It relies on aborting the waiting for the previously specified amount of time and then continuing with waiting for the remaining time computed in accordance with the new bandwidth. This modelling approach is illustrated in the next example.

Example 4.5 *Recall the example of the store-and-forward buffer in figure 4.10. Instead of having a fixed bandwidth `OutputBandwidth` with which packets are sent through port `Out`, this bandwidth may now change during the operation of the buffer. The bandwidth change is notified to the buffer by a message `NewBandwidth` carrying the new bandwidth as parameter. The time of receiving this message (through port `FlowControl`) is unknown.*



```
1   HandleOutput()()
2   |p: Packet, TimeOfSendingHead: Real|

3   [(Buffer isNotEmpty) & (OutputBandwidth > 0)]
4      p := Buffer inspectPacket;
5   Out!HeadOfPacket(p)
6      {TimeOfSendingHead := currentTime};
7   abort delay(p getSize / OutputBandwidth) with
8      BandwidthChange(p getSize - (currentTime -
9         TimeOfSendingHead) * OutputBandwidth)();
10  Out!TailOfPacket(p);
11  Buffer removePacket;
12  HandleOutput()().
```

```
12  BandwidthChange(RemainingSize: Real)()
13  |TimeOfBandwidthChange: Real|
14
15  FlowControl?NewBandwidth(OutputBandwidth)
16     {TimeOfBandwidthChange := currentTime};
17  if OutputBandwidth > 0 then
18     abort delay(RemainingSize / OutputBandwidth)
19     with BandwidthChange(RemainingSize -
20        (currentTime - TimeOfBandwidthChange) *
21        OutputBandwidth)()
22  else BandwidthChange(RemainingSize)() fi.
```

Figure 4.14: Modelling dynamically changing bandwidth.

Figure 4.14 shows the model of the described flow-controlled buffer. The output handler activity specified by method `HandleOutput` starts forwarding a packet `p` when at least one packet is available in the buffer and `OutputBandwidth` is positive, see line 3. *Atomically* *with sending the head of the packet in line 5, the current model time is stored in the local variable `TimeOfSendingHead`. Then, waiting for the duration of communicating packet `p` in accordance with the current `OutputBandwidth` is initiated. The **`delay`**-statement in line 7 is however aborted when the receive statement of method `BandwidthChange` can be executed, which occurs if a new output bandwidth is received via the `FlowControl` port. If such an abortion does not happen, the **`delay`**-statement in line 7 terminates normally, which reflects that the sending of `p` is completed. This event is indicated to the receiver of `p` by an explicit communication as proposed by approach A in figure 4.6. On the other hand, if the **`delay`**-statement in line 7 is aborted, then it must be determined how much of `p` has already been sent since the head of `p` was sent in order to compute the duration of sending the remainder of `p` according to the new bandwidth. Method `BandwidthChange` has an input parameter indicating the remaining amount of data that still has to be sent after a bandwidth change. It uses a similar approach as method `HandleOutput` to anticipate on any other bandwidth changes that may occur during sending the remainder of a packet.*

Modelling Pattern 7: Buffering Data Streams Hardware/software systems sometimes include buffers to convert an incoming type of data into another type of data. An example is the reception of video frames in a video system, which have to be converted into blocks of pixels. One option for achieving such a conversion is to first receive a video frame completely into a store-and-forward buffer and then start generating pixel blocks from it. A more common approach is to already start the generation of a pixel block when enough pixels of a video frame have been received but not necessarily the complete video frame. The idea of this streaming buffer approach is to minimise the buffer size and to minimise the latency for the data conversion.

The difficulty of developing an adequate model of a streaming buffer is that the boundaries for the different data granularities are in general not aligned. Since abstraction from both data types is desirable, using a single FIFO queue for separately storing the bytes of the data is not a suitable approach. Instead, the streaming buffer can be modelled with two subsequent FIFO queues: one for storing data of the incoming type and one for storing data of the outgoing type. What remains is to determine the moment of moving (converting) data from the first queue to the other. This moment equals the moment at which sufficient data is stored in the buffer to generate a data item of the outgoing type. Figure 4.15 sketches the proposed approach.

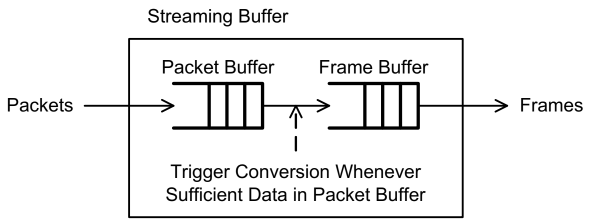
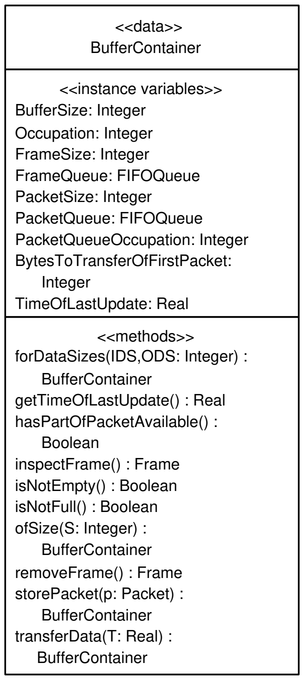

Streaming Buffer
Packet Buffer
Frame Buffer
Packets
Frames
Trigger Conversion Whenever Sufficient Data in Packet Buffer

Figure 4.15: Schematic overview for modelling a streaming buffer.

Figure 4.16 shows a container data class for modelling a streaming buffer as proposed. It assumes the reception of packets of `PacketSize` bytes into the streaming buffer and the forwarding of frames of `FrameSize` bytes, where `PacketSize` and `FrameSize` are fixed such that `PacketSize` $\geq$ `FrameSize`. The two FIFO queues


<<data>>
BufferContainer
<<instance variables>>
BufferSize: Integer
Occupation: Integer
FrameSize: Integer
FrameQueue: FIFOQueue
PacketSize: Integer
PacketQueue: FIFOQueue
PacketQueueOccupation: Integer
BytesToTransferOfFirstPacket: Integer
TimeOfLastUpdate: Real
<<methods>>
forDataSizes(IDS,ODS: Integer) : BufferContainer
getTimeOfLastUpdate() : Real
hasPartOfPacketAvailable() : Boolean
inspectFrame() : Frame
isNotEmpty() : Boolean
isNotFull() : Boolean
ofSize(S: Integer) : BufferContainer
removeFrame() : Frame
storePacket(p: Packet) : BufferContainer
transferData(T: Real) : BufferContainer

```
isNotFull() : Boolean

return(Occupation + PacketSize <= BufferSize).
```

```
hasPartOfPacketAvailable() : Boolean

return(PacketQueue isNotEmpty).
```

```
getTimeOfLastUpdate() : Boolean

return(TimeOfLastUpdate).
```

```
isNotEmpty() : Boolean

return(FrameQueue isNotEmpty).
```

```
inspectFrame() : Frame

return(FrameQueue inspect).
```

```
removeFrame() : Frame

return(FrameQueue remove).
```

```
forDataSizes(IDS, ODS: Integer) : BufferContainer

PacketSize := IDS; FrameSize := ODS; return(self).
```

```
ofSize(S: Integer) : BufferContainer

PacketQueueOccupation := 0; Occupation := 0; TimeOfLastUpdate := 0.0; Buffer-
Size := S; FrameQueue := new(FIFOQueue); PacketQueue := new(FIFOQueue); self.
```

```
storePacket(p: Packet) : BufferContainer

if PacketQueueOccupation = 0 then
   BytesToTransferOfFirstPacket := PacketSize fi;
PacketQueueOccupation := PacketQueueOccupation + PacketSize;
Occupation := Occupation + PacketSize; PacketQueue put(p); return(self).
```

```
transferData(T: Real) : BufferContainer

PacketQueueOccupation := PacketQueueOccupation - FrameSize;
if FrameSize < BytesToTransferOfFirstPacket then
   BytesToTransferOfFirstPacket := BytesToTransferOfFirstPacket - FrameSize
   else PacketQueue remove; BytesToTransferOfFirstPacket := PacketSize -
      (FrameSize - BytesToTransferOfFirstPacket) fi;
FrameQueue put(new(Frame)); TimeOfLastUpdate := T; return(self).
```

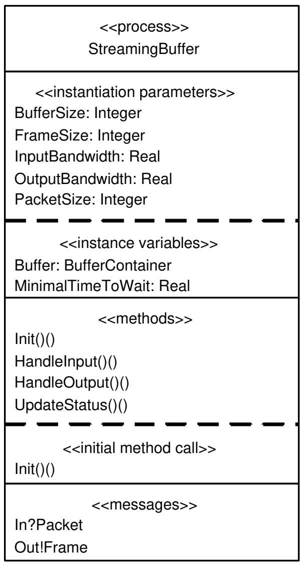
Figure 4.16: Container data class for modelling a streaming buffer.

are represented by the instance variables `PacketQueue` and `FrameQueue`. Instance variable `BufferSize` denotes the total size of the streaming buffer in bytes and `PacketQueueOccupation` and `Occupation` indicate the actual number of bytes in the `PacketQueue` and in the streaming buffer respectively.

Of interest are the methods `storePacket` and `transferData` in figure 4.16. After checking that the buffer is not yet full (with method `isNotFull`), the use of method `storePacket` enables to store a complete instance `p` of the data class `Packet` in figure 4.5 into the streaming buffer by putting it into the `PacketQueue` and increasing the occupancies of the `PacketQueue` and the streaming buffer with `PacketSize` bytes. The method `transferData` allows to move `FrameSize` bytes from the `PacketQueue` to the `FrameQueue`. It does so by generating an instance of data class `Frame`, which presents an abstraction of frames, and putting it in the `FrameQueue`. In order to adapt the contents of the `PacketQueue`, instance variable `BytesToTransferOfFirstPacket` stores the number of bytes of the first packet in the `PacketQueue` that still have to be converted and transferred to the `FrameQueue` at the moment `T` of invoking `transferData`. In case the number of bytes to transfer (`FrameSize`) is larger than the remaining part of the first packet in the `PacketQueue` (see lines 27, 28), transfer of this packet to the `FrameQueue` is completed and `BytesToTransferOfFirstPacket` is set to the remaining bytes to transfer of the next packet in the `PacketQueue`. Notice that method `storePacket` requires an **`if`**-statement for taking into account the case in which `PacketSize` equals an integer multiple of `FrameSize`, since `transferData` will then assign 0 to `BytesToTransferOfFirstPacket` if the transfer of a packet is completed, see lines 19, 20.

The use of the container data class in figure 4.16 is shown in figure 4.17. Process class `StreamingBuffer` has instance variables `InputBandwidth` and `OutputBandwidth` denoting the (fixed) bandwidths of receiving packets and sending frames respectively. After initialising data object `Buffer`, which models the streaming buffer, method `Init` initialises `MinimalTimeToWait` with the time needed to receive sufficient bytes of a packet to form a frame (see line 4). Then, it creates concurrent activities for handling the input of packets, handling the output of frames and updating the status of the buffer. The output handler activity, specified by method `HandleOutput` is similar to that of the store-and-forward buffer in figure 4.10.

When receiving the head of a packet `p`, the input handler atomically checks whether the buffer is not full (see line 13). If so, the packet is immediately stored into the `PacketQueue` and the occupancy of the `Buffer` is incremented with `PacketSize`. The **`delay`**-statement in line 15 models the duration of completing the reception of `p`, which is performed in accordance with approach B in figure 4.6. The crux of the modelling pattern in figure 4.17 lies in method `UpdateStatus`. The corresponding update activity waits until enough bytes have been received into the buffer to generate a frame that can be forwarded. In POOSL, waiting on an event can often be formalised using guards. It is however not allowed to use **`currentTime`** in a guard expression [28]. The guard in method `UpdateStatus` in line 19 synchronises the activity for updating the status of the buffer with the input handler in case a packet is available in the `PacketQueue` of the `Buffer`. Notice that this may even occur at the moment that only the head of a packet is received. The update activity must wait until enough bytes are received such that a frame can be generated. To enable the use of a **`delay`**-statement, it must be known how much time should pass until this event occurs. The time that passed since the last update of the buffer status (i.e., transferring data from the `PaceketQueue` to the `FrameQueue`) is used as a reference. In case too little time has passed to initiate the generation of a frame, the time until enough bytes have been received must be delayed (see lines 22, 23, 24). Then, data method `transferData` is invoked to update the status of the buffer. Subsequently, the output handler is allowed to forward the frame that was just generated.



```
1   Init()()

2   Buffer := new(BufferContainer) ofSize(BufferSize)
3      forDataSizes(PacketSize, FrameSize);
4   MinimalTimeToWait := FrameSize / InputBandwidth;
5   par
6      HandleInput()()
7   and
8      UpdateStatus()()
9   and
10     HandleOutput()()
11  rap.
```

```
12  HandleInput()() |p: Packet|

13  In?Packet(p){if Buffer isNotFull then
14     Buffer storePacket(p) fi};
15  delay(PacketSize / InputBandwidth);
16  HandleInput()().
```

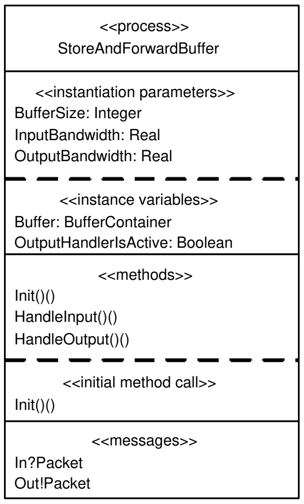
```
17  UpdateStatus()()
18  |TimePassedSinceLastUpdate: Real|

19  [Buffer hasPartOfPacketAvailable]
20     TimePassedSinceLastUpdate := currentTime -
21        Buffer getTimeOfLastUpdate;
22  if TimePassedSinceLastUpdate < MinimalTimeToWait
23     then delay(MinimalTimeToWait -
24        TimePassedSinceLastUpdate) fi;
25  Buffer transferData(currentTime);
26  UpdateStatus()().
```

```
27  HandleOutput()() |f: Frame|

28  [Buffer isNotEmpty] f := Buffer inspectFrame;
29  Out!Frame(f);
30  delay(FrameSize / OutputBandwidth);
31  Buffer removeFrame;
32  HandleOutput()().
```

Figure 4.17: Buffering data streams.

Modelling Pattern 8: Replacement of Guards Concurrent activities within a process may synchronise on shared data objects by using guards. To check whether the statement after a guard can be executed, the framework for executing POOSL models (see also appendix B) evaluates before each time transition whether the expression in blocking guards evaluate to **`true`**. As a consequence, execution speed can slow down rapidly in case there are a large number of guards that remain **`false`** for a long time. Sometimes, it is possible to replace guards by a construction that creates the blocked concurrent activity at the moment that the event on which it should synchronise occurs. This approach of creating concurrent activities only in case they are really needed is considered to be an optimisation for execution speed. Since the use of guards often yields a more intuitive model compared to the creating concurrent activities dynamically, it is advised to postpone such a replacement of guards until the construction of an intuitive model.

The replacement of guards is illustrated for the store-and-forward buffer in figure 4.10, where the output handler activity synchronises with the input handler activity by using a guard on the event of having at least one complete packet in the buffer. Figure 4.18 shows the result of adapting the model in figure 4.10. Now, process class



```
Init()()

Buffer := new(BufferContainer)
   ofSize(BufferSize);
OutputHandlerIsActive := false;
HandleInput()().
```

```
HandleInput()()
|p: Packet, NotFull, StartOutputHandler: Boolean|

In?Packet(p){NotFull := Buffer allocate};
delay(p getSize / InputBandwidth);
{StartOutputHandler := false;
   if NotFull then
      Buffer storePacket(p);
      if OutputHandlerIsActive not then
         StartOutputHandler := true;
         OutputHandlerIsActive := true
      fi
   fi};
par
   if StartOutputHandler then HandleOutput()() fi
and
   HandleInput()()
rap.
```

```
HandleOutput()() |p: Packet; Continue: Boolean|

{Continue := Buffer isNotEmpty;
   if Continue not then
      OutputHandlerIsActive := false
   else
      p := Buffer inspectPacket
   fi};
if Continue then
   Out!Packet(p);
   delay(p getSize / OutputBandwidth);
   Buffer removePacket;
   HandleOutput()()
fi.
```

Figure 4.18: Using dynamic creation of concurrent activities to replace guards.

`StoreAndForwardBuffer` includes an instance variable `OutputHandlerIs-Active` to denote whether the output handler is activated. When receiving the head of a packet `p`, the input handler atomically checks whether `p` can still be put into the buffer, see line 8. After having received packet `p`, which is the case when the **`delay`**-statement in line 9 terminates, the local variable `StartOutputHandler` is first set to **`false`**. It is then determined whether the output handler activity should be created. In case `p` is put into the buffer, there is at least one packet in the buffer. So, if the output handler was not yet active, an activity for handling the output must be started. Notice that the assignment of **`true`** to `OutputHandlerIsActive` indicates that an activity for handling the output *will be* started. The actual creation of the concurrent output handler activity is done by the **`if`**-statement in line 19.

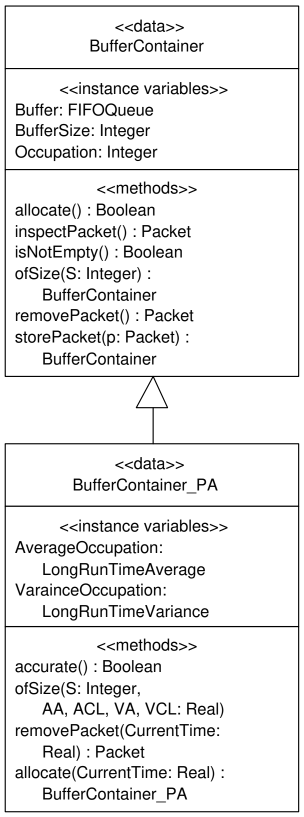
The output handler activity first checks whether there are still packets available in the buffer by using the local variable `Continue`, see line 24. If this is not the case, the activity should be terminated. Termination of the output handler is indicated by assigning **`false`** to `OutputHandlerIsActive` in line 26, which ensures that the input handler knows that the output handler is not active anymore. Otherwise, the first packet in the buffer has to be sent. The **`if`**-statement in line 30 initiates the actual sending of the first packet in the buffer if the buffer was not empty.

4.3.2 Extending Models for Performance Analysis

The approach of reflexive performance analysis proposed in chapter 3 involves extending a POOSL model with performance monitors. Such addition of variables and behaviour may however decrease the intuitiveness of the model because a clear distinction between code representing the behaviour of the system and code reflecting the monitors is often difficult to obtain [171, 180]. This section proposes an approach for extending POOSL models, which preserves the intuitiveness of a model by defining subclasses for those classes that require to be extented.

Data Classes Because monitors gather, store and process information about certain properties of the system, adding monitors to a model may have a profound impact on its data layer. Gathering, storing and processing such information is specified by expressions on certain data objects. These data objects are either introduced especially for the purpose of monitoring or are similar to data objects that already existed in the original model. In the first case, new data classes are needed. To distinguish such new data classes from data classes representing behaviour of the system, it is recommended to use a distinguishable postfix (such as `_PA`) in their class names. In the other case, extending an existing data class with additional variables and methods or with modifications to existing methods would be sufficient. Instead of doing so, it is recommended to define a subclass of the data class with the additional variables and methods. Redefining methods is then based on method overriding and explicit calls to methods of the superclass. When defining a subclass for performance analysis, it is recommended to use the name of the original data class extended with a distinguishable postfix like `_PA`.

Example 4.6 *Assume the availability of a container data class* `BufferContainer` *that models a store-and-forward buffer as shown in figure 4.9 on page 122. To analyse the average* *buffer occupancy and the variance in the buffer occupancy, this container data class can be extended with performance monitors based on the library classes discussed in section 3.3.1. Figure 4.19 shows the subclass* `BufferContainer_PA` *that extends* `BufferContainer` *with the instance variables* `AverageOccupation` *and* `VarianceOccupation` *to monitor the average occupancy and the variance in occupancy respectively. As illustrated in example 2.11 on page 52, these performance metrics can be expressed as a (conditional) long-run time average and variance respectively. The method* `ofSize` *of* `BufferContainer_PA` *overrides the method* `ofSize` *of its superclass to initialise the size of the buffer as well as the parameters of the performance monitors* `AverageOccupation` *and* `VarianceOccupation`*. The methods* `allocate` *and* `removePacket` *are also overridden. They update the estimated average occupancy and the variance in occupancy after assigning the new occupancy to instance variable* `Occupation`*. Notice that the moment of changes in the occupancy must be known in order to do so. Hence,* `allocate` *and* `removePacket` *have an additional parameter for passing on this information*[19]*. Remark also that, for both performan-*


<<data>>
BufferContainer
<<instance variables>>
Buffer: FIFOQueue
BufferSize: Integer
Occupation: Integer
<<methods>>
allocate() : Boolean
inspectPacket() : Packet
isNotEmpty() : Boolean
ofSize(S: Integer) : BufferContainer
removePacket() : Packet
storePacket(p: Packet) : BufferContainer
<<data>>
BufferContainer_PA
<<instance variables>>
AverageOccupation: LongRunTimeAverage
Varaince Occupation: LongRunTimeVariance
<<methods>>
accurate() : Boolean
ofSize(S: Integer, AA, ACL, VA, VCL: Real)
removePacket(CurrentTime: Real) : Packet
allocate(CurrentTime: Real) : BufferContainer_PA

```
ofSize(S: Integer, AA, ACL, VA, VCL: Real) :
   BufferContainer_PA

Buffer := new(FIFOQueue);
BufferSize := S; Occupation := 0;
AverageOccupation := new(LongRunTimeAverage)
   withParameters(AA, ACL);
VarianceOccupation := new(LongRunTimeVariance)
   withParameters(VA, VCL); return(self).
```

```
allocate(CurrentTime: Real) : BufferContainer_PA
|NotFull: Boolean|

NotFull := self ^allocate;
if NotFull then
   AverageOccupation rewardRC(Occupation,
      CurrentTime, false);
   VarianceOccupation rewardRC(Occupation,
      CurrentTime, false)
fi;
return(NotFull).
```

```
removePacket(CurrentTime: Real) : Packet
|p : Packet|

p := self ^removePacket;
AverageOccupation rewardRC(Occupation,
   CurrentTime, Occupation = 0);
VarianceOccupation rewardRC(Occupation,
   CurrentTime, Occupation = 0);
return(p).
```

```
accurate() : Boolean

return((AverageOccupation accurate) &
   (VarianceOccupation accurate)).
```

Figure 4.19: Container data subclass with performance monitors.

[19]Recall that **`currentTime`** may not be used in the specification of a data method [28].

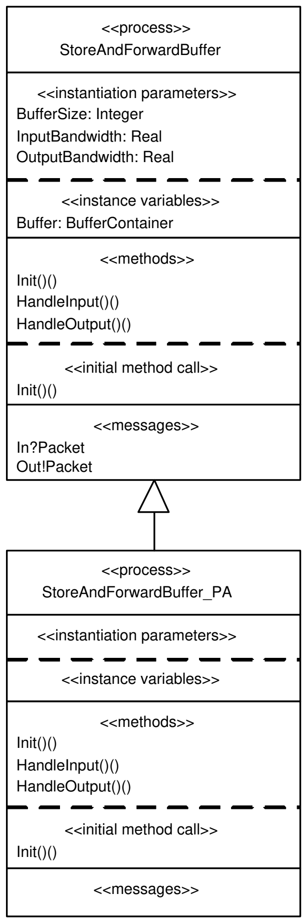
ce metrics, the emptiness of the buffer is used as a condition for approximating the beginning of a regenerative cycle (see also example 3.5). Finally, the method `accurate` *is introduced, which returns* **`true`** *in case the estimation results for both the average buffer occupancy and variance in the buffer occupancy are accurate, see also section 3.3.1.*

The use of a postfix in the names of data classes that are extended for performance analysis simplifies recognising code representing the behaviour of the system and code for evaluating performance properties. The names of the variables in the original model, which become an instance of some extended data class when extending the model with monitors, should however not be changed. This approach implies minimal modifications to data or process methods referring to these variables.

Process Classes Similar as for data classes, certain process classes may need to be extended in order to evaluate performance properties. This is especially the case if a performance monitor requires knowledge about timing-related information because such information can only be retrieved at the process layer. Instead of extending the original process class with additional variables and methods or modifying its methods, it is advised to define a subclass. Such a subclass then has the additional variables and methods, while the redefinition of methods is based on method overriding. For naming such a subclass, it is recommended to use the name of the original process class extended with a distinguishable postfix like `_PA`. The instances of such extended process classes should be named similarly as in the original model.

Example 4.7 *Consider extending the process class* `StoreAndForwardBuffer` *in figure 4.10 on page 123 for evaluating the average buffer occupancy and the variance in buffer occupancy. Observe that this can be accomplished by using container data class* `BufferContainer_PA` *of example 4.6. Figure 4.20 shows subclass* `StoreAndForwardBuffer_PA` *that extends* `StoreAndForwardBuffer` *for performance analysis. To apply the monitoring capabilities of* `BufferContainer_PA`*, all process methods need to be overridden. Only minor changes are necessary for methods* `HandleInput` *and* `HandleOutput`*. Lines* 9 *and* 17 *show how timing-related information is passed to the performance monitors. The original method* `Init` *is overridden to enable instantiating* `Buffer` *as an instance of* `BufferContainer_PA`*. In addition, it initialises the parameters for the performance monitors in line* 3 *(see also section 3.3.1). According to the* `abort`*-statement in lines* 4 *-* 6*, the behaviour of an instance of* `StoreAndForwardBuffer_PA` *is terminated when the estimation results for the average occupancy and the variance in occupancy become accurate.*

Next to the need for defining subclasses of process classes, extending a model for performance analysis may involve the introduction of new process classes. Such additional process classes may for example represent behaviour to control the simulation of an extended model. Recall that performance metrics are often defined in relation to a specific component of a system and hence, performance monitors may be included in different processes. Only in case the estimation results for all performance metrics have become accurate, a simulation may be terminated [171]. In order to do so, information on the accuracy of all performance metrics must be collected at some central location where it can then be decided whether to terminate a simulation. To minimise modifications to processes that model the behaviour of the system, it is advised to introduce a special process (and accompanying process class) to collect the accuracy-related information. When the simulation should be terminated,



```
Init()()

Buffer := new(BufferContainer_PA)
  ofSize(BufferSize, 0.9, 0.95, 0.8, 0.9);
abort
  par HandleInput()() and HandleOutput()() rap
with [Buffer accurate] skip.
```

```
HandleInput()() |p:Packet, NotFull: Boolean|

In?Packet(p)
  {NotFull := Buffer allocate(currentTime)};
delay(p getSize / InputBandwidth);
if NotFull then Buffer storePacket(p) fi;
HandleInput()().
```

```
HandleOutput()() |p: Packet|

[Buffer isNotEmpty] p := Buffer inspectPacket;
Out!Packet(p);
delay(p getSize / OutputBandwidth);
Buffer removePacket(currentTime);
HandleOutput()().
```

Figure 4.20: Example of a process subclass with extensions for performance analysis.

this process can notify all other processes to abort performing their behaviour. In order to identify this process as one that does not represent system behaviour, it is recommended to use a distinctive name such as `SimulationController` for it.

Cluster Classes In order to group processes of extended process classes in a similar way as in the original model, it may be necessary to redefine certain cluster classes. Instead of modifying the original cluster classes, it is recommended to define new cluster classes in case one of the processes in the original cluster class is to be replaced[20]. Similar naming conventions as for extended process classes are advised for such new cluster classes. In fact, it is recommended to define a complete new set of instance structure diagrams that incorporate the extended processes and clusters. The reason for this is that this approach allows for a very clear distinction between the original model and the extended model, which is supported by the SHESim tool. SHESim has the capability to open additional system-level editor windows to specify different models based on the same set of data, process and cluster classes. This

[20]As POOSL does not support inheritance for cluster classes, defining cluster subclasses is not possible.

capability allows to construct, simulate and inspect the extended model separately from the original model without modifying this original model.

The guidelines for extending a POOSL model intend to clearly separate the original model representing the system behaviour from the extended model that also includes behaviour regarding performance monitors. Next to keeping the original model intuitive (by not changing it), this separation aims at simplifying validation of the extensions that are made. Such validation is necessary to ensure that the original representation of the system's behaviour does not change [180]. Especially minimising extensions to the process layer of a model, which is often the most difficult part to validate, contributes to minimising the necessary additional validation efforts.

4.3.3 Model Validation

This section provides some guidelines that assist the designer in validating POOSL models against concept models expressed in UML according to the UML profile for SHE. These guidelines focus on three different aspects: the model's static structure, the represented dynamic behaviour and the adequacy of abstractions that are made.

Static Structure In the formulation stage of the SHE method, the application of object-oriented analysis techniques results in identifying data objects, processes and clusters and their accompanying classes. An important aspect for validation is to ensure the intuitiveness of a model by using representative names for the items composing the model. In [145], some naming conventions were introduced for developing intuitive models with the SHE method, which are extended here. An example is that data objects, processes and clusters concern resources of a system, which are named with nouns. The behaviour of data objects reflect operations on passive resources. Hence, data methods specifying such behaviour should have names starting with a verb. Data method names should be chosen such that the code of the process methods where they are used is easy to understand. Naming of process methods is more complicated. If a process method merely groups operations on data objects, it seems reasonable to use a name starting with a verb. On the other hand, if a process method represents a certain activity or (composite) state as they emerge in activity and statechart diagrams, it is recommended to name it with the noun denoting that activity or state. Finally, the names of messages exchanged between processes and clusters should clearly reflect what kind of information is actually communicated.

Validating the static structure of a POOSL model concerns checking whether the formalisation of data, process and cluster classes as well as scenarios matches with the information formulated in class, object, component, deployment, use case, sequence and collaboration diagrams. Validating the static structure of data classes mainly concerns checking that their instance variables, method headers and inheritance relations as defined in the model match with those in class and object diagrams. For process classes, the formalisation of instantiation parameters, instance variables, method headers, initial method calls, port interfaces, message interfaces and inheritance relations must be validated against the information expressed in any of the mentioned diagram types. Recall for example that inheritance relations for data and process classes should match with specialisation/generalisation relations in class diagrams. Validating cluster classes includes checking the formalisation of their instantiation parameters, port interfaces and message interfaces. Moreover, the validation of how processes and cluster are interconnected by channels is needed because the possibility of communicating messages between them as formulated in for example sequence and collaboration diagrams cannot be formalised in POOSL automatically, see section 4.2.1. Recall also that the need for cluster classes should be in accordance with aggregation/composition relations between active resources.

Dynamic Behaviour Validating the dynamic behaviour of a POOSL model concerns a check against the information expressed in especially use case, sequence, activity, collaboration and statechart diagrams. It is based on either step-by-step or trace-wise inspection of the POOSL model while executing it using the SHESim tool, see also section 4.1.1 and appendix B. SHESim allows to inspect any data object, process, cluster or message in any scenario [58]. When executing a POOSL model, SHESim automatically generates *interaction diagrams*, which reflect the sequence of messages communicated between processes and clusters. They support validation of the model's dynamic behaviour against sequence diagrams. On the other hand, inspection of clusters shows the communication of messages between processes and clusters in a similar way as expressed in collaboration diagrams. Furthermore, the messages can be inspected to observe the involved parameters (if any). Zooming into a process allows to validate its behaviour against activity and statechart diagrams by inspecting the provided graphical representation of the execution tree and the highlighted statements that are to be executed. Also, data objects encapsulated by a process and those communicated as parameters of messages can be inspected. Such inspection can however be rather involved in case a data object presents a complex data structure. It is recommended to define `printString` methods for any data class, which allows to display the most relevant information by SHESim, see also appendix B. Such a `printString` method is especially valuable for container data objects and data objects that present monitors (as discussed in section 3.3.1).

Several aspects of validating a model's dynamic behaviour require special attention:

- The behaviour that processes perform after starting a simulation may include the creation of container data objects and/or concurrent activities. Remark that this kind of behaviour is not a part of the actual system behaviour and may therefore not be expressed in the concept model. Validation includes comparing the behaviour that is initially performed by a process with what should occur after the initial method call. In other words, it must be checked whether the creation of container data objects and/or concurrent activities is successful.
- Generally, a system has to anticipate on certain events (occurring either in the environment of the system or internally). The responses to such events may be specified in isolation by means of a separate set of UML diagrams for the use case or scenario involved. Validation is done by checking whether these responses also occur when executing the POOSL model. Since the POOSL model unifies all events and responses in one model, it may be difficult to check the response to a certain event independently from the occurrence of other events. Validation includes discussions on whether such dependencies may exist indeed and if so, whether the response to the combination of events is as desired.
- Although the actual evaluation of correctness and performance properties has

to be done at the evaluation stage, premature results for certain properties may help in identifying formalisation errors. Often, the behaviour of a system should satisfy certain easy-to-check properties. An example of such a property is that decreasing some buffer size should result in an increase of the probability to loose data at that buffer. As indicated in section 4.1.2, it is recommended to use premature evaluation results for checking simple properties already during the formalisation stage in order to improve the model.

- In line with the preceding two aspects lies the validation of responses to events that occur only occasionally. Such *rare events* may greatly affect the behaviour and performance of a system, see also section 3.2.1. Validation includes checking whether the behaviour exposed by a POOSL model due to a rare event is in accordance with what is formulated in the concept model.

Adequacy of Abstractions Next to checking the formalisation of a concept model into a POOSL model, validating such a POOSL model may reveal inadequacies in the abstractions that are made. Such inadequacies are often manifested by large differences between the model's behaviour and the intended behaviour of a possible realisation of the system. Such expectations are based on knowledge of the behaviour of existing systems. To prevent constructing an inadequate model, it is advised to:

- Make a log of *any* consideration regarding the adequacy of abstractions that are made during both the formulation and formalisation stages. Several case studies have shown that certain abstractions that were considered to be adequate when they were made, turned out to be inadequate later [170]. Considerations on the adequacy of abstractions and any changes in abstractions made during the formulation and formalisation stages should be documented in the deliverables at the appropriate milestones of SHE (see figure 4.1 on page 105).
- Validate the behaviour of the model for known corner cases. It is sometimes possible to identify special situations for which some performance properties can easily be computed by hand. An example of such a corner case is the situation where all packets induced on the inputs of a router system are destined for the same output. Other corner cases may reflect a situation where certain stochasticity is removed from the model. An example is the case of inducing fixed-size packets with a fixed rate, zero inter-arrival times and fixed destinations into the mentioned router system.
- Postpone any optimisations in the code of a POOSL model in case improvement of the model's execution speed is desirable. During the formalisation stage, one should first concentrate on appropriately formalising the concepts for realising the system as represented in the concept model. Only after being confident about the adequacy of a first (intuitive but slow) model, one may consider modifying it to improve execution speed. The main reason for postponing making optimisations is that they usually reduce the intuitiveness of the model, which makes its validation much more complicated.

The best chance on revealing inadequacies in a model is during the validation of the dynamic behaviour of the constructed POOSL model as described above. Especially unexpected premature evaluation results may lead to detecting inadequacies. It is therefore useful to recall the log of considerations on the adequacy of abstractions whenever investigating the cause of unexpected premature evaluation results.

4.3.4 Related Research

UML-Based Performance Modelling Much research has been performed on the potentials of UML-based system-level design methods like the SHE method. This research included an investigation on how the characteristics of time and stochasticity can be expressed in UML. Recently, UML has been extended with profiles that assist in expressing timing and performance related information [130, 154]. Nevertheless, these profiles do not specify the exact semantics of time and stochasticity (the exact semantics of concurrency in UML is not defined either) such that tools for executing UML models require to extend these tools using a programming language like C++. A more appropriate approach is to develop a UML profile that defines the semantics of its primitives in accordance with those of a modelling language.

Of special interest is the use of SDL [88] and ROOM [155] to derive executable models from UML diagrams. For SDL, this procedure is standardised in [89]. SDL is a formal modelling language that allows to evaluate both correctness [60, 159] and performance properties [36, 14, 92]. In this respect, SDL is very similar to POOSL. However, a mathematical basis for reflexive performance analysis with SDL is missing even though stochastic behaviour can be specified using proper random number generators. A similar conclusion holds for ROOM. Another deficiency of ROOM is that time is not an inherent characteristic of ROOM models and hence, tools for executing these models like ObjectTime [155] or RoseRT [146] do not separate model time from physical time. Consequently, system-level design methods based on ROOM suffer from the problem that real-time behaviour cannot be modelled adequately [108, 81].

The need for deriving a mathematical structure from a UML model to support performance analysis was also recognised in [141]. Another relevant conclusion of [141] is the necessity to express probabilistic information in UML diagrams. Both [141] and [151] suggest to develop an integrated framework for explicitly representing performance related information in UML diagrams (as it is now standardised in [130]) and the transformation of these UML diagrams into a mathematically tractable performance model. An example of such an integrated framework is discussed in [94], where the transformation to a queueing network is based on an intermediate textual representation of the performance related information specified in UML diagrams. The SHE method is similar to [94] in that it also uses an intermediate modelling language (POOSL) to formalise a UML model.

In [141], several options are presented for generating queueing networks, (generalised stochastic) Petri nets or Markov chains from UML diagrams. References [123], [100] and [140] respectively briefly explain the ideas behind these options. In [123], the structure of queueing networks is obtained from use case and deployment diagrams, whereas the behaviour of servers in these queueing networks is determined by combining collaboration and statechart diagrams. Queueing networks impose strong restrictions on the use of probability distributions to make them mathematically tractable. As a consequence, complete industrial hardware/software systems can sometimes not be modelled adequately. The approach in [140] derives a Markov chain from UML based on a process algebra. Both [141] and [140] provide ideas for a sound basis to obtain credible performance results but do not work these ideas out in much detail. Moreover, these approaches suffer from the state-space explosion problem [126] as a consequence of generating the queueing network, generalised stochastic Petri net and Markov chain explicitly. The SHE method involves using the expressive modelling language POOSL to define a Markov chain implicitly, thereby enabling the evaluation of design alternatives for very complex systems.

Modelling Patterns Modelling patterns are comparable to design patterns [15], which capture recognisable design strategies that have demonstrated to yield good and efficient hardware or software implementations. In [54] for example, design patterns are discussed for the development of object-oriented software. The usefulness of documenting and reusing templates for modelling common aspects of hardware/software systems with POOSL was also recognised in [177]. Opposed to specialising modelling patterns for a certain application domain as proposed in [177], the modelling patterns discussed in section 4.3.1 are more generally applicable.

Model Validation Because models inherently involve abstraction from details that are not essential for evaluating the properties of interest, it is of utmost importance to check the validity of a model before taking design decisions based on it. Although model validation is often identified as one of the most difficult and time-consuming aspects of system-level design, only a few approaches provide specific guidelines that assist a designer in the validation process. Often, the guidelines for the construction of models (such as naming conventions) are considered to be sufficient for assisting the designer in validating those models. In [150], some guidelines are presented that are specific for the validation of models. These guidelines can easily be adopted by the SHE method. Similar as SHE, the system-level design method in [128] also provides specific guidelines for validating executable models against UML diagrams. Another approach for simulation-based evaluation of systems, which provides useful additional validation guidelines, is discussed in [107]. These guidelines can easily be adopted by SHE because the method in [107] is rather similar to SHE with respect to first formulating the design problem and the issues that are to be investigated, then constructing and validating an executable model followed by the actual evaluation of the properties of interest.

4.4 Conclusions

Originally, the SHE method mainly provided assistance in applying object-oriented analysis techniques for capturing the concepts and requirements of hardware/software systems. This chapter extended SHE to offer a complete approach for exploring design alternatives during system-level design. Although focussing on how to construct adequate performance models, the proposed framework integrates these performance modelling guidelines with the existing techniques, guidelines and tools.

The SHE method distinguishes the stages of formulation, formalisation and evaluation. During the formulation stage, the concepts and requirements for a system are documented as a concept model and a set of desired properties respectively using UML diagrams and explanatory texts. Such UML diagrams comply with a UML profile that intends to smoothen the derivation of an executable POOSL model and monitors from the concept model and the desired properties respectively during the formalisation stage. The SHE method supports both the model-checking and reflexive approaches for defining monitors in order to evaluate correctness and performance properties either analytically or based on simulations. After completing the formalisation stage with documenting the validated model and monitors, the actual evaluation of the properties of interest is performed during the evaluation stage. Based on the results, design decisions on the feasibility of design alternatives are taken. The proposed framework is rather generic in the sense that many other design methods implicitly include stages of formulation, formalisation and evaluation although they tend to have more detailed heuristics for guiding the formulation and formalisation in accordance with a certain application domain or modelling style.

The UML profile for SHE distinguishes data, process and cluster classes to model the resources of a system. It includes special class symbols for each of these class types. Their compartments are stereotyped in accordance with all those aspects that completely define the static structure of a POOSL model. Data classes are comparable to classes in traditional object-oriented languages and hence, their class symbol includes the usual compartments. The class symbols for process and cluster classes have a non-standard number of compartments. Process classes have an additional compartment to clearly separate messages from methods, which are not coupled as is the case for data classes. Although the class symbol of cluster classes includes a message compartment, it misses a method compartment. This originates from the restriction in SHE that clusters do not extend the behaviour of their constituents. Compared to the restricted support that UML offers for specifying the behaviour of data objects, information on the behaviour of processes can be expressed in various UML diagram types. A straightforward mapping of this information onto statements in POOSL is however not obvious, which hinders the development of a tool automating the generation of POOSL models from a set of stereotyped UML diagrams. The behaviour of clusters is based on a special diagram type, which was originally introduced for SHE. Various aspects of this diagram type are also reflected in a number of UML diagram types.

Next to the provided guidelines for deriving POOSL models from stereotyped UML diagrams, some generally applicable modelling patterns are presented. These modelling patterns provide templates for developing abstract models of common aspects in hardware/software systems, which are adequate with respect to the evaluation of performance properties. Of special importance in this context is the impact of abstraction from data, which requires to thoroughly consider the adequacy of timing behaviour in a model regarding the processing, communication and storing of data.

To evaluate performance properties according to the framework for reflexive performance analysis with POOSL, the model representing the system's behaviour must be extended with monitors. Without taking precautions, such an extension reduces the intuitiveness of the model. It is proposed to extend a model with monitors based on defining subclasses and constructing the extended model in a separate system-level editor window in the SHESim tool. This approach allows to clearly separate code representing the system's behaviour from code concerning the monitors. In addition to the separation, it is proposed to minimise changes to the process layer of a model because this layer is often considered to be the most difficult part to validate.

Validation of POOSL models against stereotyped UML diagrams is subdivided in three categories. Validating the static structure of a POOSL model is rather straightforward and will become superfluous in case a future tool is capable of automatically generating skeleton POOSL code from stereotyped UML diagrams. The dynamic behaviour of a POOSL model is more difficult to validate. An important reason for this is the different ways in which the information expressed in UML diagrams may be formalised with POOSL. Several guidelines are provided that assist in how a POOSL model can be validated against the UML diagrams and what aspects of the behaviour require special attention. Most complicated is to validate the adequacy of abstractions. A useful recommendation that may help in preventing the construction of inadequate models is to make a log of considerations on the adequacy of abstractions, which should be consulted whenever investigating the cause of unexpected evaluation results. Moreover, investigating the behaviour of a model for corner cases may help in finding model inadequacies.

Chapter 5

Case Studies

To investigate performance modelling for system-level design by using the Parallel Object-Oriented Specification Language (POOSL), various academic and industrial case studies have been carried out. These case studies provided valuable input for identifying the theoretical and practical issues that are involved when evaluating performance properties and initiated the development of theory to cope with them. Table 5.1 lists the performed industrial case studies.

Case Study	Industrial Cooperation	References
Internet Router	Alcatel Bell Antwerp	[171, 168, 169]
Dataflow System	Alcatel Bell Antwerp	[165, 50]
MA3 System	TNO Industrial Technology Eindhoven	[80]
Network Processor	IBM Research Laboratory Zürich	[172, 103]
DECT System	TNO Industrial Technology Eindhoven	[82]
PiP TV Application	Philips Research Laboratory Eindhoven	[183]

Table 5.1: Performance modelling with POOSL in industrial case studies.

In cooperation with Alcatel Bell in Antwerp, the performance of product variants of a backbone Internet Router has been evaluated as well as the performance of design alternatives for the memory arbitration mechanism in the Internet Router's input buffer subsystems (which is also called the Dataflow System). In cooperation with TNO Industrial Technology in Eindhoven, the performance of a product assembler machine called the MA3 System has been analysed as well as the performance of design alternatives for a special wireless communication system for hearing-impaired students in a class room, which is based on a modified version of the DECT protocol. The performance of a Network Processor has been analysed in cooperation with IBM Research Laboratory in Zürich. Finally, the performance of executing a Picture-in-Picture (PiP) TV Application on a multi-processor platform has been evaluated in cooperation with Philips Research Laboratory in Eindhoven.

Many of the case studies in table 5.1 were actually carried out while developing the theories proposed in this thesis. To assess the newly developed performance analysis techniques, the framework for reflexive performance analysis with POOSL and the extensions to the Software/Hardware Engineering (SHE) design method, the Internet Router case study was selected for a re-evaluation. The next section elaborates on this re-evaluation. The Network Processor case study was largely performed conform the proposed theories and section 5.2 discusses some aspects of this case study in more detail.

5.1 Internet Router

The Internet Router investigated in this section is basically an input/output buffered switch system protected by a flow-control mechanism [97]. It has a parameterisable number of inputs and outputs and its purpose is to transfer Internet packets received on the inputs to those outputs for which the packets are destined. This functionality should be performed with a minimal loss of packets, a minimal latency for transferring packets through the system, and a maximal throughput at the outputs. Notice that minimising the packet latency implies reducing the buffer capacities, which however conflicts with the requirement of minimising packet loss. Early in the design process, a certain flow-control mechanism was proposed, which was expected to minimise buffer capacities while still being capable of yielding a small packet loss and packet latency next to a high throughput. The design issue was to decide whether the proposed flow-control mechanism implied a high performance for several product variants of the system, which mainly differ in the number of inputs and outputs. Hence, the Internet Router case study involved investigating the performance of realistic product variants, determining suitable combinations of buffer capacities and other system parameters, deciding whether the proposed flow-control mechanism is appropriate and proposing improvements for it if necessary.

Formulation Figure 5.1 sketches[1] the architecture platform of the Internet Router together with an abstract view on its environment. During exploratory discussions on concepts for correct high-performance packet transfer, this sketch was used as a reference to explain how the flow control mechanism is supposed to minimise buffer capacities while maximising performance.

As shown in figure 5.1, the Internet Router has an equal number N of inputs and outputs. The packets that are to be transferred are induced on the inputs by N groups of M independent sources. Each of these sources generates Internet-traffic, where variable-sized files alternate with periods of no traffic. A file is a sequence (burst) of packets with the same destination. The packets have to be transferred in the same order as they were received (notice that it is allowed to insert packets of other files). For each output of the Internet Router, a sink deals with the traffic resulting from the forwarding of packets. The Internet Router itself consists of a switch core, N input buffers and N output buffers. Each input buffer receives and buffers the packets originating from the equally numbered group of M sources. To overcome the problem of head-of-line blocking [97], each input buffer includes N FIFO queues (which are to be implemented in a single physical memory): one for buffering the packets destined for each of the outputs. The result is that each queue in an input buffer is

[1]Figure 5.1 depicts a logical view on the Internet Router. Physically, the input and output buffers for equally numbered inputs and outputs are subsystems on the same rack-mountable printed circuit board.

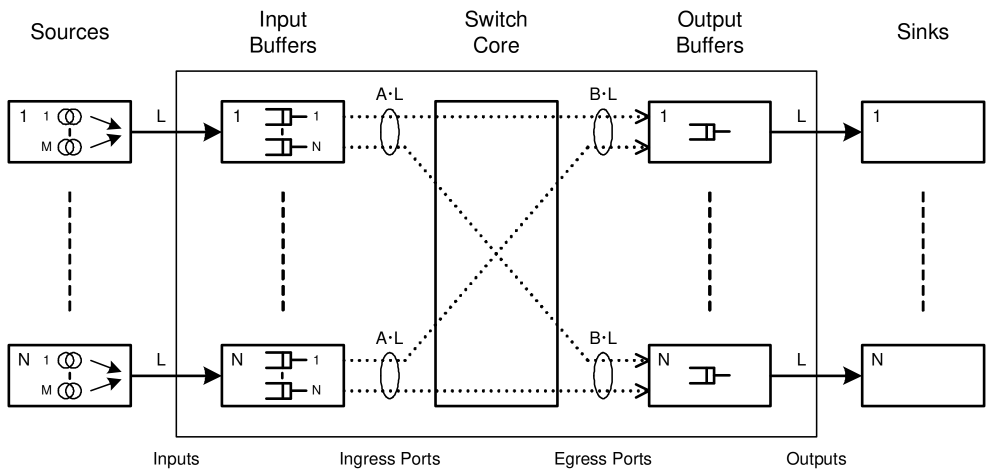


Figure 5.1: System architecture of the Internet Router.

virtually connected to the output for which the buffered packets are destined. The switch core actually realises these virtual connections between the input and output buffers. Physically, there is only a single connection available between an input or output buffer and the switch core. These connections are called ingress and egress ports respectively. Arbitration mechanisms ensure proper utilisation of these shared ports. Finally, each of the output buffers includes a single FIFO queue to take care of excessive transfer of packets to the corresponding output.

The (fixed) aggregate bandwidth with which packets are induced on an input is indicated with L and equals the bandwidth of each output. To reduce a possible loss of packets due to a full queue, the aggregate bandwidth available at each ingress port is $A \cdot L$ and at each egress port $B \cdot L$, where bandwidth factors A and B are such that $A \geq 1$ and $B \geq 1$. The flow-control mechanism is concerned with ensuring that the actual bandwidth assigned to each virtual connection is proportional to the amount of data in the corresponding queues of the involved input buffer and output buffer. Periodically updating the actual bandwidths of the virtual connections is supposed to minimise the necessary buffer capacities for a certain packet loss probability bound. Observe that minimising the buffer capacities also contributes to minimising the packet latency and maximising the throughput. On the other hand, such a flow control mechanism requires knowledge about the occupancy of both input and output buffers. The flow-control mechanism is therefore a distributed algorithm, which involves exchanging flow-control data between the input and output buffers. Such flow-control data is actually transferred by the switch core (not shown in figure 5.1). Remark that next to bandwidth factors A and B and the total capacities of the input and output buffers, also the duration of transferring packets and/or flow-control data through the switch core as well as the update period for the flow-control mechanism are parameters for a performance model.

The above explanation of the Internet Router forms the basis of applying the UML profile for SHE. From figure 5.1, it can be concluded that the basic active resources of the Internet Router are the input buffers, the switch core and the output buffers, while its environment is formed by the sources and sinks. Hence, it seems natural to model each of these components as a separate instance of an appropriate process class. However, the exact number of instances of such process classes depends on the parameters N and M (which can be large). It is therefore useful to model similar components of the system with similar concurrent activities within one process, see modelling pattern 4 in section 4.3.1. Figure 5.2 and 5.3 depict[2] the resulting instance structure diagram and class diagram respectively. All $N \cdot M$ sources are modelled as concurrent activities within a single instance of the process class `Sources`. Similarly, concurrent activities within a single instance of the process classes `InputBuffers`, `OutputBuffers` and `Sinks` model the behaviour of all N input buffers, N output buffers and N sinks respectively. The behaviour of process class `SwitchCore` represents the transfer of packets via all N^2 virtual connections as actually realised by the switch core. Finally, a cluster class named `InternetRouter` is defined to group the processes that model the Internet Router's components.

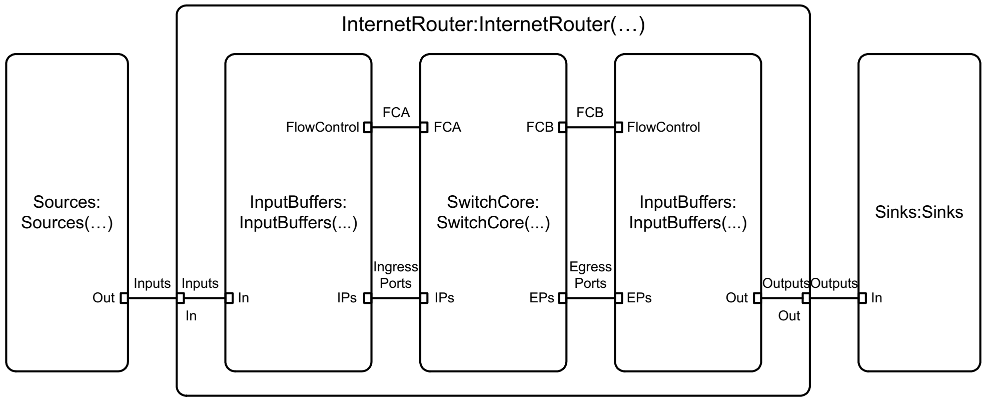


Figure 5.2: Instance structure diagram for the Internet Router (white box view).

The instance structure diagram in figure 5.2 shows separate ports and channels for transferring packets and flow-control data through the switch core. In the implementation of the Internet Router, these items will not be transferred through separate physical connections. The proposed modelling approach is considered to be adequate since only the delay for transferring flow-control data through the switch core is essential for evaluating the flow control mechanism. Remark that this aspect is logged as a consideration regarding the adequacy of abstractions.

Next to using similar concurrent activities to model similar components, many other modelling patterns of section 4.3.1 are applied. Data classes `File` and `Packet` are introduced to model the files/packets generated by the Internet-traffic sources. Instances of data class `Packet` are considered to be the elementary unit of data in the model. Communicating instances of `Packet` between the `Sources` and `InputBuffers` and between the `OutputBuffers` and `Sinks` is performed with fixed bandwidths. So, it possible to use approach B of modelling pattern 2. For modelling the transfer of packets through the switch core, possible changes in the bandwidth for each virtual connection as imposed by the flow control mechanism must be taken into account. This is accomplished by using a modified version of modelling pattern

[2]To limit the size of diagrams shown in this chapter, some information is omitted (indicated by dots).

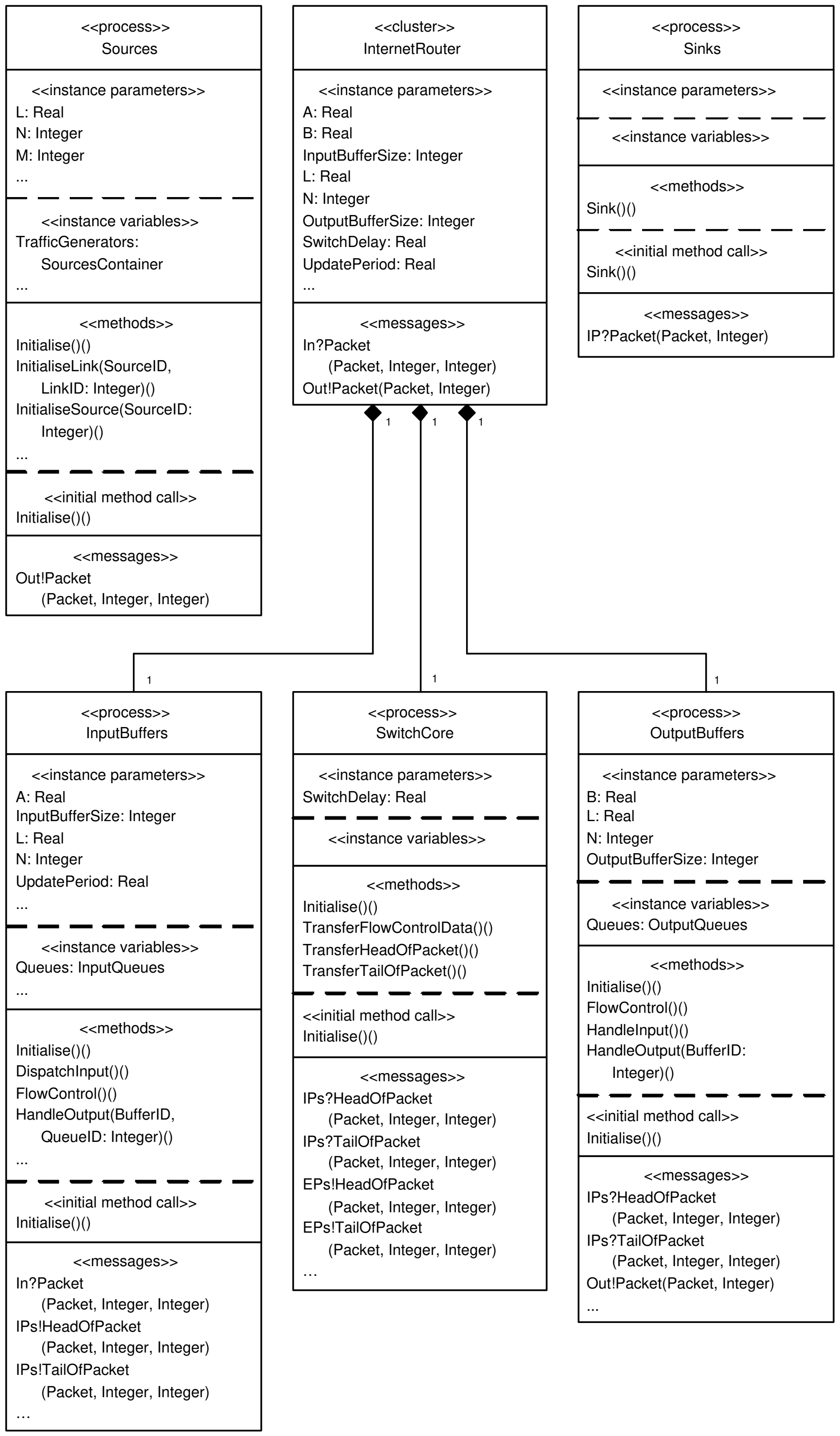


Figure 5.3: Cluster and process classes defined for the Internet Router.

5 for the `InputBuffers`. The communication of packets between the `InputBuffers` and `OutputBuffers` is based on approach A of modelling pattern 2.

In accordance with using a parameterised number of similar concurrent activities to model similar components, parameterised container data classes are introduced for the `Sources`, `InputBuffers` and `OutputBuffers` processes. These data classes provide methods for performing any data-related operation, see also figure 5.3. The `Sources` process uses data object `TrafficGenerators` to determine the next file of packets that is to be sent by any of the $N \cdot M$ source activities (similarly as in example 4.3). The `InputBuffers` process relies on a data object `Queues` of class `InputQueues` to represent all N^2 input queues, while `OutputBuffers` uses a data object `Queues` of class `OutputQueues` to represent all N output queues. The latter two data classes also include methods to perform all computations regarding the flow-control mechanism. Since no operations on data are to be performed by the `SwitchCore` and `Sinks`, no container data classes are defined for these processes.

An important aspect of modelling the Internet Router is scalability. This scalability refers to the relation between increasing the size of the system (expressed as $N \times N$) and the resulting increase in the time needed to execute the model (under the assumption of constraining the simulation time) [28]. Based on the observations summarised in table 5.2, it can be concluded that the total number of concurrent activities in the Internet Router is of order $\mathcal{O}(N(N + M))$. Although optimisations for simulation speed should in general be postponed until completing the development of an intuitive model, certain model optimisations are necessary to enable evaluating the flow-control mechanism for realistic values of N and M (which may give over a million concurrent activities) within realistic simulation time. Therefore, an initial executable model of the Internet Router was developed first, which already included some optimisations for simulation speed. This initial model uses dynamic creation and termination of concurrent activities for the `SwitchCore` and `Sinks`, which is fairly easy to model in POOSL and is still very intuitive (see also below). Moreover, the input handler activities for the `InputBuffers` and `OutputBuffers` are created and terminated dynamically in a similar way as in example 4.4 on page 126. For the `InputBuffers`, method `DispatchInput` not only receives a packet from any of the $N \cdot M$ sources and stores (dispatches) it into the appropriate input queue, but is in addition always prepared to handle a packet from another source. For `OutputBuffers`, a similar approach is used, but now in combination with approach B of modelling pattern 2 to take possible changes in the bandwidth of the involved virtual connection into account[3]. In the initial model, the output handler activities are created statically and guards are used to check the availability of packets. After validating this initial model, further optimisations were made to improve scalability. These optimisations included the replacement of the guards by dynamically created and terminated output handler activities, see modelling pattern 8.

Figure 5.4 shows two activity diagrams concerning the behaviour of the `SwitchCore` process. The left-hand side activity diagram denotes the creation of concurrent activities by method `Initialise` for transferring flow-control data and for transferring the messages for sending the head and tail of a packet. Notice that only a

[3]Notice that receiving the tail of a packet must be performed by the same activity that received the head of that packet. This can be accomplished by matching the identifier of the virtual connection (a pair of integers that are sent as parameters of the messages) and the identity of the output buffer involved.

Sources	$\rightarrow$	$N \cdot M$	Traffic Generators
Input Buffers	$\rightarrow$	$N \cdot M$	Input Handlers
	$\rightarrow$	N^2	Output Handlers
Switch Core	$\rightarrow$	N^2	Virtual Connections
Output Buffers	$\rightarrow$	N^2	Input Handlers
	$\rightarrow$	N	Output Handlers
Sink	$\rightarrow$	N	Traffic Receivers

Table 5.2: Number of concurrent activities in the Internet Router.

fork transition is drawn (and no joint transition), which indicates that the created activities are not terminated by the `Initialise` method. The right-hand side activity diagram in figure 5.4 shows the behaviour defined by method `TransferHeadOf-Packet`. After receiving a message `HeadOfPacket`, an activity is created to forward this message to the `InputBuffers` and another activity is created to handle any other message `HeadOfPacket` that might be received (and which relates to a different virtual connection). The time transition annotated with `SwitchDelay` models the duration of propagating the head of the involved packet through the switch core. After actually sending the message `HeadOfPacket` to the `OutputBuffers`, the activity concerning the packet terminates. With this approach, the number of activities for transferring a message `HeadOfPacket` equals 1 plus the number of virtual connections that are actually in use. The blocking of POOSL's receive statement ensures that no unlimited amount of concurrent activities is created. Notice that this approach is just as intuitive as statically creating N^2 activities that only perform the behaviour expressed in the right hand path of the activity diagram.

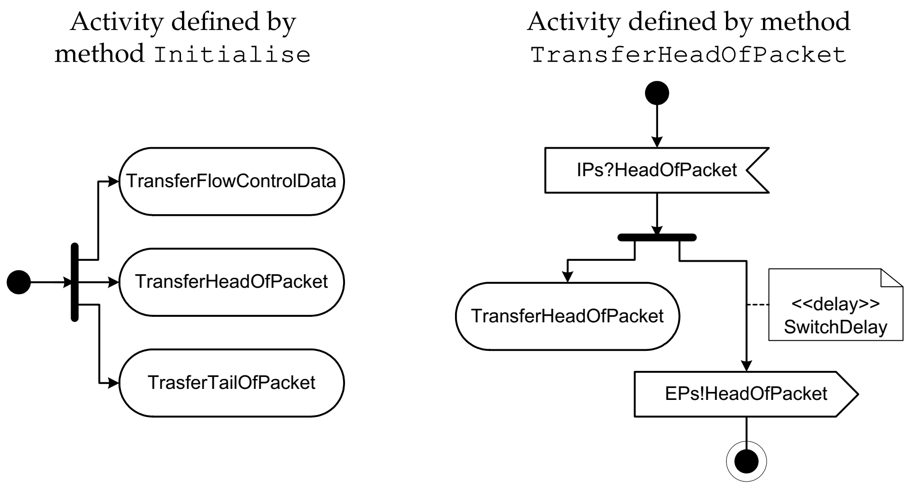


Figure 5.4: Two of the activities performed by the `SwitchCore`.

Next to documenting the concept model of the Internet Router by means of stereotyped UML diagrams and explanatory texts, a number of performance metrics is defined for the Internet Router. From the text of the initial specification at the beginning of this section, it can be concluded that the probability of loosing a packet, the

Component	Performance Metric	Type
Input Buffer	Average Buffer Occupancy	Long-run time average
	Variance in Buffer Occupancy	Long-run time variance
	Loss Probability	Long-run sample average
	Maximum Occupancy	Maximum
Output Buffer	Average Buffer Occupancy	Long-run time average
	Variance in Buffer Occupancy	Long-run time variance
	Loss Probability	Long-run sample average
	Maximum Occupancy	Maximum
Sink	Latency	Long-run sample average
	Jitter	Long-run sample variance
	Throughput	Long-run time average

Table 5.3: Performance metrics defined for the components of the Internet Router.

latency of transferring a packet through the system and the throughput are essential performance metrics. Since packets can be lost at many places (there are many buffers that can be full) and since the latency and throughput can be determined at several places in the system (e.g., at any of the outputs), it must be formulated more precisely what the exact performance metrics are. Table 5.3 gives an overview of the performance metrics defined for the individual components of the Internet Router. The performance metrics for each of the input buffers are defined as aggregate metrics for all the N input queues together. This is because loosing a packet for example depends on the aggregate occupancy of the input queues together. Next to evaluating the probability of loosing a packet at each input and output buffer, their average occupancy and maximum occupancy as well as the variance in their occupancy are to be evaluated. This evaluation gives more insight in the possibilities for dimensioning the buffer sizes. At each sink, the average duration and the variance in the duration of transferring a packet from an input to an output are evaluated (performance metrics Latency and Jitter respectively). Finally, the average utilisation of each output (throughput) is analysed at the corresponding sink.

Remark that each line in table 5.3 actually refers to N different performance metrics. Hence, it is necessary to formulate more precisely what is actually meant by *the* loss probability, latency and throughput of the Internet Router. Let p_i denote the probability of loosing a packet at input buffer i and let p_o be the packet loss probability at output buffer o. Then, *the* packet loss probability P of the Internet Router equals

$$P = \frac{1}{N}\sum_{i=1}^{N} p_i + \left(1 - \frac{1}{N}\sum_{i=1}^{N} p_i\right) \cdot \frac{1}{N}\sum_{o=1}^{N} p_o$$

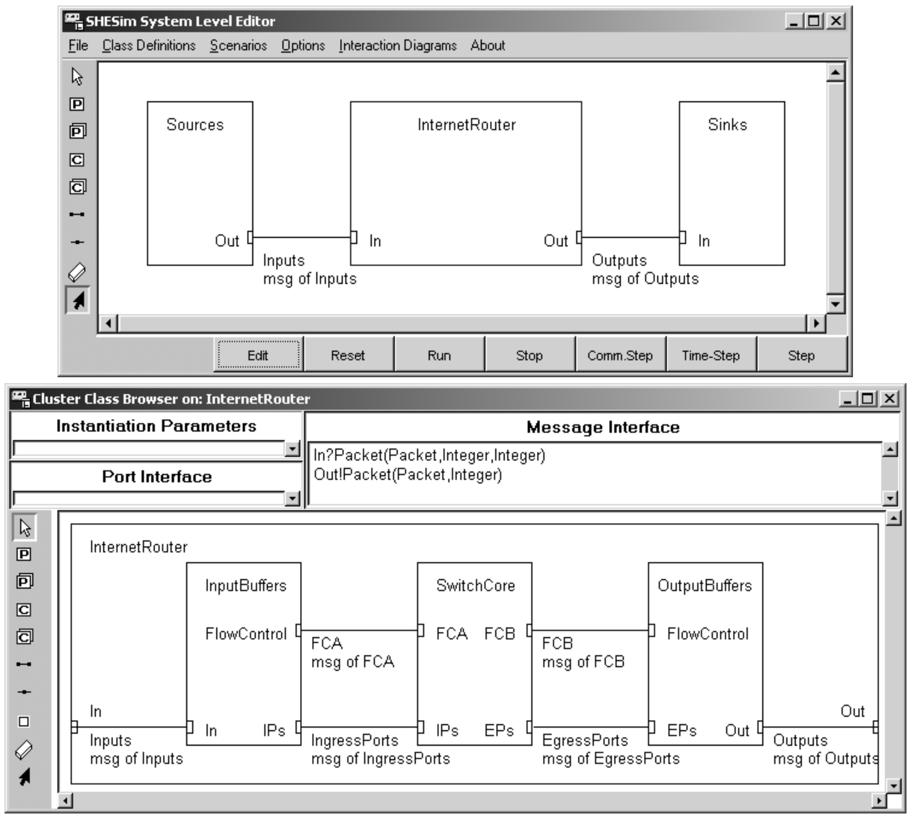
Moreover, *the* latency L of the Internet Router and *the* throughput T are defined as

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i \quad \text{and} \quad T = \sum_{i=1}^{N} T_i$$

where L_i and T_i denote the latency and throughput at sink i respectively.

Formalisation After formulating the concepts and requirements of the Internet Router, the stereotyped UML diagrams are formalised into an executable POOSL model. Formalisation of the static structure turned out to be fairly easy. Figure 5.5 shows a screendump of the formalised instance structure diagrams in the SHESim tool. During the first attempts of formalising the dynamic behaviour, the model was already extended with monitors to enable its validation based on premature performance results as advised in section 4.3.3. The premature performance results showed unexpected high input buffer occupancies and high packet losses[4]. A thorough investigation learned that changing the bandwidth for a virtual connection by the flow-control mechanism did not take immediate effect. In case of sending a packet with a previously assigned low bandwidth, the assignment of a new higher bandwidth only took effect after completely sending the packet with the lower bandwidth. This caused a delay in the removal of other packets in highly occupied buffers. According to the concept model, bandwidth changes should take effect immediately. The solution to the improper formalisation eventually resulted in modelling pattern 5 for adequately modelling dynamically changing bandwidth.

After concluding that the model represented the behaviour of the Internet Router in



```
In?Packet(Packet,Integer,Integer)
Out!Packet(Packet,Integer)
```

Figure 5.5: POOSL model of the Internet Router in SHESim.

[4]Because of the symmetrical structure of the Internet Router, performance results for the input buffers, output buffers or sinks are expected to be similar. The very first performance results obtained with the SHESim tool showed an unbalance in the buffer occupancies. This was caused by an improper implementation of the random number generators, which was corrected at that time (see also appendix B).

a proper way, the model was fully optimised for simulation speed as explained above. To illustrate the used combination of modelling patterns discussed in section 4.3.1, the `OutputBuffers` process of the optimised model is elaborated on. Next to a concurrent activity concerning the flow-control mechanism, method `Initialise` creates an activity for receiving packets from the `SwitchCore` and storing them in the appropriate output queues. The behaviour of this activity is defined by method `HandleInput`, which is depicted in figure 5.6. When receiving the head of a packet `P` in line 3, 4, it is checked atomically whether `P` can still be stored in the appropriate output buffer. If so, storage resources are reserved for `P` conform the modelling approach illustrated in figures 4.7 and 4.8 on pages 120 and 121 respectively. Then, two concurrent activities are created. One for handling `P` further (lines 6 - 18) and one for handling another packet (line 20). Notice that this other packet may be destined for the same output as `P` is. The handling of `P` is completed after waiting until receiving the tail of `P`. As noted above, *only* the concurrent activity that received the head of `P` should continue with handling `P`, which is accomplished by means of the condition in line 7. Atomically with receiving the tail of `P`, it is checked whether an output handler activity is active for the output buffer involved. Notice that the container data object `Queues` stores the information about which output handlers are active. In case an output handler activity should be created, this is done so in line 17. The formalisation of the output handler activity by method `HandleOutput` (see figure 5.6) relies on the removal of guards as proposed by modelling pattern 8.

Extending the POOSL model with performance monitors is performed in accordance with the suggestions given in section 4.3.2. Since there are performance metrics to be evaluated for the input buffers, output buffers and sinks, the behaviour of the `InputBuffers`, `OutputBuffers` and `Sinks` processes must be extended. Therefore, subclasses `InputBuffers_PA`, `OutputBuffers_PA`, `Sinks_PA` are defined. In addition, a cluster class `InternetRouter_PA` is introduced to group an instance of `InputBuffers_PA`, `SwitchCore` and `OutputBuffers_PA` in a similar way as cluster `InternetRouter` in the original model. To minimise the differences between the classes `InputBuffers_PA` and `InputBuffers` and between the classes `OutputBuffers_PA` and `OutputBuffers`, subclasses of the container data classes `InputQueues` and `OutputQueues` are defined. Their subclasses `InputQueues_PA` and `OutputQueues_PA` include the actual performance monitors for the input buffers and output buffers. In addition, a container data class `SinksContainer_PA` is introduced to evaluate the performance metrics defined for the sinks.

The long-run average performance metrics in table 5.3 are evaluated by using monitors of the POOSL library classes for accuracy analysis introduced in section 3.3.1. The local recurrence condition of an empty buffer is used to evaluate the average occupancies and variances in occupancies of the input buffers and output buffers. For the loss probabilities, the rare event of actually loosing a packet served as recurrence condition (see also example 3.6). The performance metrics defined for the sinks are estimated conform the batch-means technique. To enable evaluating the latencies at the sinks, data class `Packet` is extended with an instance variable `EntranceTime` that stores the time at which the head of a packet is received by an input buffer. The latency for each packet then equals the time at which the head of the packet is sent to a sink minus its `EntranceTime`. The maximum buffer occupancy metrics in table 5.3 are evaluated by a simple expression that tests the occupancy of the buffer for

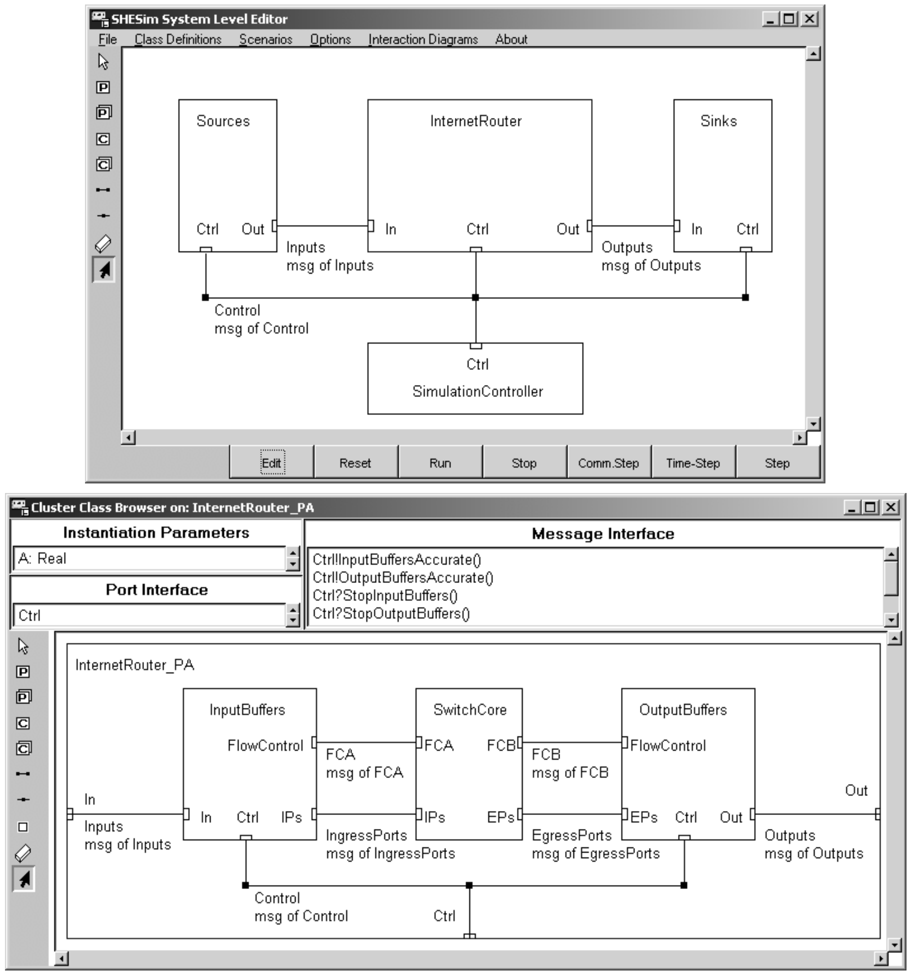
```
HandleInput()() |P, PTail: Packet, FromInputBuffer, ToOutputBuffer,
  BufferID: Integer, ReservationSucceeded, StartOutputHandler: Boolean|

EPs?HeadOfPacket(P, FromInputBuffer, BufferID)
  {ReservationSucceeded := Queues checkOccupation(P, BufferID)};
par
  EPs?TailOfPacket
    (PTail, FromInputBuffer, ToOutputBuffer | PTail getNumber = P getNumber)
    {StartOutputHandler := false;
      if ReservationSucceeded then
        Queues putPacket(P, BufferID);
        if Queues outputHandlerIsNotActive(BufferID) then
          Queues activateOutputHandler(BufferID);
          StartOutputHandler := true
        fi
      fi};
  if StartOutputHandler then
    HandleOutput(BufferID)()
  fi
and
  HandleInput()()
rap.
```

```
HandleOutput(BufferID: Integer)() |P: Packet, Continue: Boolean|

{Continue := Queues isNotEmpty(BufferID);
  if (Continue not) then
    Queues deactivateOutputHandler(BufferID)
  else
    P := Queues inspectPacket(BufferID) setTransmissionBitRate(L)
  fi};
if Continue then
  Out!Packet(P, BufferID);
  delay(P getSize / P getTransmissionBitRate);
  Queues removePacket(BufferID);
  HandleOutput(BufferID)()
fi.
```

Figure 5.6: Formalising the packet handling behaviour of the `OutputBuffers`.

being larger than the previously stored maximum occupancy. This test is performed in case reserving buffer resources for storing a packet is successful.

Next to extensions for evaluating the performance metrics in table 5.3, behaviour is added to enable automatic termination of a simulation run. For this purpose, a process class `SimulationController` is introduced and a subclass `Sources_PA` is defined for `Sources`. Figure 5.7 shows a screendump of the extended POOSL model in SHESim. When the estimation results at the `InputBuffers`, `OutputBuffers` and `Sinks` are accurate, these processes send a message to the `SimulationController`. `SimulationController` stores the overall accuracy status in a container data object of class `SimulationStatus_PA`. In case the results for all performance metrics are accurate, then the `Sources`, `InputBuffers`, `OutputBuffers` and `Sinks` are notified by the `SimulationController` to initiate the logging of the estimation results. Then, the behaviour of all these processes is terminated. Notice the addition of the channels named `Ctrl`, which are used to exchange the messages regarding the accuracy status of estimation results and the termination of a simulation.



```
Ctrl!InputBuffersAccurate()
Ctrl!OutputBuffersAccurate()
Ctrl?StopInputBuffers()
Ctrl?StopOutputBuffers()
```

Figure 5.7: Extended POOSL model of the Internet Router in SHESim.

To illustrate the extensions for automatically terminating a simulation in more detail, figure 5.8 shows the method `Initialise` of process class `OutputBuffers_PA`. After creating and initialising the container data object `Queues`, the normal behaviour of the `OutputBuffers_PA` process is started. This normal behaviour consists of an input handling activity (which starts other input handler and output handler activities whenever necessary) in line 5 and an activity regarding the flow-control mechanism in line 7. Concurrently with the normal behaviour, it is checked in line 9 whether the estimation results for the performance metrics are accurate. When the guard evaluates to **`true`**, the `SimulationController` is notified by sending the message `OutputBuffersAccurate`. The use of the **`abort`**-statement results in terminating the normal behaviour in case a message `StopOutputBuffers` is received from the `StopController`. Atomically with receiving this message, the estimation results for the performance metrics are logged to file.

The behaviour of the `SimulationController` is defined by the two methods de-

```
Initialise()()

Queues := new(OutputBuffers_PA) init(N, B, L, BufferCapacity);
abort
   par
      HandleInput()()
   and
      FlowControl()()
   and
      [Queues accurate] Ctrl!OutputBuffersAccurate
   rap
with
   Ctrl?StopOutputBuffers{Queues stopSimulation}.
```

Figure 5.8: Method `Initialise` of process class `OutputBuffers_PA`.

picted in figure 5.9. Method `Initialise` creates and initialises the container data object `SimulationStatus` of class `SimulationStatus_PA` (see also above). The behaviour of checking the accuracy of all performance metrics as specified by method `CheckAccuracy` is aborted when all performance metrics are indeed accurate (line 4). Then, the `Sources`, `InputBuffers`, `OutputBuffers` and `Sinks` are notified to log the obtained estimation results and terminate their behaviour. Notice the combination of tail-recursion and the **`sel`**-statement in the `CheckAccuracy` method to express the reception of the accuracy status from all three processes `InputBuffers`, `OutputBuffers` and `Sinks`.

```
1    Initialise()()

3    SimulationStatus := new(SimulationStatus_PA) init;
4    abort CheckAccuracy()() with [SimulationStatus accurate] Ctrl!StopSources;
5    Ctrl!StopInputBuffers; Ctrl!StopOutputBuffers; Ctrl!StopSinks.
```

```
6    CheckAccuracy()()

7    sel
8       Ctrl?SinksAccurate{SimulationStatus sinksAccurate}
9    or
10      Ctrl?InputBuffersAccurate{SimulationStatus inputBuffersAccurate}
11   or
12      Ctrl?OutputBuffersAccurate{SimulationStatus outputBuffersAccurate}
13   les;
14   CheckConfidence()().
```

Figure 5.9: Terminating the simulation when estimation results become accurate.

Evaluation The extended POOSL model has been simulated for many different configurations of N, M and other parameters of the Internet Router [169]. Although the obtained estimation results allowed assessing the quality of the flow-control mechanism and gave much information for dimensioning the buffer capacities, some deficiencies were discovered for the applied estimation techniques. For many configurations of N and M, loosing packets turned out to be really rare (as desired). Due to using the loss of a packet as recurrence condition for estimating the loss probabilities, it took unacceptably long to get accurate estimation results for these metrics.

Therefore, it was decided not to take the accuracy of the loss probabilities into account when terminating a simulation. This experience confirmed the need for future work on efficiently estimating the probability on rare events as noted in section 3.2.

5.2 Network Processor

Network processors are flexible single-chip architectures for telecommunication systems, which are capable of processing packets at wire speed [59]. The network processor considered in this section is based on the use of several existing components. It includes an embedded PowerPC together with a similar on-chip bus architecture as in [84] and network interface components from [83]. The design issue is to decide whether the proposed combination of components and memories will yield a packet processing system with a low latency and high throughput. Of special interest is the identification of potential performance bottlenecks in the system, which may surface due to the sharing of the on-chip busses or the limited capacities of memories.

Formulation The system architecture of the network processor is depicted in figure 5.10. It combines a compute part for high-speed packet processing and a peripheral part for interfacing to the network environment. In each part, a bus provides the inter-component connectivity, while communication between the two parts is based on bridges. The compute part includes the PowerPC core. It is connected to the (high-performance) Processor Local Bus (PLB) for accessing an on-chip SRAM memory and to a controller for off-chip SDRAM memory. The SDRAM memory stores the packets that will be processed by the PowerPC, while the information on where these packets are stored is available from the SRAM memory. The peripheral part consists of the (low-performance) On-chip Peripheral Bus (OPB) and a number of Ethernet Communication Macros (EMAC) from [83], which form the interface to the network environment. Each EMAC is capable of concurrently receiving and transmitting packets from/to an Ethernet medium. The compute and peripheral part are interconnected by the standard PLB-OPB and OPB-PLB bridges of [84] and a specialised bridge called Memory Access Layer (MAL). The PLB-OPB and OPB-PLB bridges are merely used during reconfiguration procedures for initialising the EMACS. Transfer of packets between the compute part and peripheral part during normal operation is performed by the MAL, which can concurrently access both PLB and OPB busses. Together, the OPB and MAL can serve up to 32 EMACS. Finally, the (low-speed) daisy-chained Device Control Register (DCR) bus is used during reconfiguration procedures for exchanging configuration and status information between the PowerPC processor core and all other components.

For analysing the performance of the network processor, the use of the PLB-OPB and OPB-PLB bridges as well as the DCR bus during reconfiguration procedures is not really relevant. Moreover, the actual content of the SDRAM memory is not relevant for evaluating its occupancy since the SDRAM's occupancy is a property that the SDRAM controller must take into account. Based on these considerations, the DRC bus, PLB-OPB and OPB-PLB bridges and the SDRAM memory are not modelled. Next to these abstractions, many others were made. The available documentation of the existing components included a lot of implementation details and hence,

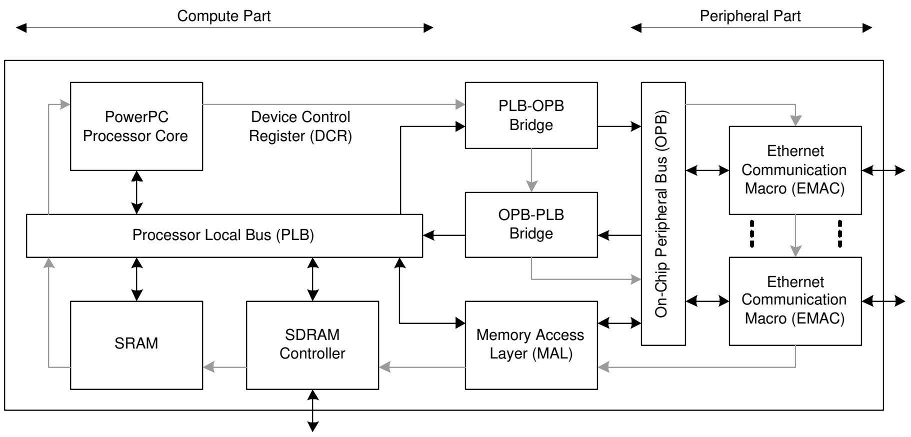

Compute Part
Peripheral Part
PowerPC Processor Core
Device Control Register (DCR)
PLB-OPB Bridge
OPB-PLB Bridge
Processor Local Bus (PLB)
On-Chip Peripheral Bus (OPB)
Ethernet Communication Macro (EMAC)
Ethernet Communication Macro (EMAC)
SRAM
SDRAM Controller
Memory Access Layer (MAL)

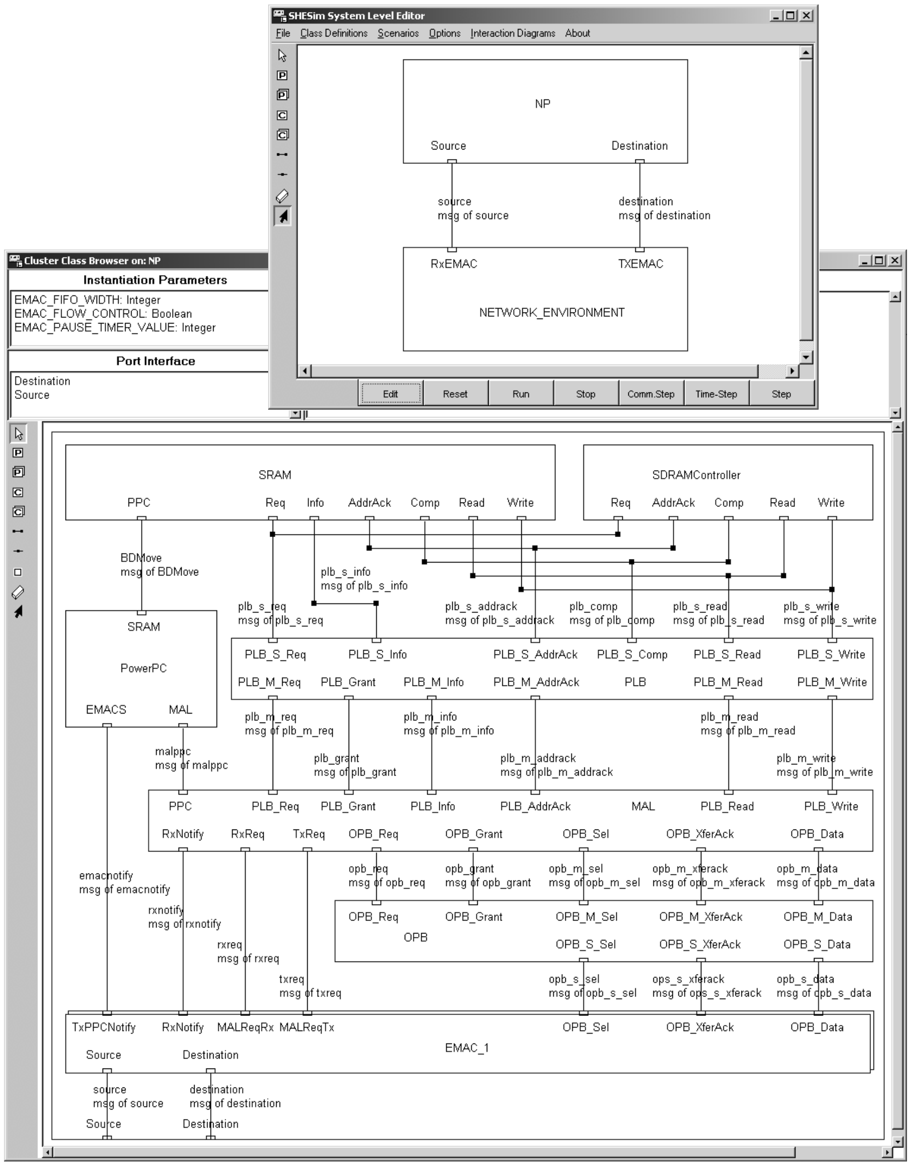
Figure 5.10: System architecture of the network processor.

a process of identifying the essential aspects for performance modelling had to be performed. Eventually, several diagrams (not necessarily expressed in UML) and explanatory texts documented the functionality of the system at a level of detail that sufficed for developing an adequate performance model [103].

This section does not intend to discuss the formulation stage for the network processor in much detail. Nevertheless, a brief overview of the working of the system is given to enable explaining the application of some of the modelling patterns discussed in section 4.3.1. In case of receiving a new packet, the involved EMAC notifies the PowerPC to set (in the SRAM memory) the location where the packet has to be stored. Then, the EMAC requests the MAL to store the packet into the SDRAM memory based on the location information in the SRAM memory. After completely storing the packet, the MAL notifies the PowerPC that processing of the packet can be started. Then, the PowerPC reads the packet from the SDRAM memory, processes it and finally updates the SDRAM memory. Upon completion of processing the packet, the EMAC concerning the destination output requests the MAL to fetch it from the SDRAM memory for transmitting it over the Ethernet medium. After transmitting the packet, its location information in the SRAM memory is cleared.

Concurrently receiving and transmitting packets over different Ethernet media involves sharing many components (including the PLB, OPB and MAL) and buffering data before accessing a shared component. Several arbitration mechanisms are required to handle accesses to these shared components. Since the input traffic is stochastic, it cannot easily be deducted how these arbitration mechanisms affect the overall performance of the network processor. To evaluate whether the arbitration mechanisms perform sufficiently well under various conditions (such as the worst-case condition where all EMACs constantly receive packets destined for a single EMAC), a number of performance metrics have been defined. These include the utilisation of the OPB and PLB busses as well as their throughput. In addition, the average and maximum occupancy of all buffers in the EMACS and MAL and also the average and maximum occupancy of the SDRAM are to be evaluated. Finally, the latency of transferring a packet through the network processor as well as the throughput at the network processor's ports to the Ethernet media are of interest.

Formalisation Based on the concept model, an executable POOSL model has been developed. Figure 5.11 depicts a screen dump of the SHESim tool showing this model. The top window shows a cluster `NP`, which models the actual network processor as depicted in figure 5.10. The cluster `NetworkEnvironment` represents the network environment. Inspection of the `NP` cluster opens the bottom window in figure 5.11. It contains the clusters and processes that model the relevant components of the system as indicated above. The EMACs are modelled using a multiple (see also section 4.2.2) of `EMAC` clusters, which share the channels for communicating control information to the `MAL` and `PowerPC` and packet data to the `OPB` respectively. Noti-



Instantiation Parameters

EMAC_FIFO_WIDTH: Integer
EMAC_FLOW_CONTROL: Boolean
EMAC_PAUSE_TIMER_VALUE: Integer

Port Interface

Destination
Source

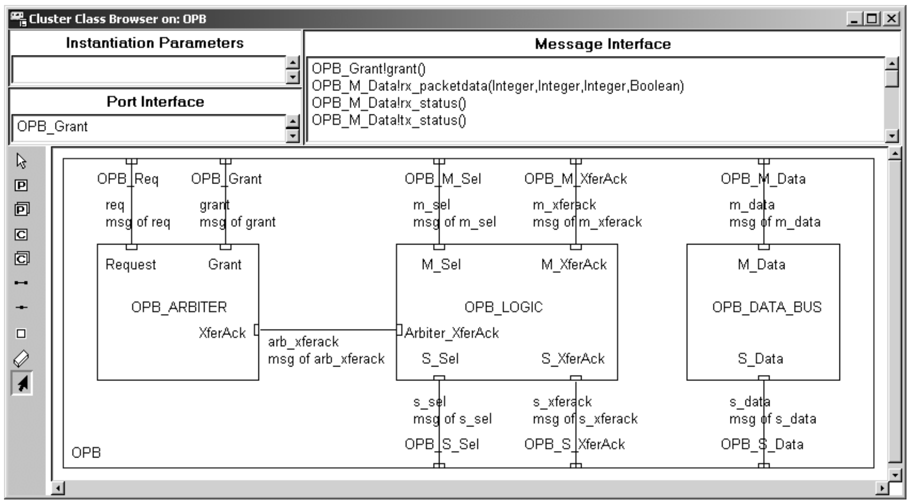
Figure 5.11: POOSL model of the network processor.

ce that the model has many more channels between the components than shown in figure 5.10. This was a consequence of the decision to represent some more detail in the model regarding the physical connections between the components to ease discussions for validation purposes with the authors of the detailed documentation available on the existing components.

The model of the network processor combines many of the modelling patterns presented in section 4.3.1, including abstraction from the details of packets. Also the implementation details of the interfaces between the components were abstracted from; only the conceptual `Request` and `Grant` messages resulting from protocols for exchanging information between components were modelled. On the other hand, the model is intended for evaluating the bus arbitration mechanisms. These arbitration mechanisms include several special features that complicate predicting their effect on the division of the available bus bandwidth. Since this effect is essential for evaluating the performance, these arbitration mechanisms are modelled in more detail.

Consider the arbitration of requests for accessing the OPB. The OPB includes an arbiter for granting access requests from connected master components. In the network processor, the MAL is the only master on the OPB, while the EMACs are all slave components. As a result, arbitration merely involves checking the presence of an access request from the MAL and granting it. Nevertheless, the arbitration mechanism facilitates *overlapped bus arbitration* [85]. This feature concerns arbitration of the next bus access concurrently with the final data transfer cycle of the current bus access. EMACs indicate the final data transfer cycle of their bus access by sending a data transfer acknowledgement to the OPB. The effect of overlapped bus arbitration is a higher OPB utilisation because it saves one idle cycle per bus access. This effect must be taken into account for developing an adequate performance model.

To model the effect of overlapped bus arbitration, access requests from the MAL are received independently from receiving data transfer acknowledgements. The model of the OPB is shown in figure 5.12. It includes the process named `OPB_ARBITER`, re-



```
OPB_Grant!grant()
OPB_M_Data!rx_packetdata(Integer,Integer,Integer,Boolean)
OPB_M_Data!rx_status()
OPB_M_Data!tx_status()
```

Figure 5.12: POOSL model of the OPB.

presenting the OPB arbiter. The behaviour of the OPB arbiter is modelled with three concurrent activities, which are created by the method `Init` depicted in figure 5.13. After creating a new container data object `Status` of class `OPB_STATUS` and initialising it by calling method `clearAll`, the **`par`**-statement is used to create the three concurrent activities. Method `ReceiveMALRequest` models the reception of bus access requests from the MAL and `ReceiveXferAck` represents the reception of a data transfer acknowledgement. Method `Arbitrate` specifies the behaviour of actually arbitrating between the available requests and granting them.

```
1   Init()()

2   Status := new(OPB_STATUS) clearAll;
3   par ReceiveMALRequest()() and ReceiveXferAck()() and Arbitrate()() rap.
```

Figure 5.13: Initialising the arbitration of OPB access requests.

Figure 5.14 presents the methods that specify the three concurrent activities in the `OPB_ARBITER`. They share container data object `Status`, which has (Boolean) instance variables named `ReqAvailable` and `AckReceived`. These variables are used to indicate whether a MAL request for bus access is received and whether a data transfer acknowledgement from an EMAC is received respectively. The status of `ReqAvailable` can be set, retrieved and cleared by the methods `setReq`, `getReq` and `clearReq` of `OPB_STATUS`. Similar methods are defined for `AckReceived`. Method `ReceiveMALRequest` in figure 5.14 specifies the reception of an access request from the MAL. When such request is received, instance variable `ReqAvailable` of `Status` is atomically set to **`true`**. Method `ReceiveXferAck` atomically sets `AckReceived` to **`true`** upon reception of a data transfer acknowledgement.

```
1   ReceiveMALRequest()()

2   Request?Request{Status setReq}; ReceiveMALRequest()().
```

```
3   ReceiveXferAck()()

4   XferAck?XferAck{Status setAck}; ReceiveXferAck()().
```

```
5   Arbitrate()()

6   [Status getReq]
7         Grant!Grant{Status clearReq};
8   [Status getAck]
9      Status clearAck;
10  Arbitrate()().
```

Figure 5.14: Synchronising concurrent activities via container data object `Status`.

In line 6, the status of `ReqAvailable` is retrieved (by using method `getReg`) in order to determine whether to grant an access request for the MAL. As long as such a request is not received, the send statement for granting the request in line 7 remains blocked. In case of receiving a request from the MAL, the send statement is executed and `ReqAvailable` is atomically reset to **`false`**. Since the OPB can handle only one access at a time, arbitration of the next MAL request must wait until `AckReceived` indicates that a data transfer acknowledgement is received. Notice that some time may elapse between receiving an access request and the data transfer acknowledgement for that access. The exact amount of time is unknown to the OPB arbiter because it depends on the responsiveness of the EMAC involved. When `AckReceived` is set by the concurrent activity for receiving the data transfer acknowledgement, it is cleared at the same moment in line 9. Then, method `Arbitrate` is called again to handle the next MAL request. Notice that if the next MAL request is obtained while receiving the data transfer acknowledgement, it is granted (at the same moment) by the next call of `Arbitrate`. This adequately models overlapped bus arbitration.

The example of modelling the OPB arbiter illustrates the application of the modelling pattern for synchronising concurrent activities. Notice that because of the timeless execution of actions, receiving the next access request may happen at the same moment (in model time) as receiving the data transfer acknowledgement for the previous access. The exact order in which the corresponding messages are received at such a moment is determined non-deterministically. Even if, at the same moment, `AckReceived` is set to **`true`** before `ReqAvailable` is, the effect is still overlapped bus arbitration (as intended). The final implementation of the network processor requires synchronising all events with a clock, including receiving requests and acknowledgements and granting requests. The model abstracts from this clock by relying on the asynchronous concurrency and the possibility to synchronise concurrent activities in only those cases for which synchronisation is really required.

After validating the POOSL model, it was extended with several performance monitors. To evaluate the utilisation of the OPB bus, the OPB arbiter model had to be extended. Conform the suggestions in section 4.3.2, a subclass `OPB_ARBITER_PA` of process class `OPB_ARBITER` is defined. Since the utilisation of the OPB is actually a long-run time average, an instance variable `Utilisation` of class `LongRunTime-Average` from the library classes for accuracy analysis discussed in section 3.3.1 is added to `OPB_ARBITER_PA`. Now, to estimate the utilisation of the OPB conform the batch-means technique, method `Arbitrate` is extended as shown in figure 5.15. Whenever a bus access request is granted, `Utilisation` is rewarded with value 1, while receiving a data transfer acknowledgement rewards `Utilisation` with 0. Notice that if overlapped bus arbitration occurs, `Utilisation` is updated twice at the same moment (in model time). Nevertheless, this does not hinder properly evaluating the utilisation of the OPB because it is a long-run *time* average.

```
Arbitrate()()

[Status getReq]
   Grant!Grant{Status clearReq; Utilisation rewardBM(1, currentTime)};
[Status getAck]
   {Status clearAck; Utilisation rewardBM(0, currentTime)};
Arbitrate()().
```

Figure 5.15: Extending method `Arbitrate` for performance evaluation.

Evaluation The extended model of the network processor has been simulated for several types of input traffic [103]. Next to using similar Internet-traffic generators as in the Internet Router case study (see section 5.1), sequences of packets transmitted over a real-life Ethernet connection were used. This was performed by reading the size of the successively transmitted packets from a file based on class `FileIn` (see also appendix B). In addition, different conditions for determining the destination of the packets were used. The obtained estimation results provided good insight in the performance of the different components and the overall latency and throughput of the network processor. In addition, the OPB was identified as a potential bottleneck in the worst-case situation were all packets are destined for the same output [103]. Finally, suggestions for dimensioning the memories and buffers in several components of the network processor were given to improve the performance.

5.3 Conclusions

The academic and industrial case studies that have been performed provided valuable input for identifying theoretical and practical issues involved in the area of performance modelling. Many of the case studies were actually carried out while developing the theories presented in this thesis. As a result, the applicability of the proposed performance analysis techniques, the framework for reflexive performance analysis with POOSL and the extensions to the SHE method were continuously of concern. Re-evaluation of the Internet Router case study enabled to assess the actual applicability of the final theories. Although performance modelling with SHE showed to be successful, several aspects were detected that need further attention. Next to the aspects discussed in chapters 2, 3 and 4, some desirable extensions for the SHESim and Rotalumis tools have been identified. An example is the need for automatically simulating a model for different settings of the system parameters.

Chapter 6

Conclusions

The increasing complexity of hardware/software systems urged the need for system-level design methods. Such frameworks for structuring the earliest phases of the design rely on developing models in order to evaluate the properties of potentially feasible design alternatives. Especially the performance of a design alternative often motivates deciding whether or not to use it as a basis for actually realising the system. Evaluating the performance without a proper mathematical foundation is insufficient for obtaining credible performance results. Modelling languages with a formal semantics allow to define a rigorous framework for applying performance analysis techniques and for unambiguously executing models. The latter is important when estimating performance properties based on simulations. However, using a formal modelling language is insufficient for assisting a designer in making design decisions. Next to guidelines for developing adequate performance models, system-level design methods should provide user-friendly modelling tools that largely automate the actual evaluation of design alternatives in order to minimise design time.

The contributions of this thesis provide ample means for obtaining credible performance results during system-level design. In addition to extending classical performance analysis techniques to enable handling the complexity of today's hardware/software systems, this thesis proposed a mathematical framework for applying them when using the formal modelling language POOSL (Parallel Object-Oriented Specification Language). Furthermore, the system-level design method SHE (Software/Hardware Engineering) is extended with a profile for UML (Unified Modelling Language) and with several guidelines that ease the development of adequate performance models with POOSL. The performed industrial case studies showed the applicability of the developed techniques and design method for very complex systems. An overview of the concrete contributions of this thesis is given in the next section, while section 6.2 discusses some aspects that need further attention.

6.1 Contributions

The contributions of this thesis can be summarised as follows:

- An inventory of the research area of performance modelling for system-level design has been made. This inventory identified the requirements for obtaining credible performance results and revealed several deficiencies in state of art performance analysis techniques, modelling languages, property specification languages, as well as system-level design methods and tools. With the inventory, it has become possible to identify where viable combinations of existing techniques, formalisms, methods and tools need to be extended/modified to ensure obtaining credible performance results during system-level design.

- Evaluating long-run average performance metrics often requires to take a certain condition into account. A simple example of such a conditional long-run average is the average duration of processing packets by a telecommunication system. For this performance metric, the value of the variable denoting the most recent processing time must only be taken into account when completing the processing of a packet. Nevertheless, straightforwardly applying classical Markov-chain based performance analysis techniques in this case requires to consider the value of the variable in all states of the system. With the proposed reduction technique, it has become possible to immediately disregard reward values in irrelevant states in the case of analytical computation as well as simulation-based estimation of conditional long-run averages. Especially in the latter case, performance evaluation speed is improved considerably, thereby greatly increasing the system complexity that can be handled.

- Many common long-run average performance metrics can be defined as an algebraic combination of simple (conditional) long-run averages. Examples of such complex performance metrics are the average occupancy of a buffer and the throughput of an on-chip bus. Although classical Markov-chain based performance analysis techniques provide ample means for computing these performance metrics analytically, a general approach for deriving the accuracy of estimation results obtained for complex long-run averages is missing. With the proposed algebra of Confidence Intervals, the accuracy of estimation results for complex (conditional) long-run averages can be analysed without the necessity to explicitly derive a Confidence Interval. Hence, the algebra of Confidence Intervals has enabled analysing the accuracy of estimation results for many common complex long-run averages for which this was not yet possible.

- Markov-chain based performance analysis techniques for estimating long-run averages require to specify a state that determines the beginning of a regenerative cycle of independent behaviour. Usually, it is impossible to specify such a recurrent state in advance and hence, it is desirable to identify a recurrent state during simulation. One could base this identification on detecting the revisiting of a state. Unfortunately, this approach is often prohibitively complex for industrial hardware/software systems and hence, it is more practical to define a certain recurrence condition that enforces the beginning of a regenerative cycle. Two approaches for defining such recurrence conditions have been proposed. One approach is generally applicable and matches the commonly used batch-means technique. This technique has however some theoretical and practical deficiencies. The other approach relies on knowledge about the behaviour of a specific component of the system, thereby yielding more credible estimation results. With these approaches, it has become possible to apply the simulation-based estimation techniques proposed in this thesis in practice.

- Instead of expressing the behaviour of a system in the form of a Markov chain, it is more convenient to use an expressive modelling language such as POOSL. To enable applying Markov-chain based performance analysis techniques in this case, POOSL models have been mathematically related to Markov chains in the context of a reflexive approach for specifying monitors. This framework for reflexive performance analysis provides a sound basis for implicit application of Markov-chain based performance analysis techniques while executing a POOSL model that is extended with performance monitors. Hence, the proposed framework has enabled a rigorous evaluation of performance metrics based on models constructed with an expressive modelling language.

- Four (conditional) long-run average performance metric types have been identified as most common for hardware/software systems. Based on the framework for reflexive performance analysis with POOSL, the proposed Markov-chain based estimation techniques and the approaches for approximating the beginning of regenerative cycles, library classes for analysing the accuracy of the four performance metric types have been developed. With these performance monitor classes, it has become possible to estimate long-run averages without knowing all details of the mathematical theories involved. The credibility of estimation results obtained when using the performance monitor classes has been assessed with an experiment. The experiment revealed that for the estimated performance metrics (one of each type), utilising knowledge about the re-occurrence of local behaviour to define a recurrence condition is favorable over using the current implementation of the batch-means technique.

- The SHE method has been extended with a framework for constructing and validating performance models with POOSL. This framework integrates several performance modelling guidelines with the existing techniques, guidelines and tools for SHE. The SHE method distinguishing the phases of formulation, formalisation and evaluation. The applicability of the SHE method has been improved by defining a UML profile that facilitates deriving POOSL models from stereotyped UML diagrams in the formalisation phase. In addition, several modelling patterns have been presented next to guidelines for extending POOSL models with monitors and for their validation. With the proposed extensions to the SHE method, a system-level design method has become available that allows rigorous evaluation of the performance (and correctness) of design alternatives for industrial hardware/software systems.

- A number of academic and industrial case studies has been performed. These case studies were very valuable for identifying the theoretical and practical issues involved in the area of performance modelling and also for assessing the developed techniques and method. Next to successfully applying the proposed performance analysis techniques and system-level design method, the actual performance results for the different industrial case studies have been valuable for thoroughly understanding the working of the involved systems and for founding decisions between design alternatives for these systems.

6.2 Future Research

System-level design is an emerging research area that poses new challenges for analysis techniques, formalisms, design methods and tools. Performance analysis is an essential aspect of system-level design because of its impact on taking design decisions. Although this thesis proposed several performance analysis techniques and a framework for applying them during system-level design, several aspects have been identified that require further attention. These include the following:

- Analytical computation of long-run average performance properties remains limited to systems with a relatively small state space. Hence, the performance of most industrial hardware/software systems will be estimated based on simulations. However, application of Markov-chain based estimation techniques suffers from the difficulty of detecting the beginning of a regenerative cycle of independent behaviour. More research is needed to develop an efficient and generally applicable approach for defining recurrence conditions that yield Confidence Intervals with excellent coverage.

- Using a modelling language that allows to express non-determinism suffers from the possibility of obtaining unrealistic performance results in case this non-determinism is not resolved properly. Future research includes examining different options for resolving non-determinism and their effect on the obtained performance results. Only after such an investigation, it can be decided whether the currently used approach for resolving non-determinism when executing POOSL models is indeed favorable. The option of not resolving non-determinism at all requires to develop techniques for deriving the complete range of results that can be obtained for the performance metric involved.

- Although the SHESim and Rotalumis tools have proven to be very valuable, several extensions are desirable. These extensions include possibilities for automatically terminating a simulation without the necessity to explicitly model the behaviour of gathering all accuracy results and invoking deadlock. In other words, all aspects related to ensuring that the estimation results for all performance metrics are accurate when terminating a simulation should be part of the tool instead of the model. Such tool extensions could rely on the `accurate` methods in the performance monitor classes. Another very desirable extension for SHESim and Rotalumis would be the possibility to have an automated way of simulating a model for different settings of system parameters.

- To improve the applicability of the SHE method further, a tool should be developed that integrates support for all three phases of formulation, formalisation and evaluation. An important aspect for developing such a tool is investigating the possibilities of automatically generating POOSL models from stereotyped UML diagrams. Although the proposed UML profile provides sufficient means to automate generating the static structure of a POOSL model, guidelines are needed for expressing the dynamic behaviour of a system in UML diagrams in such a way that the tool can assist in deriving data and process methods.

Appendix A

Mathematical Preliminaries

This appendix establishes notation for several concepts of set theory and probability theory. Based on that, a brief introduction to stochastic processes is given, together with classical techniques for evaluating their long-run behaviour. An introductory overview of some concepts of statistical analysis completes these mathematical preliminaries. This appendix is inspired by [24, 95, 46, 105] and [104].

A.1 Sets, Functions, Operations and Algebras

Sets and Elements A *set* X is an unordered collection of items, which are referred to as the *elements* of X. To indicate that item x is an element of X, $x \in X$ is written. The cardinality or *size* of X is represented with $|X|$; if the number of elements in X is infinite, then $|X| = \infty$ is written. Set X is called *countable* in case the number of elements in X is finite or denumerably infinite. A set with a single element is called a *singleton* set. The set without any elements or *empty set* is denoted by $\varnothing$.

A set can be defined by enumerating all its elements like in $\{a, b\}$ or $\{0, 1, 2, \ldots\}$. The dots in the latter example indicate that the enumeration of elements should be completed in a way that is evident from the listed elements. Another example is the set $\{0, 1, \ldots, 8, 9\}$, where the dots indicate that the enumeration of elements should be completed until the last listed elements. The set of natural numbers $\{0, 1, 2, \ldots\}$ has a denumerably infinite number of elements and is also denoted by $\mathbb{N}$. $\mathbb{R}$ represents the set of real numbers and contains an uncountable number of elements.

An alternative way of defining a set is by reference to elements of another set and specifying a certain property that has to be satisfied by those elements. For example, $\{n \in \mathbb{N} \mid n \text{ is prime}\}$ defines the set of prime numbers by referring to those elements of $\mathbb{N}$ that satisfy the property of being prime.

Set Operations A set X is said to be a *subset* of set Y, denoted by $X \subseteq Y$, if every element of X is also an element of Y. If $X \subseteq Y$, then Y is said to be a *superset* of X, which is also indicated with $Y \supseteq X$. The set containing all subsets of X is called the *powerset* of X and is denoted by $\mathbf{2}^X$. The *difference* of X and Y, denoted by $X \setminus Y$, is the set of those elements in X that are not an element of Y. The *union* of the two sets X and Y is indicated with $X \cup Y$ and their *intersection* with $X \cap Y$. X and Y are said to be *disjoint* in case $X \cap Y = \varnothing$, while the sets $X_1, X_2, \ldots$ are *pairwise disjoint* if $X_i \cap X_j = \varnothing$ for every $i \neq j$.

Cartesian Product The *Cartesian product* of a set X with a set Y is denoted by $X \times Y$ and indicates the set of all *ordered pairs* or *ordered* 2*-tuples* (x, y) with $x \in X$ and $y \in Y$. More generally, the n*-dimensional Cartesian product* $X_1 \times \ldots \times X_n$ of sets $X_1, \ldots, X_n$ is the set of all *ordered* n*-tuples* $(x_1, \ldots, x_n)$ with $x_i \in X_i$ for all i. An ordered n-tuple $(x_1, \ldots, x_n) \in X_1 \times \ldots \times X_n$ is also denoted by $\underline{x}$, where x_i is called the i^{th} *component* of $\underline{x}$. The i^{th} component of $\underline{x}$ is also often referred to by $\underline{x}_i$.

The n-dimensional Cartesian product of a set X is denoted by X^n. A *sequence* of elements of X refers to an ordered n-tuple $\underline{x} \in X^n$. A sequence $\underline{x}$ of elements in $\mathbb{R}$ is called *bounded* (by b) if there exists a $b \in \mathbb{R}$ such that $|\underline{x}_i| \leq b$ for all i. A sequence $\underline{x} \in X^n$ is called *finite* when n is finite. In case n is infinite instead, then $\underline{x}$ is called *infinite*. The *length* of a sequence $\underline{x}$ is denoted by $|\underline{x}|$. If sequence $\underline{x} \in X^n$ is finite then $|\underline{x}|$ is defined as $|\underline{x}| = n - 1$, whereas if n is infinite then $|\underline{x}| = \infty$. A finite sequence $\underline{y} \in X^n$ is called a *prefix* of sequence $\underline{x} \in X^m$ with $m \geq n$ if $\underline{y}_i = \underline{x}_i$ for all $i = 1, \ldots, n$. In such case, $\underline{y}$ is also denoted by $\underline{x}_{1..n}$.

In accordance with conventions used in probability theory, a sequence is sometimes denoted by $\{x_i \mid i \in \mathcal{T}\}$, where $\mathcal{T}$ denotes an ordered set of indices i.

Functions A *function* f from a set X to a set Y is a subset of $X \times Y$, such that for each $x \in X$ there exists at most one $y \in Y$ for which $(x, y) \in f$. If such a y exists, then $f(x)$ is said to be defined and $y = f(x)$ denotes that f assigns y to x. On the other hand, if such a y does not exist, $f(x)$ is said to be undefined, which is denoted by $f(x) = \perp$. The set $\{x \in X \mid f(x) \neq \perp\}$ is called the *domain* of f. The *range* of f is the set $\{y \in Y \mid y = f(x) \text{ with } x \in X\}$. A function f from X to Y is called *complete* in case the domain of f equals X. In such case, $f : X \to Y$ is written.

Operations An n*-ary operation* $\star$ on a set X is a function $\star : X^n \to X$, where the number n is called the *arity* of $\star$. In such case, set X is said to be *closed* under $\star$. An *algebra* is a set X accompanied with one or more n-ary operation(s) (with finite n) defined on X . The terms *unary* and *binary* are used if the arity is 1 and 2 respectively. The negation $-$ is an example of an unary operation on $\mathbb{R}$. Examples of binary operations[1] on $\mathbb{R}$ are the addition $+$ and multiplication $\cdot$. In case $x, y \in X$ and $\star$ is an unary operation on X, the *prefix notation* $y = \star x$ is often used to denote that $y = \star(x)$. Similarly, for $x, y, z \in X$ and a binary operation $\star$ on X, the *infix notation* $z = x \star y$ is commonly used to denote that $z = \star(x, y)$.

In this thesis, the binary operation $\dot{-}$ (pronounced as *monus*) is defined on $\mathbb{R}$ as follows. For any $x, y \in \mathbb{R}$,

$$x \dot{-} y = \begin{cases} x - y & \text{if } x > y \\ 0 & \text{otherwise} \end{cases}$$

[1]Notice that the division / is no binary operation on $\mathbb{R}$ but a complete function from $\mathbb{R} \times \mathbb{R} \setminus \{0\}$ to $\mathbb{R}$.

Extended Real Numbers The *real number system* refers to set $\mathbb{R}$ together with operations such as $-$, $+$, $\cdot$ and $/$. The *extended real number system* is derived from the real number system and is based on the *extended set of real numbers* $\bar{\mathbb{R}}$, which is defined as $\bar{\mathbb{R}} = \mathbb{R} \cup \{-\infty, \infty\}$ with $-\infty < x < \infty$ for all $x \in \mathbb{R}$. To allow using similar operations as those defined for $\mathbb{R}$, the rules of arithmetic for the operations $-$, $+$, $\cdot$ and / are extended as presented in [95].

A.2 Probability

σ-Algebras Consider Ω as the non-empty set of all possible outcomes of an experiment. The set Ω is often referred to as the *sample space* and the elements of Ω are called *samples* or *realisations*. A set of subsets of Ω that contains Ω and is closed under countable union and complement with respect to Ω is called a *σ-algebra*[2] on Ω. By the infinite form of De Morgan's law, a σ-algebra is also closed under countable intersection. The smallest possible σ-algebra on Ω is the set $\{\varnothing, \Omega\}$ and the largest possible is $\mathbf{2}^{\Omega}$. A σ-algebra on Ω is said to be *generated by* a set A of subsets of Ω if it is the intersection of all σ-algebras on Ω that contain A [24].

Events and Probability The elements of a σ-algebra $\mathcal{F}$ on sample space Ω are also referred to as *events* to which probability can be assigned according to some probability measure. A *probability measure* $\mathbb{P}$ is a complete function that assigns a real number in the interval [0, 1] to each event in $\mathcal{F}$ such that $\mathbb{P}(\Omega) = 1$ and

$$\mathbb{P}(\bigcup_{i=1}^{\infty} X_i) = \sum_{i=1}^{\infty} \mathbb{P}(X_i)$$

for pairwise disjoint events $X_1, X_2, \ldots \in \mathcal{F}$. The latter is known as the property of *countable additivity*.

If $\mathcal{F}$ is a σ-algebra on Ω and $\mathbb{P}$ is a probability measure on $\mathcal{F}$, then the triple $(\Omega, \mathcal{F}, \mathbb{P})$ is called a *probability space* and an event in $\mathcal{F}$ is also said to be *measurable*. For a measurable event X, $\mathbb{P}(X)$ is called the *probability* of X. If $\mathbb{P}(X) = 0$, then X is called a *null event*. In case $\mathbb{P}(X) = 1$, then X is said to be *almost sure*.

A useful result of the property of countable additivity is the following lemma, for which a proof can be found in many text books on probability theory.

Lemma A.1 *If X and Y are events in the σ-algebra $\mathcal{F}$ of a probability space $(\Omega, \mathcal{F}, \mathbb{P})$, then* $\mathbb{P}(X \cap Y) \geq \mathbb{P}(X) + \mathbb{P}(Y) - 1$.

In accordance with conventions used in probability theory, the probability on the countable intersection of events $X_1, X_2, \ldots \in \mathcal{F}$ is often denoted by $\mathbb{P}(X_1, X_2, \ldots)$.

Conditional Probability Let X, Y be events in the σ-algebra $\mathcal{F}$ of probability space $(\Omega, \mathcal{F}, \mathbb{P})$. Provided that $\mathbb{P}(Y) > 0$, the *conditional probability* on X given Y, denoted by $\mathbb{P}(X \mid Y)$, is defined as

$$\mathbb{P}(X \mid Y) = \frac{\mathbb{P}(X \cap Y)}{\mathbb{P}(Y)}$$

[2]Instead of σ-algebra, probability theory also uses the term *σ-field*.

which implies that $\mathbb{P}(X \cap Y) = \mathbb{P}(X \mid Y) \cdot \mathbb{P}(Y)$.

A.3 Random Variables

Random Variables Consider the probability space $(\Omega, \mathcal{F}, \mathbb{P})$ where $\Omega = \bar{\mathbb{R}}$ and $\mathcal{F}$ is the σ-algebra[3] generated by the set of intervals[4] $(a, b]$ with $a, b \in \bar{\mathbb{R}}$ and $a < b$. A (real-valued) *random variable* on $(\Omega, \mathcal{F}, \mathbb{P})$ is a complete function X from Ω to $\bar{\mathbb{R}}$ such that for any interval $(a, b]$ with $a, b \in \bar{\mathbb{R}}$ and $a < b$, the set $\{\omega \in \Omega \mid X(\omega) \in (a, b]\}$ is an event in $\mathcal{F}$. Hence, the probability $\mathbb{P}(\{\omega \in \Omega \mid X(\omega) \in (a, b]\})$, which is often conveniently abbreviated to $\mathbb{P}(X \in (a, b])$, exists.

The set of all random variables on $(\Omega, \mathcal{F}, \mathbb{P})$ is closed under the operations $-$, $+$ and $\cdot$ defined for $\bar{\mathbb{R}}$. Furthermore, if X and Y are random variables on $(\Omega, \mathcal{F}, \mathbb{P})$, then X/Y is a random variable on $(\Omega, \mathcal{F}, \mathbb{P})$ provided that $Y(\omega) \neq 0$ for all $\omega \in \Omega$ [95].

Associated with every random variable is a distribution and distribution function. The *distribution* of random variable X is the set of probabilities $\mathbb{P}(X \in (a, b])$ with $a, b \in \bar{\mathbb{R}}$ and $a < b$. The *distribution function* of random variable X is denoted by $\mathbf{F}_X : \bar{\mathbb{R}} \to [0, 1]$ and is defined as $\mathbf{F}_X(b) = \mathbb{P}(X \in (-\infty, b])$ for $b \in \bar{\mathbb{R}}$. A distribution of special importance is the normal distribution. A random variable X with parameters[5] $\mu \in \mathbb{R}$ and $\sigma^2 > 0$ has a *normal distribution*, denoted by $\mathrm{N}(\mu, \sigma^2)$, if

$$\mathbf{F}_X(\kappa) = \frac{1}{\sqrt{2\pi\sigma^2}} \int_{-\infty}^{\kappa} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx \quad \text{for } \kappa \in \bar{\mathbb{R}}$$

A normally distributed random variable with parameters $\mu = 0$ and $\sigma^2 = 1$ is said to have a *standard normal distribution* (denoted by $\mathrm{N}(0, 1)$). The distribution function $\mathbf{F}_X(\kappa)$ of a random variable with $N(0, 1)$ is commonly denoted by $\mathfrak{N}(\kappa)$.

Independence For any finite $n > 1$, the random variables $X_1, \ldots, X_n$ on probability space $(\Omega, \mathcal{F}, \mathbb{P})$ are called *independent* if

$$\mathbb{P}(\bigcap_{i=1}^{n} \{\omega \in \Omega \mid X_i(\omega) \in (a_i, b_i]\}) = \prod_{i=1}^{n} \mathbb{P}(X_i \in (a_i, b_i])$$

for all intervals $(a_i, b_i]$ with $a_i, b_i \in \bar{\mathbb{R}}$ and $a_i < b_i$. The random variables $X_1, X_2, \ldots$ are independent if any finite number of them is independent. Random variables that are independent and have the same distribution function are said to be independent and identically distributed, which is abbreviated to i.i.d.

Discrete Random Variables A random variable X on probability space $(\Omega, \mathcal{F}, \mathbb{P})$ is called *discrete*[6] if there is a countable subset A of $\bar{\mathbb{R}}$ for which $\mathbb{P}(X \in A) = 1$ [39]. Element a of A is called a *possible value* of X in case $\mathbb{P}(X \in \{a\}) > 0$. The probability

[3]This σ-algebra is also known as the *Borel* σ-algebra on $\bar{\mathbb{R}}$.

[4]Notice that intervals refer to subsets of $\bar{\mathbb{R}}$. For example, interval $(a, b]$ with $a, b \in \bar{\mathbb{R}}$ and $a < b$ refers to the set $\{x \in \bar{\mathbb{R}} \mid a < x \leq b\}$. In probability theory, such an interval is also called a *Borel* set.

[5]The parameters μ and σ^2 of a random variable X with $\mathrm{N}(\mu, \sigma^2)$ denote the expectation and variance of X respectively [95].

[6]Literature sometimes uses the term *simple* random variable instead of discrete random variable.

$\mathbb{P}(X \in \{a\})$ is also denoted by $\mathbb{P}(X = a)$. The distribution of X (over A) concerns the probabilities $\mathbb{P}(X = a)$ for all $a \in A$ [24].

Expectation and Variance The *expectation* or *expected value* of a discrete random variable X on probability space $(\Omega, \mathcal{F}, \mathbb{P})$, denoted by $\mathbb{E}[X]$, is the weighted sum

$$\mathbb{E}[X] = \sum_{a \in A} a \cdot \mathbb{P}(X = a)$$

For $1 \leq i < \infty$, the set of all discrete random variables X on $(\Omega, \mathcal{F}, \mathbb{P})$ for which $\mathbb{E}[|X|^i] < \infty$ is denoted by $\mathcal{L}^i$. Notice that if random variable $X \in \mathcal{L}^{i+1}$, then also $X \in \mathcal{L}^i$. In case $X \in \mathcal{L}^i$, the expected value of X^i is called the i^{th} *moment* of X. The first moment of X equals the expected value of X. The *variance* of a random variable $X \in \mathcal{L}^2$, denoted by var$[X]$, is defined as var$[X] = \mathbb{E}[(X - \mathbb{E}[X])^2] = \mathbb{E}[X^2] - \mathbb{E}[X]^2$. The result of $\sqrt{\text{var}[X]}$ is called the *standard deviation* of X and is denoted by std$[X]$.

Lemma A.2 *Let X be a discrete random variable in $\mathcal{L}^2$. In case var$[X] = 0$, then $\mathbb{P}(X = \mathbb{E}[X]) = 1$.*

Proof Assume that X is not equal to $\mathbb{E}[X]$ with probability 1. Then, there exists an $x \neq \mathbb{E}[X]$ for which $\mathbb{P}(X = x) > 0$. It has to be shown that var$[X] > 0$ in this case. Indeed,

$$\text{var}[X] = \mathbb{E}[(X - \mathbb{E}[X])^2] = \sum_{a \neq x} (a - \mathbb{E}[X])^2 \cdot \mathbb{P}(X = a) + (x - \mathbb{E}[X])^2 \cdot \mathbb{P}(X = x) > 0$$

since both $(x - \mathbb{E}[X])^2 > 0$ and $\mathbb{P}(X = x) > 0$. ■

Covariance The *covariance* between two random variables X and Y in $\mathcal{L}^2$, denoted by cov$[X, Y]$, is defined as cov$[X, Y] = \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])] = \mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y]$. Provided that var$[X] > 0$ and var$[Y] > 0$, the *correlation* between X and Y is defined as cov$[X, Y]/\sqrt{\text{var}[X]\text{var}[Y]}$. X and Y are said to be *uncorrelated* in case cov$[X, Y] = 0$. If X and Y are independent, then $\mathbb{E}[XY] = \mathbb{E}[X]\mathbb{E}[Y]$ and hence, they are also uncorrelated. The other way around is in general not true. However, if X and Y are *jointly normally distributed* (for a precise definition, see [132]) and cov$[X, Y] = 0$, then they are independent [105].

A.4 Stochastic Processes

Stochastic Processes A *discrete-time stochastic process* on some probability space $(\Omega, \mathcal{F}, \mathbb{P})$ is a sequence of random variables $\{X_i \mid i \in \mathcal{T}\}$, where all random variables X_i are defined on $(\Omega, \mathcal{F}, \mathbb{P})$ and $\mathcal{T} = \{1, 2, \ldots\}$. The set $\mathcal{T}$ is referred to as the *time domain* of the stochastic process and the elements of $\mathcal{T}$ are called *time-epochs*. Instead of writing $\{X_i \mid i \in \{1, 2, \ldots\}\}$, the notation $\{X_i \mid i \geq 1\}$ is often used.

For a discrete-time stochastic process $\{X_i \mid i \geq 1\}$ on $(\Omega, \mathcal{F}, \mathbb{P})$, where each random variable X_i is discrete and assumes values in a countable set A, the probabilities of all events in $\mathcal{F}$ are completely determined by the *n-dimensional joint probabilities*

$$\mathbb{P}(X_1 = a_1, \ldots, X_n = a_n) = \mathbb{P}(\bigcap_{i=1}^{n} \{\omega \in \Omega \mid X_i(\omega) = a_i\}) =$$
$$\mathbb{P}(X_1 = a_1) \cdot \prod_{i=1}^{n-1} \mathbb{P}(\{\omega \in \Omega \mid X_{i+1}(\omega) = a_{i+1}\} \mid \bigcap_{k=1}^{i} \{\omega \in \Omega \mid X_k(\omega) = a_k\})$$

for all $n \geq 1$ and $a_i \in A$ [39]. Often, this is conveniently rewritten to

$$\mathbb{P}(X_1 = a_1) \cdot \prod_{i=1}^{n-1} \mathbb{P}(X_{i+1} = a_{i+1} \mid X_k = a_k \text{ for all } 1 \leq k \leq i)$$

A discrete-time stochastic process $\{X_i \mid i \geq 1\}$ with $X_i \in \mathcal{L}^2$ for all $i \geq 1$ is said to be *covariance-stationary* (or *weakly stationary*) if both $\mathbb{E}[X_i] = \mathbb{E}[X_j]$ and $\text{var}[X_i] = \text{var}[X_j]$ for all $i, j \geq 1$ and $\text{cov}[X_i, X_{i+j}]$ is independent of i for all $j \geq 1$.

Long-run Behaviour Let X and $X_1, X_2, \ldots$ be random variables on some probability space $(\Omega, \mathcal{F}, \mathbb{P})$ that assume values in $\mathbb{R}$. The sequence $X_1, X_2, \ldots$ is said to *converge almost surely* to X, denoted by $X_n \xrightarrow{\text{a.s.}} X$, if

$$\mathbb{P}(\{\omega \in \Omega \mid \lim_{n\to\infty} X_n(\omega) = X(\omega)\}) = 1$$

The set of all random variables on $(\Omega, \mathcal{F}, \mathbb{P})$ that converge almost surely is closed under the negation, addition and multiplication operations defined on $\mathbb{R}$ [95]. The sequence $X_1, X_2, \ldots$ is said to *converge in distribution* to X, denoted by $X_n \xrightarrow{\text{d.}} X$, if

$$\lim_{n\to\infty} \mathbf{F}_{X_n}(b) = \mathbf{F}_X(b)$$

for all $b \in \mathbb{R}$ at which $\mathbf{F}_X$ is continuous. In [95], it is proven that $X_n \xrightarrow{\text{a.s.}} X$ implies $X_n \xrightarrow{\text{d.}} X$, whereas it is also shown that the other way around is not necessarily true.

The *long-run behaviour* of a discrete-time stochastic process $\{X_i \mid i \geq 1\}$, where each X_i assumes values in a countable subset of $\mathbb{R}$, is denoted by

$$\lim_{n\to\infty} X_n = X$$

This notation refers to almost sure convergence of the sequence $X_1, X_2, \ldots$ unless explicitly stated that another mode of convergence is meant.

Limit Theorems Several limit theorems exist for analysing the long-run behaviour of discrete-time stochastic processes. Two of them are of special importance. The *strong law of large numbers* states that if $X_1, X_2, \ldots$ are i.i.d. discrete random variables in $\mathcal{L}^1$, then

$$\frac{1}{n} \sum_{i=1}^{n} X_i \xrightarrow{\text{a.s.}} \mathbb{E}[X_1]$$

The *central limit theorem* [24] states that in case $X_1, X_2, \ldots$ are i.i.d. discrete random variables in $\mathcal{L}^2$ with expected value μ and variance $\sigma^2 > 0$, then

$$\sqrt{n} \cdot \frac{\frac{1}{n}\sum_{i=1}^{n} X_i - \mu}{\sigma} \xrightarrow{\text{d.}} \mathrm{N}(0,1)$$

A.5 Estimation of Expectation and Variance

For some $n \geq 1$, let $X_1, \ldots, X_n$ be i.i.d. random variables. *Estimation* concerns the assessment of properties of the random variables $X_1, \ldots, X_n$ such as expectation and variance by means of statistical analysis. Statistical analysis involves the evaluation of a suitable (non-constant) function of the random variables $X_1, \ldots, X_n$ for computing plausible values of the property of interest from one or more realisations. Such a function is called an *estimator*, while the evaluation result is called an *estimate*. Estimates obtained for a certain property depend on the used realisations. *Point estimation* intends to find the estimate that is as close as possible to the true value of the property, while *interval estimation* aims at finding an interval of plausible values.

Point Estimation To ensure finding a point estimate $\bar{\theta}$ close to the true value θ of the property that is being estimated, it is useful to require that the used point estimator converges almost surely to θ. A point estimator $\hat{\theta}$ that converges almost surely to θ is called *strongly consistent*[7]. Point estimator $\hat{\theta}$ is called *unbiased* in case its expected value $\mathbb{E}[\hat{\theta}]$ is equal to the true value θ of the property being estimated. If the point estimator $\hat{\theta}$ is not unbiased, the difference $\mathbb{E}[\hat{\theta}] - \theta$ is called the *bias* of $\hat{\theta}$.

For some $n \geq 1$, let $X_1, \ldots, X_n$ be i.i.d. discrete random variables in $\mathcal{L}^1$ with expected value μ and assume the objective is estimating μ. Then, the *sample average* $\hat{\mu}$ of the random variables $X_1, \ldots, X_n$, defined as

$$\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} X_i$$

is a strongly consistent point estimator for μ because $\hat{\mu} \xrightarrow{\text{a.s.}} \mu$ by the strong law of large numbers. Notice that sample average $\hat{\mu}$ is also unbiased since $\mathbb{E}[\hat{\mu}] = \mu$.

Now, for some $n \geq 1$, let $X_1, \ldots, X_n$ be i.i.d. discrete random variables in $\mathcal{L}^2$ with expected value μ and variance σ^2 and assume the objective is estimating σ^2. Then the *sample variance* $\hat{\sigma}^2$ of the random variables $X_1, \ldots, X_n$, defined as

$$\hat{\sigma}^2 = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \hat{\mu})^2$$

is a strongly consistent point estimator for σ^2 since $\hat{\sigma}^2 \xrightarrow{\text{a.s.}} \sigma^2$ by the strong law of large numbers. As shown in [46], sample variance $\hat{\sigma}^2$ is also unbiased.

[7]Notice that strong consistency is not a necessity for point estimators.

Interval Estimation To analyse how close a point estimate $\bar{\theta}$ is to the true value θ of the property that is being estimated, confidence intervals can be used. A *confidence interval* is an interval of plausible values for the property being estimated. The degree of plausibility is specified with the *confidence level*. Confidence intervals are obtained based on interval estimators for the property of interest. A γ *interval estimator* for θ is a stochastic interval $[\varphi_1, \varphi_2]$, for which the probability $\mathbb{P}(\theta \in [\varphi_1, \varphi_2])$ is greater than[8] or equal to confidence level $\gamma \in [0, 1]$. Intuitively, this means that when repeatedly performing estimations of θ, then in the long-run at least $\gamma \cdot 100\%$ of the obtained confidence intervals (realisations of $[\varphi_1, \varphi_2]$) wil indeed contain θ. In this thesis, a γ interval estimator $[\varphi_1, \varphi_2]$ for θ is also called a γ *Confidence Interval* for θ.

Definition A.1 (Confidence Interval) *Let $\gamma \in [0, 1]$ be a confidence level. A stochastic interval $[\varphi_1, \varphi_2]$, where the discrete random variables φ_1 and φ_2 assume values in $\bar{\mathbb{R}}$ such that $\varphi_1 \leq \varphi_2$, is called a γ Confidence Interval for $\mu \in \mathbb{R}$ if $\mathbb{P}(\mu \in [\varphi_1, \varphi_2]) \geq \gamma$. A realisation $[\bar{\varphi}_1, \bar{\varphi}_2]$ of the stochastic interval $[\varphi_1, \varphi_2]$ is called a γ confidence interval for μ.*

Confidence Intervals can sometimes be derived using the central limit theorem. For some $n \geq 1$, let $X_1, \dots, X_n$ be i.i.d. discrete random variables in $\mathcal{L}^2$ with expected value μ and variance $\sigma^2 > 0$. In case σ is known, then by the central limit theorem,

$$\sqrt{n} \cdot \frac{\hat{\mu} - \mu}{\sigma} \xrightarrow{\text{d.}} \mathrm{N}(0, 1)$$

which means that for every $\kappa \in \mathbb{R}$

$$\lim_{n\to\infty} \mathbb{P}\left(-\kappa \leq \sqrt(n) \cdot \frac{\hat{\mu} - \mu}{\sigma} \leq \kappa\right) = \lim_{n\to\infty} \mathbb{P}\left(\hat{\mu} - \frac{\kappa\sigma}{\sqrt{n}} \leq \mu \leq \hat{\mu} + \frac{\kappa\sigma}{\sqrt{n}}\right) = 2\mathfrak{R}(\kappa) - 1$$

Hence, for a concrete point estimate $\bar{\mu}$ obtained with the sample average of the discrete random variables $X_1, \dots, X_n$, the interval estimate $[\bar{\varphi}_1, \bar{\varphi}_2]$ with

$$\bar{\varphi}_1 = \bar{\mu} - \frac{\kappa\sigma}{\sqrt{n}} \quad \text{and} \quad \bar{\varphi}_2 = \bar{\mu} + \frac{\kappa\sigma}{\sqrt{n}}$$

is an (approximate) $2\mathfrak{R}(\kappa) - 1$ confidence interval for μ.

Notice that for $\sigma^2 = 0$, the central limit theorem cannot be applied. If $\sigma^2 = 0$, then

$$\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} X_i = \frac{1}{n}\sum_{i=1}^{n} \mu = \mu$$

by lemma A.2. Now, remark that $[\varphi_1, \varphi_2] = [\mu, \mu]$ is still a valid $2\mathfrak{R}(\kappa) - 1$ Confidence Interval for μ according to definition A.1. As a result, the condition $\sigma^2 > 0$ for obtaining valid confidence intervals can and will be discarded.

The collection of stochastic intervals satisfying the conditions in definition A.1 is also referred to as the *set of Confidence Intervals* in this thesis.

[8]The classical definition of Confidence Intervals uses $\mathbb{P}(\theta \in [\varphi_1, \varphi_2]) = \gamma$, see [46].

Appendix B

Parallel Object-Oriented Specification Language

To develop models of hardware/software systems for analysing their properties, the Parallel Object-Oriented Specification Language (POOSL) can be used. Originally introduced in [178, 179, 145], POOSL was extended in [55, 57] and [28] for expressing time and stochasticity respectively, next to several other new features. POOSL is an expressive modelling language with a small set of powerful primitives of which the semantics is defined with mathematical axioms and rules. The formal semantics of POOSL founds the application of model checking techniques for formal verification of correctness properties [57] and Markov-chain based performance analysis techniques [180]. It furthermore serves as a basis for unambiguous execution of models with two different tools: SHESim and Rotalumis. This appendix gives an introductory overview of POOSL and briefly discusses accompanying tool support.

B.1 Syntax

This section summarises the most relevant aspects of the syntax of POOSL. The complete syntax is defined in [28] using the Extended Backus-Naur Form (EBNF).

POOSL supports an object-oriented modelling approach and provides different object classes for modelling data, processes and architectural structure in three layers.

Data Layer The data layer provides the use of *data objects*, which are instances of *data classes*. Data objects are intended for modelling information that is generated, interpreted, changed and consumed by the components of a system. Information communicated between components is also modelled with data objects.

The definition of a data class involves a name, a single inheritance relation, instance variables and instance methods. Every data class defined in POOSL eventually inherits from data class `Object`, which has no superclass. The *instance variables* or global variables specify the attributes of a data object. Each data class inherits all instance variables of its superclass and may provide additional ones. The behaviour of data objects, which is purely sequential, is defined by the instance methods or *(data)*[1] *methods* of the corresponding data class. Data classes inherit all methods defined for their superclasses and may redefine methods or provide additional ones. Data objects may receive a message from data objects or from processes, which leads to the atomic execution of a method with the same name as the message. Upon completion of executing a method, the result of a calculation or the data object itself is returned. Hence, data objects are comparable to objects in traditional imperative object-oriented programming languages such as Smalltalk [62] and Java [64].

Data methods are defined according to one of the following alternative forms:

$$\begin{array}{ll} m(p_1 : D_{p_1}, \ldots, p_i : D_{p_i}) : D_r & \quad m(p_1 : D_{p_1}, \ldots, p_i : D_{p_i}) : D_r \\ \quad |l_1 : D_{l_1}, \ldots, l_k : D_{l_k}| & \quad\quad |l_1 : D_{l_1}, \ldots, l_k : D_{l_k}| \\ \quad E. & \quad\quad \texttt{\textbf{primitive}}. \end{array}$$

where m is the method name and D_r denotes the type[2] of the returned object. The lists $p_1 : D_{p_1}, \ldots, p_i : D_{p_i}$ and $l_1 : D_{l_1}, \ldots, l_k : D_{l_k}$ denote respectively the declaration of parameters and local variables for m. The body of a data method is either an expression E or is provided by axioms in the semantics of POOSL, which is denoted by the keyword **`primitive`**. Methods of the latter type are called *primitive* methods and define behaviour that cannot be captured in terms of POOSL expressions.

Next to data class `Object`, for which primitive methods like `=`, `!=`, `==`, `deepCopy` and `shallowCopy` are defined, the data classes `Boolean`, `Integer`, `Real`, `String` and `Nil` are predefined. Instances of data class `Boolean` evaluate to either **`true`** or **`false`**, while data classes `Integer` and `Real` allow instantiating data objects that represent integer and real numbers respectively. Various primitive methods are defined for these classes that reflect standard operations. Instances of data class `String` represent strings and the primitive methods of this class reflect standard operations on strings. Data class `Nil` has only a single instance; the elementary data object **`nil`**. Furthermore, data classes `Array`, `FileIn`, `FileOut` and `RandomGene-rator` are predefined. Data class `Array` allows instantiating arrays of data objects and includes primitive methods for modifying and retrieving the contents of arrays. Data classes `FileIn` and `FileOut` include primitive methods for accessing files to read or write data objects from/to files respectively. Instances of `RandomGenera-tor` denote a uniform distribution over a bounded interval. Primitive method `ran-dom` returns a `Real` according to a uniform distribution $(0, 1)$, while primitive method `randomInt`, which has `Integer` parameter p, returns an `Integer` according to a uniform distribution $[0, p)$. The primitive method `seed` initialises the initial value (seed) of the random number generator used for implementing `random` and `randomInt` (see section B.3). Primitive method `randomiseSeed` initialises the seed in an arbitrary way based on the real time at which `randomiseSeed` is evaluated.

The behaviour of non-primitive data methods is specified by *expressions*. Table B.1 summarises the syntax for the most relevant forms of expressions. Constant c refers to an instance of data class `Boolean`, `Integer`, `Real`, `String` or `Nil`. Examples

[1]Prefixing methods by the word data or process is omitted when it is clear from the context what kind of method is meant.

[2]If an object d is an instance of class D, then d is said to have type D [28].

E	$=$	c	constant
	$\mid$	x	variable
	$\mid$	**`self`**	reference self
	$\mid$	**`new`**(D)	data object creation
	$\mid$	**`currentTime`**	current model time
	$\mid$	$x := E$	assignment
	$\mid$	$E_1; E_2$	sequential composition
	$\mid$	$E\ \hat{}m(E_1, \ldots, E_i)$	(superclass) data method call
	$\mid$	**`if`** E_c **`then`** E_1 **`else`** E_2 **`fi`**	choice
	$\mid$	**`while`** E_c **`do`** E **`od`**	loop
	$\mid$	**`return`** E	return

Table B.1: Expressions.

are **`true`**, `3.8` and `"POOSL"`. The second expression evaluates to the object referred to by variable x. The keyword **`self`** refers to the data object that is evaluating expression E. Evaluation of **`new`**(D) results in the creation of a new instance of data class D[3]. Expression **`currentTime`** retrieves the current model time and can only be used in the context of processes [28]. Expression $x := E$ denotes the assignment of the value to which E evaluates to variable x. Sequential composition of expressions is denoted using a semicolon. Optional parenthesis enable to group expressions.

The next expression in table B.1 is the data method call. In case method m is defined for a data class, expression $E\ m(E_1, \ldots, E_i)$ denotes calling method m for the instance of that data class to which E evaluates. Before actually evaluating such a method call, expressions E and $E_1, \ldots, E_i$ are evaluated from left to right. If some data class D redefines or *overrides* a method m of its superclass by providing a method m with the same number of parameters, $E\ \hat{}m(E_1, \ldots, E_i)$ ensures explicitly calling the method m of the superclass for the instance of D to which E evaluates.

Expression **`if`** E_c **`then`** E_1 **`else`** E_2 **`fi`** evaluates to the result of expression E_1 in case the condition E_c evaluates to **`true`**. Alternatively, when expression E_c evaluates to **`false`**, the result is the result of expression E_2. For loop **`while`** E_c **`do`** E **`od`**, expression E is repeatedly evaluated as long as condition E_c evaluates to **`true`**. It finishes when E_c evaluates to **`false`**. On completion of a data method call, expression **`return`** E returns the data object resulting from expression E.

Process Layer Hardware/software systems are usually considered as being composed of concurrently operating components that communicate with each other. To model the basic components, POOSL provides *process objects* or *processes*, which are instances of *process classes*. Their behaviour is described in the process layer.

Defining a process class involves specifying a name, instantiation parameters and instance variables, a port interface and message interface, instance methods, an initial method call and an optional single inheritance relation. The *instantiation parameters* and *instance variables* are the attributes of a process object. The difference between instantiation parameters and instance variables is that instantiation parameters allow parameterising the behaviour of a process at instantiation. Processes may commu-

[3]Instances of `Boolean`, `Integer`, `Real` and `String` are created by denoting their constant value.

nicate (deepcopies of) their encapsulated data objects via *ports*. The *port interface* lists the names of the ports via which instances of a process class may communicate messages. The *message interface* lists the signatures of all possible messages and includes for each message the port name, the symbol ! or ? for message send and receive respectively, the message name and a (possibly empty) list of types of the message parameters. The behaviour of processes is defined by the instance methods or *(process) methods* of the corresponding process class. Methods may be called with input parameters and may return results through output parameters. The starting behaviour of a process is defined by the *initial method call*. In case an inheritance relation is defined for a process class, it inherits the instantiation parameters and instance variables, port interface and message interface as well as all the methods of its superclass.

The behaviour defined by a process method is specified using the following syntax:

$$\begin{array}{l} m(p_1 : D_{p_1}, \ldots, p_i : D_{p_i})(r_1 : D_{r_1}, \ldots, r_j : D_{r_j}) \\ \quad |l_1 : D_{l_1}, \ldots l_k : D_{l_k}| \\ \quad S. \end{array}$$

where m is the method name and declarations $p_1 : D_{p_1}, \ldots, p_i : D_{p_i}$, $r_1 : D_{r_1}, \ldots, r_j : D_{r_j}$ and $l_1 : D_{l_1}, \ldots, l_k : D_{l_k}$ denote respectively the input parameters, output parameters and local variables for method m. The body of m is defined using the (orthogonal) *statements* in table B.2.

S	$=$	E	expression
	$\mid$	$m(E_1, \ldots, E_i)(v_1, \ldots, v_j)$	process method call
	$\mid$	**par** S_1 **and** S_2 **and** ... **and** S_n **rap**	parallel composition
	$\mid$	$S_1; S_2$	sequential composition
	$\mid$	$E_p!m(E_1, \ldots, E_i)\{E\}$	message send
	$\mid$	$E_p?m(v_1, \ldots, v_i \mid E_c)\{E\}$	(conditional) message receive
	$\mid$	**sel** S_1 **or** S_2 **or** ... S_n **les**	non-deterministic selection
	$\mid$	$[E_c]S$	guarded execution
	$\mid$	**interrupt** S_1 **with** S_2	interrupt
	$\mid$	**abort** S_1 **with** S_2	abort
	$\mid$	**delay** E	time synchronisation
	$\mid$	**if** E_c **then** S_1 **else** S_2 **fi**	choice
	$\mid$	**while** E_c **do** S **od**	loop
	$\mid$	**skip**	empty behaviour

Table B.2: Statements.

Most expressions are also valid statements [28] and using optional curly braces enforces atomic (indivisible) execution of E. Statement $m(E_1, \ldots, E_i)(v_1, \ldots, v_j)$ specifies that method m is called after expressions $E_1, \ldots, E_i$ are evaluated from left to right and bound to the input parameters of m. When completing execution of m, results are returned by binding the output parameters to variables $v_1, \ldots, v_j$. Methods without output parameters may be called *tail-recursively*. The statement for parallel composition denotes the purely interleaved (non-deterministic) execution of statements $S_1, \ldots, S_n$. With this statement, POOSL offers asynchronous concurrency *within* processes next to the usual concurrency among processes. The behaviours specified with $S_1, \ldots, S_n$ are called *concurrent activities* and share the data objects encapsulated by the owing process. Since operations on data objects are atomic, potential mutual exclusion problems are solved in a natural way. Sequential composition of statements is denoted using a semicolon. Similar as for expressions, sequentially composed statements can be grouped using the optional parenthesis.

The message send statement $E_p!m(E_1, \ldots, E_i)\{E\}$ denotes the *synchronous* sending of message m to the port to which expression E_p evaluates. When a matching message receive statement is available, parameters $E_1, \ldots, E_i$ are evaluated from left to right and deepcopies of their results are bound to the parameters of the corresponding receive statement. The optional expression E between braces is evaluated atomically after the message is sent. The complementary message receive statement $E_p?m(v_1, \ldots, v_i \mid E_c)\{E\}$ describes reception of message m on the port to which expression E_p evaluates. When a matching message send statement is available, the parameters of the message are bound to variables $v_1, \ldots, v_i$. However, such synchronisation between a send statement and a receive statement can only occur if the message names match, the number of parameters is equal and the optional *reception condition* E_c (possibly depending on the received data objects) evaluates to **`true`**. The receive statement may also be followed by an atomically evaluated expression.

Use of the statement **`sel`** S_1 **`or`** S_2 **`or`** $\ldots$ S_n **`les`** for non-deterministic selection results in executing one of its non-blocking constituent statements $S_1, \ldots, S_n$. An example of a statement that may block is the statement for guarded execution $[E_c]S$. It blocks execution of S as long as the condition E_c evaluates to **`false`**.

The interrupt statement specifies that execution of S_1 is suspended when statement S_2 executes. When S_2 terminates, execution of S_1 is resumed (and can be interrupted by S_2 again). The abort statement behaves similarly, but now S_1 is terminated when S_2 executes. Statement **`delay`** E models postponing the execution of statements for E units of time. It offers the only way to express quantitative timing behaviour. This is sufficient since **`delay`** E can be combined with the interrupt and abort statements to specify more intricate timing behaviour like time-outs or watchdogs [55]. It is remarked that the time domain of a model can be discrete (then E may evaluate to instances of `Integer`) or dense (then E may evaluate to instances of both `Integer` and `Real`).

Statement **`if`** E_c **`then`** S_1 **`else`** S_2 **`fi`** results in executing statement S_1 in case condition E_c evaluates to **`true`**. If expression E_c evaluates to **`false`**, S_2 is executed instead. The loop **`while`** E_c **`do`** S **`od`** specifies repeated execution of statement S as long as condition E_c evaluates to **`true`**. The **`skip`** statement provides a means for specifying an (internal) action without an effect on the variables of a model.

Architecture Layer The architecture layer allows specification of the (hierarchical) structure of components by providing the use of processes, clusters and channels. *Clusters* are instances of *cluster classes* and group a set of processes and clusters (of other cluster classes). The behaviour of clusters is defined by the parallel composition of the included processes and clusters and therefore does not extent their behaviour. Processes and clusters are statically interconnected via their ports by *channels*, over which messages can be communicated.

The definition of a cluster class involves a name, instantiation parameters, a port interface, a message interface and a behaviour specification. The *instantiation parameters* are the attributes of a cluster. The *port interface* and *message interface* are defined in a similar way as for process classes. A *behaviour specification* defines how clusters and processes are connected by channels using similar hiding and relabelling operators as in the process algebra CCS [122]. The behaviour specification also initialises the instantiation parameters of processes and clusters with expressions [28].

Next to defining cluster classes, the architecture layer defines the *system specification* of a model. The system specification includes a behaviour specification and a list of all classes defined for a model. This behaviour specification defines how processes and clusters are interconnected at the highest hierarchical level.

Although [145, 28] define a textual syntax for denoting behaviour specifications, an intuitive graphical form of the syntax is defined as well in [145]. This thesis merely uses the graphical form, which is exemplified in figure B.1. It shows the behaviour specification of a cluster class C with parameters $p_1, \ldots, p_n$, which encapsulates instances A and B of process classes P_1 and P_2. Their parameters are initialised by expressions $PE_1, \ldots, PE_i$ and $PE_1, \ldots, PE_j$ respectively, which may depend on $p_1, \ldots, p_n$. Processes A and B may communicate via ports a and b respectively over channel ch, which is also connected to port c of instances of C. The textual representation of the behaviour specification of cluster class C is given by $(A : P1(PE_1, \ldots, PE_i)[ch/a] \parallel B : P2(PE_1, \ldots, PE_j)[ch/b])[c/ch]$. Diagrams like the one in figure B.1 are also called *instance structure diagrams* [145].

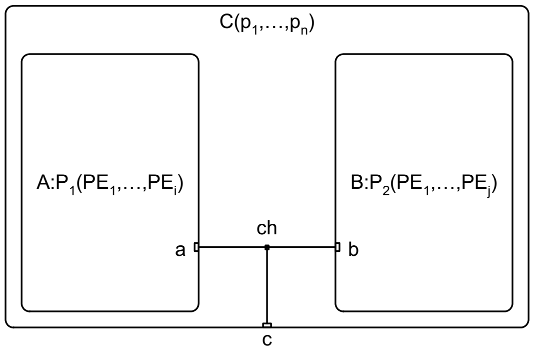


Figure B.1: Example of graphically denoting a behaviour specification.

B.2 Semantics

This section highlights some aspects of the semantics of POOSL defined in [28]. The denotational semantics of the data layer is based upon a probabilistic extension of traditional imperative object-oriented programming languages and supports the object-oriented principles of encapsulation, single inheritance (including method overriding) and polymorphism. The structural operational semantics of the process and architecture layers is based on a probabilistic real-time extension [55, 57, 28] of the process algebra CCS [122]. The probabilistic interpretation of statements specifying the behaviour of processes originates from data objects that are derived from instances of the data class `RandomGenerator` [28]. For the process layer, the semantics supports the object-oriented principles of encapsulation (of data objects) and single inheritance (including method overriding), while for the architecture layer, encapsulation (of processes and clusters) is supported.

Semantical Framework The semantics of POOSL is based on the two-phase execution model explained in [127]. The configuration (state) of a model can either change by *asynchronously* performing actions or by *synchronous* consumption of time. Time can only advance if there are no actions ready to be performed (action urgency).

The semantics of a POOSL model is given by a Plotkin-style [139] structural operational semantics and defines a *timed probabilistic labelled transition system* of the form

$$(\mathcal{C}, C_s, \mathcal{A}, \{\xrightarrow{a} \subseteq \mathcal{C} \times \mathcal{D}(\mathcal{C}) \mid a \in \mathcal{A}\}, \mathcal{T}, \{\xrightarrow{t} \subseteq \mathcal{C} \times \mathcal{C} \mid t \in \mathcal{T}^+\}) \tag{B.1}$$

It includes a countable set $\mathcal{C}$ of *configurations*, each reflecting a possible state of the model during its execution. The *initial configuration* $C_s \in \mathcal{C}$ represents the state of the model at the start of its execution. A configuration $(\mathcal{B},\mathcal{I}) \in \mathcal{C}$ includes[4] the specification of *behaviour* $\mathcal{B}$ that is to be executed in the context of *information* $\mathcal{I}$. Behaviour $\mathcal{B}$ indicates the statements describing the (future) behaviour of an executing model, while information $\mathcal{I}$ captures the data objects that are assigned to global and local variables. The timed probabilistic labelled transition system (B.1) furthermore includes a set $\mathcal{A}$ of actions, a time domain $\mathcal{T}$ and two sets of labelled transition relations. For $\mathcal{A}$, three kinds of elements are distinguished:

- the *internal action* τ and *fix action* f, representing computations that are unobservable by the environment of an executing model[5];
- *communication actions* of the following two forms. A *send action* $p!m(v_1, \ldots, v_n)$ denotes sending message m with parameters $v_1, \ldots, v_n$ to port p, whereas the *receive action* $p?m(v_1, \ldots, v_n)$ indicates receiving message m with parameters $v_1, \ldots, v_n$ from port p.

The elements of the two sets of labelled transition relations are defined based on the semantical rules and axioms for the statements in table B.2. The set $\{\xrightarrow{a} \subseteq \mathcal{C} \times \mathcal{D}(\mathcal{C}) \mid a \in \mathcal{A}\}$ denotes the *action transitions* for a model, where $\mathcal{D}(\mathcal{C})$ is the set of distribution functions[6] over $\mathcal{C}$;

$$\mathcal{D}(\mathcal{C}) = \{\boldsymbol{\pi} : \mathcal{C} \to [0,1] \mid \sum_{C \in \mathcal{C}} \boldsymbol{\pi}(C) = 1\} \tag{B.2}$$

When residing in configuration $(\mathcal{B},\mathcal{I})$, relation $(\mathcal{B},\mathcal{I}) \xrightarrow{a} \boldsymbol{\pi}$ holds if action $a \in A$ can be performed, after which the model transits to configuration $(\mathcal{B}',\mathcal{I}')$ with probability $\boldsymbol{\pi}(\mathcal{B}',\mathcal{I}')$, for each $(\mathcal{B}',\mathcal{I}') \in \mathcal{C}$. In case several (different) actions can be performed, the choice of which action is actually executed is made non-deterministically. Then, a probabilistic choice determines the resulting configuration.

[4]Here, a simplification of the actual elements in a configuration is used. See [28] for a precise definition.
[5]For a discussion on the precise difference between internal action τ and fix action f, see [28].
[6]According to [28], the sum of probabilities $\boldsymbol{\pi}(C)$ for all $C \in \mathcal{C}$ can be less then 1 for some distribution function $\boldsymbol{\pi} \in \mathcal{D}(\mathcal{C})$ due to non-terminating loops. In this thesis, such loops are not considered.

The set $\{\xrightarrow{t} \subseteq \mathcal{C} \times \mathcal{C} \mid t \in \mathcal{T}^+\}$ denotes the *time transitions* for a model. When residing in configuration $(\mathcal{B}, \mathcal{I})$, relation $(\mathcal{B}, \mathcal{I}) \xrightarrow{t} (\mathcal{B}', \mathcal{I}')$ holds if the time can pass for $t \in \mathcal{T}^+$ units of time, where $\mathcal{T}^+ = \{t \in \mathcal{T} \mid t > 0\}$. Time transitions follow the concepts presented in [181, 180]. A model only has time transitions for the *maximal* amount of time that it is willing to wait before continuing with performing action transitions, where it is implicitly understood that it is also willing to wait for a shorter time. As a result, there is at most one t and $(\mathcal{B}', \mathcal{I}')$ for each $(\mathcal{B}, \mathcal{I})$ such that $(\mathcal{B}, \mathcal{I}) \xrightarrow{t} (\mathcal{B}', \mathcal{I}')$ holds. Hence, time transitions are deterministic (taken with probability 1).

Execution Framework Execution of a POOSL model is based on the execution framework depicted in figure B.2. It includes a scheduler, a representation of all the processes in a model and a representation of all the channels that interconnect them (possibly through cluster boundaries). For each process, a virtual machine (VM) is provided to execute the operations on data objects as specified with expressions. A garbage collector (GC) automatically deletes any data object encapsulated by a process, which has become unreachable or obsolete during execution.

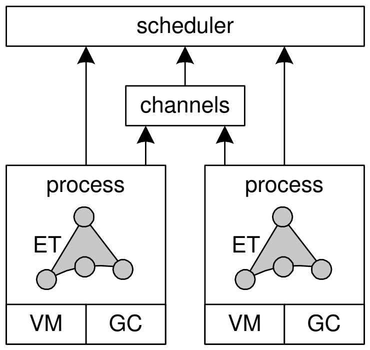


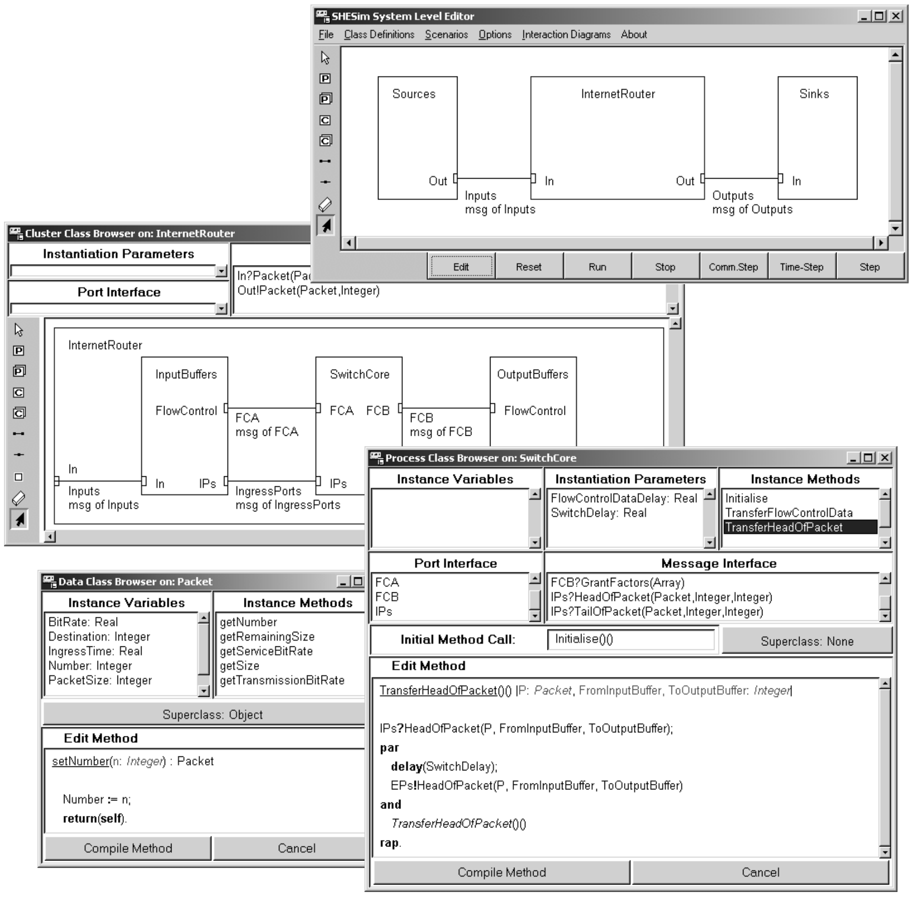
Figure B.2: Framework for executing POOSL models (from [28]).

The semantics of the process and architecture layers guides the execution of a POOSL model based on *execution trees* (ET) [57, 28]. Execution trees are data structures representing the configuration of processes and are derived from the model in accordance with the semantical axioms and rules of the statements in table B.2. Depending on its current configuration, a process issues requests for performing action and time transitions. Requests for send and receive actions are submitted to the corresponding channel. Channels combine complementary requests to single communication requests, which are submitted to the scheduler. Requests for internal actions, fix actions and time actions go directly to the scheduler. In case requests for several action transitions can be granted, the scheduler resolves the non-determinism in a probabilistic way using a uniform distribution over the set of possible action transitions.

B.3 Tool Support

Currently, two different tools are available for executing POOSL models. SHESim [58] is a graphical tool for incremental specification and modification of POOSL models that can be validated by interactive simulation, whereas the textual tool Rotalumis [28] enables high-speed simulation after completing validating a POOSL model.

SHESim SHESim allows defining data and process classes with an intuitive graphical user interface. The behaviour specification of the system specification and cluster classes are defined by drawing instance structure diagrams. Figure B.3 shows a snapshot that captures the editing of data, process and cluster classes with SHESim.



In?Packet(Pac
Out!Packet(Packet,Integer)

BitRate: Real
Destination: Integer
IngressTime: Real
Number: Integer
PacketSize: Integer

getNumber
getRemainingSize
getServiceBitRate
getSize
getTransmissionBitRate

```
setNumber(n: Integer) : Packet

  Number := n;
  return(self).
```

FlowControlDataDelay: Real
SwitchDelay: Real

Initialise
TransferFlowControlData
TransferHeadOfPacket

FCA
FCB
IPs

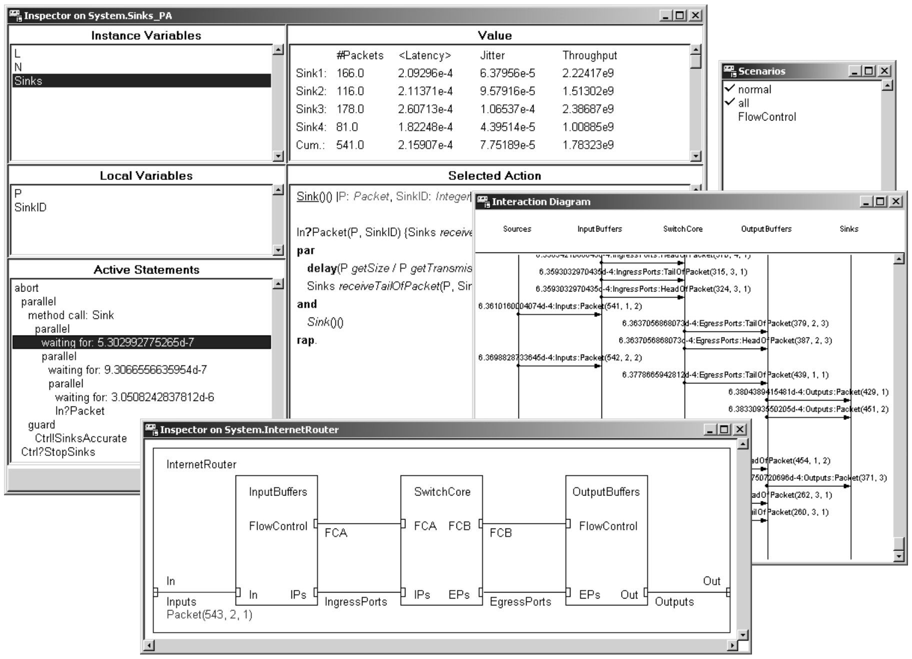
FCB?GrantFactors(Array)
IPs?HeadOfPacket(Packet,Integer,Integer)
IPs?TailOfPacket(Packet,Integer,Integer)

Initial Method Call: Initialise()()

```
TransferHeadOfPacket()() |P: Packet, FromInputBuffer, ToOutputBuffer: Integer|

IPs?HeadOfPacket(P, FromInputBuffer, ToOutputBuffer);
par
  delay(SwitchDelay);
  EPs!HeadOfPacket(P, FromInputBuffer, ToOutputBuffer)
and
  TransferHeadOfPacket()()
rap.
```

Figure B.3: Editing data, process and cluster classes with SHESim.

Next to the construction of POOSL models, SHESim supports their execution. With the buttons at the bottom of the system-level editor window in figure B.3, a model can be executed in different modes [58]. Several aspects of an executing model can be visualised. Examples are scenario-based visualisation of the messages communicated over channels in both instance structure diagrams and *interaction diagrams* [145], and the possibility to open inspectors on any data object, process, cluster or channel during executing a model. Such inspectors show, for example, the data objects assigned to variables or the current configuration of a process. As illustrated in figure B.4, the configuration of a process is reflected by a readable representation of the involved execution tree, while statements that are to be executed are highlighted.



```
L
N
Sinks
```

```
         #Packets  <Latency>    Jitter        Throughput
Sink1:   166.0     2.09296e-4   6.37956e-5    2.22417e9
Sink2:   116.0     2.11371e-4   9.57916e-5    1.51302e9
Sink3:   178.0     2.60713e-4   1.06537e-4    2.38687e9
Sink4:   81.0      1.82248e-4   4.39514e-5    1.00885e9
Cum.:    541.0     2.15907e-4   7.75189e-5    1.78323e9
```

```
P
SinkID
```

```
Sink()() |P: Packet, SinkID: Integer|

In?Packet(P, SinkID) {Sinks receive
par
  delay(P getSize / P getTransmis
  Sinks receiveTailOfPacket(P, Sir
and
  Sink()()
rap.
```

```
abort
  parallel
    method call: Sink
      parallel
        waiting for: 5.302992775265d-7
        parallel
          waiting for: 9.3066556635954d-7
          parallel
            waiting for: 3.0508242837812d-6
            In?Packet
    guard
      Ctrl!SinksAccurate
  Ctrl?StopSinks
```

```
✓ normal
✓ all
  FlowControl
```

```
6.3593032970435d-4:IngressPorts:TailOfPacket(315, 3, 1)
6.3593032970435d-4:IngressPorts:HeadOfPacket(324, 3, 1)
6.3610160004074d-4:Inputs:Packet(541, 1, 2)
6.3637056868073d-4:EgressPorts:TailOfPacket(379, 2, 3)
6.3637056868073d-4:EgressPorts:HeadOfPacket(387, 2, 3)
6.3698828733645d-4:Inputs:Packet(542, 2, 2)
6.3778665942812d-4:EgressPorts:TailOfPacket(439, 1, 1)
6.3804389415481d-4:Outputs:Packet(429, 1)
6.3833093550205d-4:Outputs:Packet(451, 2)
adOfPacket(454, 1, 2)
750720696d-4:Outputs:Packet(371, 3)
adOfPacket(262, 3, 1)
ilOfPacket(260, 3, 1)
```

Figure B.4: Simulation with SHESim.

SHESim is developed using the Smalltalk environment provided by VisualWorks [58]. For automatic collection of obsolete data objects, SHESim relies on the garbage collector incorporated in this environment. To enable a user-definable form of visualising data objects and to provide additional debugging facilities, SHESim offers two additional primitive methods for data class `Object`. Method `printString` has a `String` parameter, of which the value will be displayed when inspecting the data object. Method `error` also has a `String` parameter and if `error` is called, execution of the model is terminated and the parameter value is displayed.

Rotalumis Rotalumis is especially developed for high-speed execution of large POOSL models. Whereas SHESim executes a POOSL model in an interpretive way, Rotalumis compiles it into an intermediate byte code, which is executed on a virtual machine implemented in C++. Compared to SHESim, this improves the execution speed by a factor of about [7] 100. Obsolete data objects are deleted by a hybrid garbage collector [28] based on reference counting and Baker's Treadmill.

Figure B.5 depicts a snapshot of the screen output that Rotalumis produces during execution of a model. Rotalumis merely displays information about the progress of a simulation. For example, it indicates the running (or simulat*ion*) time (`RT`) of the execution, the model (or simulat*ed*) time (`ST`) and the number of performed execution steps (`Steps`) (action and time transitions in the timed probabilistic labelled transition system defined by a model, see also section B.2). Any simulation results that are of interest can be logged to files by using the `FileOut` data class.

[7]The actual speedup factor depends on the ratio between the number of operations on data objects (expressions) and the number of statements that have to be executed for a POOSL model.

```
*****
****  Rotalumis high-speed execution engine.
***   Programmed by L.J. van Bokhoven. (c) 2001
**    For more information visit: http://www.ics.ele.tue.nl/~lvbokhov.
*

Garbage collector: incremental Baker's treadmill.
Simulation time representation: 64-bit floating point.

Loading & compiling POOSL specification...

Running simulation...

RT: 0.22 Steps: 32769  ST: 0.0011098  GCCC: 0 VMC: 1.12746e+006
RT: 0.45 Steps: 65538  ST: 0.00194425 GCCC: 0 VMC: 2.21533e+006
RT: 0.66 Steps: 98307  ST: 0.00283991 GCCC: 0 VMC: 3.33673e+006
RT: 0.98 Steps: 131076 ST: 0.00377648 GCCC: 0 VMC: 4.44280e+006
RT: 1.24 Steps: 156434 ST: 0.00433246 GCCC: 0 VMC: 5.43233e+006
```

Figure B.5: High-speed execution with Rotalumis.

Random Number Generation To obtain random numbers for simulating probabilistic behaviour, tools like SHESim and Rotalumis use random number generators. A *random number generator* emulates drawing samples from the sample space of a random variable in accordance with its distribution. This is accomplished by producing a repetitive sequence of (pseudo) random numbers as described by a suitable algorithm. An essential issue for obtaining credible performance results is the actual randomness of the generated random numbers [136]. This randomness depends, amongst others, on the number of random numbers generated before reproducing the sequence and the statistical independency of the generated random numbers [133]. Two random number generators with good properties are the Minimal Standard [133], which was originally proposed in [109], and the recently developed Mersenne Twister [117, 47]. The Minimal Standard repeatedly generates $2^{31} - 1$ different random number, while the Mersenne Twister has a periodicity of $2^{19937} - 1$.

POOSL allows the use of multiple instances of data class `RandomGenerator` to model different distributions. To ensure the generation of nearly independent parallel sequences of random numbers for the different instances of `RandomGenerator`, two approaches can be distinguished:

- For each instance of `RandomGenerator`, a separate random number generator is used. Ensuring near independence of the parallel sequences with this approach requires to initialise the random number generators with different initial values (seeds) such that the parallel sequences of random numbers are non-overlapping[8]. Choosing these seeds should not be done without properly considering whether the parallel sequences are indeed non-overlapping [47]. Simply using another random number generator for producing the seeds is generally insufficient for obtaining credible performance results [47].

- A single (master) random number generator produces the random numbers for all instances of `RandomGenerator`. With this approach, each instance is represented as a virtual random number generator that is nearly independent from any other instance of `RandomGenerator`, thereby rendering the need for

[8]Notice that each random number generator uses the same algorithm for generating random numbers.

initialising different seeds superfluous. The methods `seed` and `randomise-Seed` may now initialise the seed of the master random number generator.

Notice that the second approach is preferred because it does not involve the difficult task of proving that parallel sequences of random number are non-overlapping. Currently, both SHESim and Rotalumis base random number generation on the first approach using a separate Mersenne Twister for each instance of `RandomGenerator`. The seeds are initialised automatically by another random number generator. Future work includes implementing the second approach with the Mersenne Twister.

Next to generating random numbers for instances of `RandomGenerator`, SHESim and Rotalumis require the use of a random number generator for probabilistically resolving non-determinism in POOSL models. Both SHESim and Rotalumis use an additional Mersenne Twister for this purpose. Its seed is fixed, which allows exact reproduction of simulations[9]. For any simulation, it is assumed that the number of random numbers needed for resolving non-determinism is much larger than the number of random numbers required for instances of `RandomGenerator`. Therefore, the random numbers generated for the different purposes are considered to be nearly independent, which enables obtaining credible performance results.

[9]Exact reproduction of simulations requires that method `randomiseSeed` is not used in a model.

Bibliography

[1] M. Abramowitz and I.A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs and Mathematical Tables*. Dover Publications, New York (U.S.A.), 1965.

[2] Accellera. Property Specification Language Reference Manual. Available from: http://www.accellera.org, April 2003. Version 1.01.

[3] C. Alexopoulos and A.F. Seila. Output Data Analysis for Simulations. In: B.A. Peters, J.S. Smith, D.J. Medeiros, and M.W. Rohrer (Eds.), *Proceedings of the Winter Simulation Conference (WSC'01)*, pp. 115–122. IEEE, Piscataway (U.S.A.), 2001.

[4] L. de Alfaro. How to Specify and Verify the Long-Run Average Behaviour of Probabilistic Systems. In: *Proceedings of the 13th IEEE Symposium on Logic in Computer Science (Indianapolis, U.S.A., June 21–24)*, pp. 454–465, 1998.

[5] A.O. Allen. *Probability, Statistics, and Queueing Theory; with Computer Science Applications*. Academic Press, San Diego (U.S.A.), 2nd edition, 1990.

[6] S. Andradóttir and N.T. Argon. Variance Estimation Using Replicated Batch Means. In: B.A. Peters, J.S. Smith, D.J. Medeiros, and M.W. Rohrer (Eds.), *Proceedings of the Winter Simulation Conference (WSC'01)*, pp. 338–343. IEEE, Piscataway (U.S.A.), 2001.

[7] P.R. d' Argenio, H. Hermanns, J.-P. Katoen, and R. Klaren. MoDeST – A Modelling and Description Language for Stochastic Timed Systems. In: L. de Alfaro and S. Gilmore (Eds.), *Proceedings of the International Joint Workshop on Process Algebra and Performance Modelling, Probabilistic Methods in Verification (PAPM-PROBMIV'01) (Aachen, Germany, September 12-14)*, pp. 87–104. Springer-Verlag, Berlin (Germany), 2001. LNCS 2165.

[8] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T Kanza, A. Landver, S. Mador-Haim, E.Singerman, A. Tiemeyer, M.Y. Vardi, and Y. Zbar. The ForSpec Temporal Logic: A New Temporal Property-Specification Language. In: J.P. Katoen and P. Stevens (Eds.), *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, pp. 296–311. Springer-Verlag, Berlin (Germany), 2002. LNCS 2280.

[9] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Model-Checking Continuous-Time Markov Chains. *ACM Transactions on Computational Logic*, 1 (1): pp. 162–170, 2000.

[10] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge University Press, 1990.

[11] C. Baier, B. Haverkort, H. Hermanns, and J.P. Katoen. Automated Performance and Dependability Evaluation Using Model Checking. In: M.C. Calzarossa and S. Tucci (Eds.), *Performance Evaluation of Complex Systems: Techniques and Tools (Performance'02)*, pp. 261–289. Springer-Verlag, Berlin (Germany), 2002. LNCS 2459.

[12] F. Balarin, E. Stentovich, M. Chiodo, H. Hsieh, B. Tabbara, A. Jurecska, L. Lavagno, C. Passerone, K. Suzuki, and A. Sangiovanni-Vincentelli. *Hardware-Software Co-Design of Embedded Systems - The POLIS Approach*. Kluwer Academic Publishers, Dordrecht (The Netherlands), 1997.

[13] F. Balarin, Y. Wantanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An Integrated Electronic System Design Environment. *IEEE Computer*, 36 (4): pp. 45–52, 2003.

[14] N. Bauer. Deployment of SDL Systems Using UML. In: *Proceedings of the 10th International SDL Forum*, pp. 107–122. Springer-Verlag, Berlin (Germany), 2001. LNCS 2078.

[15] K. Beck, J.O. Coplien, R. Crocker, L. Dominick, G. Meszaros, F. Paulisch, and J.M. Vlissides. Industrial Experience with Design Patterns. In: *Proceedings of the 18th International Conference on Software Engineering*, pp. 103–114. IEEE Computer Society, Los Alamitos (U.S.A.), 1996.

[16] P.D. Bergstrom, M.A. Ingram, A.J. Vernon, J.L.A. Hughes, and P. Tetali. A Markov Chain Model for an Optical Shared-Memory Packet Switch. *IEEE Transactions on Communications*, 47 (10): pp. 1593–1603, 1999.

[17] M. Bernardo. An Algebra-Based Method to Associate Rewards with EMPA Terms. In: *Proceedings of the 24th International Colloquium on Automata, Languages and Programming (ICALP'97) (Bologna, Italy, July 7–11)*, pp. 358–368. Springer-Verlag, Berlin (Germany), 1997. LNCS 1256.

[18] M. Bernardo and M. Bravetti. Reward Based Congruences: Can We Aggregate More? In: L. de Alfaro and S. Gilmore (Eds.), *Proceedings of the International Joint Workshop on Process Algebra and Performance Modelling, Probabilistic Methods in Verification (PAPM-PROBMIV'01) (Aachen, Germany, September 12-14)*, pp. 136–151. Springer-Verlag, Berlin (Germany), 2001. LNCS 2165.

[19] M. Bernardo, M. Bravetti, and R. Gorrieri. Towards Performance Evaluation with General Distributions in Process Algebra. In: D. Sangiorgi and R. de Simone (Eds.), *Proceedings of the 9th International Conference on Concurrency Theory (CONCUR'98) (Nice, France, September)*, pp. 405–422. Springer-Verlag, Berlin (Germany), 1998. LNCS 1466.

[20] M. Bernardo, N. Busi, and R. Gorrieri. A Distributed Semantics for EMPA Based on Stochastic Contextual Nets. *Computer Journal*, 38 (7): pp. 492–509, 1995.

[21] M. Bernardo and R. Gorrieri. A Tutorial on EMPA: A Theory of Concurrent Processes with Nondeterminism, Priorities, Probabilities and Time. *Theoretical Computer Science*, 202 (1–2): pp. 1–54, 1998.

[22] G. Berry. *The Esterel Language Primer (Version 5.20, Release 2.0)*. Ecole des Mines and INRIA, Sophia-Antipolis (France), 1999.

[23] A.K. Bhattacharjee, S.D. Dhodapkar, and R.K. Shyamasundar. PERTS: An Environment for Specification and Verification of Reactive Systems. *Reliability Engineering & System Safety*, 71 (3): pp. 299–310, 2001.

[24] P. Billingsley. *Probability and Measure*. Wiley, New York (U.S.A.), 2nd edition, 1995.

[25] D.P. Bischak, W.D. Kelton, and S.M. Pollock. Weighted Batch Means for Confidence Intervals in Steady-State Simulations. *Management Science*, 39 (8): pp. 1002–1019, 1993.

[26] M. Björkander and C. Kobryn. Architecting Systems with UML 2.0. *IEEE Software*, 20 (4): pp. 57–61, 2003.

[27] A. Bohdanowicz and J.H. Weber. Simulations of Communication Systems via Integrated Variance Reduction Techniques. In: *Proceedings of the IEEE Semiannual Vehicular Technology Conference (VTC'03)*, pp. 22–25. IEEE, 2003.

[28] L.J. van Bokhoven. *Constructive Tool Design for Formal Languages: From Semantics to Executing Models*. PhD thesis, Eindhoven University of Technology, Eindhoven (The Netherlands), 2002.

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

[30] F. Boussinot and R. de Simone. The Esterel Language. *Proceedings of the IEEE*, 79 (9): pp. 1293–1304, September 1991.

[31] O.J. Boxma. *Stochastic Performance Modelling*. Eindhoven University of Technology, Eindhoven (The Netherlands), 2002. Lecture Notes.

[32] E. Brinksma and H. Hermanns. Process Algebra and Markov Chains. In: E. Brinksma, H. Hermanns, and J.-P. Katoen (Eds.), *Proceedings of the International Workshop on Formal Methods and Performance Analysis (FMPA'00)*, pp. 183–231. Springer-Verlag, Berlin (Germany), 2001. LNCS 2090.

[33] P. Buchholz. Hierarchies in Colored GSPNs. In: M.A. Marsan (Ed.), *Proceedings of the 14th International Conference on Applications and Theory of Petri Nets (Chicago, U.S.A., June 21–25)*, pp. 106–125. Springer-Verlag, Berlin (Germany), 1993. LNCS 691.

[34] P. Buchholz. Exact and Ordinary Lumpability in Finite Markov Chains. *Journal of Applied Probability*, 31: pp. 59–75, 1994.

[35] P. Buchholz. Hierarchical Markovian Models: Symmetries and Reduction. *Performance Evaluation*, 22: pp. 93–110, 1995.

[36] M. Bütow, M. Mestern, C. Schapiro, and P.S. Kritzinger. Performance Modelling with the Formal Specification Language SDL. In: R. Gotzhein and J. Bredereke (Eds.), *Proceedings of the International Conference on Formal Description Techniques; Theory, Application and Tools*, pp. 213–228. Chapman and Hall, London (United Kingdom), 1996.

[37] Cadence. Virtual Component Codesign. Available from: http://www.cadence.com.

[38] D. Choquet, P. l' Ecuyer, and C. Léger. Bootstrap Confidence Intervals for Ratios of Expectations. *ACM Transactions on Modeling and Computer Simulation*, 9 (4): pp. 326–348, October 1999.

[39] K.L. Chung. *Markov Chains with Stationary Transition Probabilities*. Springer-Verlag, 2nd edition, 1967.

[40] C.E. Clark. Importance Sampling in Monte Carlo Analyses. *Operations Research*, 9: pp. 603–620, 1961.

[41] G. Clark, S. Gilmore, and J. Hillston. Specifying Performance Measures for PEPA. In: J.-P. Katoen (Ed.), *Proceedings of the 5th International AMAST Workshop on Real-Time and Probabilistic Systems (Bamberg, Germany)*, pp. 211–227. Springer-Verlag, Berlin (Germany), 1999. LNCS 1601.

[42] Coware. System Designer and System Verifier. Available from: http://www.coware.com.

[43] M.A. Cranes and A.J. Lemoine. *An Introduction to the Regenerative Method for Simulation Analysis*, volume 4 of *Lecture Notes in Control and Information Sciences*. Springer-Verlag, 1977.

[44] J.A. Darringer, R.A. Bergamaschi, S. Bhattacharya, D. Brand, A. Herkersdorf, J.K. Morrell, I.I Nair, P. Sagmeister, and Y. Shin. Early Analysis Tools for System-on-a-Chip Design. *IBM Journal of Research and Development*, 46 (6): pp. 691–707, 2002.

[45] C. Derman. *Finite State Markovian Decision Processes*. Academic Press, New York (U.S.A.), 1970.

[46] J.L. Devore. *Probability and Statistics for Engineering and the Sciences*. Brooks, Monterey, California (U.S.A.), 1987.

[47] P. l' Ecuyer. Software for Uniform Random Number Generation: Distinguishing the Good and the Bad. In: B.A. Peters, J.S. Smith, D.J. Medeiros, and M.W. Rohrer (Eds.), *Proceedings of the Winter Simulation Conference (WSC'01)*, pp. 95–105. IEEE, Piscataway (U.S.A.), 2001.

[48] P. van Eijk. The Design of a Simulator Tool. In: P. van Eijk, C. Vissers, and M. Diaz (Eds.), *The Formal Description Technique LOTOS*. North-Holland, Amsterdam (The Netherlands), 1989.

[49] E.A. Emerson and E.M. Clarke. Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons. *Science of Computer Programming*, 2 (3): pp. 241–266, 1982.

[50] A.J. van Ewijk. *Performance Modeling of Hardware Systems with UML*. Master's thesis, Eindhoven University of Technology, Eindhoven (The Netherlands), August 2001.

[51] M. Fabian and B. Lennartson. Petri Nets and Control Synthesis: An Object-Oriented Approach. In: P. Kopacek (Ed.), *Proceedings of the IFAC Workshop on Intelligent Manufacturing Systems (IMS'94)*, pp. 365–370. Pergamon, Oxford (United Kingdom), 1994.

[52] G.S. Fishman. *Principles of Discrete Event Simulation*. John Wiley, New York (U.S.A.), 1978.

[53] D.D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, Dordrecht (The Netherlands), 2000.

[54] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading (U.S.A.), 1995.

[55] M.C.W. Geilen. *Real-Time Concepts for Software/Hardware Engineering*. Master's thesis, Eindhoven University of Technology, Eindhoven (The Netherlands), August 1996.

[56] M.C.W. Geilen. On the Construction of Monitors for Temporal Logic Properties. In: *Proceedings of the 1st Workshop on Runtime Verification (RV'01) (Paris, France, July 23)*, 2001.

[57] M.C.W. Geilen. *Formal Techniques for Verification of Complex Real-Time Systems*. PhD thesis, Eindhoven University of Technology, Eindhoven (The Netherlands), 2002.

[58] M.C.W. Geilen, J.P.M. Voeten, P.H.A. van der Putten, L.J. van Bokhoven, and M.P.J. Stevens. Object-Oriented Modelling and Specification Using SHE. *Journal of Computer Languages*, 27 (3): pp. 19 – 38, December 2001.

[59] L. Geppert. The New Chips on the Block. *IEEE Spectrum*, 38 (1): pp. 66–68, January 2001.

[60] H. Giese, M. Kardos, and U. Nickel. Towards Design Verification and Validation at Multiple Levels of Abstraction. In: B. Kleinjohann, K.H. Kim, L. Kleinjohann, and A. Rettberg (Eds.), *Proceedings If the 17th World Computer Congress on Design and Analysis of Distributed Embedded Systems*, pp. 71–80. Kluwer Academic Publishers, Norwell (U.S.A.), 2002.

[61] P.W. Glynn and D.L. Iglehart. Importance Sampling for Stochastic Simulations. *Management Science*, 35 (11): pp. 1367–1392, 1989.

[62] A. Goldberg and D. Robson. *Smalltalk-80: The Language*. Addison-Wesley, Reading, Massachusetts (U.S.A.), 1989.

[63] H. Gomma. *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Addison-Wesley, Reading, Massachusetts (U.S.A.), 2000.

[64] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Amsterdam (The Netherlands), 1996.

[65] N. Götz, U. Herzog, and M. Rettelbach. Multiprocessor and Distributed System Design: The Integration of Functional Specification and Performance Analysis Using Stochastic Process Algebras. In: E. Donatiello and R. Nelson (Eds.), *Performance Evaluation of Computer and Communication Systems (Performance'93)*, pp. 121–146. Springer-Verlag, Berlin (Germany), 1993. LNCS 729.

[66] W.K. Grassmann, M.I. Taksar, and D.P. Heyman. Regenerative Analysis and Steady-State Distributions for Markov Chains. *Operations Research*, 33 (5): pp. 1107–1116, 1985.

[67] M. Gries. *Methods for Evaluating and Covering the Design Space during Early Design Devlopment*. Technical Report UCB/ERL/M03/32, Electronics Research Laboratory, University of California at Berkeley, California (U.S.A.), August 2003.

[68] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, Boston (U.S.A.), 2002.

[69] Object Management Group. *OMG Unified Modeling Language Specification (Version 1.5)*. Technical report, March 2003.

[70] H.A. Hanson. Time and Probabilities in Specification and Verification of Real-Time Systems. In: *Proceedings of the 4th Euromicro Workshop on Real-Time Systems*, pp. 92–97. IEEE Computer Society, Los Alamitos (U.S.A.), 1992.

[71] P.G. Harrison and B. Strulo. SPADES - A Process Algebra for Discrete Event Simulation. *Journal of Logic Computation*, 10 (1): pp. 3–41, 2000.

[72] P. Heidelberger. Fast Simulation of Rare Events in Queueing and Reliability Models. *Transactions on Modeling and Computer Simulation*, 5 (1): pp. 43–85, 1995.

[73] P. Heidelberger and P.D. Welch. Simulation Run-Length Control in the Presence of an Initial Transient. *Operations Research*, 31 (6): pp. 1109–1144, 1983.

[74] T.A. Henzinger. It's about Time: Real-Time Logics Reviewed. In: D. Sangiori and R. de Simone (Eds.), *Proceedings of the 9th International Conference on Concurrency Theory (CONCUR'98) (Nice, France, September 8–11)*, pp. 439–454. Springer-Verlag, Berlin (Germany), 1998. LNCS 1466.

[75] D.P. Heyman. Accurate Computation of the Fundamental Matrix of a Markov Chain. *SIAM J. Matrix Anal. Appl.*, 16: pp. 954–963, 1995.

[76] J. Hillston. Compositional Markovian Modelling Using a Process Algebra. In: W.J. Stewart (Ed.), *Proceedings of the International Workshop on the Numerical Solutions of Markov Chains (Raleigh, U.S.A., January 16–18)*, pp. 177–196. Kluwer, Boston (U.S.A.), 1995.

[77] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[78] R.A. Howard. *Markov Models*. Wiley, London (U.K.), 1971.

[79] J. Huang and J.P.M. Voeten. Predictability in Real-Time System Development (1) Semantics Support from Development Languages. In: *Proceedings of the Forum on Specification and Design Languages (FDL'04) (Lille, France, September 14–17)*, pp. 264–277. ECSI, Gières, France, 2004.

[80] J. Huang, J.P.M. Voeten, P.H.A. van der Putten, A. Ventevogel, R. Niesten, and W. van der Maaden. Performance Evaluation of Complex Real-Time Systems: A Case Study. In: F. Karelse (Ed.), *Proceedings of PROGRESS'02*. STW Technology Foundation, Utrecht (The Netherlands), 2002.

[81] J. Huang, J.P.M. Voeten, A. Ventevogel, and L.J. Van Bokhoven. Platform-Independent Design for Embedded Real-Time Systems. In: E. Villar and J. Mermet (Eds.), *Languages for System Specification*, Chapter 3. Kluwer Academic Publishers, Dordrecht (The Netherlands), 2004.

[82] Z. Huang, J.P.M. Voeten, A. Ventevogel, and R. Niesten. Performance Modelling of the DECT Wireless Protocol. In: F. Karelse (Ed.), *Proceedings of PROGRESS'03*, pp. 115–121. STW Technology Foundation, Utrecht (The Netherlands), 2003.

[83] IBM. Blue Logic ASIC Cores. Available from: http://www.chips.ibm.com/products/asics/products/cores.

[84] IBM. The CoreConnect Bus Architecture (White Paper). Available from: http://www.chips.ibm.com/products/coreconnect, 1999.

[85] IBM. On-Chip Peripheral Bus; Architecture Specifications (Document SA-14-2528-02). Available from: http://www-3.ibm.com/chips/techlib/techlib.nsf/productfamilies/CoreConnect_Bus_Architecture, 2001.

[86] IEEE. Standard 830: IEEE Guide to Software Requirements Specification, April 1984.

[87] D.L. Iglehart. Simulating Stable Stochastic Systems, V: Comparison of Ratio Estimators. *Naval Research Logistics Quarterly*, 22 (3): pp. 553–565, 1975.

[88] ITU-T. Recommendation Z.100: Specification and Description Language (SDL), November 1999.

[89] ITU-T. Recommendation Z.109: SDL Combined with UML, November 1999.

[90] K. Jensen. Coloured Petri Nets: A High Level Language for System Design and Analysis. In: G. Rozenberg (Ed.), *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets (Bonn, Germany, 1990)*, pp. 342–416. Springer-Verlag, Berlin (Germany), 1991. LNCS 483.

[91] W.B. Joerg and K.T. Campbell. PSIM - A Simulator for Concurrent Execution of Net-Based Programs. In: *Proceedings of the IEEE Pacific Rim Conference on Communications, Computer and Signal Processing 1995*, pp. 517–520. IEEE, New York (U.S.A.), 1995.

[92] G. de Jong. A UML-Based Design Methodology for Real-Time and Embedded Systems. In: *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'02)*, pp. 776–779. IEEE Computer Society, Los Alamitos (U.S.A.), 2002.

[93] S. Juneja and P. Shahabuddin. Fast Simulation of Markov Chains with Small Transition Probabilities. *Management Science*, 47 (4): pp. 547–562, 2001.

[94] P. Kähkipuro. UML-Based Performance Modeling Framework for Component-Based Distributed Systems. In: R. Dumke (Ed.), *Proceeding of the 2th International Conference on the Unified Modeling Language (UML'99)*, pp. 167–184. Springer-Verlag, Berlin (Germany), 1999. LNCS 2047.

[95] A.F. Karr. *Probability*. Springer-Verlag, New York (U.S.A.), 1993.

[96] J.P. Katoen and P.R. d' Argenio. General Distributions in Process Algebra. In: *Lectures on Formal Methods and Performance Analysis*, pp. 375–429. Springer-Verlag, New York (U.S.A.), 2001. LNCS 2090.

[97] S. Keshav. *An Engineering Approach to Computer Networking; ATM Networks, the Internet and the Telephone Network*. Addison-Wesley, Reading, Massachusetts (U.S.A.), 1998.

[98] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System Level Design: Orthogonalization of Concerns and Platform-Based Design. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, 19 (12): pp. 1523–1543, 2000.

[99] B. Kienhuis, E.F. Deprettere, P. van der Wolf, and K. Vissers. A Methodology to Design Programmable Embedded Systems - The Y-chart Approach. In: E.F. Deprettere, J. Teich, and S. Vassiliadis (Eds.), *Embedded Processor Design Challanges: Systems, Architectures, Modeling and Simulation (SAMOS'02)*, pp. 18–37. Springer-Verlag, Berlin (Germany), 2002. LNCS 2268.

[100] P.J.B. King and R.J. Pooley. Using UML to Derive Stochastic Petri Net Models. In: N. Davies and J. Bradley (Eds.), *Proceedings of the 15th UK Performance Engineering Workshop (UKPEW'99) (Bristol, United Kingdom, July 22–23)*, pp. 45–56. University of Bristol, Bristol (United Kingdom), 1999.

[101] J.P.C. Kleijnen. *Statistical Techniques in Simulation (Part I)*, volume 9 of *Statistics: Textbooks and Monographs*. Marcel Decker, New York (U.S.A.), 1974.

[102] L Kleinrock. *Queueing Systems, Volume 1: Theory*. Wiley Interscience, New York (U.S.A.), 1975.

[103] R.D.J. Kramer. *Performance Modelling of a Network Processor using POOSL*. Master's thesis, Eindhoven University of Technology, Eindhoven (The Netherlands), January 2003.

[104] E. Kreyszig. *Introductory Mathematical Statistics*. John Wiley and Sons, 1970.

[105] A. Law and W. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, New York (U.S.A.), 3rd edition, 2000.

[106] A.M. Law and J.S. Carson. A Sequential Procedure for Determining the Length of a Steady-State Simulation. *Operations Research*, 27 (5): pp. 1011–1025, 1979.

[107] A.M. Law and M.G. McComas. How to Build Valid and Credible Simulation Models. In: B.A. Peters, J.S. Smith, D.J. Medeiros, and M.W. Rohrer (Eds.), *Proceedings of the Winter Simulation Conference (WSC'01)*, pp. 22–29. IEEE, Piscataway (U.S.A), 2001.

[108] S. Leue. Specifying Real-Time Requirements for SDL Specifications - A Temporal Logic-based Approach. In: P. Dembinski and M. Sredniawa (Eds.), *Protocol Specification, Testing and Verification XV*, pp. 19–34. Chapman and Hall, London (United Kingdom), 1997.

[109] P.A.W. Lewis, A.S. Goodman, and J.M. Miller. A Pseudo-Random Number Generator for the System/360. *IBM System's Journal*, 8: pp. 136–143, 1969.

[110] P. Lieverse, P. van der Wolf, K. Vissers, and E. Deprettere. A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems. *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, 29 (3): pp. 197–207, 2001.

[111] G. Lopez. *Modélisation, Simulation et Vérification d'un Protocole de Telecommunication*. Master's thesis, Eindhoven University of Technology, Eindhoven (The Netherlands), 1998.

[112] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York (U.S.A.), 1992.

[113] M.A. Marsan, A. Bobio, and S. Donatelli. Petri Nets in Performance Analysis: An Introduction. In: W. Reisig and G. Rozenberg (Eds.), *Lectures on Petri Nets I: Basic Models*, pp. 211–256. Springer-Verlag, Berlin (Germany), 1998. LNCS 1491.

[114] M.A. Marsan, G. Conte, and G. Balbo. A Class of Generalised Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems. *ACM Transactions on Computer Systems*, 2 (2): pp. 93–122, 1984.

[115] P. Marwedel. *Embedded System Design*. Kluwer Academic Publishers, Dordrecht (The Netherlands), 2003.

[116] Mathworks. The Simulink Tool Suite. Available from: http://www.mathworks.com.

[117] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudorandom Number Generator. *ACM Transactions on Modeling and Computer Simulations: Special Issue on Uniform Random Number Generation*, 8 (1): pp. 3–30, January 1998.

[118] D.C. McNickle, K. Pawlikowski, and G. Ewing. Experimental Evaluation of Confidence Interval Procedures in Sequential Steady-State Simulation. In: J.M. Charmes, D.J. Morrice, D.T. Brunner, and J.J. Swain (Eds.), *Proceedings of the Winter Simulation Conference (WSC'96)*, pp. 382–389. SCS International, San Diego (U.S.A.), 1996.

[119] M.S. Meketon and B. Schmeiser. Overlapping Batch Means: Something for Nothing? In: S. Sheppard, U.W. Pooch, and C.D. Pegden (Eds.), *Proceeding of the Winter Simulation Conference (WSC'84)*, pp. 227–230. IEEE, New York (U.S.A.), 1984.

[120] M. Meo, E. de Souze e Silva, and M.A. Marsan. Efficient Solution for a Class of Markov Chain Models of Telecommunication Systems. *Performance Evaluation*, 27/28: pp. 603–625, 1996.

[121] S.P. Meyn and R.L Tweedie. *Markov Chains and Stochastic Stability*. Springer-Verlag, Berlin (Germany), 1993.

[122] R. Milner. *Communication and Concurrency*. Prentice-Hall, Englewood Cliffs, New Jersey (U.S.A.), 1989.

[123] R. Mirandola and V. Cortelessa. UML Based Performance Modeling of Distributed Systems. In: A. Evans, S. Kent, and B. Selic (Eds.), *Proceedings of the 3rd Conference on the Unified Modeling Language (UML'00)*, pp. 178–193. Springer-Verlag, Berlin (Germany), 2000. LNCS 1939.

[124] M.K. Molloy. Performance Analysis using Stochastic Petri Nets. *IEEE Transactions on Computers*, 31 (9): pp. 913–917, 1982.

[125] D.C. Montgomery and G.C. Runger. *Applied Statistics and Probability for Engineers*. John Wiley and Sons, 1994.

[126] K. S. Namjoshi. *Ameliorating the State Space Explosion Problem*. PhD thesis, University of Texas at Austin, 1998.

[127] X. Nicollin and J. Sifakis. An Overview and Synthesis on Timed Process Algebras. In: K. Larsen and A. Skou (Eds.), *Proceeding of the 3rd International Workshop on Computer Aided Verification (CAV'91) (Ålborg, Denmark, July 1–4)*, pp. 376–398. Springer-Verlag, Berlin (Germany), 1991.

[128] A. Niemegeers, G. de Jong, and M. de Langhe. Early System Co-Validation Based on a Hierarchical Specification Flow. In: *Proceedings of the 4th International High Level Design Validation and Test Workshop (HLDVT'99)*. IEEE Computer Society, Los Alamitos (U.S.A.), 1999.

[129] W.D. Obal and W.H. Sanders. State-Space Support for Path-Based Reward Variables. *Performance Evaluation*, 35 (3–4): pp. 233–251, 1999.

[130] OMG. *UML Profile for Schedulability, Performance and Time Specification*. OMG Adopted Specification ptc/02-03-02, Object Management Group, March 2002.

[131] M. Rettelbach T. Ackermann P. Liggesmeyer, M. Rothfelder. Qualitätssicherung Software-basierter technischer Systeme - Problembereiche und Lösungsanzätze. *Informatiek-Spektrum*, 21 (5): pp. 249–258, 1998.

[132] A. Papoulis. *Probability, Random Variables and Stochastic Processes*. McGraw-Hill, London (U.K.), 2nd edition, 1984.

[133] S.K. Park and K.W. Miller. Random Number Generators: Good Ones are Hard to Find. *Communications of the ACM*, 31 (10): pp. 1192–1201, October 1988.

[134] B. Partee, A. ter Meulen, and R. Wall. *Mathematical Methods in Linguistic*. Kluwer Academic Publishers, 1990.

[135] K. Pawlikowski. Steady-State Simulation of Queueing Processes: A Survey of Problems and Solutions. *ACM Computing Surveys*, 22 (2): pp. 123–170, June 1990.

[136] K. Pawlikowski, H. D. J. Jeong, and J. S. R. Lee. On Credibility of Simulation Studies of Telecommunication Networks. *IEEE Communications Magazine*, 40 (1): pp. 132–139, 2002.

[137] A.D. Pimentel, P. van der Wolf, E.F. Deprettere, L.O. Hertzberger, J.T.J Van Eijndhoven, and S. Vassiliadis. The Artemis Architecture Workbench. In: J.P. Veen (Ed.), *Proceedings of PROGRESS'00 (Utrecht, The Netherlands, October 13)*, pp. 69–78. STW Technology Foundation, Utrecht (The Netherlands), 2000.

[138] J. Plantin and E. Stoy. Aspects of System-Level Design. In: *Proceedings of the 7th International Workshop on Hardware/Software Codesign (CODES'99)*, pp. 209–210. ACM, New York (U.S.A.), 1999.

[139] G.D. Plotkin. *A Structural Approach to Operational Semantics*. Technical Report DAIMI FN-19, University of Aarhus, Aarhus (Denmark), 1981.

[140] R.J. Pooley. Using UML to Derive Stochastic Process Algebra Models. In: N. Davies and J. Bradley (Eds.), *Proceedings of the 15th UK Performance Engineering Workshop (UKPEW'99) (Bristol, United Kingdom, July 22-23)*, pp. 23–34. University of Bristol, Bristol (United Kingdom), 1999.

[141] R.J. Pooley and P.J.B. King. The Unified Modelling Language and Performance Engineering. *IEE Proceedings - Software*, 146 (1): pp. 2–10, February 1999.

[142] Y. Pribadi, J.P.M. Voeten, and B.D. Theelen. Reducing Markov Chains for Performance Evaluation. In: F. Karelse (Ed.), *Proceedings of PROGRESS'01 (Utrecht, The Netherlands, October 18)*, pp. 173–179. STW Technology Foundation, Utrecht (The Netherlands), 2001.

[143] M. Priestley. *Practical Object-Oriented Design with UML*. McGraw-Hill, London (U.K.), 2nd edition, 2004.

[144] P.H.A. van der Putten. *Specification of Complex Hardware/Software Systems*. Eindhoven University of Technology, Eindhoven (The Netherlands), 2002. Lecture Notes.

[145] P.H.A. van der Putten and J.P.M. Voeten. *Specification of Reactive Hardware/Software Systems*. PhD thesis, Eindhoven University of Technology, Eindhoven (The Netherlands), 1997.

[146] Rational. Rose Real Time (Also known as Rose Technical Developer). Available from: http://www-3.ibm.com/software/awdtools/developer/technical.

[147] M. Ribaudo. Stochastic Petri Net Semantics for Stochastic Process Algebras. In: *Proceedings of the 6th International Workshop on Petri Nets and Performance Models (Durham, U.S.A., October 3–6)*, pp. 148–157. IEEE Computer Society, Los Alamitos (U.S.A.), 1995.

[148] C. Rowen. Reducing SoC Simulation and Development Time. *IEEE Computer*, 35 (12): pp. 29–34, December 2002.

[149] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Amsterdam (The Netherlands), 1999.

[150] R.G. Sargent. Verification, Validation, and Accreditation of Simulation Models. In: J.S. Joines, R.R. Barton, K.Kang, and P.A. Fishwick (Eds.), *Proceedings of the Winter Simulation Conference (WSC'00) (Orlando, Florida, December 10–13)*, pp. 50–59, 2000.

[151] A. Schmietendorf and E. Dimitrov. Possibilities of Performance Modelling with UML. In: R. Dumke (Ed.), *Proceedings of the 4th Conference on the Unified Modeling Language (UML'01)*, pp. 78–95. Springer-Verlag, Berlin (Germany), 2001. LNCS 2047.

[152] L.W. Schruben. Detecting Initialization Bias in Simulation Output. *Operations Research*, 30: pp. 569–590, 1982.

[153] L.W. Schruben. Confidence Interval Estimation using Standardized Time Series. *Operations Research*, 31 (6): pp. 1090–1108, 1983.

[154] B. Selic. The Real-Time UML Standard: Definition and Application. In: B. Werner (Ed.), *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'02)*, pp. 770–772. IEEE Computer Society, Los Alamitos (U.S.A.), 2002.

[155] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modelling*. Wiley and Sons, New York (U.S.A.), 1994.

[156] P. Shahabuddin. Rare Event Simulation in Stochastic Models. In: C. Alexopoulos, K. Kang, W.R. Lilegdon, and D. Goldsman (Eds.), *Proceedings of the Winter Simulation Conference (WSC'95)*, pp. 178–185. IEEE, New York (U.S.A.), 1995.

[157] G.S. Shedler. *Regenerative Stochastic Simulation*. Statistical Modeling and Decision Science. Academic Press Inc., London (U.K.), 1993.

[158] T.J. Sheskin. A Markov Partitioning Algorithm for Computing Steady-State Probabilities. *Operations Research*, 33 (5): pp. 228–235, 1985.

[159] N. Sidorova and M. Steffen. Verifying Large SDL-Specifications Using Model Checking. In: R. Reed and J. Reed (Eds.), *Proceedings of the 10th International SDL Forum*, pp. 403–420. Springer-Verlag, Berlin (Germany), 2001. LNCS 2078.

[160] K. Skadron, M. Martonosi, D.I. August, M.D. Hill, D.J. Lilja, and V.S. Pai. Challenges in Computer Architecture Evaluation. *IEEE Computer*, 36 (8): pp. 30–36, 2003.

[161] P.J. Smith, M. Shafi, and H. Gao. Quick Simulation: A Review of Importance Sampling Techniques in Communication Systems. *Journal on Selected Areas in Communications*, 15 (4): pp. 597–613, 1997.

[162] A. Sokolova and E.P. de Vink. On Relational Properties of Lumpability. In: F. Karelse (Ed.), *Proceedings of PROGRESS'03 (Nieuwegein, The Netherlands, October 22)*. STW Technology Foundation, Utrecht (The Netherlands), 2003.

[163] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts (U.S.A.), 3rd edition, 1997.

[164] Synopsys. System Studio. Available from: http://www.synopsys.com.

[165] B.D. Theelen. *Deliverable 2.1: Performance Modelling and Architecture Exploration of the Dataflow System*. Alcatel – TUE frame project: Structured System Design, Eindhoven University of Technology, Eindhoven (The Netherlands), January 2002.

[166] B.D. Theelen. *Deliverable 2.2: Guidelines for System-Level Methods on Abstraction and Refinement*. Alcatel – TUE frame project: Structured System Design, Eindhoven University of Technology, Eindhoven (The Netherlands), January 2002.

[167] B.D. Theelen, P.H.A. van der Putten, and J.P.M. Voeten. Using the SHE Method for UML-based Performance Modelling. In: E. Villar and J. Mermet (Eds.), *System Specification and Design Languages*, Chapter 12, pp. 143–160. Kluwer Academic Publishers, Dordrecht (The Netherlands), 2003.

[168] B.D. Theelen and J.P.M. Voeten. *Deliverable 1.1: Modelling the MPSR Switch System using POOSL; Performance Analysis for System-Level Design*. Alcatel – TUE frame project: Performance Analysis, Technische Universiteit Eindhoven, Eindhoven (The Netherlands), March 2000.

[169] B.D. Theelen and J.P.M. Voeten. *Deliverable 1.2: Simulating the Evolved POOSL Model of the MPSR Switch System; Performance Analysis for System-Level Design*. Alcatel – TUE frame project: Performance Analysis, Technische Universiteit Eindhoven, Eindhoven (The Netherlands), May 2000.

[170] B.D. Theelen and J.P.M. Voeten. *Deliverable 1.3: Guidelines for System-Level Methods and Tools; Performance Analysis for System-Level Design*. Alcatel – TUE frame project: Performance Analysis, Eindhoven University of Technology, Eindhoven (The Netherlands), June 2000.

[171] B.D. Theelen, J.P.M. Voeten, L.J. van Bokhoven, P.H.A. van der Putten, A.M.M. Niemegeers, and G.G. Jong. Performance Modeling in the Large: A Case Study. In: N. Giambiasi and C. Frydman (Eds.), *Proceedings of the 13th European Simulation Symposium (ESS'01) (Marseille, France, October 18–21)*, pp. 174–181. SCS-Europe, Ghent (Belgium), 2001.

[172] B.D. Theelen, J.P.M. Voeten, and R.D.J. Kramer. Performance Modelling of a Network Processor using POOSL. *Journal of Computer Networks*, 41 (5): pp. 667–684, April 2003.

[173] B.D. Theelen, J.P.M. Voeten, and Y. Pribadi. Accuracy Analysis of Long-run Average Performance Metrics. In: F. Karelse (Ed.), *Proceedings of PROGRESS'01 (Veldhoven, The Netherlands, October 18)*, pp. 261–269. STW Technology Foundation, Utrecht (The Netherlands), 2001.

[174] L. Thiele, S. Chakraborty, M. Gries, and S. Künzli. Design Space Exploration of Network Processor Architectures. In: M.A. Franklin, P. Crowley, H. Hadimioglu, and P.Z. Onufryk (Eds.), *Network Processor Design: Issues and Practices Volume*, Chapter 4. Morgan Kaufman, 2002.

[175] H.C. Tijms. *Stochastic Models; An Algorithmic Approach*. John Wiley & Sons, Chichester (England), 1994.

[176] M.Y. Vardi. Automatic Verification of Probabilistic Concurrent Finite-State Programs. In: *Proceedings of FOCS'85*, pp. 327–338. IEEE Computer Society Press, 1985.

[177] M. Verhappen, J.P.M. Voeten, and P.H.A. van der Putten. Traversing the Fundamental System-Level Design Gap Using Modelling Patterns. In: *Proceedings of the Forum on Specification and Design Languages (FDL) 2003 (Frankfurt, Germany, September 23–26)*. Kluwer Academic Publishers, 2003.

[178] J.P.M. Voeten. *POOSL: An Object-Oriented Language for the Analysis and Design of Hardware/Software Systems*. EUT report 95-E-290, Eindhoven University of Technology, Eindhoven (The Netherlands), 1995.

[179] J.P.M. Voeten. *Semantics of POOSL: An Object-Oriented Specification Language for the Analysis and Design of Hardware/Software Systems*. EUT report 95-E-293, Eindhoven University of Technology, Eindhoven (The Netherlands), 1995.

[180] J.P.M. Voeten. Performance Evaluation with Temporal Rewards. *Performance Evaluation*, 50 (2/3): pp. 189–218, November 2002.

[181] J.P.M. Voeten, M.C.W. Geilen, L.J. van Bokhoven, P.H.A. van der Putten, and M.P.J. Stevens. A Probabilistic Real-Time Calculus for Performance Evaluation. In: G. Horton, D. Möller, and U. Rüde (Eds.), *Proceedings of the 11th European Simulation Symposium (ESS'99) (Erlangen, Germany, October 26–28)*, pp. 608–617. SCS-Europe, Delft (The Netherlands), 1999.

[182] P.D. Welch. The Statistical Analysis of Simulation Results. In: S.S. Lavenberg (Ed.), *The Computer Performance Modeling Handbook*, pp. 267–329. Academic Press, New York (U.S.A.), 1983.

[183] F.N. van Wijk. *System-Level Modelling and Design-Space Exploration*. PhD thesis, Eindhoven University of Technology, to appear.

[184] F.N. Van Wijk, J.P.M. Voeten, and A.J.W.M. ten Berg. An Abstract Modeling Approach Towards System-Level Design-Space Exploration. In: E. Villar and J. Mermet (Eds.), *System Specification and Design Languages*, Chapter 22, pp. 267–282. Kluwer Academic Publishers, Dordrecht (The Netherlands), 2003.

[185] G.A.J. Willemse. *Research on Design Methods for Hard/Software Systems; Use of UML in the context of SHE*. Master's thesis, Eindhoven University of Technology, Eindhoven (The Netherlands), March 2001.

[186] V.D. Živković and P. Lieverse. An Overview of Methodologies and Tools in the Field of System-Level Design. In: *Embedded Processor Design Challenges: Systems, Architectures, Modeling and Simulation (SAMOS'01)*, pp. 74–88. Springer-Verlag, Berlin (Germany), 2002. LNCS 2268.

Glossary of Symbols

This glossary lists where some (non-standard) mathematical symbols are introduced.

Symbol	**Description**	**Page**
$\perp$	Undefined	170
$x \dot{-} y$	Monus Operator	170
$\bar{\mathbb{R}}$	Extended Reals	171
$\lvert X \rvert$	Set Cardinality	169
$\mathbf{2}^X$	Powerset	169
$\mathbb{P}(X)$	Probability	171
$\mathbb{P}(X \mid Y)$	Conditional Probability	171
$\mathbb{P}^*(U)$	Conditional Probability	21
π_T	Equilibrium Probability	23
$\mathrm{N}(\mu, \sigma^2)$	Normal Distribution	172
$\mathrm{N}(0,1)$	Standard Normal Distribution	172
$\mathfrak{N}(\kappa)$	Distribution Function of $\mathrm{N}(0,1)$	172
$v^i_{S_r}$	Random Variables	24
$t^k_{S_r}$	Random Variables	24
$Y^k_{S_r}$	Random Variables	24
$U^k_{S_r}$	Random Variables	25
$L^k_{S_r}$	Random Variables	26
$\mathcal{L}^i$	$\mathcal{L}^i$ Space	173
$\mathbb{E}[X]$	Expectation	173
$\mathrm{var}[X]$	Variance	173
$\mathrm{std}[X]$	Standard Deviation	173
$\mathrm{cov}[X]$	Covariance	173
$\bar{\theta}$	Point Estimate	175
$\hat{\theta}$	Point Estimator	175
$X \xrightarrow{\text{a.s.}} Y$	Almost Sure Convergence	174
$X \xrightarrow{\text{d.}} Y$	Convergence in Distribution	174
$\underline{x}$	Sequence	170
$\underline{x}_i$	Component of Sequence	170
$\lvert \underline{x} \rvert$	Sequence Length	170
$\underline{x}_{1..n}$	Sequence Prefix	170
$\underline{x}^{\rightsquigarrow}$	Thin Cylinder	20
$U^{\rightsquigarrow}$	Generalised Cylinder	21

Symbol	Description	Page
$\mathcal{Z}_S^T$	Set of State Sequences	22
$U \circ V$	Concatenation	22
$X^{\restriction c}$	Reduced to c	33
$\mathcal{T}$	Time Domain	173, 183
$\mathcal{T}^+$	Positive Time	184
$\mathcal{D}(\mathcal{C})$	Set of Distribution Functions	183
$C \xrightarrow{a} \boldsymbol{\pi}$	Action Transition	183
$C \xrightarrow{t} C'$	Time Transition	184
$C \overset{a,p}{\Longrightarrow} C'$	Action Transition	66
$C \overset{t}{\Longrightarrow} C'$	Time Transition	67
$-$	No Transition	67
Θ	Previous-Reward Operator	51
$\top$	Progress in Model Time	50
$currentTime$	Current Model Time	72
Δ	Duration in Model Time	50

Index

Curriculum Vitae

Bart Theelen was born on November 15, 1974 in Heerlen, the Netherlands. In 1993, he received his Atheneum diploma from the Bisschoppelijk College Sittard.

From 1993, Bart Theelen studied Information Technology at the Department of Electrical Engineering of the Eindhoven University of Technology in the Netherlands. As a trainee, he did a project on concepts for a single-chip multi-processor platform with an on-chip real-time operating system kernel and a project on the detection of symmetries in data flow graphs to improve the efficiency of constraint-analysis based design automation tools. Furthermore, he performed a project at the Forschungszentrum Jülich in Germany, in which he contributed to the development of a tool for analysing the communication of messages between the processors of Cray T3E supercomputers. His graduation project was on modelling WDM networks and resulted in an extensive report on coherencies between several telecommunication standards of IEEE, ITU, ISO and ANSI. He received his M.Sc. degree in Information Technology from the Eindhoven University of Technology in 1999.

Since 1999, Bart Theelen worked towards his Ph.D. degree on performance modelling for system-level design at the Information and Communication Systems group of the Eindhoven University of Technology. He contributed to the development of techniques for performance analysis and proposed a method for applying them during system-level design of hardware/software systems. In addition, he performed three industrial case studies. In cooperation with Alcatel Bell in Antwerp, Belgium, he analysed the performance of product variants of a backbone Internet Router and the performance of design alternatives for the memory arbitration in the Internet Router's input buffer subsystems. In cooperation with IBM Research Laboratory in Zürich, Switzerland, he contributed to analysing the performance of a network processor. While obtaining his Ph.D. degree, he also performed various educational tasks including the coaching of four graduate students and five trainees.

List of Refereed Publications

- B.D. Theelen, A.C. Verschueren, V.V. Reyes Suárez, M.P.J. Stevens, and A. Nuñez. A Scalable Single-Chip Multi-Processor Architecture with On-Chip RTOS Kernel. In: M. Edwards and L. Józwiak (Eds.), *Journal on Systems Architecture*, 49 (12–15): pp. 619–639, December 2003.

- B.D. Theelen, P.H.A. van der Putten, and J.P.M. Voeten. Using the SHE Method for UML-based Performance Modelling. In: E. Villar and J. Mermet (Eds.), *System Specification and Design Languages*, Chapter 12, pp. 143–160. Kluwer Academic Publishers, Dordrecht (The Netherlands), 2003

- B.D. Theelen, J.P.M. Voeten, and R.D.J. Kramer. Performance Modelling of a Network Processor using POOSL. *Journal of Computer Networks*, 41 (5): pp. 667–684, April 2003.

- B.D. Theelen, P.H.A. van der Putten, and J.P.M. Voeten. Using the SHE Method for UML-based Performance Modelling. In: F. Pourchelle (Ed.), *Proceedings of the Forum on Specification and Design Languages (FDL'02) (Marseille, France, September 24–27)*. ECSI, Gieres (France), 2002.

- B.D. Theelen and A.C. Verschueren. Architecture Design of a Scalable Single-Chip Multi-Processor. In: M. Edwards (Ed.), *Proceedings of the Euromicro Symposium on Digital System Design (DSD'02) (Dortmund, Germany, September 4–6)*, pp. 132–139. IEEE Computer Society, Los Alamitos (U.S.A.), 2002.

- B.D. Theelen, J.P.M. Voeten, P.H.A. van der Putten, H.J.S. Dorren, and M.P.J. Stevens. Concurrent Support of Higher-layer Protocols over WDM. *Photonic Network Communications*, 4 (1): pp. 47–62, January 2002.

- B.D. Theelen, J.P.M. Voeten, L.J. van Bokhoven, P.H.A. van der Putten, A.M.M. Niemegeers, and G.G. de Jong. Performance Modeling in the Large: A Case Study. In: N. Giambiasi and C. Frydman (Eds.), *Proceedings of the 13th European Simulation Symposiom (ESS'01) (Marseille, France, October 18–21)*, pp. 174–181. SCS Europe, Ghent (Belgium), 2001.

- B.D. Theelen, J.P.M. Voeten, and Y. Pribabi. Accuracy Analysis of Long-run Average Performance Metrics. In: F. Karelse (Ed.), *Proceedings of PROGRESS'01 (Veldhoven, The Netherlands, October 18)*, pp. 261–269. STW Technology Foundation, Utrecht (The Netherlands), 2001.

- B.D. Theelen, J.P.M. Voeten, L.J. van Bokhoven, G.G. de Jong, A.M.M. Niemegeers, P.H.A. van der Putten, M.P.J. Stevens, and J.C.M. Baeten. System-Level Modelling and Performance Analysis. In: J.P. Veen (Ed.), *Proceedings of PROGRESS'00 (Utrecht, The Netherlands, October 13)*, pp. 141–147. STW Technology Foundation, Utrecht (The Netherlands), 2000.

- B.D. Theelen, J.P.M. Voeten, P.H.A. van der Putten, H.J.S. Dorren, and M.P.J. Stevens. Modelling Optical WDM Networks using POOSL. In: J.P. Veen (Ed.), *Proceedings of ProRISC'99 (Mierlo, The Netherlands, November 24–26)*, pp. 503–508. STW Technology Foundation, Utrecht (The Netherlands), 1999.

Stellingen

behorende bij het proefschrift

Performance Modelling for System-Level Design

van Bartholomeus Desiderius Theelen

1. In case a tool relies on a modelling language for which a rigorous framework to compute performance metrics is missing (such as for C++ based languages), simulation-based estimation with this tool cannot lead to credible results.
2. The hardest part of system-level design is making adequate abstractions when developing models of design alternatives.
3. The success of a design method based on a new superior modelling language does not only depend on the availability of accompanying user-friendly and efficient tools, but also on the willingness to learn the new modelling language.
4. Moore's Law is al lang geen gevolg van de technologische vooruitgang meer, maar eerder een oorzaak.
5. Terminologie kan heel verhelderend werken maar kan net zo goed veel verwarring veroorzaken. Zo weten hardware en software ontwerpers goed wat er met een 'architectuur' wordt bedoeld, totdat ze met elkaar gaan praten.
6. Om de computers van morgen efficiënt te kunnen ontwerpen zouden we vandaag al over de computers van overmorgen moeten beschikken.
7. Het onderbouwen van de kwaliteit en originaliteit van de ideeën die aan een proefschrift ten grondslag liggen kost vaak meer tijd dan het bedenken van die ideeën.
8. Het succes van een uitvinding is vooral te meten aan de eenvoud waarmee deze een probleem oplost.
9. De Nederlandse regering zou als een goed gedirigeerd orkest moeten werken. Helaas is het in de praktijk vaak te vergelijken met een rommelig toneelstuk waarbij onduidelijk is waarvoor de acteurs staan.
10. Het zeen neet allein de vlaaj, het vouksleed, de väöle fiesten, en het sjoene landjsjap det Limburgers oet Belsj en Holles mit ein vereinigt, mer veural de sósjaal-kulturele versjille mit angere Vlaminge en Hollenjers.