A Lightweight Formal Method for the Prediction of Non-Functional System Properties

Der Technischen Fakultät der

Universität Erlangen-Nürnberg

zur Erlangung des Grades

DOKTOR-INGENIEUR

vorgelegt von

Jörg Barner

Erlangen — 2005

Als Dissertation genehmigt von
der Technischen Fakultät der
Universität Erlangen-Nürnberg

Tag der Einreichung: 20.06.2005

Tag der Promotion: 14.10.2005

Dekan: Prof. Dr.-Ing. Alfred Leipertz

Berichterstatter: Prof. Dr. Fridolin Hofmann, Prof. Dr. Khalid Al-Begain

Acknowledgments

First of all, I would like to thank *Prof. Dr. Fridolin Hofmann* for supervising this dissertation and for the open environment he created together with *Prof. Dr.-Ing. Wolfgang Schröder-Preikschat* in our department which allowed me to finish this project. Furthermore, I am particularly grateful to *Prof. Dr. Khalid Al-Begain*, my co-promoter, whose interest in applications and the encouraging feedback he gave were always a great pleasure.

I owe a lot of thanks to *Dr.-Ing. Gunter Bolch*, my "Ana" research group leader for many inspiring discussions which helped me a lot to sharpen my thoughts on important issues and prevented me from losing my direction.

Without the commitment that *Björn Beutel* and *Patrick Wüchner* exhibited during the preparation of their master theses some of the concepts presented in this dissertation would have remained purely theoretical.

My former office mate *Dr.-Ing. Michael Schröder* proof-read various intermediate versions of this dissertation. Together with him, *Stephan Kösters* supported me in the nerve-wracking final phase before submitting the thesis by doing some valuable last-minute proof-reading.

It has been a lot of fun to work at the department for distributed systems and operating systems with all the colleagues and students. The pleasant working atmosphere was extraordinary, and I will always remember my time there with happiness! Special acknowledgments go to *Dr.-Ing. Uwe Linster* who gave me the opportunity to participate in the interesting *Univ*IS project and to *Dr.-Ing. Jürgen Kleinöder* for his support concerning various funding and working contract issues.

Finally, I would like to express my gratitude to my parents *Ilsetraut* and *Klaus Barner* for their continuous and loving support over all these years. Far away in "down under" *Leonie*, *Christopher* and *Ellen Barner* always kept their fingers crossed for me and my project — Thank you all!

Abstract

The goal of performance and reliability evaluation during the development of computer-based systems is to predict the compliance of the projected system with a set of non-functional requirements. As an example, consider the expected mean end-to-end delay of data packets in a communication network, which can be predicted on the basis of a stochastic model during the early conceptual design phase. A glance at the software engineering practice reveals that despite their high financial savings potential academic methods of performance evaluation are not applied on a large scale. Frequently quoted reasons for these findings are the cumbersome notation of some stochastic description techniques and the impractical presentation of the theoretical foundations. The successful application of performance evaluation methods thus remains reserved to a handful of well-educated experts. This phenomenon, termed *insularity problem* by some members of the academic performance evaluation community, can be regarded as a special case of the general problem of *low formal method acceptance* in industrial software development. The first formal methods were developed by several computer scientists about 40 years ago with the ambitious goal to create programs which are proven to be *correct by construction*. Because of the quite revolutionary approach to programming and the inability of their developers to show that formal methods are appropriate to solve larger real-world problems they did not become widely accepted and used by the software engineering industry. A promising attempt to bridge this "industry-academia gap" was initiated in the early-mid 1990s with the advent of "lightweight" formal methods for the verification of functional system requirements, some of which could be applied successfully in industrial software development projects.

The initial point of view which is adopted in this dissertation is to regard performance evaluation as a branch of requirements verification techniques which are applied during the development of software-based systems. Because of the similarities in the structure and the objectives of the methods in both areas, we assume that the transfer of the lightweight formal method concept to the area of performance evaluation will have a positive effect on the solution of the insularity problem. In order to corroborate this assumption, a new lightweight formal method for the prediction of nonfunctional system requirements is developed within the scope of this thesis. The method consists of a formal description technique which is based on a well-defined syntax and semantics. It enables the user to specify the system structure and behaviour, the interaction of the system with its enviroment as well as the interesting nonfunctional system requirements as a stochastic model on a high abstraction level. The syntax of our description language combines the advantages of the pragmatic network-oriented modelling paradigm with the succinctness of a pure textual notation. The evaluation of the system properties is carried out in a user-friendly manner by the automatic analysis of the model in the evaluation environment prototype which was implemented in the context of this thesis. In order to demonstrate the applicability of the method we present some detailed case studies from the area of communication systems (WLAN, GSM). The keynote of this dissertation is to promote the transfer of technology from academia to industry by the integration of formal methods for performance and reliability evaluation in the early phases of the software development process.

Parts of the material presented in this thesis has previously been published as:

1. Al-Begain, Khalid; Barner, Jörg; Bolch, Gunter; Zreikat, Aymen: The Performance and Reliability Modelling Language MOSEL-2 and its Applications. *International Journal on Simulation: Systems, Science and Technology*, 3(3 – 4)(66-80), 2002.

2. Barner, Jörg: Performance and reliability modelling with MOSEL-2. Tutorial. *ASMTA-Track of the 17th European Simulation Multiconference: Foundations for Successful Modelling & Simulation (ESM'03)*, Nottingham Trent University, Nottingham, England, Juni 2003

3. Barner, Jörg; Bolch, Gunter: MOSEL-2: Modeling, Specification and Evaluation Language, Revision 2 . In: Sanders, William (Hrsg.): *Proc. 13th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation* (Performance TOOLS 2003 Urbana-Champaign, Illinois, 2 - 5.9 2003). 2003, S. 222 – 230.

Table of Contents

List of Figures

1. Introduction

"Mind the gap!"
LONDON UNDERGROUND security announcement, since approx. 1960

1.1. Motivation

The complexity of modern technical systems, such as telecommunication networks, computer systems, manufacturing systems or automobiles and aircrafts has increased steadily over the last decades. This complexity stems from the large number and heterogeneity of a system's components, from the multi-layered architecture used in the system's design and from the unpredictable nature of the interactions between the system and its environment.

The vast majority of modern systems consist of both hardware and software which controls the operation of the system. Unfortunately, the quality of the software components, regarding their functional correctness, i.e. absence of deadlocks or live-locks, proper reaction in unpredictable exceptional situations, as well as their performance characteristics like mean system response time, throughput and resource utilisation, often failed to keep in step with the computational power offered by the computing hardware and the flexibility featured by the operating systems available. These observations culminated in the identification of a *software crisis*[1] by EDSGER DIJKSTRA [Dij72] in the beginning of the 1970ies.

As a reaction to the crisis, researchers and practitioners in the field of *software engineering* (SWE), which emerged in the end of the 1960ies (see NAUR et al. [NR68]), have created numerous methods for the systematic development of complex (software) systems. One of the key principles of SWE is to separate the system design from coding; the system development process should be carried out in several phases. During each of these phases representations of the envisioned system should be created at appropriate levels of abstraction in order to specify and study the structure and behaviour of the final product.

Moreover, models of the system can be employed for an automated, tool-based analysis of functional and nonfunctional properties, provided that they rely on some precise or *formal* mathematical framework. For the *model-based* prediction of nonfunctional system properties, classical *Performance Evaluation* (PE) formalisms such as Markov models [MS06], Queueing Networks [Jac54], various types of stochastic Petri nets [Pet62] [Mol81] or the newer Stochastic Process Algebras [NY85] can be used. Some of these formalisms are much older than SWE: Markov models have been in use for *capacity planning* of telephone switching centres since 1917 (ERLANG [Erl17]) and Queueing Networks were introduced in the 1950ies to solve problems in the area of Operations Research (JACKSON [Jac57]). Since

[1] In 1979 ROBERT W. FLOYD suggested that in view of the continuing problems in software development *"software depression"* would be a more appropriate term [Flo79].

all the modelling formalisms mentioned above map system descriptions on *stochastic processes* [Ros83] as the underlying mathematical framework, the generic term *stochastic modelling* is commonly used for model-based performance evaluation activities.

Although it is obvious that methods from the fields of Software Engineering and Performance Evaluation should be used conjointly in order to improve the quality of complex software systems, practice reveals that the interplay of both sub-disciplines is often disturbed by the following deficiencies:

- CONNIE SMITH spotted a *"fix-it-later"* mentality [Smi03] among software engineers concerning the assessment of non-functional system properties. They focus on functional properties during the system development process. Performance — if at all — becomes an issue during the system test phase. The fact is ignored that most performance shortcomings have their origin in a faulty design, and thus are introduced in the early development phases. If the system's performance requirements are not met, a costly redesign is the consequence (see Sect. 2.1.3). Nevertheless, the declining attitude towards early performance evaluation is common among software engineering practitioners and also widespread within academia. One of the favourite citations frequently quoted by tenacious opponents of PE is the following statement of D. KNUTH ([Knu74]) [2]:

  > *... We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. ...*

- According to DOMENICO FERRARI the Performance Evaluation community has long been suffering from an *"insularity"* problem, as he reported in his self-critical and controversial discussed article [Fer86]. He claims that Performance Evaluation research is carried out in isolation of the mainstream computer science and engineering fields. Many PE researchers provide theoretical solutions to outdated problems, e.g. the evaluation of monoprocessor computer systems while the rest of the world concentrates on the design of multiprocessor systems. This attitude is promoted by a lack of multidisciplinary spirit and the reluctance to present the theoretical concepts in a way that they can be easily accessed by the practitioners. In a recently published contribution [Fer03] FERRARI reassesses the situation and draws the conclusion that although many Performance Evaluation experts "left the island" during the last 20 years, there is still a lot of work to be done on the part of the PE community in order to integrate and synchronise their work with the mainstream.

- A grave problem is the *"industry-academia gap"* concerning the importance and rôle of formal methods in system development. Academics are often too enthusiastic about the achievements of formalisation. In their idealistic view a complete formalisation of all system aspects during the various stages of the development process will lead to an almost completely correct system, i.e. a final product which satisfies all the requirements. On the other hand many practitioners are deterred from using formal

[2] Although not stated explicitly in [Knu74] this statement was probably not coined by KNUTH himself but rather has to be placed to the credit of his colleague and friend R. FLOYD.

methods because of the abstract and sometimes cumbersome syntax used in formal description techniques. Even in the cases where practitioners are willing to use a new formal method in their project they often have to discover that the application of the theory does not solve their problem. In [Hol96] GERALD HOLZMANN accurately describes their experiences as follows:

> *... Many practitioners are interested in new solutions. They are prepared to listen to you and to try. However, the key to their problems delivered by researchers usually does not fit, and when the practitioner comes back complaining, he is told that it is not the key which is wrong, but the lock, ... and the door, and the wall...*

- Scientific research in general, and particularly in PE, is often carried out in a *"put the cart before the horse"*-like fashion, i.e. a *solution* is developed *before* a concrete *problem* was identified. After the development of new foundational theories, methods and algorithms a case study — the problem — is *"designed"* in order to prove that the application of the new theory can solve it. This mentality also dominates the academic perspective on the rôle of modelling which is quite different from the engineering view. In [BJ96] FREDERICK P. BROOKS hits the mark, as he states:

  > *The scientist builds in order to study — the engineer studies in order to build.*

  Unfortunately, the models created in many "real-world" case studies presented in academic contributions are distorted by oversimplification, unrealistic assumptions, and modelling artefacts. As a consequence of this, the statements which are obtained by reasoning in the formal world of the model cannot be translated back to any 'useful' statement about the system in the informal language of the real-world domain.

In summation, the points mentioned above open out into a problem of *technology transfer* and *method integration* [Kro92]: The Performance Evaluation community has developed a lot of foundational theories and solutions for the model-based *analysis* and evaluation of quantitative system properties which are based on *formal methods*. In the field of Software Engineering, on the other hand, a lot of effort has been put into the establishment of *software development processes* and *specification and description standards* which enable practitioners to cope with the problems in the design and implementation of complex software systems. The technology transfer challenge consists of making the formal PE methods accessible by the software engineering community through a seamless integration into the SWE tool-chain. Fig. 1.1 illustrates the situation: The industry-academia gap which hampers the use of formal PE methods in SWE projects is located in the centre. To the left of the gap, on the industrial bank of SWE, practitioners working on a project from a specific problem domain use standardised description techniques, such as the *Unified Modelling Language* (UML, [OMG02b]) or the *Specification and Description Language* SDL [Uni00]. With the help of CASE-tools, which are based on the description standards, engineers capture the various aspects of the planned system. To the right of the gap, on the bank of academia, PE researchers created a multitude of tools for the evaluation of non-functional system properties. These tools take as input a *stochastic model* of the system to be evaluated which is ex-

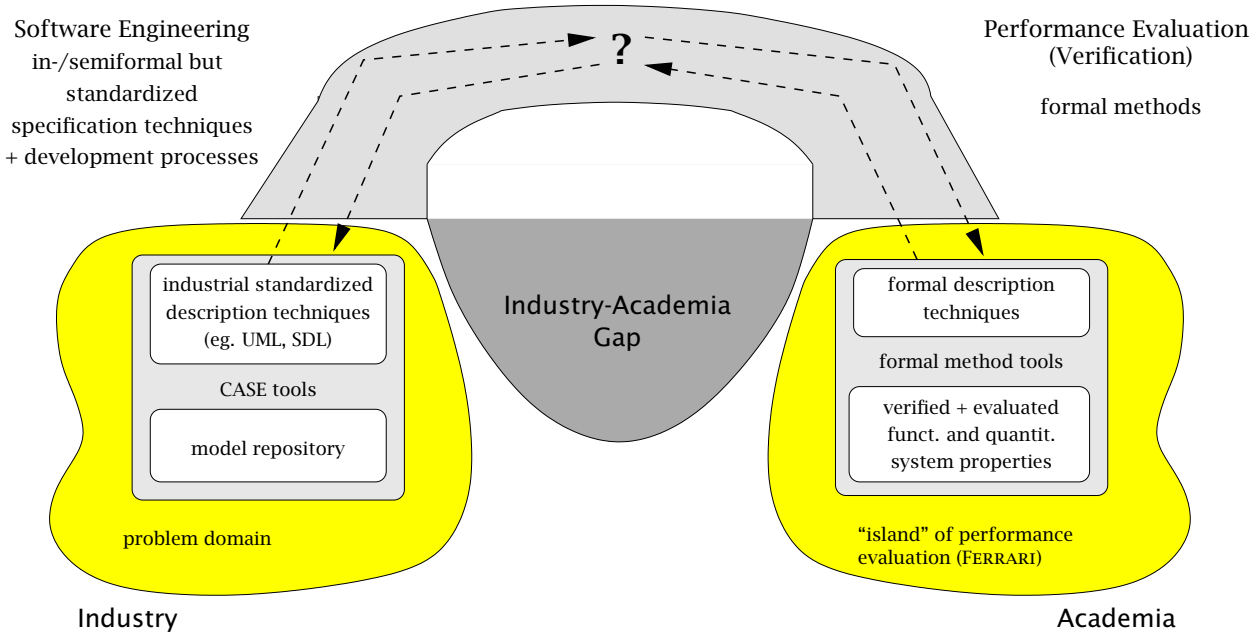

Fig. 1.1: The "Industry-Academia" Gap

pressed by employing one of many non-standardised (semi-)formal description techniques (FDT). The bridge over the industry-academia gap symbolises the desirable integration of PE methods into the industrial SWE process, thus reducing the performance problems in industrial software development processes and at the same-time letting PE researchers escape from their "island".

1.2. Objectives

The goal of this thesis is to contribute to the construction of the technology-transfer bridge depicted in Fig. 1.1. Our main contribution is located on the academic bank and consists of the development of a *Lightweight Formal Method* (LWFM) [JW96] for the evaluation of quantitative system properties. The MOSEL-2 (MOdelling, Specification and Evaluation Language, 2nd revision) LWFM serves as an integration platform for many existent formal methods and solution algorithms. The LWFM approach has its roots the area of formal system verification (see Sect. 2.1.4) and is based on a set of recommendations which have been developed by a group of software engineering researchers in the mid 1990s. They serve as a guideline on how to construct a formal method in such a way that it is easily accessible by practitioners. The MOSEL-2 LWFM consists of a stochastic modelling language and an associated evaluation environment which were developed under the supervision of the author during the last years ([ABBBZ02], [Beu03], [Wüc03], [Wüc04], [WABBB04]). The modelling language is based on its predecessor MOSEL which was created by HELMUT HEROLD as part of his thesis [Her00] during the years 1995-2000. In contrast to HEROLD, who introduced the application of the MOSEL *language* following the straightforward but informal *"explain by example"* approach by presenting numerous, mostly small-sized case studies, we emphasise the formal nature of our *method*: The MOSEL-2 LWFM is based on a sound methodological framework,

which comprises an *ontology* as an explicit specification of the conceptualization that has to be adopted by the modeller. Moreover, a precise definition of the method's formal description technique, including formal syntax, formal semantics and a description of the underlying formal logical system is provided.

In addition to this methodological progress, we focussed on the extension of the expressiveness of the modelling language. The class of real-world phenomena which can be expressed adequately in MOSEL-2 is much larger than the set of systems that can be described by its predecessor. The increased expressiveness of the modelling language demands an extension of the associated evaluation environment used to perform the automated evaluation of the system properties. This is achieved by the integration of the Petri net analysis tool TimeNET 3.0 and the simulation component of the stochastic Petri net package SPNP 6.1, which were recently connected to the MOSEL-2 evaluation environment (see [Beu03], [Wüc03]).

The second objective of this thesis is to describe how the central part of the bridge over the industry-academia gap can be constructed. The basic idea is to map performance enhanced UML models to MOSEL-2 descriptions. Fortunately, preparatory work to facilitate this plan has been carried out on the industrial side: The *"UML Profile for Schedulability, Performance, and Time Specification* [OMG02a] defines a set of notational enhancements of the UML for the purpose of Performance Evaluation.

1.3. Thesis Outline

This thesis is organised as follows: In chapter 2 we collect the central ideas and results from the areas of software engineering and performance evaluation on which the remainder of the thesis is built. Section 2.1 commemorates the emergence of software engineering as an independent area of research in computer science. Thereafter, some of its key concepts, namely software development life-cycle models, the benefit of decomposing complex software systems into modules [Par72a] and the the impact of structured programming [Dij69], [DDH72] on the evolution of programming languages and software engineering are emphasised. Section. 2.1.2 discusses the origin and some key aspects of the standardised modelling language UML, which is regarded as *the* contemporary modelling language used in the software engineering industry. In Sect. 2.1.4 we take a glance at some application areas for methods which are based on a formal system description. The importance of an early evaluation of non-functional system properties is emphasised in Sect. 2.1.3 from an economic viewpoint. In Sect. 2.2 we highlight the origins of performance evaluation and introduce standard definitions for the most important types of non-functional system properties. Three frequently used PE modelling formalisms are presented, namely queueing networks, stochastic Petri nets and stochastic process algebras. Section 2.3 concludes the chapter with an investigation of the evolution of formal models in computer science. This rather elaborate treatment serves the purpose to perceive clearly, that two different stochastic modelling paradigms exist today which inherited not only the notational style but also the analysis goals and methodology from their non-stochastic ancestors. This insight has a considerable influence on the development of the core concept of this thesis which is presented in chapter 3.

In Sect. 3.1 the importance of a methodological foundation for a successful method application is pointed out. To base the following considerations on a precise terminology, we define the central concepts related to methods and modelling in the beginning of Sect. 3.2. In particular, we establish the term *lightweight formal method*, which is proposed by JACKSON, WING ET AL. [JW96] as *the* solution to overcome the acceptance problems of formal methods for functional system verification. In Sect. 3.3 we recall the main arguments of the formal methods debate in verification and draw the conclusion that in view of the numerous parallels between functional verification and performance evaluation methodology, the transfer of the lightweight approach to the area of performance evaluation will likewise be suited to promote the early integration of PE in the software development process.

In Sect. 3.4 we elaborate on the architecture of the MOSEL-2 lightweight formal method and exemplify its application by means of a small-sized running example. The real-world, the syntactic and semantic domains and the stochastic process foundation constitute the four layers of the MOSEL-2 LWFM. Starting on the real-world layer with an informal description of the example system in English prose, we move on to a formal model of the system using the formal description technique of MOSEL-2. The semantic mapping of the formal model into the semantic domain of labelled transition systems [Kel76] is executed automatically by the MOSEL-2 evaluation environment. Depending on the stochastic information included in the formal system description, the resulting semantic model can be transformed into a stochastic process which belongs to one of three different stochastic process classes.

Chapter 4 contains a detailed description of the concrete MOSEL-2 syntax. We differentiate between core language constructs, which determine the expressiveness of MOSEL-2 and additional elements of the language, which aim at increasing the modelling convenience. The structure of a MOSEL-2 model as a fixed sequence of constituting parts and the core language constructs used therein are explained. The most important language construct with respect to increased modelling convenience is the *loop* construct which allows identical components of the system to be "folded" into one and thus considerably reducing the size of the model description.

Chapter 5 is devoted to the description of the MOSEL-2 evaluation environment. In contrast to the stratified logical view of the MOSEL-2 LWFM architecture, we now focus on the actual implementation of the transformations between the three lower layers of the MOSEL-2 LWFM. The *integrative* philosophy of the MOSEL-2 evaluation environment is emphasised by showing how existing stochastic modelling tools have been connected in our method.

In chapter 6 we report on some description techniques and tools for performance evaluation and formal system verification which are insofar related to our approach as that they are either possible candidates for a future integration into the MOSEL-2 evaluation environment or they themselves follow an *integrative* approach and therefore suggest a comparison to MOSEL-2.

In order to give a proof of the concepts introduced in this thesis, we demonstrate the applicability of the formal MOSEL-2 description technique and the power of the evaluation environment by giving a detailed performance evaluation study of a wireless LAN network in chapter 7.

In chapter 8 we give a retrospective view on the main achievements of this dissertation. We summarise the components of the technology-transfer bridge that spans the industrial-academia gap which were developed throughout the preceding chapters. We emphasise that the adoption of the lightweight formal method approach by the performance evaluation community is almost inevitable, if this community is seriously interested in a technology transfer of their methods into the engineering practice. We argue, that the tendency to heavyweight approaches, which seems to be a current trend followed by many academic performance evaluation research groups, will undoubtedly produce a multitude of interesting foundational results but on the other hand are likely to deepen the industry-academia gap of formal methods adoption. We conclude the dissertation with some remarks on possible directions for further research.

The appendix contains a presentation of the MOSEL-2 syntax in extended Backus-Naur form (App. A) and a list of command line options of the MOSEL-2 evaluation environment (App. B).

2. Software Engineering meets Performance Evaluation

All models are wrong, but some are useful.

GEORGE E.P. BOX *[Box79]*

In this chapter we put together some important concepts from the areas of software engineering and performance evaluation which we consider to be relevant with respect to the objectives of this thesis. Section 2.1 reports on the origins and core concepts of software engineering and presents the basic ideas of the standardised industrial modelling language UML. The necessity to verify functional and non-functional, quantitative system requirements early in the development process is motivated from an economic viewpoint. Furthermore, the potential application areas for formal verification methods are examined including a brief explanation of some standard techniques. Section 2.2 introduces performance evaluation as a model-based verification method for quantitative system requirements: The three classes of non-functional system properties which can be subsumed under the general term performance are defined in sect. 2.2.1. The process of providing a system description which includes the definition of the desired performance properties is initiated by the modeller, who generates a formal system and requirements specification using one of the *high-level* stochastic modelling formalisms presented in sect. 2.2.2. We emphasise that the three presented formalisms differ in their expressiveness and are based on two entirely different modelling *paradigms*. The historical background of the developments in theoretical computer science that led to the different stochastic modelling paradigms is highlighted in section 2.3. Based on this investigation we conclude how a formal description technique for the generation of stochastic models should best be designed in order to support the goal of PE method integration in the software development process.

2.1. SWE – Standardised Development Processes and Description Techniques

2.1.1. Origins and Concepts of Software Engineering

As a reaction to the software crisis a new area of research was initiated in 1968 under the title *software engineering*[1] (SE) [NR68], when a broad international cross-section of participants from industry, government and academia met in Garmisch-Partenkirchen, south

[1] The triggering event which lead the NATO Science Committee to sponsor this conference were the severe problems in IBM's OS/360 development project.

Germany, to spot the causes for unreliable and faulty system software and to discuss possible solutions to the identified problems. A year later, a follow-up conference entitled *software engineering techniques* [BR69] took place in Rome, Italy.

One of the most important insights gained during the Garmisch meeting was that the implementation of a complex software system should be separated from its design. The development of a software system should be carried out in several phases and in order to manage complexity, larger systems should be partitioned into separate modules. During the last decades, several models for structuring the system development process — also called *software development life-cycle* (SDLC) models — have been proposed and applied in practice. Notable examples are:

- WIN ROYCE's *waterfall model* [Roy70] (1970) which was the first attempt to systemise the software development process,
- BARRY BOEHM's *spiral model* [Boe88] (1988) views the software development process as interactive-incremental task which can be visualised as a spiral,
- the *V-model* [GMo97] as the German industrial standard for SE-projects commissioned by public authorities (1992),
- *Extreme Programming* [Bec99] and the *Crystal Clear Process* [Coc04] as representatives of the family of *agile* development processes [Man01],
- IBM/Rational's *Rational Unified Process* (RUP) [Kru00] which is strongly connected to the *Unified Modeling Language* (see sect. 2.1.2).

With the exception of the agile development processes in which the separation and documentation of the several phases is not so strict, the other traditional life-cycle models, although quite different in detail, contain more or less the following phases:

- *requirements engineering*, find out which system has to be built, see NUSEIBEH and EASTERBROOK [NE00] for an overview of current practices and trends in this area,
- *conceptual design*, determine the overall system structure/architecture and desired behaviour of the system,
- *detailed design*, create a refined description for each system module using, e.g. the technique of *stepwise refinement* [Wir71][Mor87],
- *implementation*, transform the module descriptions into executable code written in a suitable programming language,
- *module test*, test each module on a stand-alone basis,
- *system test*, test all system modules together [Whi00].

The application of the MOSEL-2 method is linked to the requirements engineering and the conceptual design phases, since the level of detailedness needed at that stage can be captured with MOSEL-2 and moreover, the system performance evaluation should take place early in the system development life-cycle (see sect. 2.1.3).

The 1969 Rome conference centred around the question on how to make the development of computer applications and programs more engineering-like. The idea of standardised parts that could be reused was raised. The concept of *information hiding* and decomposing larger software systems into modules was proposed later by PARNAS in [Par72a], [Par72b]. In the same year O.-J. DAHL, E.W. DIJKSTRA and C.A.R. HOARE published their monograph entitled *"Structured Programming"* [DDH72], in which they described the key properties that programming languages should possess in order to facilitate the development of complex software systems. The monograph had a large influence on how programming languages — as the primary tool in software development — were designed thereafter.

A modern survey on the fundamental ideas of contemporary software engineering is given in the textbook of C. GHEZZI ET AL. [GJM02]. In the following section, we examine the origins and basic ideas of the most widely used design notation in modern software engineering.

2.1.2. The Unified Modelling Language UML

The proliferation of the *object-oriented programming* (OOP) paradigm [Nyg86] during the 1980s is considered as a major improvement of the quality of software systems. The application of OOP languages like C++ [Str88] in SE projects promises to increase the re-usability, adaptability, and maintainability of the created software. Many software engineers perceived that in order to avoid a disturbing shift of paradigm in-between the phases of the software development process, object-orientation should also be used in the early analysis and design phases. In order to realise this idea, an abundance of rivalling OO-based modelling techniques were developed in the beginning of the 1990s. To put an end to this "method-war" GRADY BOOCH, IVAR JACOBSEN and JAMES RUMBAUGH joined forces and integrated their OO-methods to form a standardised OO modelling language based on an agreed terminology. Their Unified Modelling Language UML [BJR96] was completed by incorporating ideas from other sources, most notably the *statecharts* notation of DAVID HAREL [Har87]. The proliferation of UML within the software industry is promoted by the *Object Management Group* (OMG), a consortium of industrial companies and academic institutions, which acts as the standardisation body of UML and other computer industry standards, such as the *Common Object Request Broker Architecture* (CORBA). UML version 1.1 was adopted as an OMG *industrial standard* in 1997 and is scheduled to be adopted as the ISO/IEC standard No. 19501. Since then it has widespread among industry and academia, and can now be regarded as the "de facto" OO-modelling standard in industrial software engineering.

According to [Obj97] the UML is

> *... a standard language for specifying, visualising, constructing, and documenting the artifacts of software systems, as well as for business modelling and other non-software systems.*

The UML is a collection of various diagrammatic notations (see Fig. 2.1) which facilitate the generation of models of a system based on different views. Each view emphasises a certain structural or behavioural aspect of the system, such as possible user-system interactions (use-case diagram) or distribution of software components on processing nodes

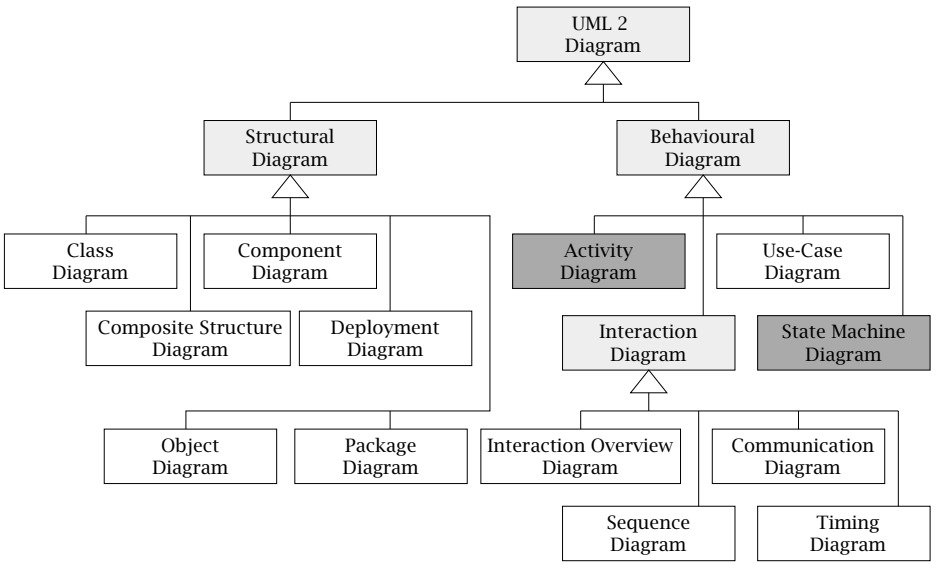

Fig. 2.1: UML 2.0 diagram types

(deployment diagram). The views are projections of a complete model, the well-formedness of the whole model, i.e. the consistency of the different system views in the *model repository* is maintained by the UML *metamodel*. The UML diagram types that are the most interesting for our purposes are the *activity diagram* and the *state machine diagram*, since they can be used to describe the behaviour of a large class of discrete event systems.

An important point to notice is that the UML is not intended to be used exclusively by software engineers and programmers. UML use-case diagrams for example play a central rôle during the requirements analysis phase, where they serve as a highly intuitive communication platform on which software engineers and their stakeholders, like clients, future system users or application domain experts — people which usually don't have a strong background in programming or formal methods — can come to an agreement about the requirements that the envisioned system has to fulfil.

In [HR04] D. HAREL and B. RUMPE recently pointed out, that in the definition of modelling languages the term *semantics* is surrounded by much confusion and that this is all the more true in the context of UML. Despite of the recent introduction of *action semantics* as a reference semantics for UML version 1.5, UML has to be regarded as a semiformal language, which means that it has a precisely defined syntax but imprecise semantics. This imprecision is intentional, since the availability of *"semantic variation points"* is closely related to another valuable feature of UML, which was introduced in version UML 1.3: builtin *extension mechanisms* — called *stereotypes* and *tagged values* — allow the adaption of UML diagrams to specific modelling situations, which can not be covered by the standard diagrams. The advantage of using the builtin extension mechanisms is that although existing UML concepts are changed by specialisation, the resulting diagrams stay conform to standard UML and therefore compatible to UML-based CASE tools. An alternative to the use of builtin extension mechanisms is a *heavyweight* extension by changing the UML-metamodel using OMGs *Meta Object Facility* (MOF). Since this is a difficult and error-prone task, it should be avoided if the goal can be reached by creating a UML *profile* using the builtin extension

mechanisms. Another disadvantage of the heavyweight extension is, that it defines another version of UML, and therefore the compatibility with existing UML tools would get lost. An interesting profile which is related to performance evaluation issues, is the UML Profile for Schedulability, Performance, and Time Specification [OMG02a]. It includes the *performance sub-profile* which defines how UML behavioural diagram types can be enhanced by performance evaluation related notations.

2.1.3. Early System Verification — An Economic Motivation

So far we focussed mainly on the *synthesis* and description oriented activities in software engineering, which were developed to facilitate the *implementation* of complex software systems. Although it is indisputable that the application of concepts like life-cycle models, structured programming, modularisation, and information hiding in a software engineering project contribute vitally to an increased quality of the final product, no guarantee can be given that no system failures and other unwanted behaviour will be encountered during the operation of the software system. In order to ensure that the delivered product will contain as few errors as possible, most development processes contain several test phases during which the — probably unfinished — system is executed in order to find errors. The drawback of this approach is pointed out by DIJKSTRA in [Dij69] where he remarks that

> *Program testing can be used to show the presence of bugs, but never to show their absence!*

This means that although extensive testing of the system is carried out, there is no guarantee that even all of the most critical errors are detected during the test phases. In order to overcome the difficulties associated with testing, the software development process should be augmented by *analysis* oriented activities which aim at *proving* that the final software system will behave as intended by the developers. According to BOEHM [Boe84], the term *verification* is used to designate checking that a software system conforms to its specification, whereas the term *validation* is used to designate checking that the specification captures the actual needs (or the expectations) of its customers. In the following, we give an economic motivation why the analysis oriented activities for system *verification* should be executed *early* in the system development process.

The consideration of economic issues is important in all areas of engineering. In her essay on prospects for an engineering discipline of software [Sha90] M. SHAW states:

> *Engineering is not just about solving problems; it is about solving problems with economical use of all resources, including money.*

In order to provide the reader with concrete figures about the financial consequences which may arise from ignoring the necessity of verification activities in the development process, we quote from a study conducted by LIGGESMEYER et al. [LRRA98]. They interviewed several German software engineering companies about the introduction of errors into their software development projects and revealed that only about 50% of all faults which showed up during the operational phase can be traced back to the programming phase, i.e. the phase where actual code is generated. The conceptual design phase also significantly contributes

to the introduction of errors. An extreme example is mentioned by LUTZ [Lut92] who reports that 194 of the 197 faults that were characterised as the cause of catastrophic failure during the integration testing of the Voyager and Galileo spacecraft were traced back to a problem in the specifications and only three were coding errors. The solid and dotted lines in the lower diagram of Fig. 2.2 illustrate the progression of introduced and detected errors quantitatively. As shown in the upper part of Fig. 2.2 the percentage of fulfilled functional system requirements drops from approx. 80% to 20% after the design tests are performed, the loop-back indicates the return to the system design phase, where the insights gained during the tests are incorporated into the system specification and implementation. A second, smaller loop-back can be observed during the system test phase where again some errors are found which have to be corrected in the system implementation.

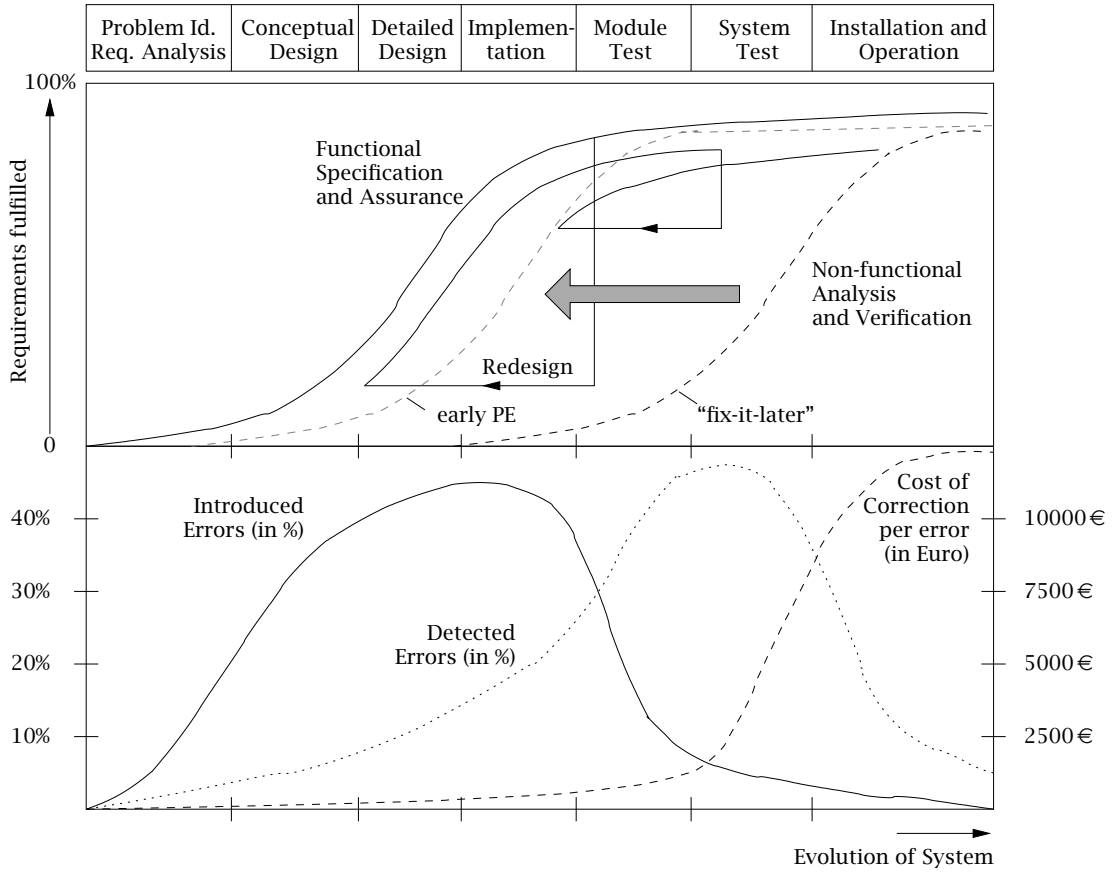

Fig. 2.2: Fulfilled requirements during system evolution (upper plot); introduced and corrected errors and the cost of error correction (lower plot)

The economic consequences become evident if we take into account that the cost of bug-fixing increases dramatically during the system development process: According to MÖLLER [Möl96] the cost of removing an error increases from 250 € during analysis, design and programming to over 1000 € during module test. An error detected during system test costs 3000 €, whereas an error corrected during the late installation and operation phases produces a cost of 12500 € on the average. This effect is illustrated by the dashed line in the lower plot of Fig. 2.2. In order to reduce the cost caused by the introduction and removal of software bugs most efficiently, the verification process should thus begin *early*

in the development process, namely during the conceptual design phase when the curve of introduced errors starts to increase.

The verification of non-functional, performance-related system properties deserves separate treatment. In the introduction we already mentioned the *"fix-it-later"* mentality [Smi03] most software engineers show towards performance related issues. Recently, HERZOG [Her03] reports about the current state of performance evaluation in system design. He points out that despite of many success stories where system designers, performance specialists and software engineers cooperated efficiently, the ignorance of performance aspects during the system development process is still very common. This phenomenon is illustrated by the rightmost dashed line in the upper plot of Fig. 2.2.

> *Unfortunately, however, very often we can see a dangerous lack of cooperation: It is common practice to fully design, implement and functionality test hardware/software systems before an attempt is made to determine their performance characteristics. Major redesigns of software and/or hardware are frequently the undesired consequence. And dramatic examples with enormous financial implications and delivery delays are known.*

On the other hand, almost all performance problems can be traced back to wrong design decisions, i.e. to an early development phase where the cost of fixing an error is relatively low. Thus, switching from *fix-it-later* to *early performance evaluation* as indicated by the grey arrow in Fig. 2.2 bears the highest potential of reducing the overall development cost in a software engineering project.

Recapitulating the substance of what we elaborated above, we identified that the early verification of functional *and* non-functional system properties is an important activity during the system development process. In the next section we turn towards the methods which were developed to assist the software engineers in their verification and validation tasks.

2.1.4. Requirements Verification using Formal Methods

In the following paragraph we present some of the most important applications for formal methods in the software development process. As a precise definition of the term *formal method* will be given in chapter 3.2, we use it here in an intuitive way as a designation for a set of model and tool based techniques which help the developers to get predictive answers to questions concerning crucial properties of the system under construction. Although formal methods may be used also in the later stages of the system life cycle (see e.g. DALIBOR [Dal01] or TRETMANS [TB03]), we restrict our considerations to the area of automatic verification of system requirements during the early phases of the development process. In the following, we focus on the verification of various kinds of system properties which are based on a formal specification of the system and the relevant requirements.

Various questions that can be posed about specific system properties give rise to different types of requirements that have to be verified. Furthermore, various *notions of time* are associated with the different requirement types. We identify the following classes of questions that can be investigated by formal verification methods:

- Is the system *correct*, i.e. does the system or program indeed implement the function that it is expected to do? The systems that are verified for pure functional properties belong to the class of sequential and terminating programs written in an imperative programming language. In order to verify functional requirements for these systems, no timing information needs to be captured. The question of program correctness is the oldest application domain for formal verification methods and reaches back to the late 1960ies, where the verification methods of FLOYD-HOARE logic [Flo67], [Hoa69], MANNA-PNUELI [MP69] propositional logic or DIJKSTRA's weakest precondition reasoning [Dij76] were developed.
- Is the system *safe*, i.e. is it sure that "something bad can never happen" [OG76], [OL82]? Is it *live*, in the sense of "something good will eventually happen" [AS85]? These are questions which are of interest in the context of infinitely running concurrent systems, especially communication systems. Another question is to ask about *fairness*, i.e. do all processes which belong to the system and want to do something good will eventually get the chance to do so? The evolution of a system is typically interpreted as a sequence of steps (execution paths) which are temporally ordered, but no quantitative information about the durations between two steps is included in the requirements specification.
- Questions about *schedulability* and *timeliness* typically arise in the context of real-time system safety [Bat98]. In order to verify real-time safety requirements, an inclusion of absolute "wall-clock" time information in the formal model is necessary. The evolution of the system is regarded to be triggered by the occurrence of *events* which model the completion of system activities to which deterministic durations are associated. The terms *deterministic timing* or *absolute timing* are common.
- Questions about the *performance*, *reliability* or *dependability* of a projected system are typically expressed in terms of mean values or probabilities (see sect. 2.2.1), i.e. requirements which are afflicted with some uncertainty. Two notions of time are used in the verification of this type of quantitative, non-functional properties. In the first approach, the evolution of the system is interpreted as a sequence of states where transitions to the next state occur at multiples of a fixed time unit. Possible alternatives for the next state are resolved according to discrete probability distributions. This notion of time for *synchronously* evolving systems is therefore mostly referred to as *probabilistic timing*. In the alternative approach, the system may change its state at any moment in time and the *durations* between two consecutive state changes are determined by continuous probability distributions. This way of including time in the description of *asynchronously* evolving systems is usually entitled *stochastic timing*.

The steps that are carried out during the formal methods application can be summarised as follows (see Fig. 2.3):

1. Having in mind the categorisation that we elaborated before, the system developer determines the problem domain and the goal that he wants to reach by applying the formal method. The particular formal method, i.e. a formal description technique which is supported by an appropriate tool, has to be selected.

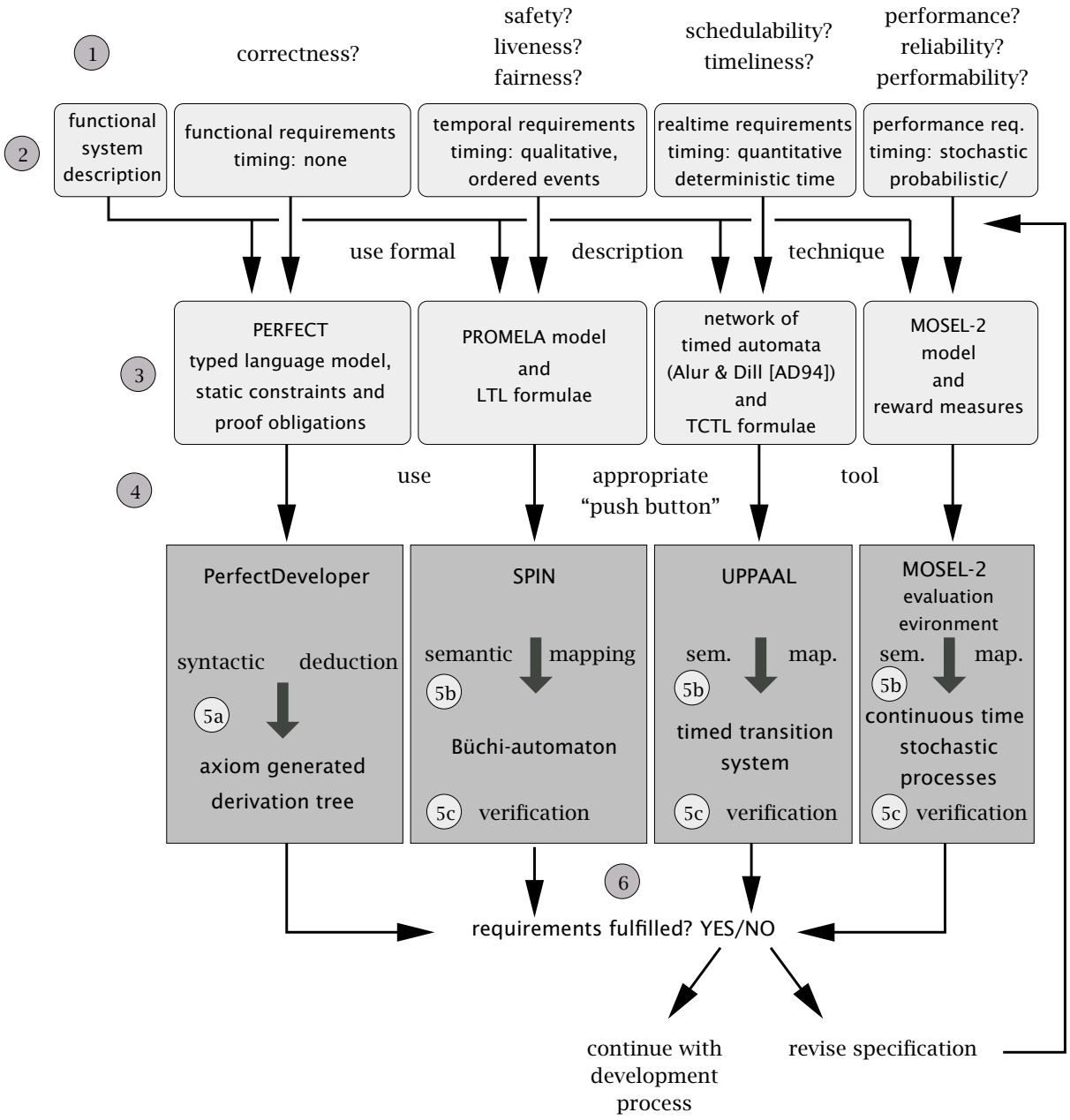

Fig. 2.3: Potential application areas of formal methods for requirements verification and example tools

2. Collect information about the desired system structure, functional behaviour and the requirements that the system has to fulfil. This work is carried out during the requirements engineering phase [NE00] of the software development process. Usually, informal (e.g. natural language descriptions) or semi-formal (e.g. UML use case diagrams) notations are employed in the generation of the requirements specification. Note that the creation of a requirements specification is mandatory for every larger industrial software development project and has to be carried out anyway, regardless of the use of formal methods.

3. The application of the formal method takes place during the conceptual design phase of the system life cycle. Based on the requirements specification, the system developer creates a formal system description (a model) including a precisely defined representation of the requirements which have to be verified. During the generation of the formal model the system developer may abstract away from details which are described in the requirements specification but which are considered to be irrelevant for the chosen purpose of verification or which cannot be captured by the formal description technique.

4. The automatic verification of the requirements against the formal model is initiated by invoking the analysis tool. For tools which perform the verification without further user interaction, the term *"push button"*-analysis [FDR98] is commonly used.

5. Two different approaches for performing the automated verification of within a tool exist:

   a) Tools as PerfectDeveloper [CC04], which belongs to the class of *automated program correctness verifiers* execute on the syntactic level and derive, i.e. prove the validity of the requirements based on a logical calculus.

   b) In tools like SPIN [Hol97], UPPAAL [LPY97] or the MOSEL-2 evaluation environment [ABBBZ02], the system description is mapped on a semantic model, which can be represented by some kind of relational structure, as e.g. a Büchi automaton, a timed transition system or a continuous time stochastic process.

   c) The automated verification of the requirements is performed on the generated semantic object. Depending on the type of requirements to be verified, and the structure of the semantic object, different verification techniques are applied.

6. The results of the verification are presented in textual and/or graphical notation. If the requirements are not fulfilled, the formal model has to be revised and the verification procedure starts anew.

In many cases formal methods exclusively perform the analytic task of requirements verification. Some formal methods, however, include also a synthesis component, i.e. during the verification process either system code is generated which is ready to compile or at least code fragments which can be used as a "launching pad" for the system developers in the implementation phase. Special application areas for formal methods are the development of self-optimising or self-reconfiguring systems which contain a decision model as a part of

the formal model that can be mapped onto an underlying (Markov) decision processes (see e.g. DE MEER [dM92]). Another example of a formal method which comprises modelling, verification and synthesis are the methods used in control theory, e.g. methods based on the *supervisory control theory* of RAMADGE and WONHAM [RW87, RW89].

Two frequently used techniques for requirements verification are *model-checking* and *program construction*: Model checking (see KATOEN [Kat99]) is a fully automated analysis technique which can be used for the verification of temporal and real-time requirements. The method requires as input the formal requirements specification given as a *temporal logic* (for temporal requirements) or a *real time temporal logic* (for real time requirements) *formula* ϕ and a formal system description $\mathcal{D}$. The model checker first generates a semantic model for $\mathcal{D}$ as a relational structure which represents the complete state space of $\mathcal{D}$ and then checks whether the requirements ϕ are valid in every state. If ϕ is valid in all states, the requirements are verified. The big advantage of the model-checking approach is, that if the algorithm finds a state in which ϕ is invalid, the path leading to that state can be output (counter-example generation), which provides useful information to the system developer regarding the sequence of events that lead to a violation of the requirements. The disadvantage of model checking is that it is afflicted by the state-space explosion problem, i.e. that the number of global states increases exponentially with the size of the system description $\mathcal{D}$. For complex systems, memory efficient representations of the state space based on *(ordered) binary decision diagrams* (O)BDD [Bry92] have been developed. In Fig. 2.3 two tools based on model checking techniques are referenced. SPIN (Simple Promela INterpreter) [Hol97] is a tool for checking temporal requirements specified as LTL (Linear Time temporal Logic) [Pnu77] formulae on a system description given as a PROMELA (PROcess MEta LAnguage) program. UPPAAL [LPY97] is a model checking tool for the verification of real-time requirements. The system is specified graphically as a network (a parallel composition) of *timed automata* [AD94], the real-time requirements as TCTL (Timed Computational Tree Logic) formulae [AD94].

Interactive, semiautomatic program construction and automatic theorem proving (see Sect. 3.2.1) are alternative techniques to model-checking. For a modern introduction to the application of theorem proving for verification purposes, we refer the reader to the survey of RUSHBY [Rus01]. Fig. 2.3 refers to the OO-based program construction and proving tool PerfectDeveloper [CC04] which facilitates the generation of C++ programs for which functional properties can be *proved* automatically.

2.2. Performance Evaluation — Stochastic Modelling and Analysis Techniques

As pointed out in the previous section, the early verification of functional *and* non-functional system requirements can contribute largely to the development of correct software systems which can be delivered within the agreed time and without exceeding the estimated budget. We will now concentrate on the fourth column of Fig. 2.3, questions about performance, reliability and performability. We first give definitions for some important non-functional system properties, i.e. characteristics or attributes that a system or component must ex-

hibit while performing its functions, and at the formal methods which are available for their verification. For the sake of convenience, we subsume these properties under the simplifying term “performance” when we refer to non-functional system properties as a whole. Since the underlying principle of all the presented methods is to *evaluate* the performance properties by somehow “executing” a low-level stochastic model derived from a high-level system description, the term *performance evaluation* is commonly used instead of performance verification.

2.2.1. Non-Functional System Properties

The classes of non-functional system properties that are in the focus of the method developed in this thesis can be grouped into three categories:

- Performance properties;
- Reliability and availability properties;
- Dependability and performability properties;

All these properties refer to quantitative descriptions of the *service* that is delivered by a system, i.e. its behaviour as it is perceived by its user(s) during the operation. A commonality of all the non-functional properties is that they are characterised by values which are afflicted with some probabilistic or stochastic uncertainty. Usually probabilities and mean values are sufficient, but sometimes values for higher moments or complete discrete probability distributions are requested. The categorisation above is derived from the amount of additional information which has to be included in the system description. For classical performance properties, an ideal faultless system behaviour is usually assumed. Therefore, in the description only the stochastic and probabilistic information for the desired functionality has to be provided. In order to verify reliability or availability properties, the failure and repair behaviour of the system has also to be captured in the system description. Dependability and performability properties are usually considered in the context of *fault tolerant*, especially *gracefully degrading* [Mey78] systems, which are able to offer their service at varying degrees of quality during the operation. Consequently, the information which characterises the different service levels and the transitions between them has to be included in the system description.

Performance properties: They are the oldest targets of performance evaluation and have been estimated already in the pre-software engineering era for non-computing systems like telephone switching centres [Erl17] or patient flows in hospitals [Jac54] using closed-formula descriptions from applied probability theory. Typical properties to be evaluated are the mean throughput of served customers, the mean waiting or response time and the utilisation of the various system resources. In many cases, the performance properties of the system are evaluated in the so-called *steady state*, i.e. after the transient phenomena that are usually observed after the system is switched on have faded away. The IEEE standard glossary of software engineering terminology [IEE90] contains the following definition:

Definition 2.1 ***(Performance)*** *The degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage.*

As an example for a performance requirement of a software comprising system, consider the mean throughput of packets that can be routed through a communication network according to a protocol under defined traffic conditions. The throughput of packets belonging to a certain application which uses the service offered by the communication network should not fall below a certain threshold in order to keep the users of the application satisfied.

Reliability and availability: Requirements of these types can be evaluated quantitatively if the system description contains information about the failure and repair behaviour of the system components. In some cases it is also necessary to specify the conditions under which a new user cannot get access to the service offered by the operational system. The information about the failure behaviour of system components is usually based on heuristics which are reflected in the parameters of probability distributions. Depending on the nature of the component, various kinds of probability distributions can be used, e.g. the Weibull distribution for modelling the infant-mortality and wear-out failure behaviour of hardware components [Nel85]. The IEEE standard glossary of software engineering [IEE90] defines software reliability as:

Definition 2.2 ***(Reliability)*** The *probability* that the software will not cause the failure of the system for a specified time under specified conditions.

More precisely: The reliability is the probability that a system *continuously* delivers its specified service during the interval $[0, \ldots, t)$, provided that it was able to do so at time $t = 0$. In other words

$$R(t) = \mathrm{Prob}(Z > t)$$

where Z is a continuous random variable which characterises the time until the first faulty system behaviour is observed. Closely related to $R(t)$ is the *mean time to failure* (MTTF), which records the mean length of the time interval from the moment $t = 0$ until it becomes unavailable due to a failure. A large MTTF is desirable for a reliable system. If a failed system is considered to be repaired, then the *mean time to repair* (MTTR) expresses the mean length of the interval during which the system is inaccessible due to maintenance. The *mean time between failures* (MTBF) designates the time period between two consecutive occurrences of failures including the repair after the first failure.

System reliability is a measure for the continuity of correct service, whereas availability measures for a system refer to its readiness for correct service, as stated by the following definition from [IEE90]:

Definition 2.3 ***(Availability)*** *Ability of a system to perform its required function at a stated instant or over a stated period of time. It is usually expressed as the availability ratio, i.e. the proportion of time that the service is actually available for use by the Customers within the agreed service hours.*

As it should become clear from the definition above, there exist several forms of availability: *instantaneous* or *point availability* is the probability that the system will be operational at time t. At any given time t the system will be operational if the following conditions are met [Tri02]: The system was continuously operational during the interval $[0, \ldots t)$ with probability $R(t)$ or it worked properly since the last repair at time u $(0 < u < t)$, with probability $\int_0^t R(t-u)m(u)\,\mathrm{d}\,u$, where $m(u)$ is the *renewal density function* of the system. Then the point availability is the summation of these two probabilities:

$$A(t) = R(t) + \int_0^t R(t-u)m(u)\,\mathrm{d}\,u\,.$$

The *average uptime availability* or *mean availability* is the proportion of time during a mission or time period during which the system is considered to be accessible. It represents the mean value of the instantaneous availability function over the period $(0, t]$ and is given by

$$\overline{A(t)} = \frac{1}{t}\int_0^t A(u)\,\mathrm{d}\,u$$

The *steady state availability* is the limit of the instantaneous availability function as time approaches infinity: $A(\infty) = \lim_{t\to\infty} A(t)$. The *inherent availability* A_I is equal to the steady state availability when only the corrective down-time of the system is taken into consideration and can be expressed using the MTTF, MTTR, and MTBF [BCS69]:

$$A_I = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}\ \text{(single component)}, \quad A_I = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}\ \text{(complete system)}$$

Note that reliability and availability are orthogonal system properties: a system which — during a mission time of 100 days — fails on average every two minutes but becomes operational again after a few milliseconds is not very reliable but nevertheless highly available. As an example of a system for which the evaluation of reliability and availability properties are of interest, consider a cellular mobile phone system such as the European GSM network [ETS00]. Connections of users which move from one cell of the network to an adjacent one are "handed over" to the base station of the adjacent cell as soon as the radio contact to the original base station fades out. In most cases the mobile phone user does not notice the handover, however, during times of high load in the adjacent cell it can happen that the connection request of the incoming handover call cannot be fulfilled because no channels are available. In this case the call of the user gets lost during handover, which from the users perspective is interpreted as an unreliable behaviour of the GSM system. Another scenario that occurs in a heavily loaded cell is that a user cannot initiate a new call due to limited channel availability and is blocked by the base station, i.e from the user's view the GSM system is unavailable. Stochastic models and reliability and availability analysis techniques are used to predict the *call loss* and *call block probabilities* during the development of optimised *call admission* and *handover handling* algorithms for modern mobile telecommunication systems, e.g. in Begain et al. [ABAK03], [ABBT00], Lindemann et al. [LLT04], or Mišić and Tam [MT04].

Dependability and performability: These terms and the definitions for them originated from the area of *dependable* and *fault tolerant* computing. The terminology used in this sector of engineering was worked out and is continuously refined by the members of the workgroup 10.4 of the *International Federation for Information Processing* (IFIP WG 4.10). The most recently published article [ALRL04], contains the following definition for dependability:

Definition 2.4 ***(Dependability)*** *The dependability of a computer system is the ability to deliver a service that can justifiably be trusted. The service delivered by a system is its behaviour as it is perceived by its user(s); a user is another system (physical, human) that interacts with the former at the service interface.*

This is a rather general definition which comprises the five attributes availability, reliability, maintainability[2], integrity[3] and safety[4]. Older contributions of the IFIP WG 10.4 members, e.g. [LAK92] or [ALR01] also list *security* as an attribute of dependability, but there is an ongoing trend to treat security and dependability as coequal system properties which can be evaluated using the same model-based methods [NST04]. The term performability was coined by JOHN F. MEYER [Mey78] as a measure to assess a system's ability to perform when performance degrades as a consequence of faults:

Definition 2.5 ***(Performability)*** *The probability that the system reaches an accomplishment level y over a utilisation interval $(0, t)$. That is, the probability that the system does a certain amount of useful work over a mission time t.*

Informally, the performability of a system is its ability to perform taking into account that faults, i.e. breakdowns of some of the system's components, may occur. This measure is especially interesting to be evaluated for *gracefully degrading systems* which according to BEAUDRY [Bea77] form one of the four subclasses of fault-tolerant systems:

- *Massive redundant systems*, execute the same task on each equivalent module and vote on the output.
- *Standby redundant systems*, execute tasks on q active modules. Upon a failure of an active module, the systems attempt to replace the faulty unit with a spare.
- *Hybrid redundant systems* are composed of a massive redundant core with spares to replace failed modules.
- *Gracefully degrading systems* use all modules to execute tasks, i.e., all failure-free modules are active. When a module fails, the system attempts to *reconfigure* to a system with one less module.

Performability properties are frequently defined as *reward measures* which are calculated accumulatively over the system utilisation interval. Reward measures can be defined in the high-level formal model, e.g. in a model based on stochastic activity nets (SAN) [SM00] or

[2] The systems ability to undergo modifications or repairs
[3] The absence of improper system alterations.
[4] A measure for the continuous delivery of service free from occurrences of catastrophic failures.

stochastic reward nets (SRN) [CMT89]. During the generation of the underlying semantic model the high-level reward specifications can be mapped onto the following types of state-space level rewards:

- *state rewards* where a (state-dependent) value is added to the cumulative measure when the system stays in a specific state,
- *impulse rewards* where the cumulative measure is incremented when a specific transition in the state-space level representation is taken,
- *path-based rewards* where a reward value is associated with a specific sequence of states and transitions (a path) and the cumulative reward measure is incremented after the system traversed the path [OS98].

We refer the reader interested in a particular performability related reward measure to chapter 2 of the monograph [BGdMT98], which contains an extensive list of well-established reward measure definitions. Note that the definition of reward measures is also possible in a MOSEL-2 model (see section 4.2.3).

During the last years the verification of performability requirements has been applied successfully in the area of *Quality of Service* (QoS) assessment for modern telecommunication systems [Mey01]. The flexibility of the reward measure mechanism enables the specification and evaluation of various QoS parameters which take into account complex constraints of the system like differentiated services.

2.2.2. Stochastic High-Level Modelling Formalisms

In the following, we introduce the origins and concepts behind three high-level stochastic modelling formalisms which are commonly used in PE today. The term "high-level" indicates that the level of abstraction on which these formalisms reside is appropriate to be used during the conceptual design phase of the software development process.

Queueing Networks were introduced by Rex Raymond Phillip Jackson as a stochastic model of patient flow in hospitals for a study commissioned by the U.K. National Health Service in 1954 [Jac54]. He connected two *queueing systems* or *service centres*, which were studied intensively by applied probability researchers in the beginning of the 1950ies, in a serial or "tandem" configuration. In two contributions from 1951 [Ken51] and 1953 [Ken53] David George Kendall defined a notation for the various types of queueing systems and introduced an analysis method based on the solution of imbedded[5] Markov chains.

The queueing network formalism was generalised by James R. Jackson in 1957 [Jac57] and a follow-up contribution in 1963 [Jac63]. His aim was to model the processing of *jobs* in a *machine shop*. A job enters the shop at a randomly determined machine group and joins a queue. A major improvement in James R. Jackson's QN-formalism was the introduction of *routing* or *branching probabilities*, according to which a job that completed service at one service centre (machine group) either leaves the shop or joins the queue of other

[5] today, the term *embedded* is used more frequently than *imbedded*, which appears in the title of [Ken53].

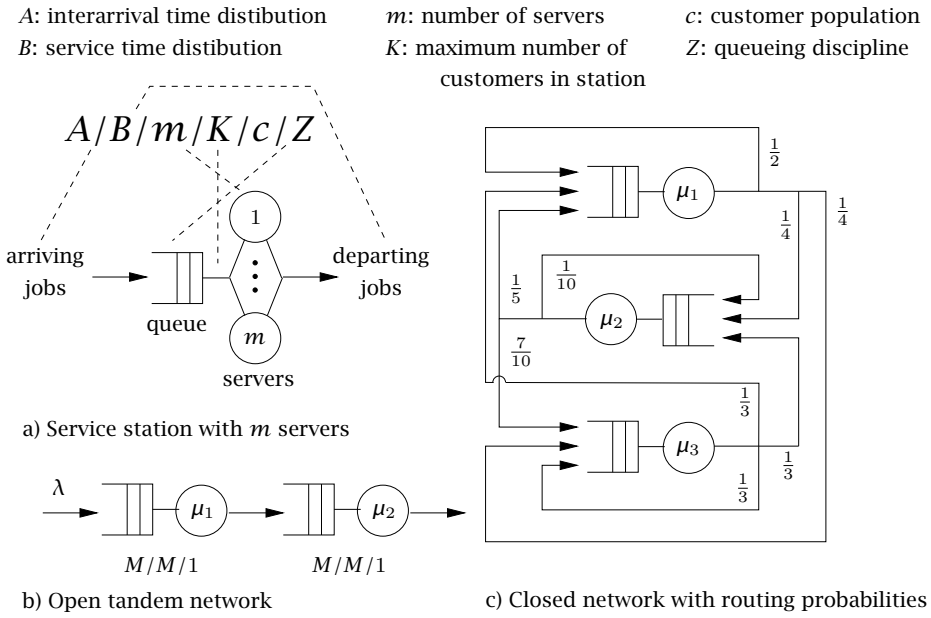

Fig. 2.4: Service station, open tandem queueing network and a closed queueing network with routing probabilities

machine groups with a certain probability. The notion of routing probabilities allows to connect several service centres to form an arbitrary complex network topology. Figure 2.4 a) shows a service station with multiple servers and the KENDALL notation. For the arrival and service time distributions the most commonly used variables are: M for the exponential distribution, G for a general distribution, D denotes a deterministic distribution, and E_k an Erlang distribution with k phases. The queueing discipline Z is usually FCFS (first come first served), other possible disciplines include LCFS (least come first served), RR (round robin), and IS (infinite server). For a detailed explanation of queueing systems and networks, the reader is referred to the monograph [BGdMT98]. Fig. 2.4 b) illustrates a simple *open* tandem queueing network, which consists of two $M/M/1$ service stations. As in this example, the variable λ denotes the rate of the exponential inter-arrival time distribution, whereas μ_i are the rates of the exponentially distributed service time distributions. In Fig. 2.4 c) a *closed* queueing network including the routing probabilities between the service stations is shown. The aim of QN analysis is typically the mean or distribution of the number of jobs at a service station, the mean waiting time for a job at a station, or the mean job throughput.

Shortly after the publication of [Jac63], the QN-formalism was carried over to non-jobshop interpretations, most notably as an abstract model of the first packet switched networks by LEONARD KLEINROCK in 1964 [Kle64]. The success of queueing networks relies to a large extent on the early availability of very efficient analysis algorithms and tools for the class of *product form* networks [BCMP75], such as BUZEN*'s algorithm* [Buz73] or the *mean-value analysis* [RL80]. Although queueing networks have been extended in various directions, e.g. in order to model the forking and synchronisation of jobs [HH79] or finite capacity queues with blocking [Per94], the QN formalism is not suitable for the modelling of arbitrary systems, but specialised to the application area of shared resource systems where individual customers may be considered independent of each other. In a recently published

retrospective on his modelling formalism JAMES R. JACKSON concludes [Jac02]:

> *In every case that I've heard of, the networks-of-queues model provides an abstract and grossly simplified model of messy phenomena, which one hopes has provided a foothold from which more realistic views can be developed. Perhaps the most interesting such use has been that of* LEONARD KLEINROCK *and his associates as a first formal conceptualisation of the systems from which the Internet has descended.*

Stochastic Petri nets are a time-enhanced version of the visual formalism of Petri nets, which were introduced in 1962 by CARL ADAM PETRI to model communication and synchronisation in concurrent systems [Pet62]. A comprehensive introduction into the theory of untimed Petri nets is presented in the survey of TADAO MURATA [Mur89]. Petri nets can be visualised as a bipartite directed graph where the two types of vertices, called *places* and *transitions* are connected by two types of arcs. *Input arcs* are directed from places to transitions, whereas *output arcs* point from transitions to places. Each transition may have several *input places* (connected via input arcs) as well as several *output places* (connected via output arcs). Each place, drawn as a circle, can hold a number of (indistinguishable) tokens, which are illustrated as small black dots inside the place. The distribution of tokens among the places of the Petri net is called the net's *marking*. The Petri net marking represents a *distributed overall state* of the system modelled by the net structure. One possibility to describe the dynamic behaviour of a Petri net is as follows: for a given (initial) marking of the net a transition is called to be *enabled* if each input place of the transition contains as many tokens as there are input arcs between the input place and the transition. Any transition enabled in a given marking may *fire*, i.e. may trigger a change of the net marking which represents an evolution step of the Petri net. If a transition fires, as many tokens are removed from each input place as there are input arcs between the input place and the transition. To each output place, as many token are added as there are output arcs from the transition. The removal and adding of tokens is atomic, a *token flow* from input to output places is induced by the firing of a transition. Figure 2.5 shows a simple Petri net consisting of three places (p_1, p_2, p_3) and three transitions (t_1, t_2, t_3). In the marking shown in the upper part of Fig. 2.5, transitions t_1 and t_2 are enabled concurrently and any of them might fire first. The lower part of Fig. 2.5 shows the resulting marking after t_2 has fired. A possible *firing sequence* in this situation is t_1, t_2, t_2 which leads to a net marking where p_1 and p_3 contain no tokens and p_2 contains three. In this *absorbing* marking no transition is enabled and the evolution of the Petri net has reached a final state. Petri nets which model the behaviour of a system with infinite behaviour (e.g. a communication network) should not contain any absorbing marking.

Untimed Petri nets are typically analysed for functional properties of the systems they represent, e.g absence of deadlocks or boundedness. On the one hand, this can be done statically by checking structural properties of the Petri net which are independent of the marking using linear algebraic methods based on the *incidence matrix* of the net (see [Mur89], p. 551–553). On the other hand, two general views on the dynamic behaviour of a Petri net exist, which lead to two different Petri net semantics: The first one — called the *sequential*

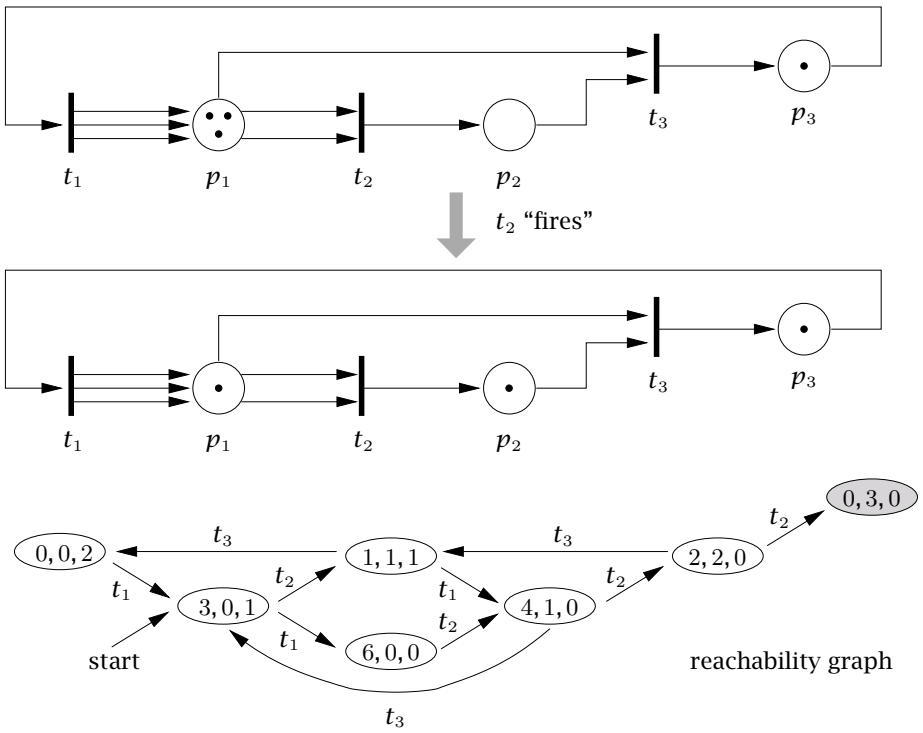

Fig. 2.5: A simple Petri net

or *total order* semantics — looks at the set of *firing sequences* of a net as it is described above. The second one — called the *causal* or *partial order* semantics — looks at the set of partially ordered runs (or processes) of a net. Although C.A. PETRI had designed his network formalism with having a partial order semantics in mind because of its ability to handle true concurrency, the area of Petri net applications is dominated by the sequential semantics because of their easy and straight-away definition. The sequential semantics interpretation of the Petri net dynamics is based on the generation and examination of the *reachability graph* (RG) (see [Mur89], p. 550). The nodes of the RG consist of all net markings (= states) which can be reached from an *initial marking* by exploring all possible firing sequences. The arcs of the RG are labelled with the name of the transition whose firing caused the corresponding state change (see Fig. 2.5).

In order to employ Petri nets for performance evaluation, a notion of time has to be added to the basic formalism. In 1980 respectively 1981 STÉPHANE NATKIN [Nat80] and MICHAEL KARL MOLLOY [Mol81] — independently of each other — introduced stochastic Petri nets (SPN) by attaching *exponential firing rates* to the transitions of the basic Petri net model. The timed transitions of a SPN fire after a delay which is sampled from the exponential distribution determined by the firing rate. If the firing rate of transition t_i is λ_i, its firing time distribution is exponential with mean $\frac{1}{\lambda_i}$. The transitions firing time is measured from the instant the transition is enabled to the instant it actually fires. If more than one transition is enabled in a marking of a SPN usually a *minimum firing time execution policy* is used to determine which transition fires next. With this policy, the transition whose firing time elapses first is the one that fires. Alternatively, a *preselection execution policy* may be used, in which the next transition to fire is chosen according to additionally specified

firing probabilities (see [Cia87]). The analysis of a SPN typically aims at calculating the mean throughput of a transition, the mean number of tokens in a place or the probability of there being no token in a place. Since a SPN is equivalent to an untimed Petri net regarding net structure and token flow, its reachability graph can be generated by applying the same algorithm. The only difference is that the arcs of an SPN RG are additionally labelled with the firing rates of the corresponding transitions. If one abstracts from the transition names and the net markings stored in the RG nodes, the reachability graph is isomorphic to a continuous time Markov chain (CTMC) — a low-level stochastic system description which can be analysed by standard numerical algorithms (see sect. 3.4.4).

Shortly after their introduction, SPN have been extended in several directions. The extensions either aim at enhancing the stochastic expressiveness, or at increasing the modelling convenience of the SPN formalism.

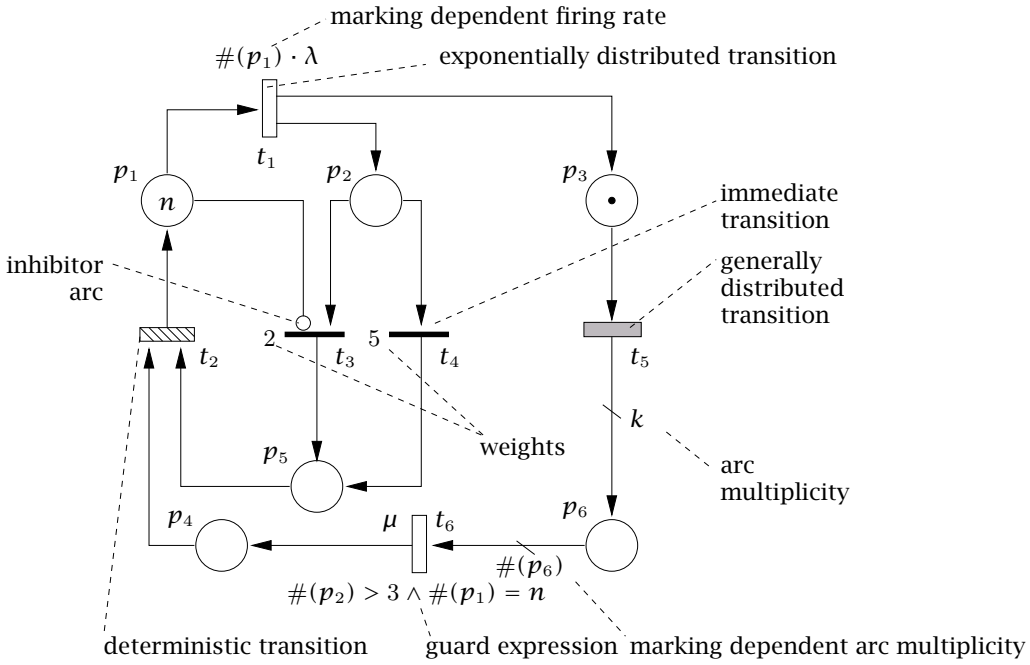

Fig. 2.6: Various notational enhancements for extended stochastic Petri nets

In 1984 Marco Ajmone Marsan et al. introduced generalised stochastic Petri nets (GSPN) which contain *immediate* transitions with zero firing times. In 1993 they redefined the GSPN formalism by attaching a *weight* to each immediate transition [CMBC93] in a GSPN, and thereby introduced the concept of *branching probabilities* — which we already encountered in connection with the QN formalism — to Petri Net modelling. GSPNs increase the modelling convenience but not the expressiveness of SPNs, since SPNs and GSPNs are both mapped to CTMCs as the underlying stochastic model. In 1987 Marco Ajmone Marsan et al. [AMC87] also established deterministic stochastic Petri nets (DSPN) by including transitions with fixed firing times. Extended stochastic Petri nets (ESPN) where the firing times can be sampled from arbitrary probability distributions have been introduced in 1984 by Joanne Bechta Dugan et al. [BDTGN84]. In [GL93] Reinhard German and Christoph Lindemann define extended DSPNs which can be analysed by the *method of supplementary variables* ([Cox55a]). Stochastic Activity Networks (SAN) by Ali Movaghar and John F.

MEYER [MM84] and GIOVANNI CHIOLA's Stochastic Well-Formed Nets [CDFH93] are structural high-level extensions of GSPNs which provide notational elements to capture symmetries in the net structure which can be exploited to create compact representations of the net's state-space. KISHOR S. TRIVEDI and VIDHADYAR G. KULKARNI developed the Fluid Stochastic Petri Nets (FSPN) formalism in which additional *fluid* places contain a "fluid" amount of token mass which flows continuously to other fluid places as long as corresponding continuous transitions are enabled. FSPNs aim at modelling *hybrid systems* which possess a combined discrete/continuous dynamics. Since we restrict ourselves to the modelling and analysis of discrete event systems throughout this thesis, the FSPN formalism is not suited for our purposes.

Fig. 2.6 contains a compendium of notational enhancements which are frequently used in extended SPN models today. As it should become clear from the figure, the inclusion of additional textual syntax annotations on the one hand causes a compactification of the model, but on the other hand spoils the intelligibility of a pure graphical representation.

Stochastic Process Algebras (SPAs) are an alternative high-level stochastic modelling formalism that can be used to specify concurrent systems. Like SPNs are based on untimed Petri nets, stochastic process algebras are a timed extension of untimed process algebras (PA) from which they inherited the basic modelling primitives. An excellent introduction to the theory of process algebras can be found in the report of BAETEN [Bae04]. Readers interested in an elaborate treatment of the theoretical SPA foundations are referred to the survey of HERMANNS, HERZOG and KATOEN in [HHK02]. Modelling with SPAs is based on the notion of processes as abstractions of independent threads of control which — in order to capture the behaviour of a complex concurrent system — are able to interact in a precisely defined way. An algebra is a mathematical structure with a set of elements and a collection of operations on the elements. These operations enjoy algebraic properties such as commutativity, associativity, idempotency and distributivity (see Fig. 2.7). In a process algebra, the atomic elements of behaviour are *actions* and, e.g. *parallel composition* is defined to be a commutative and associative operation on them. Any process algebra *term* which is generated by applying the algebraic laws to the atomic elements can be defined as a process (meta-)variable. Once defined, process variables can themselves be used as operands in the algebraic laws. With this technique, complex systems can be specified from smaller building blocks in a compositional manner. Most process algebras support the concept of *hiding* (see. Fig. 2.7) which provides a convenient abstraction mechanism. The construct "**hide** a **in** P" implies that the action a has to be regarded as an *internal action* of the process term P and is "invisible" from the outside. In particular, a cannot be used for synchronisation if P is engaged in a parallel composition. A central issue in the definition of a particular process algebra is whether there exists a law which connects parallel composition to choice and sequential composition. The inclusion of such an dynamic *expansion law* as it is shown in Fig. 2.8 leads to an *interleaving* interpretation of concurrently executing activities, in which the parallelism can be removed step by step. Process algebras without an expansion law, such as the one presented in [BKLL95], have a *non-interleaving* semantics.

In a stochastic process algebra, time information is introduced by augmenting the actions

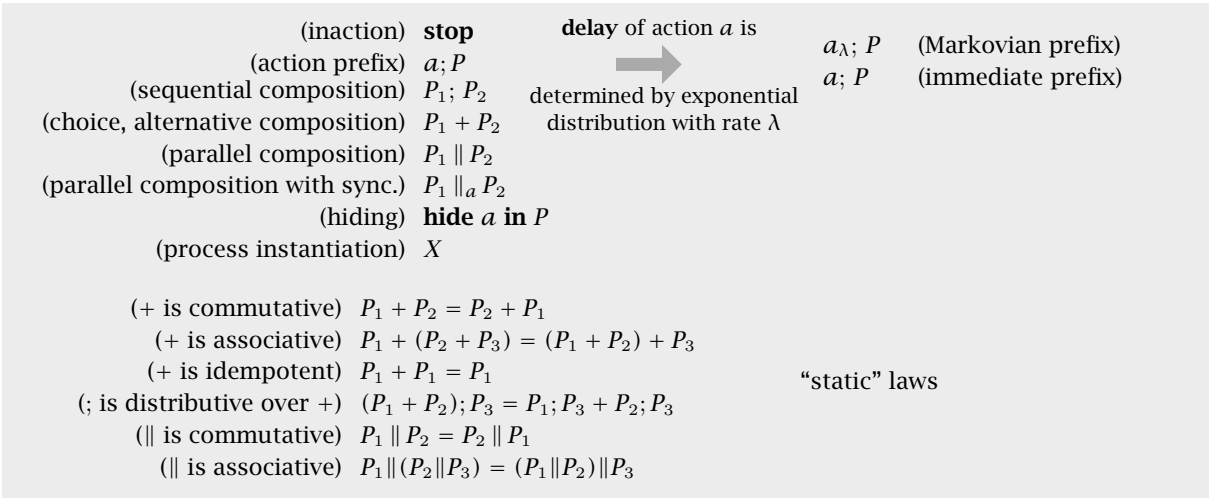

Fig. 2.7: Operations and static laws of a sample stochastic process algebra

with delays sampled from (continuous) probability distributions. The delays in the earlier, so-called "Markovian" SPA implementations TIPP [GHR92], MPA [Buc94], EMPA [BG97] and PEPA [Hil94a] were restricted to be sampled from exponential distributions only. As illustrated in Fig. 2.9 the introduction of exponential delays in SPAs causes some difficulties with respect to the definition of laws for the parallel composition of processes, if they have to synchronise on the execution of an exponentially delayed action.

The designers of a particular SPA have to decide how the delay of the process resulting from a synchronisation of two process components on a delayed action has to be calculated. Various solutions are possible and implemented in the different Markovian SPAs (see Fig. 2.9). In HOLGER HERMANNS algebra of *interactive Markov chains* (IMC) [HR98] the synchronisation problem is skillfully circumvented by strictly separating exponential delays and actions. All the Markovian SPAs mentioned above are equipped with an expansion law (see Fig. 2.8 upper right corner), so that the execution of concurrently evolving processes is interpreted according to the interleaving approach. The underlying semantic state-space level model of a SPA description is generated by the application of inference rules called *structured operational semantics* (SOS) to a set of stochastic process definitions. This method for providing a semantics for an algebraic system description was already developed in 1981 in the context of non-stochastic process algebras by GORDON PLOTKIN [Plo81].

The algorithmic application of the SOS inference rules to a SPA description coincides with the reachability analysis algorithm in a stochastic Petri net tool. The generated *derivation graph* which in the context of process algebras is usually called a *labelled (multi-)transition system* (LTS) is isomorphic to a CTMC if one abstracts away the action names which decorate the transitions of the LTS.

During the last years a couple of non-Markovian stochastic process algebras in which the delays are sampled from arbitrary probability distributions have been developed, e.g. ♠ by D'ARGENIO [D'A99] and the algebra of *interactive generalised semi-Markov processes* (IGSMP) of MARIO BRAVETTI [Bra02]. System descriptions are mapped onto *generalised semi-Markov processes* (GSMPs) which then are usually analysed by discrete event simulation. An elabo-

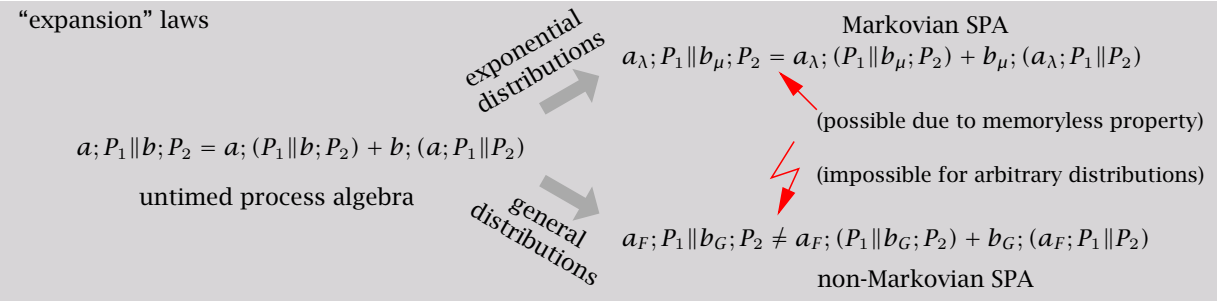

Fig. 2.8: Expansion laws can be defined for untimed and Markovian stochastic process algebras but not for non-Markovian SPA

exponential delays lead to different interpretations about the *nature of synchronization* [Hil94b]

$a; P_1 \|_a (a; P_2 + a; P_3) = a; (P_1 \|_a P_2) + a; (P_1 \|_a P_3)$

$\Rightarrow a_\lambda; P_1 \|_a (a_\mu; P_2 + a_\nu; P_3) = a_{\mathbf{?}}; (P_1 \|_a P_2) + a_{\mathbf{??}}; (P_1 \|_a P_3)$

intuitvely: assume $F_{\mathbf{?}} = \max(F_\lambda, F_\mu)$, which is **not** exponential

TIPP [GHR92]: use product of individual rates, i.e. $\mathbf{?} = \lambda \cdot \mu$
EMPA [BG97]: let one of the communication partners determine the resulting rate, e.g. ?=??=λ
PEPA [Hil94a]: use scaled max. of mean durations, $?=\frac{\lambda \cdot \mu}{\mu + \nu}$ if $\lambda \leq \mu + \nu$, and ?=μ otherwise

Fig. 2.9: Markovian SPA: different solutions to define the resulting rate in case of synchronisation

rate treatment of non-Markovian stochastic process algebras can be found in the contribution of Bravetti and D'Argenio [BD02].

In his survey entitled *Formal Methods for Performance Evaluation* [Her01] Ulrich Herzog, who belongs to the earliest and distinguished promoters of the SPA formalism, expresses the somewhat surprising statement:

> *The main disadvantage of SPA is that they are not completely developed; acceptance is also still low because of their unconventional theoretical foundation.*

We argue that — quite the contrary — the theoretical foundation of SPA is rather conventional, since it re-uses to a large extent methodology, notation, and terminology of the existing algebraic concurrency and process theory. The low acceptance among the engineering practitioners is more likely due to the intrinsic subtleties and semantic variations of the different SPA implementations which deters the practitioners from using them. On the other hand, the inherent *constructiveness*[6] property of the SPA formalism is an attractive feature which supports the formal method expert in his task of specifying and verifying nonfunctional requirements of complex concurrent systems.

The advantages and drawbacks of the QN, SPN and SPA modelling formalisms are discussed against the background of their potential integration in the software development process in the end of the following section.

[6] Compositionality, abstraction, and existence of a calculus which enables the rewriting of process terms.

2.3. The Roots of Stochastic Modelling

The term *paradigm* (from greek παράδειγμα *paradeigma*, meaning pattern, example) is a central concept in the following considerations. The definition of a modelling paradigm is related to the notions of *conceptualization* and *ontology* which are introduced in the beginning of Chapt. 3. In this section it is used in the sense of the philosopher T.S. KUHN, who defines a scientific paradigm as *"...a constellation of concepts, values, perceptions and practices shared by a community which forms a particular vision of reality that is the basis of the way a community organizes itself."* (see [Kuh96]). The importance of choosing the right paradigm is pointed out by E. GÖKTÜRK and M.N. AKKØK in [GA04]: Since the choice of a paradigm for a modelling activity is basically the same as the choice of a conceptualization/communication language, then according to the SAPIR-WHORF thesis [Who56][7], *"it is of utmost importance because choice of language decides to a large degree what we see, how we see and conceptualize what we see"*. AKKØK and GÖKTÜRK conclude that, since software engineering activities like analysis, design and implementation are communication and reasoning-intensive activities, setting out with an unsuitable or ill-defined paradigm would be very unlucky.

In order to get a better understanding on how the different paradigms of stochastic modelling evolved from existing non-stochastic formalism and for which reasons we decided to let MOSEL-2 be based on a particular modelling paradigm, we set out on a brief historical excursus: The diagram in Fig. 2.10 depicts in a nutshell a genealogy of formal models in computer science during the last 70 years. The three leftmost columns list formal description methods categorised by the notational style on which they are based. Each notational style emphasises a different purpose which is pursued by the application of the particular method. From left to right, we distinguish between:

- The *machine oriented* view, where the formal description is based on a diagrammatical representation of an abstract device or automaton. The advantage of this approach is the ability to *visualise* structural and *animate* behavioral aspects of the system within the model. The problem with diagram-based descriptions is that they tend to be unintelligible for complex systems or littered with too many different notational elements.

- The *algorithm oriented* view, where a system is *expressed* as a sequence of instructions which describe its behaviour. The system description resembles a program with keywords selected according to the application domain and constitute a word of a formal language[8] whose syntax can be represented by a production system or formal grammar. In comparison with the machine-oriented view, the advantage of the expression-oriented modelling is that a compact but yet comprehensible description of complex systems is possible if the modelling language is carefully designed.

- The *algebraic-logical* view, in which the emphasis is put on the properties of an algebraic structure which represents the system on a formal level. System descriptions

[7] This thesis of linguistics states that there are certain thoughts of an individual in one language that cannot be understood by those in another language and that the way people think is strongly affected by their native languages.

[8] A detailed survey of the origins of formal languages can be found in GREIBACH [Gre81].

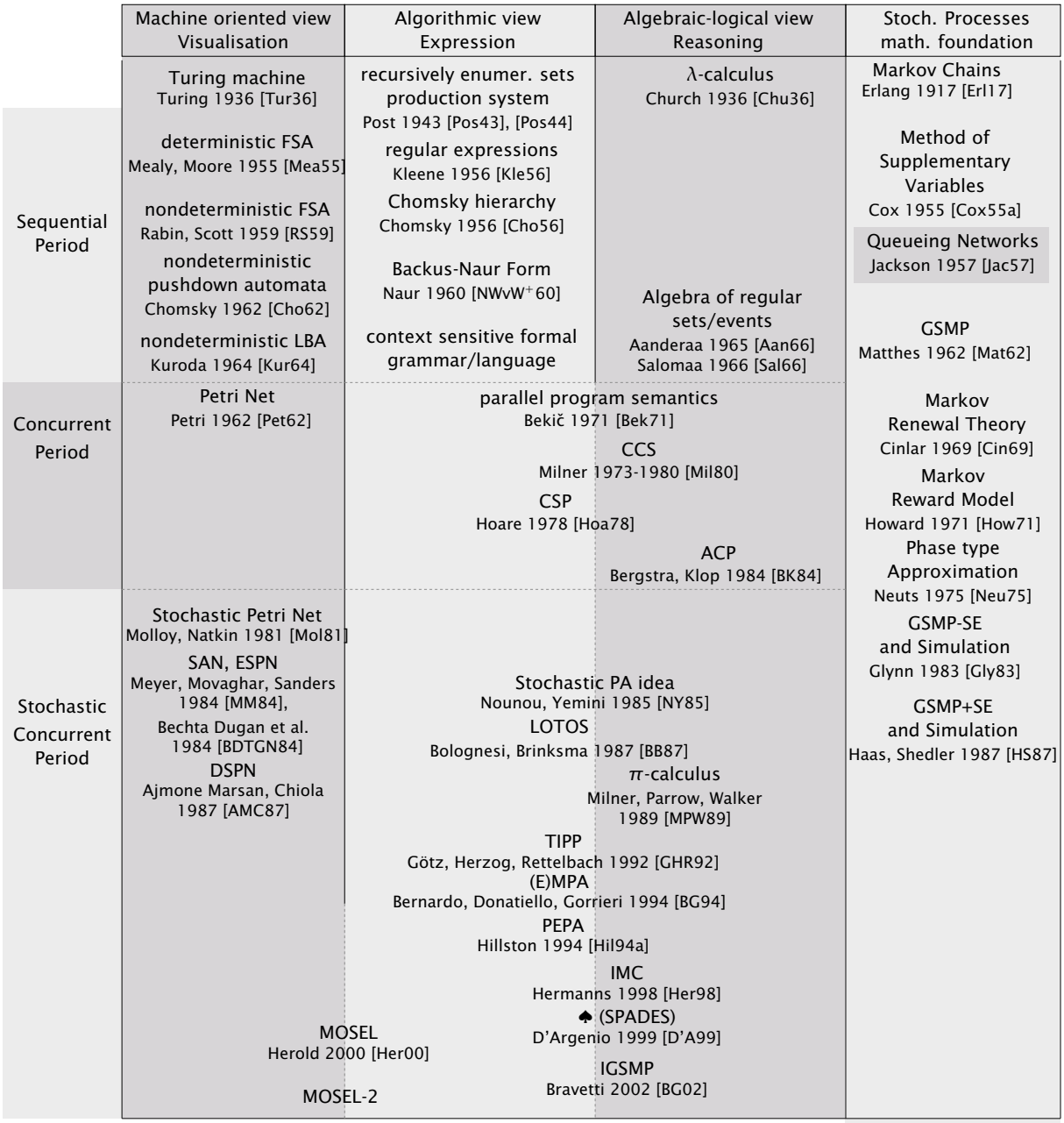

Fig. 2.10: The ancestry of stochastic from sequential and concurrent modelling formalisms

are given as a set of well-formed formulae of a formal language using a vocabulary of mathematical and logical symbols. Most frequently, *first-order* languages, i.e. first-order predicate calculi are used. The advantage of this approach is that the descriptions are directly amenable to automated *deductive* and/or *equational reasoning* which enables the comparison and transformation of different system descriptions as well as the automated syntatic deduction of system properties.

Note that the column in which the algorithmic, language-oriented formalisms are listed in Fig. 2.10 is intentionally placed in-between the other two columns, since formal languages are the most flexible notational style in which either models based on a machine-oriented or an algebraic-logical view can be expressed. The horizontal placement of a particular formalism on or near the border between two columns indicates that the origin of the formalism is either machine-oriented or algebraic-logical, but is expressed in a language for reasons of modelling convenience or improving the intelligibility of the notation.

In order to complete the genealogy with respect to stochastic modelling, the rightmost column in Fig. 2.10 lists some important achievements from the field of applied probability theory which were developed independently of the formalisms in the other columns. Note that queueing networks — although they fit well into the machine-oriented view — are listed here, since they have no ancestral link to the sequential automaton formalisms.

On the vertical time-line we partition the evolution of formal modelling into three periods:

- The *sequential period* (from 1936 onwards), during which the theoretical foundations of computability and computational complexity were developed.
- The *concurrent period* (from 1962 onwards), dominated by research on the nature of concurrent behaviour and communication.
- The *stochastic concurrent period* (from 1981 onwards), during which the methods and notations of the concurrent period were reused and combined with results from applied probability theory for the purpose of quantitative system evaluation.

This classification should not be misinterpreted in the sense that the ideas which heralded a new area let the research on the older theories come to a complete standstill. Sequential automata theory for example is still an active area of research and this also applies to non-stochastic concurrency and process theory.

The research of each period was stimulated by finding answers to different questions. During the sequential period the central problems were *computability*, i.e. which functions can be calculated on a computing device, and *computational complexity*, i.e. what does it mean to say that function f is more difficult to compute than g?[9]. In 1936 two different formalisations of effective computability were published independently by ALONSO CHURCH [Chu36] and ALAN TURING [Tur36]. Both authors used their formalisms to give a negative answer to the *Entscheidungsproblem*[10]. Although the two notions of computability are

[9] for an overview of computational complexity see COOK [Coo83].

[10] the question if there exists a general algorithm which decides for given statements in first-order logic whether they are universally valid or not.

equivalent, they are based on quite different notations. Whereas TURING's formalism is centred around the definition of an abstract computing device — the Turing machine, CHURCH's concept is based on the machine-independent mathematical/algebraical λ-*calculus* [Chu41]. Over the set of computable functions, called λ-expressions an *equivalence relation* is defined[11] that captures the intuition that two λ-expressions denote the same program. In 1943 EMIL L. POST [Pos43] presented another definition of computability based on the generation of *recursively enumerable sets* by rewriting or *production* systems[12], which can be regarded as the ancestor of *formal grammars*. Another model of computability based on grammar-like string-rewriting systems called *normal algorithm* was formulated by A.A. MARKOV JR. in [MJ54]. Since the memory unboundedness of Turing machines was felt to be unrealistic for an abstract representation of a real computing device, researchers concentrated on studying the computational power of various kinds of *finite state automata*. HUFFMANN [Huf54], MEALY [Mea55] and MOORE [Moo56] defined variants of *deterministic* finite automata (DFA) to be used for the formal synthesis of switching circuits. STEPHEN S. KLEENE [Kle56] showed that DFA possess the power to compute the class of *regular* formal languages or *regular expressions*. An essential contribution was the enhancement of finite state automata with *nondeterminism* by RABIN and SCOTT in 1959 [RS59], which facilitated a substantially compacted automata construction in comparison to the expression equivalent DFA. The development of the programming language ALGOL boosted two important concepts related to the definition and translation of the class of *context-free* languages: The *stack-principle* or *pushdown storage* of SAMELSON and BAUER [SB59], which led to the definition of the nondeterministic pushdown-store automaton (NPDA) [Cho62], and the BACKUS-NAUR form (BNF) [NWvW+60] as a convenient formal description of the context free syntax of programming languages. In 1964 the CHOMSKY hierarchy [Cho56] of formal languages and their accepting automata was completed by KURODA who showed that his nondeterministic *linear bounded automata* were able to accept the class of *context-sensitive* languages [Kur64].

The concurrent period of theoretical computer science began in the year 1962 when CARL A. PETRI submitted his dissertation entitled *"Kommunikation mit Automaten"* ("Communication with Automata") [Pet62]. Therein he pointed out a discrepancy in the system view used in theoretical physics and theoretical computer science: Theoretical physics describes systems as a collection of interacting particles (subsystems), without a notion of global clock or simultaneity. Theoretical computer science describes systems as sequential virtual machines going through a temporally ordered sequence of global states. PETRI stated that the existing sequential automata models of computation were inadequate to express communication between concurrently evolving parties. According to ROZENBERG [Roz91], PETRI has deliberately chosen an ambivalent title for his thesis. One interpretation of the title is inspired by TURING's *"imitation game"*[13] where a person communicates with two other "persons" (one of them is in fact a computer) which are hidden behind a curtain. The other interpretation refers to the rôle of computers as the connecting link in the communication between humans. The main result of PETRI's dissertation was the proof that a description

[11] by the so-called α-conversion rule and the β-reduction rule

[12] POST named them *semi-Thue* systems

[13] A thought-experiment which is described in *"Computing Machinery and Intelligence"* [Tur50]. This article is regarded as the seminal paper on the topic of artificial intelligence.

of a universal Turing machine is possible in accordance with the laws of physics (see also [Pet67]). The most influential part of the thesis was the introduction of the diagrammatical, machine-oriented formalism of Petri nets, which remained up to this day the most thoroughly studied concurrent automaton type. Petri net theory is regarded to be the origin of *concurrency theory*.

The transition from the sequential theories of computability and computational complexity to concurrency theory is characterised by the following shifts in the researchers system perception and ways of reasoning:

- The transformational program view has to be replaced by an *interactive* or *reactive* [Pnu86] one. A concurrent system can no longer be regarded merely as a function which maps input to output by the execution of a sequence of calculation steps. The possible interaction between several processes during the execution of a concurrent system spoils the functional character.

- For many concurrent applications the computational aspect is of minor interest, the emphasis is put on the nature of concurrent behaviour, including phenomena which involve synchronisation, e.g. mutual exclusion, resource contention, and preemption.

- Due to the possible spatial distribution of a concurrent system the use of *global variables* has to be abandoned and to be replaced by suitable *communication mechanisms*.

- Nondeterminism is more the rule than an exception for concurrent systems that have to react to stimuli triggered by their *environment* which is usually unpredictable.

- Infinite behaviour has to be regarded as a desirable feature, and not as a bug. Many systems, e.g. a telephone switching system or a computer operating system should *not* terminate. For systems that run indefinitely, the sequence of system states is often much more important than the possibly undesirable termination condition. An important property of many concurrent systems is *cyclicity*, which means that under all circumstances the system will eventually return to its *initial state*.

- The question of *system equivalence* becomes difficult. In the sequential world the equivalence of two Turing computable programs f and g is defined via the equivalence of the λ-expressions for f and g in the λ-calculus[14]. For deterministic DFA, equivalence can be defined and decided via *language equivalence*. Two algebras that allow equational reasoning about DFA are the algebra of regular expressions (Aanderaa [Aan66]) and the algebra of *regular events* (Salomaa [Sal66]). In the concurrent world it turns out that the question of program/system equivalence is much more complicated. Many different types of behavioural equivalences, e.g. *trace equivalence* or *bisimulation* (see Park [Par81]) have been proposed.

- For concurrent *discrete event* systems, two possible interpretations of parallelism exist which are supported by different semantic models [LL91]: *partial order* or *true concurrency* models, i.e. models based on a causality structure on an event-set, reflect

[14]Note that sequential program equivalence is in general undecidable, i.e. there exists no algorithm that decides for two arbitrary λ-expressions, if they are equal, i.e. compute the same function, or not.

directly that a dynamic system consists of a set of events on which a causal relationship on the elements is defined. A preceding event may have an effect on a subsequent one, so that the preceding event is a necessary precondition for the occurrence of the subsequent one. *Total order* concurrency models[15] are defined as a set of states, a set of initial states and a transition relation (the set of transitions). The transition system defines a set of possible or allowed executions which are represented by state-transition sequences beginning in a start state. A state represents the global status of the entire concurrent system, i.e. the combination of all local states of the processes. All local state transitions are considered to be atomic and occur instantaneously at selected time points, so that also the executions of the entire systems form a totally ordered sequence of transitions. Such a global transition sequence is generated by the interleaving of the local transitions which model the execution of the single concurrent processes.

The algebraic-logical branch of the concurrent period started in 1971, when HANS BEKIČ began to study the semantics of *parallel programs*. His report [Bek71] contained a precursory definition of an algebraic law for the parallel composition of processes. The first coherent and comprehensive concurrent processes theory was the *Calculus of Communicating Systems* (CCS), developed by ROBIN MILNER in the years 1973-1980 [Mil80], who was like BEKIČ motivated by tackling the problem to define a semantics for parallel programs. TONY HOARE introduced the *message passing* paradigm of synchronous communication in his language of *Communicating Sequential Processes* (CSP) in 1978 [Hoa78]. The term *process algebra* was first used by BERGSTRA and KLOP in the title of a report from 1982 [BK82], which contained a preliminary version of the *Algebra of Communicating Processes* (ACP) which was published in 1984 [BK84].

A characteristic feature of process algebra based concurrency theories is the existence of a *formal semantics* in which the meaning of the expressions in the algebraic system description is captured. The objective of a formal semantics is to create a consistent and unambiguous framework for reasoning about concurrent systems. Along with the development of a large number of process algebras for describing concurrent systems, an almost equally large number of different semantics has been proposed. Since a large portion of the reasoning enabled by the formal semantics is devoted to the question of concurrent system equivalence, this part of process algebra theory is nowadays regarded as a separate area of research, called *process theory* [Bae04] or *comparative concurrency semantics* [BB01].

The concurrent stochastic period began in the early 1980s[16] and is characterised by the combination of the formalisms developed during the concurrent period with results from the area of applied probability theory which are presented in the rightmost column of Fig. 2.10. The primary intention of the resulting formalisms is to provide a framework for the description of concurrent systems, where stochastic time information is included in order to enable the automatic, i.e. tool-based verification of nonfunctional system properties. According to their ancestry from non-stochastic formalisms, SPNs follow a machine-

[15] In the context of process algebras total-order semantics are often confused with interleaving semantics, but the characterisations interleaving versus non-interleaving and total order versus partial order are orthogonal for process algebras [BB01].

[16] With the exception of QN, of course.

oriented and SPAs an algebraic-logical oriented style of system description. In other words, they are based on two different kinds of *modelling paradigms* (see Sect. 3.2): The *resource-usage-flow* based paradigm of SPN and QN in which the system structure is visualised as a net, i.e. an interconnection of abstract machines, and in which the system behaviour is captured by the flow of tokens (jobs, customers) through the network. The occupation of the network nodes (places, queues, servers) by the tokens indicates the usage or status of the system resource which is represented by the associated network node(s). This is contrasted with the *process-(inter)action* based modelling paradigm of SPA in which the system is described as the composition of process terms which are expressed using either an abstract symbolic notation or extensions of non-stochastic process algebra languages[17]. Another distinction of the two modelling paradigms can be based on the viewpoint that has to be adopted by the modeller: *intensional* models, like QN and PN, focus on describing what systems do, whereas *extensional* models, as SPA, are based on what an outside observer sees (see CLEAVELAND et al. [CS96]).

Note that for the purpose of performance evaluation QN-, SPN- and SPA-based models are most frequently mapped onto continuous time stochastic processes (see Sect. 3.4.4). As a consequence of this, we are forced to interpret the evolution of the system using a sequential or total order semantics, since the probability for sampling an identical time instant for two random variables defined on a continuous support is zero! On the other hand, the total-order assumption fits well to the interpretation of the system behaviour on the reachability graph of a stochastic Petri Net or the labelled transition system generated by applying the SOS inference rules to an interleaving SPA description. Another notable property of the standard concurrent stochastic formalisms is, that they have their focus on modelling the control flow of the concurrent system. In all three presented formalisms, the aspect of data manipulation, which is of course important in a synthesis-oriented system description, is underrepresented in standard QN, SPN and SPA models. In the basic SPN and QN descriptions for example, the tokens or jobs are typeless and have no associated value which could be exploited for modelling a control decision. In some cases, this inability to express the transfer and manipulation of typed data adequately forces the inclusion of "modelling artefacts" into the system description (see e.g. the WLAN model of Sect. 7.2).

Conclusions We use the insights gained during the historical excursus as a decision guidance which helps us to determine how a formal description technique should be designed in order to be well suited to support the superior goal of formal PE methods integration in the software development process. The systems that we want to describe and verify belong to the class of concurrent discrete event systems. We need to design a formal description technique, i.e. a modelling language, which is optimal with respect to the following criteria:

- *expressiveness*, i.e. which real-world phenomena can be adequately described in the language? Does the application of the language impose too many abstractions?
- *compositionality*, can complex systems be described by the composition of smaller building blocks in a style which is supported by the description technique?

[17] For example the notation used in the SPA TIPP [HHK+00] is derived from basic LOTOS [BB87].

- *intelligibility*, i.e. how easy can the notation of the description technique be understood by non-experts? This applies especially to syntax elements which carry a lot of semantic meaning.

- *integration potential*, i.e. how far does the modelling style of the formalism differ from the notation used by the UML?

Regarding the first criterion, we have to state that QN are the least expressive of the presented stochastic modelling formalisms. Many real-world situations in which synchronisation among concurrently evolving processes is important cannot be expressed in a queueing network model. On the contrary, SPN- and SPA-based models allow the expression of almost arbitrary synchronisation mechanisms and are well-suited to describe most of the phenomena which are present in concurrent discrete event systems.

Compositionality is undoubtedly one of the strong points of the SPA approach. It is an inherent feature of this formalism and can be employed for describing arbitrary complex systems as a hierarchical composition of smaller constituents. To some extent, compositionality is also available in many SPN-based formalisms, see e.g. [MM84], [Roj97] or [BDH01]. In general, the availability of compositional structuring mechanisms is a desirable feature of any description technique for complex systems.

A comparative assessment of the intelligibility is complicated. The advantage of the SPN and QN formalisms is that they are based on a few modelling primitives only and that smaller models can be visualised in a convenient way. Moreover, SPNs are based on a simple and unambiguous syntax definition. The drawback of the visualisation of networks is that the models become unintelligible if the systems they represent are getting more complex. In order to reduce the size of the diagrams, a lot of textual annotations have been introduced for SPNs but with a mixed diagrammatical/textual network description the ability to animate the dynamic behaviour in a sensible way gets lost (as shown in Fig. 2.6). The intelligibility of SPA descriptions is sometimes hampered by the abstract symbolic notation of some SPA packages, although other implementations use a more user-friendly syntax which is based on programming language-like notation. From the practitioners point of view the ambiguities in the selection and definition of the operators in a particular SPAs is confusing, no common definition of the basic modelling primitives is available as it is the case for the most SPN-based approaches. Due to their ancestry from classical process algebras the SPAs "inherited" the feature to be a suitable framework for the equational reasoning about concurrent systems. Although this feature has spawned a valuable state-space optimisation strategy for systems which can be mapped onto CTMCs[18], we like to point out that, in general, equational reasoning is not the primary goal of performance evaluation. The current trend in SPA research to develop frameworks which combine classical performance evaluation with comparative concurrency semantics and functional requirements verification, represents a *methodological dilemma* against the background of the desired applicability in the software development process: On the one hand, combined frameworks are more powerful and promise to "kill two birds with one stone", e.g. to verify functional and non-functional

[18]This strategy is known as *compositional aggregation* [HS99] and has been applied successfully in the generation of the underlying Markov chain of complex systems, see e.g. [HK00].

system properties simultaneously. On the other hand the syntax and semantics of the combined formalisms are more complex and therefore harder to understand for the engineering practitioner. As HEINZ ZEMANEK points out in [Zem66] pragmatic considerations are very important if a formal description technique is going to be used by practitioners. In this context he emphasises the relation of syntax, semantics and pragmatics:

> *The syntactic elements of constructed languages carry an important portion of meaning. Pragmatically, this is very important because the human being when reading has a kind of meaning-assignment mechanism running, which produces many errors if the artificially assigned meaning is different from the usual, natural one.*

For many people — especially non computer scientists — it is much harder to conceptualise reality in terms of *intangible* things like processes than in terms of *tangible* entities like a set of SPN places as an abstraction of a system resource. Using the machine-oriented SPN or QN formalisms it is often possible for the practitioner to establish a one-to-one correspondence between components of the network and parts of real-world systems in which regulated flows, such as parts flow in a production line, jobs flow in a computer system or work flow in an office play a rôle.

The last criterion which asks about the potential that a particular stochastic modelling formalism possesses with respect to the seamless integration into the UML-dominated modelling world of software engineering can be answered clearly in favour of the machine-oriented SPN and QN formalisms since the corresponding behavioural diagram types of the UML are also based on abstract machine descriptions. Transforming UML behaviour models to process-interaction based SPA descriptions and vice versa requires the application of a disturbing shift of paradigm.

Taking into consideration the advantages and disadvantages of the available formalisms with respect to the four criteria, we come to the conclusion that a stochastic modelling formalism which is well-suited for the application by the engineering practitioner should best be based on the machine-oriented, viz. network-oriented view in which the models are expressed using a programming language-like syntax. This approach combines the advantages of the machine-oriented view concerning the intelligibility of the system dynamics with the convenient way in which complex systems can be described in a language. Since for the purpose of non-functional system property verification we are not interested so much in equational reasoning and comparative concurrency semantics, the rather "heavyweight" SPA approaches will not be in the focus of our considerations. Instead, we base our modelling formalism on a SPN-oriented, "lightweight" approach as it is explained in the following chapter.

3. The MOSEL-2 Lightweight Formal Method

Software development needs methodological guidance.

MANFRED BROY *[Bro97]*

In this chapter the methodological framework and the architecture of the MOSEL-2 lightweight formal method are introduced. Section 3.1 contains a motivation to which extent the successful application of a formal method depends on the existence of a framework. The main task of the methodology is to escort the software engineer through the most difficult stage of the system development process, which is the transition from the informal reality to the formal system description. During this transition a worldly computational problem has to be captured in a symbolic representation so that the statements obtained by the following automated reasoning in the conceptual world are *useful* statements once translated back into the language of the real-world. According to W.L. SCHERLIS [Sch89] this formalization problem is *"the essence of requirements engineering"*. A good methodological framework should specify explicitly how the objects in the real-world, their behaviour and the relations among them should be *conceptualized* and expressed in the formal method's language. The goal is to ensure that method designer and method user share a common view of the problem domain. The focus of Sect. 3.2 is put on the second important aspect of a methodological framework: It should provide the user with a precise description of the central notions 'method', 'formal method' and 'formal description technique' (FDT). The definition of the latter comprises a detailed explanation of the formal language *syntax*, the nature of the underlying *semantics* and of the principles of the logical *formal system* which is used for automated reasoning about the system properties. The two most frequently used logical frameworks for the definition of a FDT are introduced in Sect. 3.2.1.

As it was pointed out in the previous chapter, the evaluation of non-functional system properties is in this thesis considered as a branch of system requirements verification. Hence, the methodology of MOSEL-2 as a method for nonfunctional system requirements is influenced by the insights that have been gained during the development of verification methods for functional system requirements. The origin of the problems that hampered the application of formal methods in software system development are discussed in Sect. 3.2.3. The proposed solution for these problems — the lightweight formal method approach — is described in Sect. 3.2.4. Afterwards, the lightweight approach is transferred to the evaluation of nonfunctional properties.

The architecture of the MOSEL-2 lightweight formal method is introduced in Sect. 3.4 following the methodological principles that were developed before. Section 3.4.1 discusses the possible application areas of MOSEL-2 in the real-world. The abstract syntax of the MOSEL-2 FDT is formalized as a *multiset rewriting system* in Sect. 3.4.2. The operational semantics of MOSEL-2 are defined formally in a model-theoretic style in Sect. 3.4.3. Fi-

nally, three classes of continuous time stochastic processes as the mathematical foundation on which various nonfunctional system requirements can be calculated, are presented in Sect. 3.4.4.

3.1. "Explain by example" vs. "Explain by methodology"

In order to teach the application of a novel method to the target user group and to give them an understanding of its merits, the method designer may choose among different strategies. A straightforward and frequently selected approach is to *"explain by example"*, in which the use of the method is introduced on the basis of some small to moderate sized case studies[1]. The syntactic constructs of the language are explained "on demand" in the order of their appearance in the case studies. The semantics of the language constructs is often described only implicitly: First, the results of the example system analysis, which are obtained by the invocation of the method's tool, are presented in graphical or tabular form. Afterwards, an interpretation of the syntax is established implicitly by relating the calculated results to the real-world system that is described in the method's language. The advantage of this approach is that the novice user is invited to explore the method on his own at an early stage of his learning process by repeating the presented case studies, applying some small modifications to them and analyzing the modified examples by invocation of the tool. In this way, the user gets an encouraging rapid feedback. The major drawback of the "explain by example" strategy is that, due to the informal treatment of the language semantics, some core postulates of the method are left implicit and thus are subject to interpretation by the novice user. This is a dangerous situation, since it may happen that the user establishes an interpretation which is different from the one intended by the method developer. As a consequence thereof, the user may draw inadequate conclusions about the expected behaviour of the real-world system under construction when he applies the method in his development project. In exceptional cases, the user may even misuse the method by trying to apply it to problems which are out of the methods scope.

An improved strategy for the introduction of a method is to heed BROY's initally quoted advice. According to this, the method designer boosts the learning process of the novice user by providing him with a precise methodological framework, i.e. with a systematic study of the method. This approach is chosen in this dissertation, and is called *"explain by methodology"* in the following. Of course, the methodological framework has to be confirmed by a set of carefully crafted case studies, but complex examples should certainly not be presented until the user has understood the crucial parts of the methodology.

The term *ontology* (from the Greek οντος = 'being' and λὸγος = 'word/speech') is borrowed from philosophy where it refers to a systematic account of existence. A widely used, but liberal definition for an[2] ontology in computer science is given by T.R. GRUBER in [Gru93] according to whom it is *"an explicit specification of a shared conceptualization"*. Specialised

[1] This strategy was chosen amongst many others by H. HEROLD in his thesis *"MOSEL A Universal Language for Modeling Computer, Communication and Manufacturing Systems* [Her00].

[2] Note that since in philosophy there is just *one* ontology, philosophers become chilled by the more liberal use of the term in computer science which allows for multiple ontologies (see e.g. [Jan01]).

definitions for the term ontology originate from the areas of knowledge-based systems engineering and intelligent information systems integration: A 'domain ontology' [MSSS00] is used to *"capture domain knowledge in a generic way and to provide a commonly agreed upon understanding of a domain."* A 'method ontology' [GGM98] describes the *"conceptualization of a problem solving method"* (PSM). Recently, ontologies play an important rôle in the *semantic web* project [DOS03], which gave rise to the development of the OWL *Web Ontology Language* [W3C04].

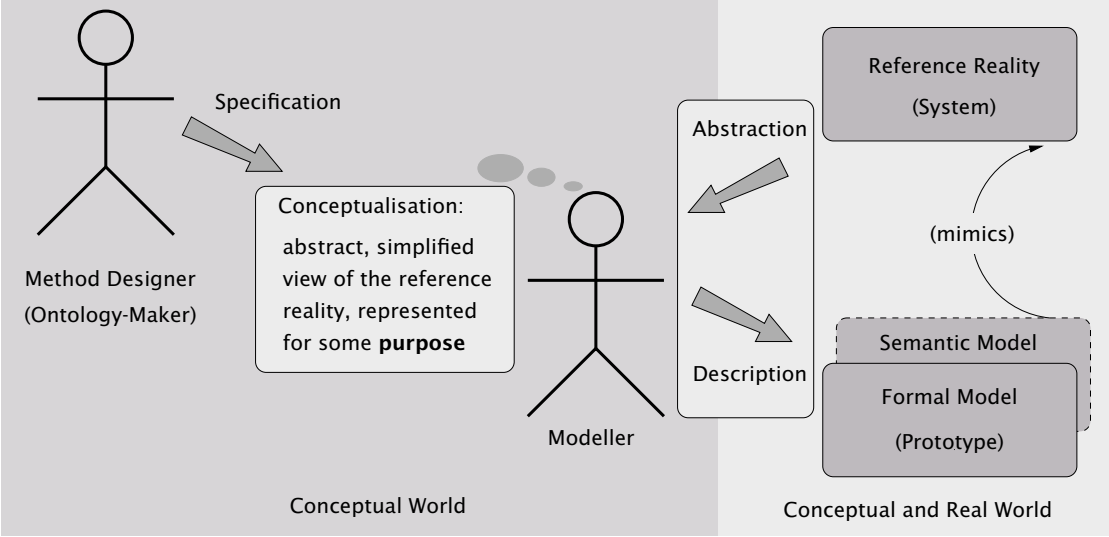

Fig. 3.1: Methodology: the rôles of method designer and modeller in the modelling process

For the purposes of this thesis a description of the term ontology in English prose which is based on the liberal definition of GRUBER will be sufficient: The rôle of the method designer and the modeller in the application of a formal method is illustrated in Fig. 3.1: To solve a class of verification problems in the development of a real-world system the method designer provides a formal method including an ontology for it. This means that the method designer — as the ontology maker — specifies a conceptualization, which is an abstract, simplified view of the world that has to be represented for the purpose of requirements verification. The modeller, once he became acquainted with the ontology, uses the conceptualization as a central part of his modelling activity. As it is shown in the right part of Fig. 3.1, modelling is an act of abstraction, i.e. a reduction with respect to the original, and conceptualisation by a human agent for a given purpose. Taking into account the abstractions which are specified in the conceptualisation, the modeller describes a real-world system which belongs to his *reference reality* or *universe of discourse* as a model. In other words, the modeller generates a *prototype* or *executable specification* of a system in his application domain which *mimics* the problem relevant properties in the real-world. Many formal methods are built upon a formal logical system in which the syntactic prototype model is mapped onto a corresponding so-called *semantic model* (see Fig. 3.1). If such a *semantic mapping* or *interpretation* exists, the method designer should precisely describe the structure of the semantic model and which abstractions are applied during the transition from the syntatic to the semantic level. A suitable approach to exemplify the abstractions employed by the method's ontology is to describe them via a *modelling*

paradigm. The properties of a stochastic modelling paradigm, which was introduced in Sect. 2.3 as a constellation of shared concepts, values, perceptions and practices which form a particular vision of reality, thus play an important rôle in the methodological/ontological considerations of the present thesis. In their contribution entitled *"Paradigm and Software Engineering"* [GA04] GÖKTÜRK and AKKØK summarize the relation of modelling, ontology and modelling paradigm as follows:

> *The act of modelling starts with a mapping between the elements from "the" ontology, what is out there in the universe, to "an" ontology, the ontology of the modelling/design paradigm...*

In the next section some important methodological aspects concerning terminology and structure of formal methods are introduced.

3.2. Methods and Formal Systems

In spite of the large amount of work spent on formal methods, the notion of a 'formal method' is often not stated sufficiently precise [Bro96]. Seemingly, many authors assume that the reader has an intuitive understanding of what a formal method should be. In particular, academic contributions from the area of PE are often afflicted with a vague or missing terminology with respect to formal methods and related terms[3]. This lack of precision in terminology is one indication for an unsatisfactory methodological foundation of PE which has to be improved in order to increase the acceptance of PE within the software engineering community. One of the few elaborate definitions for 'method' and 'formal method' available in the computer science, viz. software engineering literature are given by M. BROY and O. SLOTOSCH in [BS98]:

Definition 3.1 ***(Method)*** *A method consists of description concepts, rules for constructing and relating descriptions, and development strategies explaining how and when to apply the method in a* goal directed *manner. A method should possess a set of guidelines or a style sheet that tells the user the circumstances under which the method can and should be applied most effectively.*

Although BROY and SLOTOSCH do not use the term methodology explicitly in this definition, they paraphrase it by the words, 'set of guidelines' and 'style sheet' which fits well in the view of an ontology as a 'specification of a conceptualization captured in a modelling paradigm'. But definition 3.1 also points out that the methodological guidance for a successful application of a method has to go beyond than just providing an ontology. It is also important to explain how — once the user adopted the ontology — he should apply the method in order to solve his problems. An important prerequisite for doing this is a detailed understanding of the formal method's architecture:

[3] As an example for these findings, the reader is referred to the survey of U. HERZOG entitled *"Formal Methods for Performance Evaluation"* [Her01]. The term 'formal method' occurs exactly three times in the entire document: in the title, the abstract and the conclusion. No definition or even an informal description of a 'formal method' is provided.

Definition 3.2 ***(Formal Method)*** *A* formal method *consists of a formal description technique (FDT) and a tool which executes the automatic verification of the specified system properties on the formal model described via the FDT.*

The FDT constitutes the 'front end' of the formal method and implements a user-interface via which the modeller gets access to the automated verification techniques provided by the formal methods tool. As indicated by the word 'description' in FDT, the modeller uses the FDT to describe 'something', but what is described exactly? The following components of a formal description can be identified:

- the *prototype* of the real-word system under development (see Fig. 3.1),
- the *system requirements*, i.e. the properties that the final product has to exhibit,
- the *interaction* of the system with its enviroment.

Note the difference between the first two components: the prototype describes — on a high level of abstraction — what the system under development will do, whereas the requirements state what the final system is supposed to do. The third component of a formal description does not exist in all FDTs, since the mathematical foundations of some formal methods enable automated reasoning about the fulfilment of the requirements solely on the basis of the prototype desciption. Other formal methods, including the MOSEL-2 LWFM, usually depend upon the presence of the system-environment interaction in the formal description. In many cases the description of the system requirements is called a *specification* or — more precisely a *requirements specification*:

Definition 3.3 *A (requirements) specification is a description of the required properties that reflects all relevant details that a system has to fulfil for a given problem/perspective.*

The somewhat confusing fact is that on the other hand, many definitions for *specification language* comprise the requirements as well as the prototype description aspects, as for example in the following definition from the IEEE glossary of software engineering terminology [IEE90]:

Definition 3.4 ***(Specification language)*** *A language, often a machine-processible combination of natural and formal language, used to express the requirements, design, behaviour, or other characteristics of a system or component. For example, a design language or requirements specification language.*

In most cases where a specification language is used as a part of a formal method, e.g. as in the formal methods presented in Fig. 2.3 in the previous chapter, the system requirements as well as an abstract prototype of the system are expressed by means of the language. The acronym MOSEL-2 for MOdelling, Specification and Evaluation Language-2nd version captures the prototype (modelling), the requirements (specification) and the automated analysis (evaluation) aspects of a formal description.

The definition of a formal description technique that is used in this thesis is also borrowed from [BS98]:

Definition 3.5 ***(FDT)*** *A description technique is* fully formal *if it has a formal syntax (including both context free syntax and context correctness), a formal semantics (described in a precise mathematical way), and a logical theory that supports reasoning about the descriptions and their properties. A description technique is called* semiformal *if it has a formal syntax but not a formal reference semantics nor a logical theory.*

How syntax, semantics and the mathematical foundations of a FDT can be defined and how they relate to each other is described below.

3.2.1. Syntax, Semantics and Formal Systems

Two different approaches for providing a formal verification method with a *formal system* as its mathematical foundation exist. The first is based on *proof theory* [TS00], the second on *model theory* [Hod93], which are both studied as branches of mathematical logic. The main difference between both approaches is the way in which automated reasoning about the system properties to be verified is enabled. In general, a formal system comprises a universal algebraic *structure* $\mathcal{A}$, a *formal language* $\mathcal{L}$ and a *theory* $\mathcal{T}$. The structure consists of a set A, called the *domain* or *carrier* of $\mathcal{A}$ along with some distinguished n-ary functions $f^{\mathcal{A}} : A^n \to A$ for various n, *constants* $c^{\mathcal{A}} \in A$ which can be viewed as 0-ary functions, and n-ary *relations* $R^{\mathcal{A}} \subseteq A^n$ for various n. The list of distinguished functions and relations of $\mathcal{A}$ together with their *arities* is called the *signature* of $\mathcal{A}$. The signature is usually represented by an alphabet Σ consisting of a set of function symbols F and a set of relation symbols R one for each function or relation defined for $\mathcal{A}$, each with a fixed associated arity. Based on the symbols of Σ, the *formation rules* of the formal language $\mathcal{L}$ determine which sequences of symbols express well-formed formulae (wff) of $\mathcal{L}$. The theory $\mathcal{T}$ is based on a set of *non-logical* and *logical axioms* which are expressed as well-formed formulae of $\mathcal{L}$. The non-logical axioms are a *syntactic* description of the relational structure $\mathcal{A}$ and its properties of interest, whereas the logical axioms embody the general truths about proper reasoning in the theory.

The basic principle of the proof theoretic approach is shown in Fig. 3.2: In addition to $\mathcal{A}$, $\mathcal{L}$ and $\mathcal{T}$ the formal system is equipped with an *inference system* or *logical calculus*. An *inference* or *transformation rule* of the logical calculus defines how from a given set of well-formed formulae, called the *premises*, a new well-formed formula — the *consequence* — can be derived, or *inferred*. A well-known inference rule is the *modus ponens*: $\varphi, (\varphi \to \psi) \vdash \psi$, i.e. given the premises φ and $(\varphi \to \psi)$ infer ("$\vdash$") the consequence ψ. As shown in Fig. 3.2, various logical calculi exist for different formal proof systems. Note that the application of the inference rules is a purely syntactic activity which can be "mechanized", i.e. carried out by an algorithm. The verification of system requirements in a proof theoretic formal system proceeds as follows: The axioms of the theory $\mathcal{T}$, which are defined to be *true statements* or *theorems*, serve as a starting point. A system requirement which has to be verified is specified as a well-formed formula Φ of the language $\mathcal{L}$. The task is to show that Φ is a theorem of $\mathcal{T}$, i.e. that it can be inferred from the axioms by application of a finite sequence of inference rules. In Fig. 3.2 two related approaches are shown: *program construction* is an semiautomatic/interactive technique in which a system or program specification $\mathcal{D}$ in con-

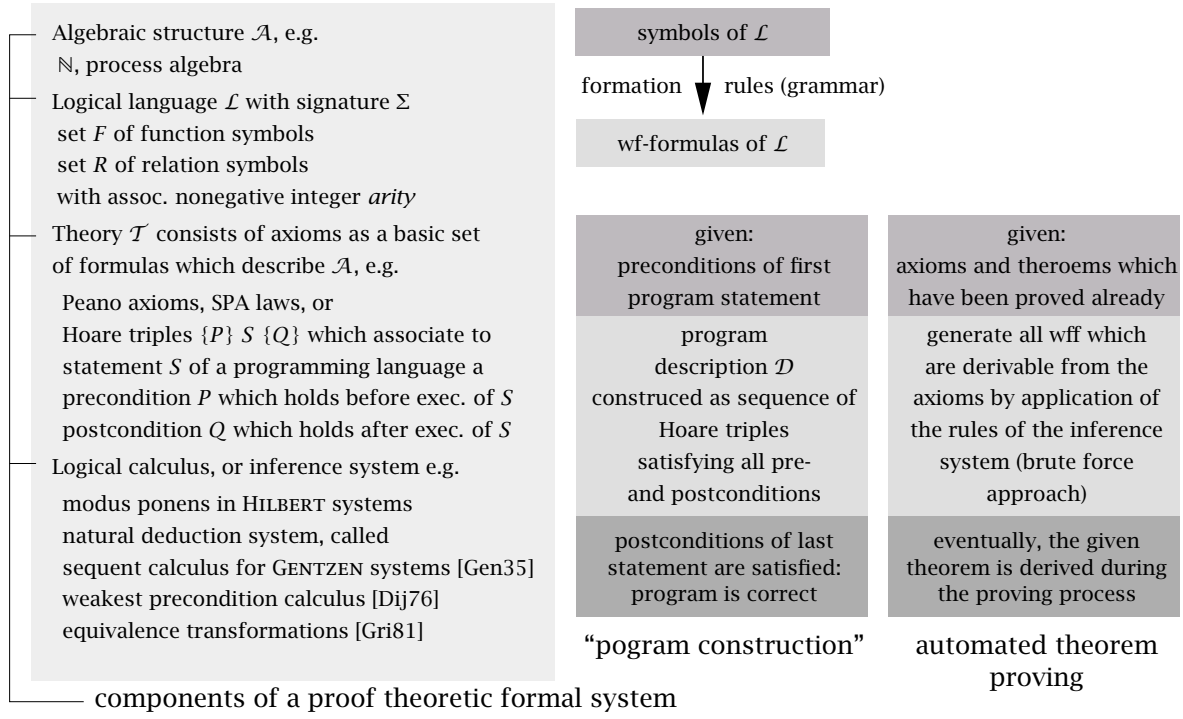

Fig. 3.2: Principle of the proof-theoretic approach in the definition of FDTs

structed by the modeller as a sequence of program statements, which are annotated with well-formed formulae called pre- and postconditions. A verification procedure invoked by the modeller tries to infer Φ as the postcondition of the last program statement by application of the inference rules along the path which was prescibed by the modeller via the pre- and postcondition annotations. In automatic theorem proving the initial preconditions of the program as a set of theorems and the final conclusion Φ which has to be inferred are given. The theorem proving algorithm tries to find a derivation sequence, without being guided by a program specification. This is a brute force approach which may take long and there is no guarantee that a proof for Φ can be found at all. The semantics of a specification language under a proof-theoretic viewpoint is often called *axiomatic* [Hoa69] or *propositional*, since the meaning of a system/program specification is defined by relating the basic language statements to a set of axioms. The logical axioms of proof theoretic formal systems are usually based on the 'classic' propositional or first order predicate logics. For these types of logics GÖDEL's *completeness* theorem [Göd29] and the *soundness* theorem ensure that any *universally valid*[4] formula can be proved, i.e. the theory $\mathcal{T}$ is complete, and that only universally valid statements can be proved, i.e. $\mathcal{T}$ is sound. The completeness and soundness of theories $\mathcal{T}$ based on first order logic guarantee that the theorems obtained by *syntactic* reasoning express true facts about the algebraic structure $\mathcal{A}$ and that only the facts which are considered to be true for $\mathcal{A}$ can be inferred in $\mathcal{T}$.

The basic idea behind the model-theoretic approach for formal systems is to provide an interpretation $\mathcal{I}$, which is a mapping from the domain of syntactic system descriptions $\mathcal{D}$

[4] One has to resort to set theory in order to define what precisely is meant by 'universally valid'.

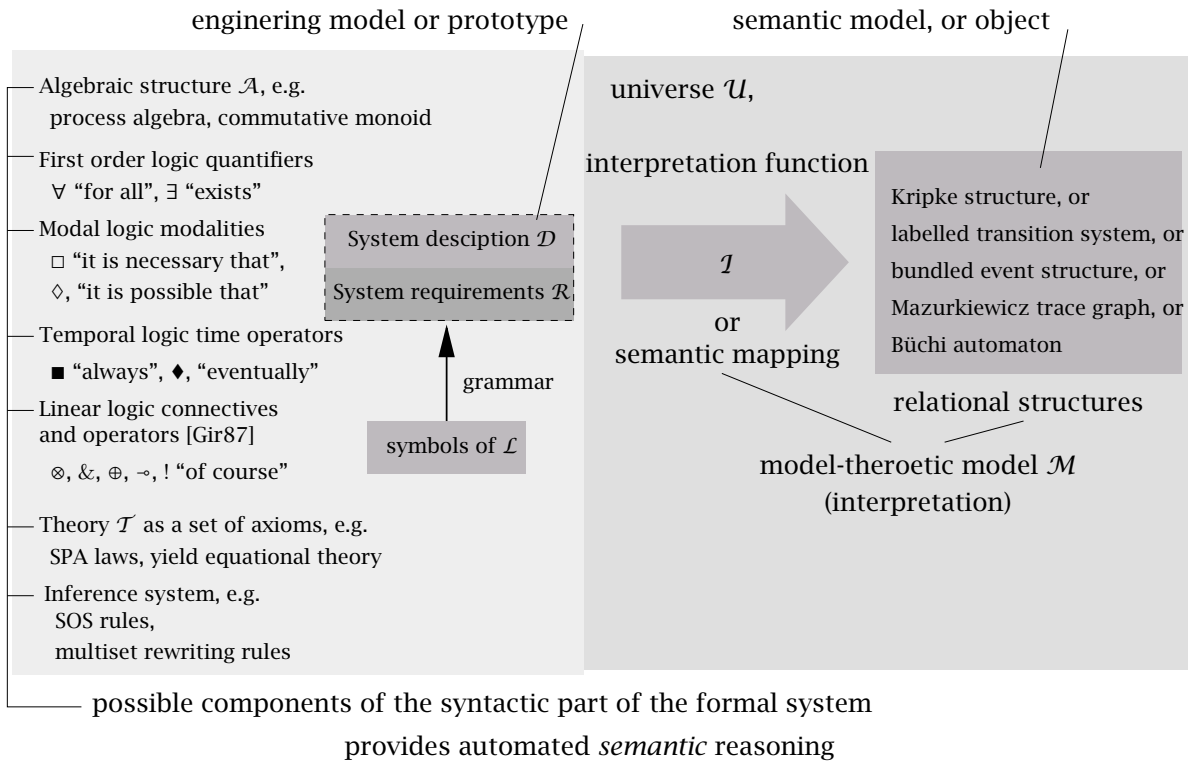

Fig. 3.3: Principle of the model-theoretic approach in the definition of FDTs

and requirements specifications $\mathcal{R}$ (with $\mathcal{D}$, $\mathcal{R} \subset \mathcal{L}$) as well-formed formulae of the language $\mathcal{L}$ to a set of objects in the *universe of discourse* $\mathcal{U}$. Automated reasoning about system properties is then performed on the objects of $\mathcal{U}$ by showing that the requirements $\mathcal{R}$ are *satisfied* in the semantic model which is an element of $\mathcal{U}$. Fig. 3.3 shows the principle of a model-theoretic formal system in more detail: The differences to the proof-theoretic formal systems lie in the absence of logical axioms and the way how the rules of the inference system are used. Instead of providing a means for deductive syntactic reasoning the inference rules — which are based on the non-logical axioms of $\mathcal{T}$ — constitute the core of an algorithm which implements the interpretation function or semantic mapping $\mathcal{I}$. This algorithm takes as input the syntactic system description $\mathcal{D}$ and generates a corresponding semantic model as a relational structure in $\mathcal{U}$. The system requirements $\mathcal{R}$, which are also expressed using a dedicated set of symbols from $\mathcal{L}$, are not used by the algorithm but employed later for automated reasoning on the semantic model. The existence of an algorithm which implements the interpretation function as a mapping from a syntactic system description in $\mathcal{L}$ based on an algebraic structure $\mathcal{A}$ to a relational structure in the universe $\mathcal{U}$ is often referred to as an *operational semantics*. As it should become clear from Fig. 3.3 the term 'model' appears here in three different meanings: The first use of it refers to the well-formed formulae of $\mathcal{L}$ which describe the system and its requirements on the syntactic side. They are usually referred to as the engineering or syntactic model which corresponds to the *prototype* of the system shown in Fig. 3.1.

Definition 3.6 ***(Model, engineering)*** *An engineering model is an abstract description of a system that reflects all details which are relevant for a given problem/perspective.*

The second use of the term model is the designation of the model-theoretic or mathematical model, which according to W. HODGES comprises the interpretation function and the objects in the universe of discourse [Hod93]:

Definition 3.7 ***(Model, model-theoretic)*** *A model is formally defined in the context of some symbolic language $\mathcal{L}$. The model consists of two things: A universe $\mathcal{U}$ which contains all the objects of interest (the* domain of discourse*) and a mapping from $\mathcal{L}$ to $\mathcal{U}$, called the evaluation mapping or intepretation function, which has as its domain all constant, predicate and function symbols in the language. The associated theory $\mathcal{T}$ is defined as a set of sentences of $\mathcal{L}$.*

This is a generalization of the original model definition of A. TARSKI [Tar44], [Tar53] in which $\mathcal{T}$ was considered in a classical first order predicate logic framework and *"A possible realization in which all valid sentences of a theory $\mathcal{T}$ are satisfied is called a model of $\mathcal{T}$."*. As shown in Fig. 3.3, a model-theoretic formal system can also be based on the non-classical *modal* and *temporal logic* frameworks, which allows for the automated verification of temporal system requirements.

The third use of the term model in Fig. 3.3 is to serve as a name for an object in the universe of discourse $\mathcal{U}$ which is the image of the engineering model under the interpretation function $\mathcal{I}$. The objects of $\mathcal{U}$ are also called *semantic models*. Note, that the term model is frequently used in the computer science literature without stating explicitly which of the three meanings is intended. In particular, much confusion can be caused if the syntactic engineering model is not distinguished clearly from the corresponding semantic model. In mathematical contributions from the area of model-theory the engineering model as the description of the system is not called a model but belongs to a theory, based on a formal language. For engineers, such a notion seems somewhat strange: In [Sup60] P. SUPPES points out that in the empirical branches of science, theories serve as basis for building (engineering) models of natural phenomena or technical devices, yet in model-theory this approach is turned upside down and (model-theoretic) models are used as a formal basis to show that a mathematical theory is consistent.

A stringent application of model-theoretic concepts in computer science is the method of *abstract state machines* (ASMs) which was introduced by the model-theorist Y. GUREVICH in the mid 1980s [Gur84] and has since been used to provide a formal semantics for many programming and specification languages (see BÖRGER [Bör02]). A further generalisation of a model-theoretic definition for programming and specification languages, which is based on so-called *institutions* can be found in the contribution of J. GOGUEN and R. BURSTALL [GB92].

3.2.2. Properties of Formal Description Techniques

The advantage of using a formal description technique instead of working with an informal or semiformal one is that some important properties of the system descriptions which determine their "usefulness" in the development process can be defined in a precise way. The two most important properties of this kind are *unambiguity* and *consistency*. Informally, a system description or specification is unambigous if it has exactly one meaning.

In contrast to natural language specifications which are inherently ambiguous, i.e. may be intepreted in various ways, the precise definition of the interpretation or semantic mapping in a formal description technique excludes this ambiguity. Consistency of a specification implies that during the automated reasoning no mutually exclusive answers about system properties can be derived, i.e. that the verification task is possible at all. Seen from a model-theoretic viewpoint a specification given as an engineering model is consistent or *satisfiable* if there exists a semantic model as the corresponding image of the specification under the interpretation *I* of the formal system. In [Win01], J. WING gives the following definitions for *unambiguity* and *consistency* in which Syn denotes the syntactic and Sem the semantic domain; $\mathrm{syn} \in \mathrm{Syn}$ is an engineering model and $\mathrm{sem} \in \mathrm{Sem}$ a semantic model; Sat denotes the *satisifies relation*:

Definition 3.8 ***(Unambiguous specification)*** *Given a FDT* $\langle \mathrm{Syn}, \mathrm{Sem}, \mathrm{Sat} \rangle$*, a specification* $\mathrm{syn} \in \mathrm{Syn}$ *is* unambiguous *iff* Sat *maps* syn *to exactly one specificand set.*

Definition 3.9 ***(Consistent specification)*** *Given a FDT* $\langle \mathrm{Syn}, \mathrm{Sem}, \mathrm{Sat} \rangle$*, a specification* $\mathrm{syn} \in \mathrm{Syn}$ *is* consistent *(or* satisfiable*), iff* Sat *maps* syn *to a non-empty specificand set.*

Strictly speaking, the availability of the two definitions above is *the* main rationale behind any formal methods application in system development. A method which is not based on a formally defined syntax and semantics is inherently ambiguous and it is also impossible to reason conclusively about the consistency of specifications. An example for a notation which gives rise to ambiguous specifications is the KENDALL notation for queueing systems [Ken53] — more precisely — the abbreviations for some queueing or scheduling strategies used herein. The JSQ (join shortest queue) strategy for example, introduces ambiguity because it does not state at all how a job should be scheduled if several queues have equal length. This kind of nondeterminism in a specification, named *underspecification*, is considered a feature in situations where it enables *implementation freedom*, but unwanted in situations where specifications are expected to give rise to 'executable' specificands.

3.2.3. Problems in Formal Method Application

The search for formal ways of reasoning about the correctness of software systems is almost as old as computer science itself (see C.B. JONES [Jon03]). After the existence of a software 'crisis' was openly admitted on the Garmisch NATO conference in 1968 [NR68], the interest in formal methods as a promising remedy for the crisis grew rapidly. A new approach to prove the correctness of software, called *program derivation* was introduced by a group of mostly European scientists, among them E.W. DIJKSTRA [Dij68][Dij75], R.W. FLOYD [Flo67], T. HOARE [Hoa69] and P. NAUR [Nau66]. The basic idea is to *"develop proof and program hand in hand."* One starts with a formal specification of what a program is supposed to do and applies syntactical transformations, based e.g. on the *refinement calculus* [Mor87, BvW98], to the specification until it is turned into a program that can be executed. The resulting program is then known to be *correct by construction*. The alternative approach of *a posteriori verification*, i.e. to construct a proof of correctness not until the executable program is available, has the drawback that the proof can only be constructed if the final

program is indeed correct. If it is incorrect the necessary debugging process often requires to rework large parts of the program. Furthermore, it turned out that post facto verification of non-trivial programs is not practicable since this method does not scale up (see JONES [Jon03], p. 21).

The advent of program derivation as the earliest formal methods can be regarded as the cause for the industry-academia gap to break open: the inventors of program derivation, headmost E.W. DIJKSTRA, aggressively promoted their constructive way of program development as a *revolutionary* approach, and refused any attempts to *integrate* the new ideas into existing industrial development practices. Since DIJKSTRA and some of his followers regarded computer science to be rather a new branch of mathematics than an engineering discipline, he derogatively denoted 'software engineering' as the *"doomed discipline"* that deals with the subject of *"how to program if you cannot"* [Dij89]. If programmers do not possess the intellectual capacity to master the formal techniques needed for the application of program derivation, *"then the software crisis will remain with us and will be considered as an incurable disease"* [Dij00].

Unsurprisingly, the extreme view of DIJKSTRA concerning the rôle of formal methods was not approved by many software practitioners but also provoked opposition within academia, e.g. from D.L. PARNAS who objected that in formal methods research *"there is an unreasonable focus on proof of correctness. In other areas, engineers rarely prove theorems; instead, they use mathematics to derive important properties of their proposed designs"* [Par96]. A detailed survey of the controversy concerning the approaches for program verification can be found in [Fet88].

Below, some of the most frequently mentioned complaints from practitioners who tried to apply formal methods in their development projects are listed:

- The tool support provided by many formal methods is either poor or simply non-existent [Cra91].
- Systems are rarely developed from scratch, so in these cases the program derivation method is not very efficient [AL98].
- The learning curve needed to get acquainted to the mathematical theory is felt to be too steep [Hei98].
- The notation employed by many formal methods is often reported to be cumbersome and overly symbolic [CGR95].
- The formal specifications are not readable by the intended customers/users of the software product [Gla96].
- Advocates of a particular formal method do not show that their method scales up to be applicable in a real-world situation, why can they have faith that the practitioner will succeed in doing so [Hol96]?
- the majority of existing FDTs do not address practical issues such as usability and maintainability and require access to experts with a strong mathematical background [WC99].

- Since there is an abundance of formal methods made available from academia[5], a question frequently asked by practitioners is "which is the best notation/method to use?" for which they often get no satisfying answer [DLM02].

For these reasons many software engineers simply do not consider formal methods as a practical alternative to the established processes. Moreover, many companies would be reluctant to throw away current best practice and investments required by such a revolutionary approach if the benefits of the formal methods application are not evident. The disillusioning experiences in many formal method applications spread various myths and misconceptions which deter newcomers from trying to use them (see A. HALL [Hal90], J.P. BOWEN and M.G. HINCHEY [BH95]).

3.2.4. Practical Formal Methods: The Lightweight Approach

Despite of the slow takeup of formal methods by the software engineering industry, various success stories are reported in the literature. Most of them originated from the area of *safety critical* systems, such as in the development of the BOS[6] control system for the Dutch storm surge barrier near Rotterdam [TWC01], or the door management system of the Paris Metro trains [ALR98], but also successful FM application in the development of non-safety critical systems are known [BS93]. The interesting point in these case studies is *how* the formal methods were applied herein. Different formal methods were used in the BOS project during the various phases of the development process. The first application was to formalize the functional specification (FS), a mixture of a requirements specification enhanced with implementation decisions and algorithm descriptions, which was input to the project by the customer[7] as a 700 page natural language text document augmented with various data-flow charts. The communication protocols of the BOS and the interaction with its environment were modelled in PROMELA [Ger97] and the functional requirements of the FS verified on the models using the model checker SPIN [Hol97]. A bunch of violations of the requirements in the original algorithms were found, correct alternatives were modelled and the results proposed to the customer. Other formal methods were applied sucessfully during the detailed design phase of the BOS development.

The style in which formal methods were used in the BOS and other projects follow a trend which emerged in the end of the 1980s and was later termed *formal methods 'light'* [Jon96] or *lightweight* as by D. JACKSON and J. WING [JW96]:

Definition 3.10 ***(Lightweight Formal Method)*** *A lightweight formal method is a formal method which has the following properties:*

1. Partiality in language: *the FDT syntax contains only a small set of basic language constructs which leads to a restricted expressiveness of a specification formulated using the FDT syntax;*

[5] In the words of C.A.R. HOARE [Hoa90]: *"Models are like seeds scattered in the wind. Most will perish but others will root and flourish. The more seeds the better."*
[6] Dutch: *Beslis & Ondersteunend Systeem*, i.e. decision and support system
[7] The Dutch Ministry of Transport, Public Works and Water Management

2. Partiality in modelling: *the application of a LWFM is restricted to reach a single goal. The system under development is modelled from one particular point of view, i.e. a single semantic abstraction function A is applied.*

3. Partiality in analysis: *a formal method should focus rather on the detection of possible errors in specifications than on the more ambitious goal to conduct proofs about the correctness of programs.*

4. Partiality in composition*, the composition of the LWFM with other methods in which other views on the system are taken has not to be complete in the sense of a formally established multi-view consistency.*

The idea behind the first recommendation is that the design of a formal specification language should be oriented towards the automated analysis of the specifications by a tool. Developers of traditional formal methods, especially those of Z [Spi88], have focused primarily on the form of the model, and have treated analysis as secondary, favoring succinctness of reasoning over amenability to automation [JR00]. The expressiveness of a lightweight formal specification language should not go beyond which can be exploited afterwards by the automated analysis. The formal system underlying a LWFM should enable automated verification maintaining a careful balance between descriptive power and calculational reasoning power. Moreover, restricting the language to a few basic modelling primitives results in a more gradual learning curve that has to be mastered by the practitioner [Din03]. In a lightweight approach, the rapid application of the method is also possible for novice users [Fea98], so lightweight formal methods yield results in a cost-effective and timely fashion.

The second point emphasises to put the focus of a formal method application on those aspects of a system for which the analysis merits the cost of formalization. One should not assume that formalization is useful in its own right, but instead know exactly for which purpose the formal specification is constructed and which kind of problems can *and cannot* be solved by the method application.

The third item addresses the question of expressiveness vs. decidability. Since specification languages which are expressive enough to capture all interesting phenomena in the application domain are undecidable[8], which implies that the existence of a correctness proof cannot be assured in general, the lightweight approach proposes to restrict the expressiveness of the specification language in a way that it remains decidable. One abstraction which is usually applied to obtain a decidable specification language within the model-theoretic framework is to restrict the semantic models of the FDT to finite relational strutures. With a decidable specification language the application of exhaustive analysis techniques, such as model-checking (see Sect. 2.1.4), for the verification of a restricted class of requirements is possible. In other words, a lightweight formal method sacrifices the possibility to conduct proofs for expressive specifications in favour of finding errors in restricted specifications reliably and early.

The last part of the LWFM definition proposes to use a single view FDT side by side with other formal or semiformal description techniques without providing a precise link between

[8] RICE's theorem states that all interesting semantic questions for a Turing-complete language are undecidable [Ric53].

them. To find suitable ways of defining consistency in formal multi-view specifications is complicated and no generally applicable strategy has been developed yet. If only the syntax of multiple description techniques is defined formally, as for example in the various diagram types of the UML, the consistency of the system views specified via the different notations can be fomally defined on a meta level, as in the UML *meta-model*, the M2 level of the UML four layer architecture [OMG01]. In his thesis on *value-based requirements engineering* [Gor02] J. GORDIJN follows the lightweight recommendation by relating different viewpoints[9] on e-commerce applications loosely through operational scenarios.

In short, the lightweight approach to formal methods application is *integrative* instead of *revolutionary*. Formal methods should not follow the ambitious *correct by construction* principle but instead serve as an *early defect detection* technique. A good LWFM supports the system developers in their tasks of understanding the system requirements and taking the proper design decisions during the requirements engineering and conceptual design phases of the SDLC. Formal methods should be treated not as an alternative but complementary approach to traditional best practice methods of industrial software engineering. Expressed in the words of M. JACKSON and J. WING [JW96]:

> *A lightweight approach, in comparison to the traditional, lacks power of expression and breadth of coverage. A surgical laser likewise produces less power and poorer coverage than a light bulb, but it makes more efficient use of the energy it consumes, and its effect is more dramatic.*

3.3. The LWFM approach for Performance Evaluation

In the following the methodology and formal framework of the MOSEL-2 method is presented based on the definitions and ideas which were elaborated in the preceeding sections.

3.3.1. MOSEL-2 methodology

The starting point to specify a conceptualisation for the MOSEL-2 LWFM is to define for which kind of real-world systems it can be applied and to state the goal that one wishes to reach by using the method: The *application domain* of MOSEL-2 are the real-world systems whose dynamic behaviour can be characterized as a *discrete event* triggered evolution. According to RAMADGE and WONHAM [RW89] a *Discrete Event System* (DES) *"is a dynamic system that evolves in accordance with the abrupt occurrence, at possibly unknown irregular intervals, of physical events."* DES originate from various reference realities, such as in manufacturing, robotics, vehicular traffic, logistics, and computer and communication networks. Typical events in systems from these reference realities are for example the arrival of a customer in a queue, the departure of a workpiece from a press in a production line, or the transmission of a data packet in a communication network. On the other hand, the MOSEL-2 method is not applicable for the analysis of systems whose behaviour is characterized by a continuous change of its state. As stated before, the goal of the MOSEL-2

[9] For a definition of the terms *view* and *viewpoint* and the subtle difference between them, see [FKN+92] and [AWG00].

method is the verification of time-related, probabilistically quantified non-functional system requirements.

This information about the methods potential application areas and goal forms the origin for the ontological part of the MOSEL-2 LWFM which is illustrated in Fig. 3.4: The MOSEL-2 ontology-maker specifies a conceptualization which mandates to view the real-world DES following the machine oriented, network-token-flow modelling paradigm (see Sect. 2.3). In order to end up with an intelligible prototype for the DES during the formalization, the MOSEL-2 ontology lays down the notational form of the prototype to be purely textual.

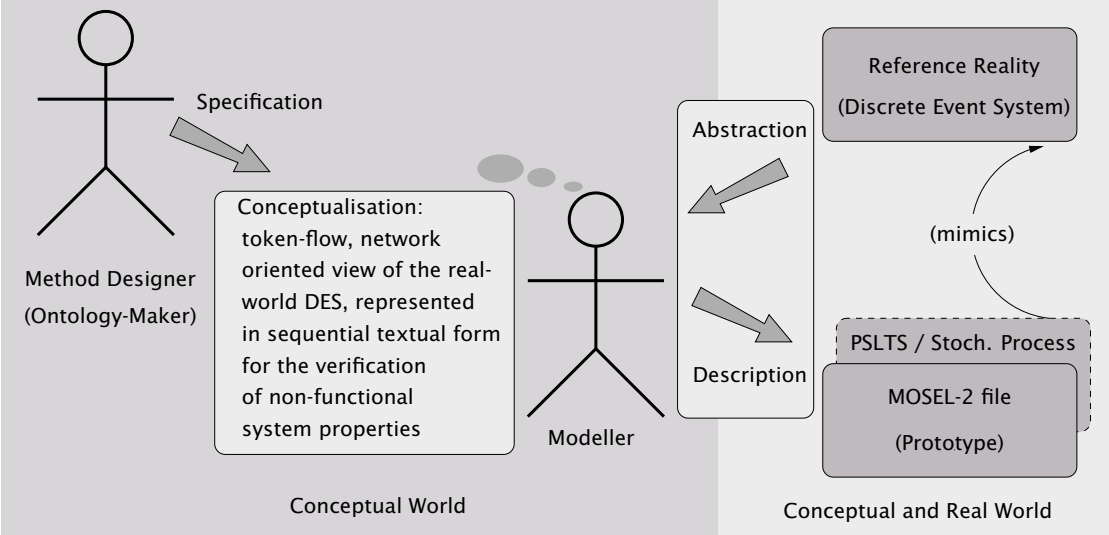

Fig. 3.4: Ontological part of the MOSEL-2 methodology

The following two main abstractions, i.e. reductions with respect to the real-world DES, are applied during the MOSEL-2 formalization:

1. Data manipulation, or computational aspects are *not* in the focus of the method and are largely abstracted from. The tokens used in MOSEL-2 descriptions are anonymous and have no type. For example, it is not possible to model the processing steps which are applied to a workpiece by the different machines of a production line as changes within the token representing the workpiece in the MOSEL-2 description. The only information conveyed by the presence of the token in a MOSEL-2 node representing the processing unit of a machine is the fact that the machine is loaded with a part and supposed to be working. Based on this information, the possible future evolution of the DES being in the state where the machine is busy can be expressed in the model. In other words, the *emphasis of a* MOSEL-2 *description is on modelling the control-flow aspects* of the real-world DES.

2. MOSEL-2 descriptions belong to the class of *executable specifications* in which the evolution of a DES is interpreted over a continuous time interval. Since most of the real-world DES in the target application domains are *interactive* or *reactive* systems which usually exhibit *nondeterministic* behaviour, an abstraction mechanism is needed to resolve this nondeterminism during the formalization in order to obtain an executable

prototype. As usual in stochastic modelling, the abstraction is to express the real-valued durations between the event-triggered state changes of the DES by continous time random variables (see Sect. 3.4.4). The continous time domain is chosen because most of the targeted DES are made up of several components which evolve, i.e. change their state *asynchronuously*. Together with the *interleaving interpretation of concurrency* in the semantic models of MOSEL-2 (see Sect. 3.4.3) a *probabilistic resolution of nondeterminism* is achieved, since at any moment in continuous time every possible event has a well-defined probability to occur and the next event to trigger the system state change can be uniquely determined. Note that an inadequate, i.e. unrealistic choice of the distribution type and parameter values for the random variables by the modeller is the most frequent source of 'distortion' in the prototype. In order to obtain useful quantitative statements about the system behaviour from the MOSEL-2 LWFM application, the modeller should ensure that the stochastic information that he provides in the system description approximates the timely behaviour of the DES as accurately as possible. If available, parameters based on empirical results should be used.

The MOSEL-2 LWFM is well-suited for the specification and analysis of concurrent DES since the modelling language provides syntactic constructs which allow the modeller to express even complex synchronisation and coordination mechanisms in a compact and intelligible way. The language features are in particular tailored to model control decisions needed in multiprocessor computing systems, communication network protocols and parallel production systems which turns these reference realities into ideal targets for the application of the MOSEL-2 LWFM.

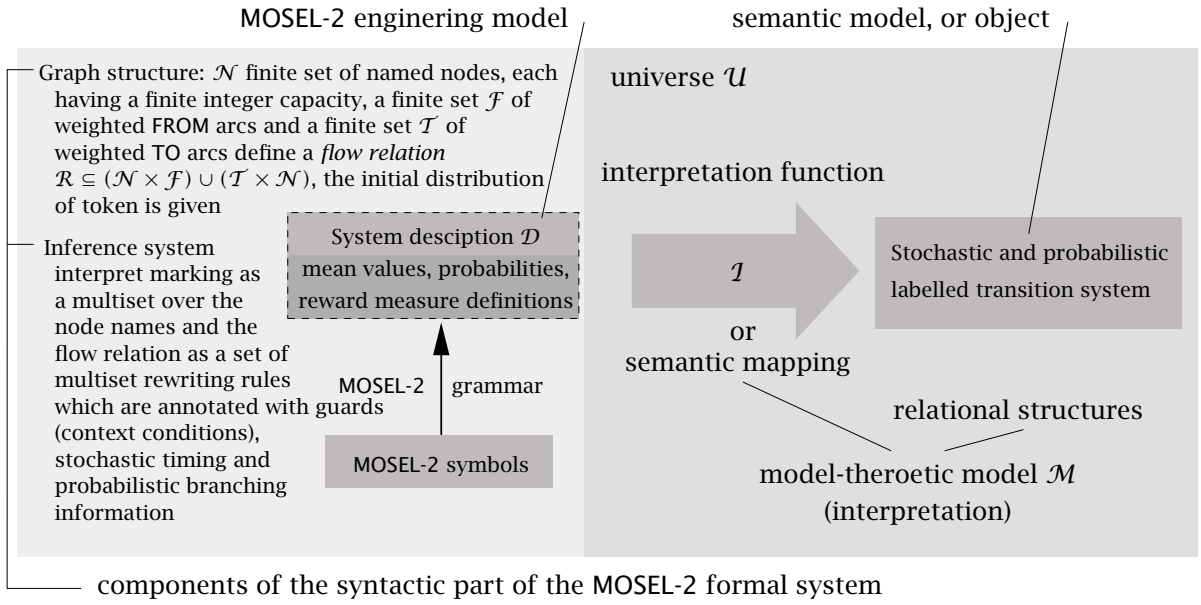

Fig. 3.5: The model-theoretic formal framework of MOSEL-2

3.3.2. The MOSEL-2 formal framework

The second important part of the methodology for MOSEL-2 is illustrated in Fig. 3.5: The formal framework of MOSEL-2 is of the model-theoretic type, i.e. automated reasoning is performed on the semantic level. The MOSEL-2 engineering model which is created by the modeller according to the MOSEL-2 ontology contains a system description $\mathcal{D}$ and a specification of the nonfunctional system requirements that the modeller wants to be evaluated. These requirements are specified as mean values, probabilities or probability distributions. Based on the set of MOSEL-2 symbols, the well-formedness rules of the language, which are given as a context-free grammar, determine which sequences of symbols form a syntactically correct MOSEL-2 description. The network structure is specified as a set of *named* nodes $\mathcal{N}$ followed by a set of *unnamed* rules $\mathcal{R}$. Each node has an associated integer capacity denoting the maximum number of tokens that can be stored in the node. Each node can either be initialized with a number of token up to its capacity or initialized with zero, which is the default. The initial *marking*, i.e. the token values in all nodes represent the global start state of the DES described by $\mathcal{D}$. The rules define a *flow relation*, i.e. the connection of the nodes using a set of *weighted* FROM and TO arcs. Besides this structural information, the rules can be augmented by various constructs which express behavioural aspects of the modelled DES. Each rule can be regarded to describe the effects of the completion of an associated event of the DES on the system state by changing the number of tokens in the nodes used in the rule.

The *inference system* of the MOSEL-2 formal framework is based on an alternative, non graph-oriented representation of the basic modelling primitives: the net markings are formalized as *multisets*[10] over the set of node names. The MOSEL-2 rules are then interpreted as *multiset rewrite rules*[11] which are used to express the dynamics of the modelled DES starting with the initial marking, viz. the initial multiset. A detailed description of the MOSEL-2 inference system is provided in Sect. 3.4.2. The multiset rewrite rules constitute the core component of an algorithm which implements the *interpretation function* $\mathcal{I}$ (see Fig. 3.5). The operational semantics maps every MOSEL-2 description $\mathcal{D}$ onto a probabilistic and stochastic *labelled transition system* (PSLTS) [Kel76], [Sie02]. This underlying semantic model mimics the possible behaviour of the concurrent DES in an interleaving interpretation. The detailed stucture of the PSLTS of MOSEL-2 is described in Sect. 3.4.3.

Remarks: The structure of the MOSEL-2 formal framework of Fig. 3.5 does not exhibit all properties of the full model-theoretic setup presented in Fig. 3.3. The main difference is that in the MOSEL-2 formal system the algebraic properties of the syntactic structures are not considered. The symbols of MOSEL-2 do not contain algebraic operators and the MOSEL-2 rules define no algebraic laws. It should be noted that, depending on the goals that have to be reached by the formal methods application, there exist many other ways to set up a formal system for a concurrent system specification language. For example, is also possible to interpret marked bipartite graphs, viz. Petri nets, as *semantic models* for formal

[10] frequently, the alternative term *bag* is used. See [Bli89] for an elaborate introduction to multisets and multiset theory.

[11] for a precise definition of a multiset rewrite system, see e.g. [Cer94].

logic languages of concurrency which are based on J.-Y. GIRARDs *intuitionistic linear logic* [Gir87, Wad93]. Here, the marked places of the net form atomic propositions of the logic and the transitions are interpreted as proof rules (see [EW90]). Another approach to concurrency theory is based on *category theory* [ML98, BW99]. This general mathematical theory of structures and systems of structures is suitable to formalize different complementary properties of them, such as geometric, algebraic-logical, computational and computational ones, within a common framework. Algebraic properties can be included in terms of *commutative monoids*[12] [MM90] or the more expressive GIRARD *quantales*[13] [Gir87]. The survey [MOM91] of N. MARTÍ-OLIET and J. MESEGUER contains a unified treatment of Petri net theory, linear logic and category theory. In [BG95] C. BROWN and D. GURR show how category theory and linear logic can be used to develop a *compositional* framework for Petri nets.

The design of the MOSEL-2 method follows the four aspects of the lightweight formal method definition: Concerning the required *partiality in language*, the MOSEL-2 FDT is based on two modelling primitives only, namely a set of nodes and a set of rules. These are sufficient to describe most of the dynamic aspects of concurrent DES which are needed for nonfunctional system property evaluation. From the beginning, the design of the MOSEL-2 FDT was carried out hand in hand with the development of the associated evaluation environment, which ensures that every MOSEL-2 description can be analysed by the evaluation environment. The limitations of the syntactic component of MOSEL-2's formal system to consist only of textual description of a network structure and an associated inference system is a consequence of the *partiality in modelling* of a LWFM. In order to reach MOSEL-2's unique goal of nonfunctional system property verification it is not necessary to provide the calculational power to reason about behavioural equivalence of DES on the semantic level. Consequently, the symbols and well-formed formulae of the language are insufficient to express equivalences on the syntactic level in order to keep the balance between descriptive and calculational reasoning power. MOSEL-2 is *partial in analysis* since it does not deal with proving the correctness of programs but enables the modeller to detect the violation of performance, reliability, availability and dependability related requirements in a high-level specificaction at early stages of the development process. Concerning the *partiality in composition*, MOSEL-2 fulfils this property of a LWFM, since there exists no formally defined composed view of MOSEL-2 descriptions with other structural or behavioural specifications in which the DES is seen from different perspectives.

3.4. Architecture of the MOSEL-2 LWFM

In this section, a description of the components of the MOSEL-2 LWFM is presented. The four layers of the MOSEL-2 architecture are illustrated in Fig. 3.6 from top to bottom in the sequence of their traversal during the the modelling and evaluation process.

[12] An algebraic structure with an associative and commutative binary operation and identity element, e.g. the positive integers under addition form a commutative monoid.

[13] A quantale is a commutative monoid on a complete join semilattice, for a precise definition see e.g. [EW90].

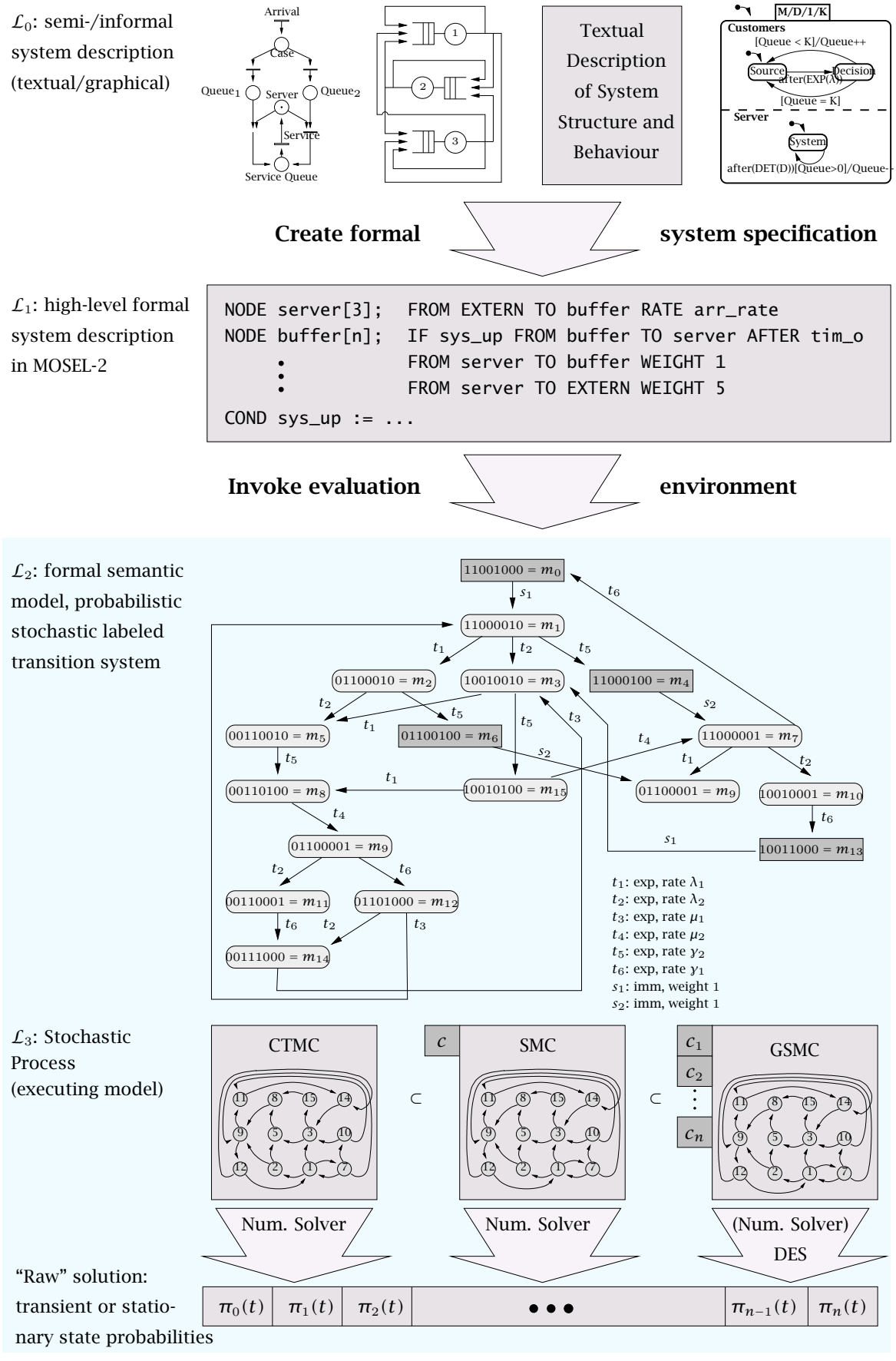

Fig. 3.6: The four constituting layers of the MOSEL-2 lightweight formal method architecture

3.4.1. Informality: The Real World

For the sake of completeness a level $\mathcal{L}_0$ is included in Fig. 3.6 on which informal or semi-formal descriptions and specifications of DES are located and which, in the strict sense, has considered to be outside the MOSEL-2 LWFM. Nevertheless, informal system descriptions and ambiguously stated user requirements is the information which is closest to the real-world application area and every formalization process during a LWFM application needs some informal input to start with. The descriptions/specifications on $\mathcal{L}_0$ can either be a natural language requirements document, which describes some of the structural and functional aspects that the system under development, or parts of it, should fulfil. Other types of suitable information are diagrammatical descriptions of the system structure, its internal behaviour, or explain how the system interacts with other systems in the application domain, which also include the human users of the system under development. Since most of the DES from the application domains are *reactive systems* (see A. PNUELI [Pnu86]), and since the model has to be executable, the modeller has to capture also these influences or stimuli that the environment imposes on the system under development in the formal system description. In PE the specification of the interaction between system and its environment is frequently called *workload modelling*.

The reactive systems that we consider throughout this thesis usually exhibit *nondeterministic* behaviour since the external stimuli from the system's environment are in the most cases unpredictable. Since we are interested in the creation of abstract system representations (models) which are in some sense *executable*, we will have to resolve the inherent nondeterminism of the reactive systems somehow. A widespread solution in this situation is the introduction of probabilities in the model: Possible alternative evolutions of the system as well as possible interactions with the environment are associated with a probability for their occurence. The execution of the model can then be regarded as some kind of repeated random experiment. In this manner, the probabilistic resolution of nondeterminism is achieved.

3.4.2. Formalization: From Reality to Formal Specification

A convenient mathematical foundation for a formal system description $\mathcal{D}$ on the syntactic level $\mathcal{L}_1$ are *multisets* and *multiset rewrite systems*. A survey on multiset theory can be found in the contribution of W. BLIZARD in [Bli89]. Multiset rewriting systems are treated in detail by I. CERVESATO in [Cer94]. An interesting extension of multiset rewriting, which includes the definition of "guards" to represent side conditions for the applicability of a rewrite rule is introduced by MARTINELLI in [MBC+03].

A MOSEL-2 description can be viewed as a multiset rewriting system defined over a finite alphabet Σ which represent the set of named nodes of the MOSEL-2 description. The initialization of the nodes with integer values is interpreted as a multiset of symbols from Σ: k occurrences of the symbol n in the multiset correspond to k as the initial value of the node n. Moreover, the MOSEL-2 rules can be interpreted as multiset rewriting rules over symbols in Σ which together constitute a muliset rewrite system:

Definition 3.11 ***(MOSEL-2 model)*** *A MOSEL-2-model* $\mathrm{syn}_{\mathit{MOSEL\text{-}2}}$ *is a quintuple* $\langle \mathcal{N}, \mathcal{R}, \mathcal{A}, C, W \rangle$

where

- *a finite set of* nodes $\mathcal{N}$, $|\mathcal{N}| = n$
- *a finte set of* rules $\mathcal{R}$ *which are disjoint from the nodes*
- $\mathcal{A} \subseteq (\mathcal{N} \times \mathcal{R}) \cup (\mathcal{R} \times \mathcal{N})$ *is a set of arcs defined by a binary relation which is called the* flow relation.
- $C : \mathcal{N} \to \mathbb{N}_0$ *is the* capacity function.
- $W : \mathcal{A} \to \mathbb{N}_0$ *is the* weight function.

Definition 3.12 ***(from-parts, to-parts)*** *Given a MOSEL-2 model* $\text{syn}_{\textit{MOSEL-2}} = \langle \mathcal{N}, \mathcal{R}, \mathcal{A}, W \rangle$ *and* $r \in \mathcal{R}$*, we define* ${}^\bullet r = \{n \in \mathcal{N} \mid (n, r) \in \mathcal{A}\}$ *and* $r^\bullet = \{n \in \mathcal{N} \mid (r, n) \in \mathcal{A}\}$*. We call* ${}^\bullet r$ *and* $r^\bullet$*, respectively, the* from parts *and* to parts *of the rule* r*.*

Definition 3.13 ***(Marking)*** *A* marking *for a* MOSEL-2 *model* $\text{syn}_{\textit{MOSEL-2}} = \langle \mathcal{N}, \mathcal{R}, \mathcal{A}, W \rangle$ *is a multiset over* $\mathcal{N}$*, i.e. a mapping* $\mathcal{M} : \mathcal{N} \to \mathbb{N}$*.*

3.4.3. What does it mean? Semantic Mapping

The transition from the syntactic level $\mathcal{L}_1$ to the semantic level $\mathcal{L}_2$ can be defined formally by the specification of an algorithm which takes as its input on the $\mathcal{L}_1$ level, a MOSEL-2 description $\mathcal{D}$ formalized as a multiset rewrite system including a given *inital multiset* or *start marking*, and maps this syntactic model $\mathcal{D}$ onto a PSLTS by calculating the *reflexive-transitive hull* of the initial multiset. During the execution of this algorithm all multisets, which can be reached by the application of muliset rewrite rule sequences on the start multiset, are generated. They are stored in a data structure together with the probabilistic and stochastic information that is associated with each of the multiset rewrite rules and the antecedent of the reached mulitset. When the semantic mapping algorithm terminates, the data structure contains an PSLTS as the underlying semantic model of $\mathcal{D}$. Since the semantic mapping algorithm or interpretation function $\mathcal{I}$ mimics all possible sequences of operations of the formal system description $\mathcal{D}$, it is said to define an *operational semantics*.

Interleaving vs. “real” concurrency and continuous stochastic timing

For a formal method for concurrent systems a true concurrency, partial order or a total order, interleaving execution semantics are possible. The majority of the approaches choose a semantic domain where concurrency is interpreted in the interleaving way. In stochastic modelling, the choice of the concurreny interpretation usually depends on the nature of the index set of the stochastic processes which are used to determine the performance indices. For the MOSEL-2 LWFM the index set is continuous, since the behaviour of the DES is interpreted over continuous time intervals. In this case, the interleaving semantics is enforced by the laws of measure theory, which is the foundational theory of stochastic process theory, and which states that the probability of two events represented by continous random variables occurring at the same time instant has the measure zero, i.e. a synchronous firing

of two MOSEL-2 rules is theoretically impossible in the continuous stochastic interpretation. Interleaving is also applied in the reflexive-transitive hull algorithm which implements the interpretation function $\mathcal{I}$ of the MOSEL-2 LWFM because the algorithm always applies one multiset rewrite rule after another, even if more than one multiset rewrite rule may be enabled. Note that the standard interleaving semantics used in stochastic modelling is at odds with the original ideas of C.A. PETRI which are based on a partial or true concurrency semantics for his communicating automata model. True concurrency semantic models used in non-stochastic applications of Petri nets, are the trace graphs of MAZURKIEWICZ [Maz96].

3.4.4. The Mathematical Foundation: Stochastic Processes

The last level of the MOSEL-2 architecture consists of the computational machinery which performs the automated reasoning about nonfunctional DES properties. The transition from $\mathcal{L}_2$ to $\mathcal{L}_3$ in the MOSEL-2 architecture is characterized by replacing the PSLTS as a *discrete* relational structure with an intepretation of the system behaviour in a *continuous* domain. In contrast to the formalization in the syntactic and semantic levels $\mathcal{L}_1$ and $\mathcal{L}_2$, which is governed by discrete mathematical theories, $\mathcal{L}_3$ makes intensive use of continuous mathematics, most notably of *measure theory* as the foundation of *probability theory*. Various algorithms based on results from numerical and seminumerical mathematics are employed to calculate the interesting nonfunctional system properties.

The basic principle behind the transition from the PSLTS to the representation of the system behaviour as a stochastic process is to interpret the information about distribution parameters and branching probabilities stored in the PSLTS arc labels, as well as PSLTS structure as a *stochastic process*. A detailed treatment of stochastic process theory can be found in [Ros83], for a concise and comprehensive stochastic process primer the reader is referred to chapter 6 of [CL99]. In the following, the three classes of stochastic processes employed by the MOSEL-2 LWFM are presented and the most frequently applied solution techniques are introduced.

Classification of Stochastic Processes

Definition 3.14 *A* stochastic process *is a family of random variables* $\{X_t \mid t \in \mathcal{T}\}$ *in which each random variable* X *is indexed by parameter* t *where* t *varies over the index set* $\mathcal{T}$*. The random variables are defined over a common probability space* $(\Omega, \mathbb{E}, P)$*.*

The index parameter t is usually interpreted as time if $\mathcal{T} \subseteq \mathbb{R}_+ = [0, \infty)$. If $\mathcal{T}$ is a countable set, the stochastic process is called a *discrete-parameter* process ($\mathcal{T} \subseteq \mathbb{N}$), otherwise we refer to it as a *continuous-parameter* process. The values of the random variables X_t are called *states* and the set of all possible states S constitutes the *state space* of the stochastic process. If S is a finite or countable infinite set, then the stochastic process is a *discrete-state* process which is often referred to as a *chain*. Stochastic processes with continuous state space S also exist, but since the stochastic processes in the MOSEL-2 LWFM are derived from the finite discrete PSLTS from which they inherit the state space, continuous state space processes are out of the scope of MOSEL-2.

Alongside this classification of stochastic processes with respect to the nature of their parameter set $\mathcal{T}$ and the state space S, the statistical properties of the random variables defining the process are important. A random variable X is completely characterized by specifying a cumulative distribution function (cdf) $F_X(x) = \mathrm{Prob}(X \leq x)$ for it. Equivalently, a stochastic process $\{X_t \mid t \in \mathcal{T}\}$ is fully characterized by the *joint cdf* that describes the interdependence of *all* random variables that constitute the process. With the random vector

$$\vec{X} = (X_{t_0}, X_{t_1}, \ldots, X_{t_n})$$

which can take values $\vec{x} = (x_0, x_1, \ldots, x_n)$ we have to specify a joint cdf

$$F_{\vec{X}}(x_0, \ldots, x_n; t_0, \ldots, t_n) = \mathrm{Prob}(X_{t_0} \leq x_0, \ldots, X_{t_n} \leq x_n)$$

for all possible $(x_0, x_1, \ldots, x_n)$, all $(t_0, t_1, \ldots, t_n)$ and all n.

In general, it is a difficult problem to specify the joint cdf of an arbitrary stochastic process. Instead of looking for a general solution to this problem, we focus on classes of stochastic processes for which the interdependence of the random variables X_t posesses restrictive properties. These properties cause the specification of a joint cdf for the stochastic process to be much simpler.

At first, we introduce the probably most popular class of stochastic processes in the field of performance evaluation which is characterized by the following restrictive property:

Definition 3.15 ***(Markov process)*** *A stochastic process* $\{X_t \mid t \in \mathcal{T}\}$ *constitutes a* Markov process*, if for any sequence of time points* $t_0 \leq t_1 \leq \ldots \leq t_k \leq t_{k+1}$*,*

$$\mathrm{Prob}(X_{t_{k+1}} \leq x_{k+1} \mid X_{t_k} = x_k, X_{t_{k-1}} = x_{k-1}, \ldots X_{t_0} = x_0) = \mathrm{Prob}(X_{t_{k+1}} \leq x_{k+1} \mid X_{t_k} = x_k)\,. \tag{3.1}$$

Equation (3.1) can be interpreted as follows: if we think of t_k representing the present time (3.1) states that the evolution of a Markov process at a future time t_{k+1}, conditioned on its past and present values, depends only on its present value $X_{t_k} = x_k$. X_{t_k} contains all information about the evolution of the process in the past which is necessary to determine its future distribution. Equation (3.1) is thus called the *memoryless* or *Markov* property.

A Markov process with continuous time parameter and discrete state space is called a *continuous time Markov chain* (CTMC); if the time parameter is discrete we obtain a *discrete time Markov chain* (DTMC).

It is important to realize that the Markov property (3.1) comprises *two* types of memorylessness: Suppose that at time t_1 a Markov chain entered state x. At time $t_2 > t_1$ let the state still be x, that is, no state transitition has yet occurred. Based on (3.1), if at t_2 we try to predict the future given the information "the state is x at time t_2" we should come up with the same result as having the information "the state is x and it has been x over the whole interval $[t_1, t_2]$". For this reason, not only the past history of the Markov process, i.e. the states it occupied before entering state x at t_1, but also the amount of time that has elapsed since t_1 must be irrelevant for the future evolution of the process. The two aspects

of the memoryless property are summarized in the following statments:

> All past state information is irrelevant (no *state* memory needed), and (NSM)
> how long the chain has been in the present state is irrelevant (NSA)
> (no *state age* memory needed).

The (NSA) property of a Markov chain has a strong influence on the nature of the random variable which specifies the length of the time interval between consecutive state transitions of the chain. In order to fulfil (NSA), the cdf of the state sojourn times of a Markov chain has to be the *exponential* distribution function for a CTMC, or the *geometric* distribution in the DTMC case. Recall that we use stochastic processes as the underlying mathematical model for the analysis of discrete event system behaviour. The restriction to exponentially distributed *sojourn times* on the Markov chain level which correspond to the *inter-event* times in the high-level DES description substantially constrains the class of real-world systems which can be modeled properly by a CTMC or DTMC. Nevertheless, for a lot of real-world applications the assumption of exponentially distributed inter-event times is realistic or can at least be justified to a large extent. The popularity of Markov chains in performance evaluation is due to fact that they can be analyzed by the use of efficient numerical standard algorithms (see Sect. 3.4.4).

The second class of stochastic processes which are used in the MOSEL-2 LWFM consitutes a superset of Markov Chains and is characterized by the following property:

Definition 3.16 ***(Semi-Markov chain)*** *A* semi-Markov *chain is a Markov chain where the condition (NSA) is relaxed.*

The sojourn times of a semi-Markov chain are not restricted to be exponentially distributed. A state transition of a semi-Markov chain can occur at any time and the sojourn times can have arbitrary distributions. As a consequence, the stochastic timing aspects of a larger class of real-world phenomena can be captured by semi-Markov chains.

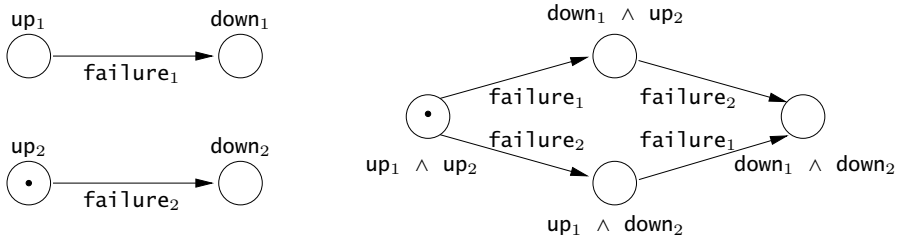

Fig. 3.7: State transition diagrams for a simple computer network

As an example, consider a computer that can be in one of two states `up` and `down`. An event `failure` causes a transition from state `up` to `down` at a random point in time. We assume that the delay until the transition occurs is governed by the uniform distribution $U(0,1)$. A system consisting of a single such computer can be modeled as a semi-Markov process.

The left part in Fig. 3.7 shows the *state transition diagrams* for two computer systems running asynchronously. If we assume now, that the two computers are connected via a

network and view the resulting computer network as a single stochastic process, we obtain the state transition diagram on the right side of Fig. 3.7. Note that for continuous delay distributions, the probability is zero that the two events trigger at exactly the same time, so there is no direct transition from state $\mathtt{up}_1 \wedge \mathtt{up}_2$ to state $\mathtt{down}_1 \wedge \mathtt{down}_2$. If the delay distributions associated with events $\mathtt{failure}_1$ and $\mathtt{failure}_2$ are both uniform $U(0,1)$, the stochastic process underlying the computer network is no longer a semi-Markov process. If after 0.3 time units the $\mathtt{failure}_1$ event causes a transition from state $\mathtt{up}_1 \wedge \mathtt{up}_2$ to $\mathtt{down}_1 \wedge \mathtt{up}_2$, event $\mathtt{failure}_2$, which is enabled in the new state, has already been enabled for 0.3 time units, as it was enabled in the previous state without triggering. Consequently, the delay distribution for $\mathtt{failure}_2$ is not $U(0,1)$ at this point, but $U(0,0.7)$.

In general, the inter-event time distributions at a particular point in time may depend on parts or the entire *execution history* of the stochastic process and this *history dependence* cannot be captured by a semi-Markov process since it has no memory to record the execution history. In order to describe the behaviour of the computer network above by a stochastic process, we have to generalize the definition of the process states in such a way that the process history becomes an integral part of each state $x \in S$. For each event e_j in the finite set of events $\mathcal{E} = \{e_1, e_2, \ldots, e_k\}$ that may trigger a state transition of the process, we define a *stochastic clock variable* $c_j \in \mathbb{R}_+$ which records the remaining time until the event e_j would trigger a transition. For each state $x \in S$ let $\mathcal{E}(x)$ denote the set of events $e \in \mathcal{E}$ that are "active" or "scheduled" when the process is in state x, i.e. all events that may cause a state transition from x. The set $C(x)$ of all possible clock-reading *vectors* is then defined as follows:

$$C(x) = \{(c_1, \ldots, c_k) \mid c_j \geq 0 \text{ and } c_j > 0 \text{ iff } e_j \in E(x)\} \tag{3.2}$$

We use $C(x)$ to construct a Markov chain $\{(S_t, C_t) \mid t \in \mathbb{R}_+\}$ which records the values of the state variable and clock vector at successive transition epochs. The state space of (S_t, C_t) is given by

$$\Sigma = \bigcup_{x \in S} (\{x\} \times C(x)) \subset S \times \mathbb{R}^k . \tag{3.3}$$

Definition 3.17 ***(General State Space Markov Chain)*** *The chain* $\{(S_n, C_n) \mid n \in \mathbb{R}_+\}$ *fulfils the Markov property*

$$\mathrm{Prob}((S_{n+1}, C_{n+1}) \in \mathcal{A} \mid (S_n, C_n), \ldots, (S_0, C_0)) = \mathrm{Prob}((S_{n+1}, C_{n+1}) \in \mathcal{A} \mid (S_n, C_n)) \tag{3.4}$$

for all $n \geq 0$ *and measurable subsets*[14] $\mathcal{A}$ *of* Σ *is called a* general state space Markov chain *(GSSMC) (or* generalized semi-Markov ordered pair *[Gly83]).*

Remark: GLYNN [Gly83] defines GSMP without Simultaneous events. Note that this definition fits well to the standard interleaving interpretation of concurrency in a (continuous time) stochastic process. The GSMP definition of HAAS and SHEDLER [HS89] instead, allows simultaneous events and therefore is suitable for a stochastic interpreteation of true concurrency.

[14] For the definition of measurability see [Tay97]

The term "general state space" emphasizes the continuous nature of the state space Σ. Since, on the other hand, the state space S is discrete, we still refer to the process as a "chain". The class of stochastic processes in which the future evolution may depend on the whole process history is now defined on top of a GSSMC:

Definition 3.18 ***(Generalized Semi-Markov Process)*** *Let* $\{(S_n, C_n) \mid n \in \mathbb{R}_+\}$ *be a GSSMC and* $\zeta_n \in \mathbb{R}_+$ *be the point in time of the n-th state transition. Then, the continuous-time process* $\{X_t \mid t \in \mathbb{R}_+\}$ *with* $X_t = S_{N_t}$ *and* $N_t = \max\{n \mid \zeta_n \leq t\}$ *is a* generalized semi-Markov process *(GSMP).*

Figure 3.8 shows a possible evolution of the GSMP for the connected computer system augmented by a transition `repair` which puts the failed system back to the initial state $\mathsf{up}_1 \wedge \mathsf{up}_2$.

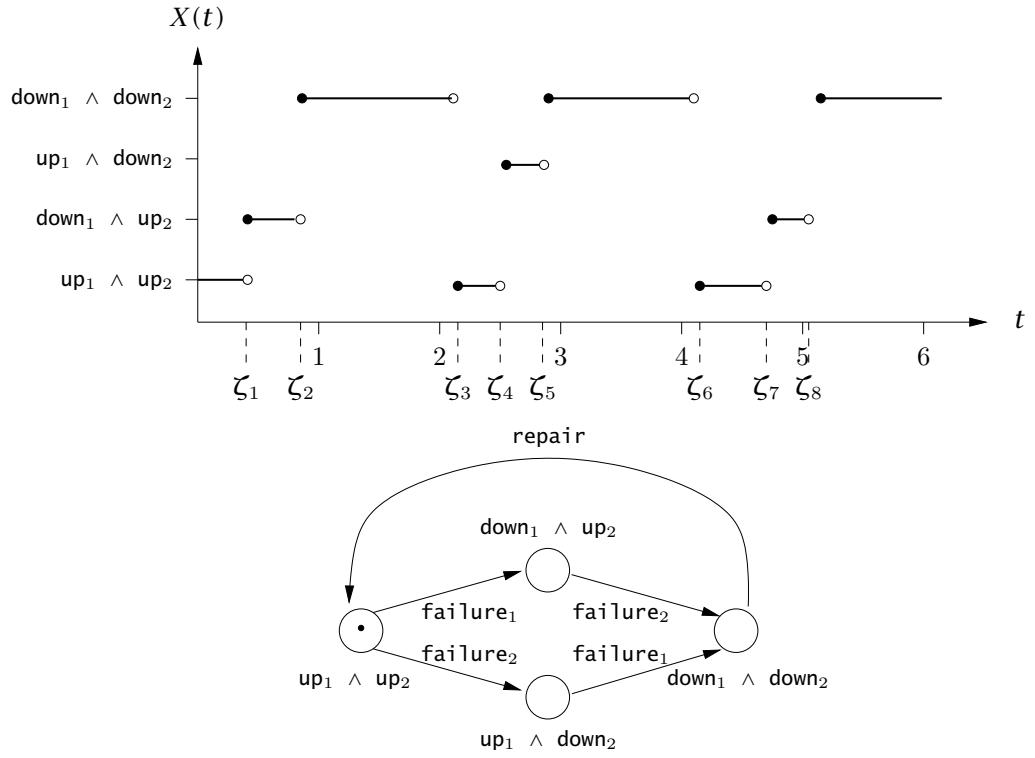

Fig. 3.8: Possible evolution of a generalized semi-Markov process

The class of GSMPs includes both Markov processes and Semi-Markov processes. In the case of continuous time parameter t we obtain the stochastic process hierarchy shown in Fig. 3.9: CTMCs are the least expressive class of stochastic proceses because in order to fulfil the properties (NSM) (no state memory) *and* (NSA) (no state age) the state-sojourn times have to be exponentially distributed. For CTSMCs (continuous time semi Markov chains) the restriction to the exponential distribution is relaxed, but to ensure execution history independence at most one non-exponentially distributed event can be scheduled to cause a transition in any state of the CTSMC. In a GSMP at any time a finite number of non-exponentially distributed events many cause a transtion to the next state, after each state transition the *execution history*, i.e. how long the other non-exponentially distributed events

have been active in the previous process states is recorded in the clock reading vector.

Two other issues introduced by the history-dependence of GSMPs are the phenomena of *preemption* and *re-enabling*: The term preemption is used to denote the disabling of an active event caused by the occurrence of another event. In [AMBB+89] AJMONE MARSAN et al. show how *execution policies* specified in SPN descriptions can be mapped onto re-enabling or memory policies on the stochastic process level. The same technique is applied in the MOSEL-2 LWFM (see Sect. 4.2).

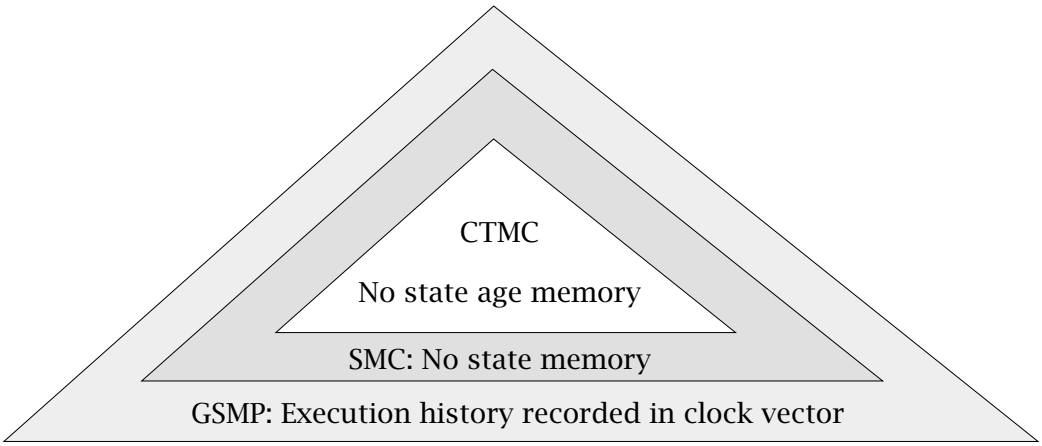

Fig. 3.9: A hierarchy of discrete state continuous time stochastic process classes

Analysis of Stochastic Processes

Now that the three classes of stochastic processes which may appear in the MOSEL-2-method have been identified, it remains to explain how the quantitative system properties that are specified in the MOSEL-2 description can be derived by numerical calculational reasoning for the different stochastic process classes. To "analyze" a stochastic process means to determine a *state probability distribution* vector $\vec{\pi}(t) \in \mathbb{R}^n$ where n is the number of states of the stochastic process. Depending on the high-level evaluation problem, two types of analyses may be performed: in *transient analysis* one is interested in the short-term behaviour of the DES and $\vec{\pi}(t)$ is determined for a specific time-instant $t > 0$, given the initial state probability distribution $\vec{\pi}(0)$. A *stationary* analysis aims at the verification of long-term average system properties, which means that the *steady-state* or *equilibrium* probablity vector $\lim_{t\to\infty} \vec{\pi}(t) = \vec{\pi}$ has to be calculated. As the limiting probabilities do not exist for every stochastic process, it is useful to check in advance for properties which guarantee their existence, e.g. to check a CTMC for *ergodicity*[15].

In principle, every numerical stochastic process analysis procedure is based on deriving a *system of state equations* which captures the dynamics of the process and providing an appropriate algorithm which computes $\vec{\pi}(t)$ as the solution of the system of state equations. The nature and complexity of the state equations depends on the class to which the

[15] Informally, a CTMC is ergodic, if during an infinite time interval each state of the chain is visited infinitely often, each state is reachable from each other state and there are no fixed number of transitions until a state is reached again.

stochastic process belongs: For some CTMCs a system of linear equations is sufficient to determine $\vec{\pi}(t)$ in the stationary case by standard solution algorithms. On the contrary, the stationary analysis of a subclass of GSMPs requires the solution of a system of *multidimenisonal Volterra integral equations* by rather sophisticated algorithms (see [Thü03]).

In the following, systems of state-equations for CTMCs, CTSMCs and GSMPs are derived and references to the relevant literature are provided. A detailed description of the solution algorithms can be found there.

CTMC Analysis

In the following, it is assumed that the CMTCs are *time-homogeneous*, i.e. $\mathrm{Prob}(X_t \leq s \mid X_{t_n} = s_n) = \mathrm{Prob}(X_{t-t_n} \leq s \mid X_0 = s_n)$. The system of state equations for the time-dependent behaviour is then derived as follows:

The elements of state probability vector $\vec{\pi}(t) \in \mathbb{R}^n$ represent the probability that the CTMC is in state j at time t, i.e.

$$\pi_j(t) = \mathrm{Prob}(X_t = j)\,, \quad t \in \mathcal{T}\,,\ j \in S \tag{3.5}$$

In order to express the dynamics of the CTMC we define $p_{ij}(\Delta t) := \mathrm{Prob}(X_{t+\Delta t} = j \mid X_t = i)$ as the probability for a transition from state i to state j within a time interval Δt. Using $p_{ij}(t)$, Eq. (3.5) and the law of total probability, we get (note that $p_{ij}(t+\Delta t) = p + ij(t)$ because of the memoryless property and the time-homogeneity)

$$\pi_j(t+\Delta t) = \sum_{i \in S} \pi_j(t) p_{ij}(t+\Delta t) = \sum_{i \in S} \pi_j(t) p_{ij}(\Delta t) \tag{3.6}$$

for the state probabilities at multiples of the time step Δt. Let $\mathbf{P}(\Delta t)$ be the matrix of all transition probablilies $p_{ij}(\Delta t)$. Equation (3.6) can then be rewritten in matrix notation as follows ($\mathbf{I}$:=identity matrix):

$$\begin{aligned}
&\vec{\pi}(t+\Delta t) = \vec{\pi}(t) \cdot \mathbf{P}(\Delta t) \\
\Leftrightarrow\ &\vec{\pi}(t+\Delta t) - \vec{\pi}(t) = \vec{\pi}(t) \cdot \mathbf{P}(\Delta t) - \vec{\pi}(t) \\
\Leftrightarrow\ &\frac{\vec{\pi}(t+\Delta t) - \vec{\pi}(t)}{\Delta t} = \frac{\vec{\pi}(t) \cdot \mathbf{P}(\Delta t) - \vec{\pi}(t)}{\Delta t} = \vec{\pi}(t) \cdot \frac{\mathbf{P}(\Delta t) - \mathbf{I}}{\Delta t}
\end{aligned}$$

If we take the limit $t \to 0$, we observe that the $\vec{\pi}(t)$ can be obtained as the solution of the following sytem of ordinary differential equations

$$\frac{\mathrm{d}\,\vec{\pi}(t)}{\mathrm{d}\,t} = \vec{\pi}(t) \cdot \mathbf{Q} \quad \text{for a given start distribution } \vec{\pi}(0) \tag{3.7}$$

which are known as the *Chapman-Kolmogorov*[16] equations where

$$\mathbf{Q} = \lim_{\Delta t \to 0} \frac{\mathbf{P}(\Delta t) - \mathbf{I}}{\Delta t}\,, \quad q_{ij} = \lim_{\Delta t \to 0} \frac{p_{ij}(\Delta t)}{\Delta t}\,,\ (i \neq j)\,, \quad q_{ii} = \lim_{\Delta t \to 0} \frac{p_{ii}(\Delta t) - 1}{\Delta t}\ (\textit{exit rate})$$

As an exponentially distributed random variable X is fully determined by its distribution function

$$F_X(t) = \mathrm{Prob}(X \leq t) = 1 - \mathrm{e}^{-\lambda t}$$

[16] Sydney Chapman (1888-1970), english geo-physicist; Andrei N. Kolmogorov (1903-1987) russian mathematician

which contains only one parameter λ, called *rate*, the entries q_{ij} $(i \neq j)$ of the matrix $\mathbf{Q}$ correspond to the rates of the exponential distributions which govern the sojourn time in state i before making a transition to state j.

For the solution of the Champan-Kolmogorov equations, standard numerical solution methods for ODEs – as the *Runge-Kutta* method (see e.g. [SK04]) – can be used. A direct computation of $\vec{\pi}(t)$ using the closed form solution

$$\vec{\pi}(t) = \vec{\pi}(0)\,\mathrm{e}^{\mathbf{Q}t} = \sum_{k=0}^{\infty} \vec{\pi}(0)\frac{(\mathbf{Q}t)^k}{k!} \tag{3.8}$$

of (3.7) given the initial distribution $\vec{\pi}(0)$ is infeasible in practice because of numerical instabilities of the algorithm for computing the series expansion for $\mathrm{e}^{\mathbf{Q}t}$ [Gra77]. Instead, a method which is kown as *uniformization* or *randomization* technique is used frequently: It is based on the transformation of $\mathbf{Q}$ into a stochastic *probability* matrix $\mathbf{P} = \mathbf{Q} \cdot \Delta t + \mathbf{I}$ of a DTMC, where $\frac{1}{\Delta t} = q$ must be larger than the maximum exit rate $q_{ii\,\max} = \max\{q_{ii} | i \in S\}$ of the CTMC. The DTMC is said to be *embedded* in the original CTMC and $q = \frac{1}{\Delta t}$ is called the *uniformization* constant. If we substitute $\mathbf{Q} = \mathbf{P} \cdot q - \mathbf{I} \cdot q$, we get $\mathrm{e}^{\mathbf{Q}t} = \mathrm{e}^{\mathbf{P}qt} \cdot \mathrm{e}^{-\mathbf{I}qt} = \mathrm{e}^{\mathbf{P}qt} \cdot \mathbf{I} \cdot \mathrm{e}^{-qt} = \mathrm{e}^{\mathbf{P}qt} \cdot \mathrm{e}^{-qt}$. Using this substitution (3.8) turns into

$$\vec{\pi}(t) = \vec{\pi}(0)\,\mathrm{e}^{\mathbf{Q}t} = \vec{\pi}(0) \cdot \mathrm{e}^{\mathbf{P}qt} \cdot \mathrm{e}^{-qt} = \vec{\pi}(0) \cdot \sum_{k=0}^{\infty} \mathbf{P}^k \cdot \frac{(qt)^k}{k!} \cdot \mathrm{e}^{-qt}\ . \tag{3.9}$$

The factors $\frac{(qt)^k}{k!} \cdot \mathrm{e}^{-qt}$ are the *Poisson probabilities* of the embedded DTMC. For the approximate algorithmic calculation of $\vec{\pi}(t)$ only a finite number of terms of the infinite sum in (3.9) are evaluated, i.e. $\sum_{k=L}^{R} \mathbf{P}^k \cdot \frac{(qt)^k}{k!} \cdot \mathrm{e}^{-qt}$ is calculated, where the left and right truncation points L and R depend on qt and on desired precision ε of the solution. A detailed treatment of the algorithmic issues of the uniformization technique can be found in [FG88] or [Ger00], pp 67-72.

The steady-state analysis of a CTMC is simplified considerably if the chain possesses the so-called *irreducibility*-property. Informally, a CTMC is called *irreducible* if every state is reachable from every other state in the state-transition graph of the chain. In this case the limiting distribution $\vec{\pi}$ is independent of the initial distribution $\vec{\pi}(0)$ and by applying the limit operator to both sides of Eq. (3.7), we obtain

$$\vec{0} = \vec{\pi} \cdot \mathbf{Q} \qquad \text{normalization condition: } \sum_{i=1}^{n} \pi_i = 1. \tag{3.10}$$

The Chapman-Kolmogorov equations are thus reduced to a system of linear equations, out of which $\vec{\pi}$ can be calculated by different standard numerical algorithms, such as Gaussian elimination, Gauß-Seidel iteration, or the SOR method. The reader interested in the details of numerical solution algorithms for CTMCs is referred to the monograph of STEWART [Ste94].

CTSMC Analysis

The calculation of $\vec{\pi}(t)$ or $\lim_{t\to\infty} \vec{\pi}(t) = \vec{\pi}$ for a Semi-Markov Chain is substantially more complicated than it is in the CTMC case. Since some of the state sojourn times in a SMC

can be governed by arbitrary probability distributions – which are usually determined by more than just one parameter as the case of the exponential distribution – it is impossible to represent the dynamic behaviour of the SMC by solely using a transition rate matrix $\mathbf{R} \in \mathbb{R} \times \mathbb{R}$. Since a SMC does not possess the "no state age memory" property (NSA) but only the "no state memory" property (NSM), we need to include the state age information into the description of the SMC dynamics.

There exists a large body of literature dealing with the analysis of different types of Semi-Markov chains. Due to space reasons we will not go into the details of each of these results here. Instead, we explain the basic ideas of the two most frequently used approaches in SMC analysis in a nutshell, namely the *method of supplementary variable(s)*, which was developed by D.R. Cox in the mid 1950s [Cox55a] and extended by R. German in [Ger00], and the *Markov renewal theory* of E. Cinlar [Cin69].

The Supplementary Variable Approach: The keynote of this analysis technique is to enhance each state of the stochastic process by a real-valued variable, which records the *state age* starting from zero when the process enters the state. If we restrict us to the class of SMCs for which in every state *at most one* generally distributed transition is possible, the supplementary variables can be interpreted as recording the *elapsed firing time* of the general transition. The state space S of the SMC can thus be decomposed in the two disjunct subsets S^G denoting the states in which a general transition is active and S^E which consists of the states in which only exponential transitions are possible. A SMC with this restrictions can be defined as a hybrid discrete/continuous state process $\{(N_t, C_t) \mid t \in \mathcal{T}\}$ where $N_t \in S$ is the discrete state at time t and $C_t \in [0, x^g_{\max})$ is the elapsed activation time of the general transition if $N_t \in S^G$. The following functions are needed for the derivation of the general state equations (notation borrowed from [Ger00]):

$$\Pi_i(t,x) = \mathrm{Prob}(N_t = i\,, X_t \le x)\,,\ \ i \in S^G \qquad \text{age } \textit{distribution} \text{ function}$$

$$\pi_i(t,x) = \frac{\partial}{\partial x}\Pi(t,x)\,,\ \ i \in S^E \qquad \text{age } \textit{density} \text{ function}$$

$$p_i(t,x) = \frac{\pi_i(t,x)}{\overline{F}^g(x)}\,,\ \ i \in S^G\,, x < x^g_{\max} \qquad \text{age } \textit{intensity} \text{ function}$$

The state probability $\pi_i(t) = \mathrm{Prob}(N_t = i)$ can be obtained from the age density function as

$$\pi_i(t) = \int_0^\infty \pi_i(t,x)\,\mathrm{d}\,x\,,$$

Using the functions above and the matrices $\mathbf{Q}$, $\overline{\mathbf{Q}}$ and $\mathbf{\Delta}$ which describe the structure of the SMC, the following system of state equations for the transient SMC state probabilities can be derived (see, e.g. [Ger00], pp. 120):

$$\frac{\mathrm{d}}{\mathrm{d}\,t}\vec{\pi}^E(t) = \vec{\pi}^E(t)\mathbf{Q}^{E,E} + \vec{\varphi}(t)\mathbf{\Delta}^{G,E} + \vec{\pi}^G(t)\overline{\mathbf{Q}}^{G,E} \qquad \text{ODEs}$$

$$\frac{\partial}{\partial t}\mathbf{p}^G(t,x) + \frac{\partial}{\partial x}\mathbf{p}^G(t,x) = \vec{p}^G(t,x)\mathbf{Q}^G \qquad \text{PDEs}$$

$$\mathbf{p}^G(0,x) = \vec{\pi}^G(0)\delta(x)\,, \qquad \text{initial conditions}$$

$$\mathbf{p}^G(t,0) = \vec{\pi}^E(t)\mathbf{Q}^{E,G} + \vec{\varphi}(t)\mathbf{\Delta}^{G,G} + \vec{\pi}^G(t)\overline{\mathbf{Q}}^{G,G} \qquad \text{boundary conditions}$$

using the integrals

$$\vec{\varphi}(t) = \sum_{g \in G} \int_0^{x_{\max}^g} \vec{p}^g(t,x) f^g(x)\,\mathrm{d}x\,, \qquad \vec{\pi}^G(t) = \sum_{g \in G} \int_0^{x_{\max}^g} \vec{p}^g(t,x) \overline{F}^g(x)\,\mathrm{d}x$$

and for the stationary case (see [Ger00], pp. 129):

$$\vec{0} = \vec{\pi}^E \mathbf{Q}^{E,E} + \vec{\varphi} \mathbf{\Delta}^{G,E} + \vec{\pi}^G \overline{\mathbf{Q}}^{G,E} \qquad \text{balance equations.}$$

$$\frac{\mathrm{d}}{\mathrm{d}x} \vec{p}^G(x) = \vec{p}^G(x) \mathbf{Q}^G$$

$$\vec{p}^G(0) = \vec{\pi}^E \mathbf{Q}^{E,G} + \vec{\varphi} \mathbf{\Delta}^{G,G} + \vec{\pi}^G \overline{\mathbf{Q}}^{G,G}$$

$$\vec{\varphi} = \sum_{g \in G} \int_0^{x_{\max}^g} \vec{p}^g(x) f^g(x)\,\mathrm{d}x\,, \qquad \vec{\pi}^G = \sum_{g \in G} \int_0^{x_{\max}^g} \vec{p}^g(x) \overline{F}^g(x)\,\mathrm{d}x\,, \qquad \vec{\pi}\vec{e} = 1$$

In order to calculate the state probabilities from the systems of integro-differential equations above, a couple of theorems and algorithms for the solution of initial value problems from numerical mathematics have to be applied. A detailed presentation of the solution procedure would go beyond the scope of this summary and can be found in Chapters 8 and 9 of [Ger00].

Markov Renewal Theory: In this analysis method a different point of view is taken, in which the continuous SMC is considered at *discrete* time instants $0 = T_0 \le T_1 \le T_2 \le \ldots$, the so-called *regeneration points*. At the regeneration instants the SMC describing the global state changes of the DES is memoryless, i.e. properties (NSM) and (NSA) are fulfilled, and its future evolution is a replica of itself. Fig. 3.10 shows a sample path in the evolution of a SMC which represents a M/G/1 queueing system. The regeneration points (black dots) are chosen at the time instants when either the service begins, i.e. the DES represented by the queue switches from idle to busy, or when the generally distributed service of a customer is completed. Between two regeneration points other customers may arrive according to the Poisson arrival process.

An *embedded* discrete time Markov chain is defined at the regeneration points as $Y_k = N(T_k^+)$. The dynamics of the process at time t can be expressed by means of the following quantities:

$$\pi_{ij}(t) = \mathrm{Prob}(N(t) = j \mid N(0) = i) \qquad \text{conditional state probabilities}$$

$$e_{ij}(t) = \mathrm{Prob}(N(t) = j, T_1 > t \mid Y_0 = i) \qquad \text{conditional probabilities}$$

$$k_{ij}(t) = \mathrm{Prob}(Y_1 = j, T_1 \le t \mid Y_0 = i)$$

Using matrix notation, $\mathbf{E}(t)$ which is referred to as the *local kernel* and $\mathbf{K}(t)$ as the *global kernel* are used in the definition of the *generalized Markov renewal equation* (see [GT99]):

$$\mathbf{\Pi}(t) = \mathbf{E}(t) + \mathbf{K}' * \mathbf{\Pi}(t)\,. \quad (' \text{denotes derivation, } * \text{ matrix convolution})$$

In order to calculate the state probabilities the Laplace-transformed version

$$\mathbf{\Pi}^*(s) = \mathbf{E}^*(s) + s\mathbf{K}^*(s)\mathbf{\Pi}^*(s) = (\mathbf{I} - s\mathbf{K}^*(s))^{-1}\mathbf{E}^*(s)$$

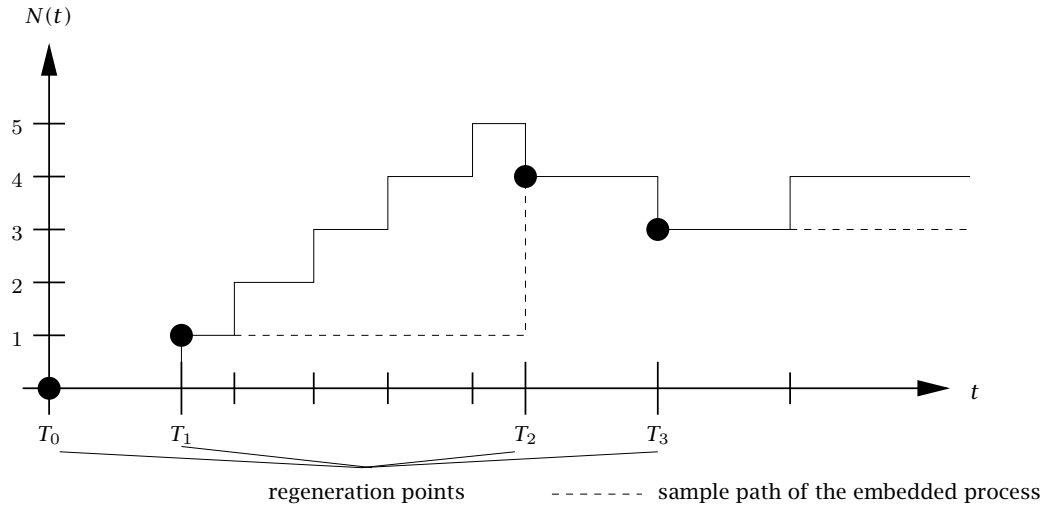

Fig. 3.10: Regeneration points for a SMC representing an M/G/1 queueing system (from [Ger00])

is better suited. In [GT99] GERMAN and TELEK derive another representation of the Laplace transformed generalized Markov renewal equation, which is well suited for a transient analysis. They describe also in detail, how a stationary analysis of a SMC can be performed based on the Markov renewal approach.

GSMP Analysis

The computation of transient or stationary state probabilities via the construction of state equations and customized numerical solution algorithms is also possible for some restricted subclasses of GSMPs: LINDEMANN, SHEDLER and THÜMMLER developed solution methods for the stationary [LS96] and transient solution [LT99, Thü03] of finite-state GSMPs with exponential and *deterministic* transitions, where in any state at most two deterministic transitions are active. TELEK and HORVÁTH [TH98] derive state equations and an associated numerical solution algorithm for a subclass of GSMPs which describe the marking process of a special class of stochastic Petri Nets called *Age-MRSPNs*. The evident disadvantage of these approaches is that they are applicable only if the formal system description on the $\mathcal{L}_1$-level can be mapped onto a GSMP which belongs to the solution-specific subclass. Since it is often impossible to determine the type of the underlying stochastic process of an arbitrary model on the $\mathcal{L}_1$-level, it is desirable to have a solution method at hand which can be applied in any situation. Fortunately, such a method exists and is described in a nutshell in the following:

Discrete Event Simulation (DES) The relation of GSMPs and a large class of discrete event simulations was identified by GLYNN in [Gly83]. The basic idea is to mimic the GSMP dynamics by an *simulation algorithm* which is presented in Fig. 3.11 as a NASSI-SHNEIDERMAN diagram [NS73][17]. The executing GSMP simulation algorithm generates a sequence of *obser-*

[17]It should be noted that real implementations of ESPN-GSMP based simulation engines usually do not calculate the state probability vector, as it is presented here.

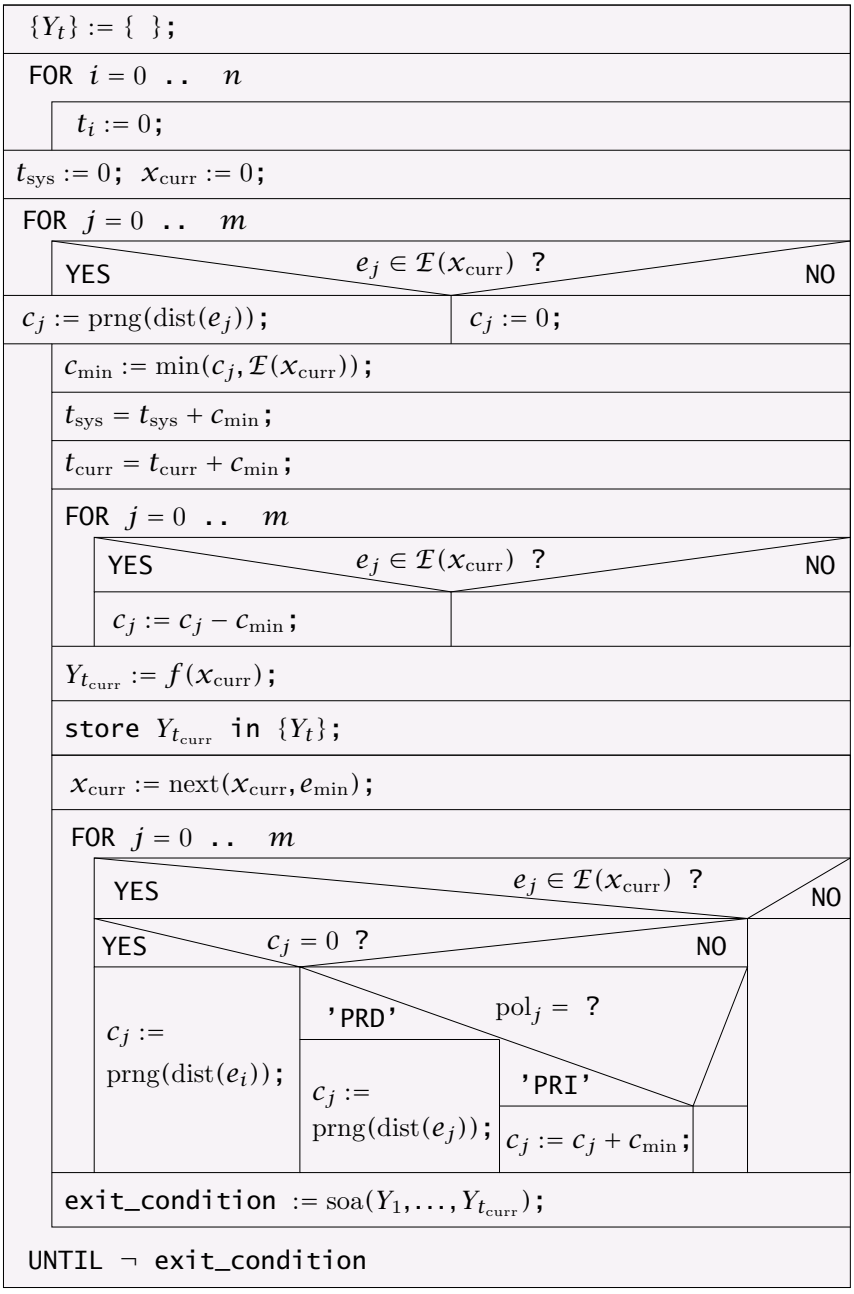

Fig. 3.11: GSMP-simulation algorithm

vations of the evolving stochastic process, as a *sample path* through its state-space. Since the ultimate goal of stochastic process analysis is to evaluate a nonfunctional system property Θ, the simulation algorithm also creates a sequence $\{Y_0, Y_1, \ldots\}$ of obervations of a random variable Y where $\Theta = E(Y)$. A key component of the simulation is the *pseudo random number generator* (PNRG), an algorithm which generates a sequence of samples according to the probability distribution functions associated with the sojourn times of the GSMP states. The term "pseudo" reflects the fact that the PRNG as an algorithm running on a deterministic computer cannot exhibit true random behaviour. The simulation algorithm starts with resetting the variables for the state residence times t_i, the sequence $\{Y_t\}$ for storing the observations of Y and the system time t_{sys}. The current state x_{curr} is set to the initial state x_0 which corresponds to the initalization of the nodes in the MOSEL-2 description. The set of active events $\mathcal{E}(x_{\text{curr}})$ is determined[18] and for each active event an initial random variate is generated by the PRNG and the values stored in the clock vector components c_j. Then the main loop body of the simulation algorithm is executed until the `exit_condition` becomes true: The clock vector component c_{min} containing the smallest value is determined[19], which means that the associated event e_{min} is scheduled to trigger the next state transition of the GSMP. Then the time variables t_{sys} and t_{curr} are updated and c_{min} is subtracted from all c_j. The next state of the GSMP determined by the the occurrence of e_{min} is selected and stored in x_{curr}. Inside the inner `for`-loop, the new set $\mathcal{E}(x_{\text{curr}})$ is determined and — according to the memory policy pol_j of the active event — either a new remaining lifetime c_j is sampled by the PRNG, or c_j is restored to the previous value by adding c_{min}. The `exit_condition` is evaluated by a *simulation output analysis* procedure $\text{soa}()$ based on the sequence $\{Y_1, \ldots, Y_{t_{\text{curr}}}\}$ of observations generated so far.

The purpose of the output analysis routine is to determine the precision of $\{Y_1, \ldots, Y_{t_{\text{curr}}}\}$ as an *estimator* $\hat{\Theta}$ for the performance measure Θ by measuring the *expected value* $E(Y) = \Theta$ and its *variance*. Whereas a sufficiently precise *point estimator* for $E(Y)$ is given as the *aritmetic mean* $\hat{Y}(n) = \overline{y} = \frac{1}{n} \sum_{i=1}^{n} Y_i$ from n given observations, an estimation for the variance is more difficult. The most frequently applied technique for variance estimation is to construct approximate *confidence intervals*[20] and determine if after k runs of the simulation algorithm at least $k \cdot (1 - \alpha)$ of the constructed confidence intervals contain the expected value (see, e.g. KELLING [Kel95], pp. 45). A good estimation of the variance is assumed if the *confidence level* $(1 - \alpha)$ exceeds 98%. For a detailed description of the principles and problems in simulation output analysis see, e.g. Ch. 11 of [BCNN01].

The following advantages and pitfalls of the DES approach should always be kept in mind:

- In general, the explicit generation of the state-transition diagram of the GSMP prior to the start of the simulation algorithm is not necessary. Many implementations generate the set of enabled events "on the fly" at every state transition out of the $\mathcal{L}_1$ level

[18] In analogy to the procedure used in the algorithm which implements the semantic mapping $\mathcal{I}$ described in Sect. 3.4.3.

[19] Due to the interleaving interpretation of concurrency in MOSEL-2 we assume here, that two events are never scheduled to occur at the same time instant, i.e. there is always exactly one smallest clock vector component.

[20] exact confidence intervals can only be constructed in the case when the observations are for normal distributed random variables.

system description. The advantage of this strategy is that the potentially large state space does not need to be stored. On the other hand, it is impossible to guarantee, that the complete state space is explored during a simulation run.

- If the simulation is used in a reliability study, in which the events which model failures occur extremely rare, e.g. one event on average within 10^7 time units, the run times needed by standard simulation algorithms tend to be unacceptably long. Many simulation packages provide sophisticated methods tailored to the analysis of rare-event models. A survey of rare event simulation methods can be found in [GLFH03].

- One of the main problems with the implementations of the GSMP simulation algorithm is the choice of the PRNG function used. A *good PRNG* should generate random variates with a *high equidistribution* in *many dimensions*, with long *period* — the maximum number of different variates generated in a sequence — at high speed. The standard `rand()`-function of the standard C library, does not possess these properties and the usage of `rand()` in a "from-scratch" implementation of a GSMP-simulation is likely to render the obtained results useless. A good and publicly available PRNG is the MERSENNE-*Twister* [MN98] , which generates 623-dimensional $[0, 1)$-variates with a period of $2^{19937} - 1$. Unfortunately, the use of bad PRNGs is very common, as stated for example by PAWLIKOWSKI in [Paw03].

The MOSEL-2 evaluation environment provides access to the simulative analysis of discrete event systems via the SNP based tools SPNP 6.1 and TimeNET 3.0 which both contain simulation engines as well for stationary and for transient analysis. The TimeNET 3.0 simulator is equipped with sophisticated techniques (see Ch. 5 of [Kel95]) which speed up the simulation of certain models considerably. The SPNP 6.1 simulator features different specialized simulation techniques for *rare-event* simulation, which turns this part of the MOSEL-2 evaluation environment into an ideal tool for the analysis of reliability and performability models.

This chapter is concluded by the description of another stochastic process analysis technique which is less generally applicable than the simulation but nontheless useful in various situations.

Approximation of general distributions by exponential phases: The common idea behind the various techniques for phase type (PH) approximation is to replace the generally distributed transitions between two states in a SMC or GSMC by a set of additional states and exponentially distributed transitions and exit probabilities between them. The extra states are interpreted as phases and the time spent moving through the phases on different possible paths models the delay of the original general distribution. The PH approach encodes the memory of a general distribution in the set of discrete phases. One technique for continuous time stochastic processes was developed in the mid 1970s by M. NEUTS [Neu75, Neu95] who proposed to replace a generally distributed transition with an *absorbing* CTMC whose *time to absorption* approximates the firing time of the original transition. Another method was introduced by D.R. COX in [Cox55b] where he showed that every distribution function, whose Laplace transform is rational with m singularities, can be represented by a sequence of m exponential phases with complex valued rates and routing

probabilities. Subclasses of NEUTS PH distributions have been considered by K. BEGAIN in [AB93] as a means to model activities whose distribution parameters have been empirically determined by measurements on real-systems. The extended features of MOSEL-2 allow for the definition of empiric delay distributions, for which mean and variance parameters can be specified. The *rule preprocessor* of the MOSEL-2 evaluation environment automatically substitutes suitable phase-type distribution for the empric ones, before the subsequent analysis steps are executed. A detailed description of the `EMP (mean, var)` construct is given in Sect. 4.2.4.

From the analytic point of view, the PH approximation technique is appealing because under well-defined side conditions concerning the re-enabling of the generally distributed transitions, numerical analysis algorithms for CTMCs can be applied. The main drawback of the PH-approximation approach is that the resulting expanded chain has considerably more states than the original process, which means that for larger models the notorious *state-space explosion* problem becomes even more severe. Algorithms which perform the PH replacement of generally distributed transitions on the semantic $\mathcal{L}_2$ level are described by BOBBIO and TELEK in [BT98].

Summary

This chapter introduced the concept of the lightweight formal method approach for non-functional system property prediction. The need for a *methodological guidance* in a general formal methods application was motivated by the advantages it has with respect to the solution of the *formalization problem*. Any method developer should state explicitly the scope, viewpoint, and modelling paradigm of his method instead of introducing it just by the presentation of a couple of application examples. This methodological guidance helps the practitioner to master the difficult transitions from the informal reality to the formal world of the model and back to the application domain, and ensures that the method application produces *useful* statements about the system under development.

The two main components of a formal method are the FDT and the associated tool. An FDT has a formally defined syntax, semantics and logical system which follows either a proof-theoretic or model-theoretic style and determines if useful statements are inferred on the syntactical or semantical level.

Problems in the application of early formal methods led to the development of a lightweight formal method definition by JACKSON and WING. Formal methods of this kind were successfully applied to detect errors in functional system specifications during the early system development phases. The ideas of methodological guidance and lightweight formal methods were transferred to the area of Performance Evaluation: The resulting MOSEL-2 LWFM is equipped with a model-theoretic formal system, and has a four-layered architecture. Informal system descriptions are formalized using the token-flow, network-oriented modelling paradigm and a pure textual notation. The resulting formal specification is automatically mapped onto an underlying semantic model by the MOSEL-2 evaluation environment and afterwards transformed into an executing semantic model from which statements about nonfunctional system properties are derived either numerically or by discrete event simulation.

4. The MOSEL-2 Formal Description Technique

> *Most programming languages are partly a way of expressing things in terms of other things and partly a basic set of given things.*
>
> PETER J. LANDIN *[Lan66]*

This chapter deals with the concrete syntax of the MOSEL-2 formal description technique. As the concrete syntax defines the "interface" between the modeller viz. practitioner and the MOSEL-2 evaluation environment, pragmatic considerations play an important rôle. The formal interpretation of a MOSEL-2 description as a multiset-rewriting system was convenient for describing the formal relation between the syntactic and semantic layers of the MOSEL-2 architecture. Here, the interpretation as a network with a controlled "flow" of tokens is better suited and is also suggested by the MOSEL-2 ontology described in Sect. 3.1. The basic design idea of the language is to express the two-dimensional network structure, the token flow-control mechanisms as well as the routing probabilities and the stochastic delay information in a one-dimensional textual notation. The pragmatic qualities of this approach are twofold: The first consists of the one-to-one correspondence the basic modelling primitives of the MOSEL-2 *nodes* have with the real-world entities they represent. The second advantage is that the other modelling primitive, the MOSEL-2 *rules*, capture the controlled token flow, including possible delays and routing information in a concise way by meaningful keywords[1]. CRAIGEN et al. [CGR95] stated after interviewing software engineering practitioners about their experiences with formal methods in software development projects, that the ergonomics of language design and the ease with which the underlying semantics can be described as *"cannot be underestimated in developing a successful formal method."*

A MOSEL-2 model is stored in a text file and is divided into the six parts shown in Fig. 4.1. The sequence of the parts in the description is fixed. A minimal model contains only a node part and a rule part. All other parts are supplementary, but a MOSEL-2 description for which non-functional system properties have to be evaluated has to contain a result part as well. MOSEL-2 is a *format-free* language, which means that indentation and line breaks have no special meaning. Any sequence of spaces, tabulator stops and newline characters is called *whitespace* and is ignored.

The MOSEL-2 syntax can roughly be divided into two groups of constructs. The *core* language constructs described in Sect. 4.1 determine the *expressiveness* of the language, i.e. which kind of real-world phenomena can be captured in a MOSEL-2 description. Typically, every real-world system which exhibits discrete event behaviour is considered to be representable by the core MOSEL-2 language constructs. For modelling complex systems,

[1] A list of the context-free production rules which constitute the MOSEL-2 grammar can be found in the appendix (App. A).

MOSEL-2 provides additional syntax elements, which either aim to increase the modelling convenience or can be used to capture complex aspects of the system's topology. Some of these advanced features of MOSEL-2 are introduced in Sect. 4.2.

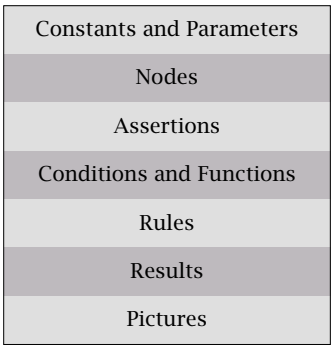

Fig. 4.1: MOSEL-2 file structure

4.1. Fixing the Expressiveness: Core Language

The two most important basic modelling primitives of MOSEL-2 are the *node* and *rule* declarations which have to be specified in the *node part* respectively *rule part* of a MOSEL-2 file. The set of named nodes constitute the state vector of the modelled system and the set of transitions describe how the contolled flow of tokens — which are stored in the nodes — changes the global system state in a discrete event manner as continuous time evolves.

4.1.1. The node construct

A node is associated with a name and a value range. The values are integer numbers ranging from 0 to a node-specific maximum, called the node's *capacity*.

```
node-def ::= "NODE" node "[" max-value "]" [":=" initial-value] ";" .
max-value ::= int-expr .
initial-value ::= int-expr .
```

`node-def` defines a node with name *node* and a value range $\{0,\ldots,$`max-value`$\}$. The node's initial value will be `initial-value` or 0, if `initial-value` is omitted. `initial-value` must be an integer number in $\{0,\ldots,$`max-value`$\}$.

4.1.2. The rule construct

A rule is composed of the following parts:

- A *precondition*, which describes the subspace of states in which the rule is enabled.
- One or more *actions*, which describe the changes of the current state that take place when the rule fires.

- A *firing distribution*. When the rule gets enabled, it may fire after a delay which is sampled from the firing distribution.

- A *re-enabling policy*. When a rule gets disabled, it may remember or forget the time that has elapsed while the rule was enabled. This has impact on the remaining firing delay used when the rule gets re-enabled.

- A *priority* and a *weight*. If several rules are enabled and may fire at the same time, only one of the rules with maximum priority will do so. Let S be the set of all rules that are enabled, may fire at a certain time point, and have maximum priority. Let t be the sum of weights of all those rules. Then each rule of S fires with a probability w/t, where w is the rule's weight. Timed rules always have a weight of 1 and a priority of 0. Immediate rules have a default priority of 1, but they can be assigned higher priorities.

The specification of a firing distribution and weight in a rule are mutually exclusive, i.e. there exist two classes of rules which are used to model two different aspects of the system:

- rules for which a firing distribution is specified, are called *timed* rules and are used to capture the time that an activity in the modelled system, which is associated with the rule, needs from its enabling to its completion. In other words, the timed rules capture the stochastic behaviour of the abstract MOSEL-2 system description.

- the second type of rules for which weights are specified, are called *immediate* rules. These rules are used to capture the probabilistic resolution of nondeterminism in the executable MOSEL-2 specification. In the moment when a timed rule fires, several immediate rules may become enabled which indicates that different sucessor states are possible in this situation. To resolve this nondeterminism probabilistically, the successor state reached by the firing of immediate transition $j \in S$ is taken with a probability w_j/t. This probabilistic resolution of nondeterminism is assummed to be completed in no time, i.e. the sucessor state is reached immediately.

Figure 4.2 illustrates some frequently used MOSEL-2 rule constructs together with a Petri net like graphical equivalent, namely:

- The sequential execution of two delayed activities, the first one delayed with a deterministic firing time `t_w`, the second one is exponentially delayed with rate `mu_r`.

- An example for probabilistic branching where there is equal probability for the two possible next states.

- A timeless duplication of a process or fork-construct.

- A join of two tasks which run in parallel; the marking of the Petri net illustrates the situation where `task_1_fin` waits for `task_2_fin` to complete before their joined execution continues.

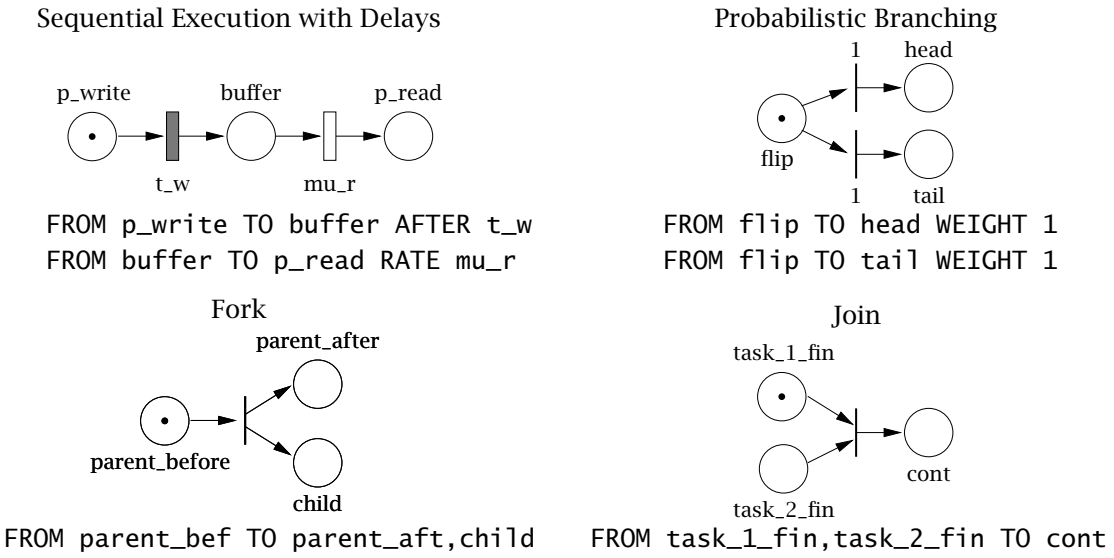

Fig. 4.2: Frequently used rule constructs

Enabling conditions

A condition is defined as follows:

```
       condition ::= and-condition { "OR" and-condition } .
   and-condition ::= not-condition { "AND" not-condition } .
   not-condition ::= ["NOT"] simple-condition .
simple-condition ::= state-expr compare-oper state-expr
                   | "(" condition ")" .
    compare-oper ::= "=" | "/=" | "<=" | ">=" | "<" | ">" .
```

A condition is state-specific. It is used in PROB result definitions and in IF rule parts. OR-ed and AND-ed conditions are shortcut-evaluated, so "`node > 0 AND 1/node = 1`" is a valid expression, although "`1/node = 1`" is illegal if node is zero.

Firing time distributions

The firing time distributions which are available in MOSEL-2 are summarized in Tab. 4.1. It should be noted that in contrast to MOSEL [Her00], where only the exponential firing time distribution was available, the existence of a rich set of firing time distributions contributes much to the usefulness of the MOSEL-2 language. Many real-world phenomena can now be described much more accurately, e.g. delays in internet traffic using heavy-tailed lognormal or Pareto distribution [CLR00] or wear-in or wear-out behaviour of mechanical system components using the Weibull distribution [Nel85].

Reenabling Policies

A frequently occurring situation in various types of DES is *preemption*, which means that a running activity is stopped because another activity completed and changed the global

Tab. 4.1: Available firing time distributions in MOSEL-2

Distribution	MOSEL-2 syntax	Distribution	MOSEL-2 syntax
beta	`BETA (a, b)`	hypoexponential	`HYPO (a, b, c, d)`
binomial	`BINOM (a, b, c)`	lognormal	`LOGN (a, b)`
Cauchy	`CAUCHY (a, b)`	negative binomial	`NEGB (a, b, c)`
deterministic	`AFTER a`	normal	`NORM (a, b)`
Erlang	`ERLANG (a, b)`	Pareto	`PARETO (a, b)`
exponential	`RATE a`	Poisson	`POIS (a, b)`
gamma	`GAMMA (a, b)`	uniform	`AFTER a .. b`
geometrical	`GEOM (a, b)`	Weibull	`WEIB (a, b)`
hyperexponential	`HYPER (a, b, c)`		

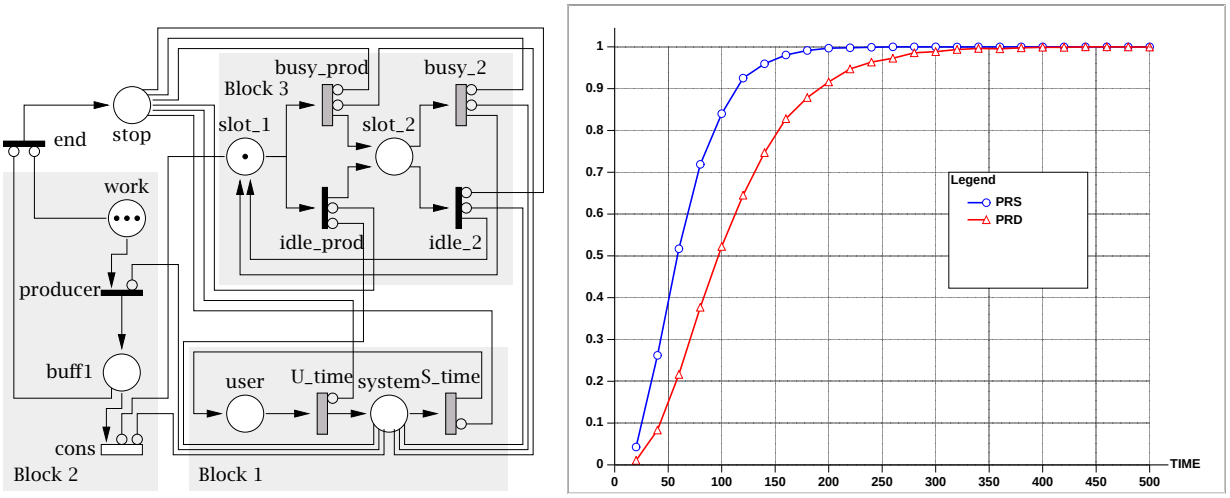

Fig. 4.3: Preemption in a multitasking environment

system state in a way that prohibits the execution of the former one. For rules with non-exponentially distributed delays, e.g. uniformly or deterministically distributed ones, it can be important in order to reflect the behaviour of the real system correctly, to consider the time that has elapsed between the initial enabling of the rule and the moment when it was preempted by the firing of another enabled rule. If later the enabling conditions for the previously preempted rule become true again, should the time spent enabled earlier be neglected or not? Three different approaches are common and were also included in the description of the GSMP-simulation algorithm in Ch. 3 (see Fig. 3.11):

- preemptive resume (PRS), i.e. take the time elapsed before into consideration. This is also called the “work-conserving” reenabling policy.
- preemptive repeat different (PRD), i.e. get a new sample for the firing time of the rule once it becomes enabled again. This is also called the “work-neglecting” policy.
- preempitve repeat identical (PRI), i.e. resample with the same value that was used the last time.

As an example for the effect of selecting different reenabling policies for non-exponentially timed rules on the system performance we consider the model of an abstract multitasking system shown Fig. 4.3. The Petri net based model was presented in [FPTT98] and contains

three blocks: Block 1 models the alternation of the system between productive and maintenance phase, the two places and transitions in Block 2 represent the two sequential phases of job processing, Block 3 captures the alternation during the operative phase between the phase of pre-processing and the one of processing of jobs. The MOSEL-2 description for the system reads as follows:

```
 1  CONST N := 3;
 2
 3  CONST prod_start := 0.5;   CONST prod_end := 1.5;   CONST U_time := 1.0;
 4  CONST S_time := 1.0;       CONST busy_prod := 0.1;  CONST busy2 := 0.1;
 5  CONST cons1 := 0.1;
 6
 7  // NODES for block 1
 8  NODE user[1] := 1;   NODE system[1];
 9
10  // NODES for block 2
11  NODE work[N] := 3;   NODE buff1[N];
12
13  // NODES for block 3
14  NODE slot1[1] := 1;  NODE slot2[1];
15
16  // global NODE indicating completion of service
17  NODE stop[1] := 0;
18
19  // RULES
20  IF (stop == 0 AND system == 0) FROM user TO system WITH U_time PRD;
21  IF (stop == 0 AND user == 0) FROM system TO user WITH S_time;
22
23  IF (slot2 == 0 AND system == 0) FROM work TO buff1 AFTER prod_start..prod_end  PRD;
24  IF (slot1 == 0 AND system == 0) FROM buff1 WITH cons1;
25
26
27  IF (system == 0 AND stop == 0) FROM slot1 TO slot2 AFTER busy_prod PRD;
28  IF (system == 0 AND stop == 0 AND work == 0) FROM slot1 TO slot2 PRIO 1;
29  IF (system == 0 AND stop == 0) FROM slot2 TO slot2 AFTER busy2 PRD;
30  IF (system == 0 AND stop == 0 AND buff1 == 0) FROM slot2 TO slot1 PRIO 1;
31
32  // RULE modeling the condition which leads to system halt
33  IF (work == 0 AND buff1 == 0) TO stop PRIO 2;
34  IF (stop == 1) FROM stop TO stop;
```

If the reenabling policy for all deterministically and uniformly timed rules is changed from PRD to the work conserving PRS-strategy, the probability fir a completed work increases earlier, as shown in the plot on the right side of Fig. 4.3.

4.1.3. Constant and Parameter Declarations

Instead of specifying constant numerical values for the parameters of the firing time delays in the rules, it is also possible to declare names for these parameters in the constant and parameter part of the MOSEL-2 file and assiociate with the name a single value for a CONST-declaration, or a list of values for a PARAMETER-declaration. The advantages of this feature are twofold: If mnemonic names are chosen by the modeller for the delay and other parameters, the MOSEL-2 file becomes much more readable than just with numeric values. Moreover the PARAMETER-declaration provides a very convenient way to turn a MOSEL-2 file

into a descripion of a class of systems with the same structure but different input parameters.

Examples for parameters in systems described by MOSEL-2 can be the number of processes in the application, the communication delay, the clock speed, the network topology, etc. Usually, these systems fulfil their non-functional system requirements for some values of the parameters and for others they do not. If a MOSEL-2 description contains a PARAMETER-declaration, the MOSEL-2 evaluation environment executes an *experiment*, which means, that the MOSEL-2 description is analysed for each varying input parameter. If more than one PARAMETER-declaration is included, the experiment is conducted for every input parameter set. In this way, the modeller can easily obtain information about the range of input parameters for which the non-functional system requirements are fulfilled. It is the aim of *parameter synthesis* to derive the exact conditions on the parameters that are needed for correct system operation. These conditions are usually called *parameter constraints* and knowing these constraints provides useful information for building correct systems. The convenience for MOSEL-2 experiments is increased further if the MOSEL-2 description also contains a graphics part which allows for the specification of plots showing all the results calculated during the experiment graphically.

4.2. Advanced language constructs

The enhanced language constructs described below fall either into the category of "syntactic sugar" which makes the system specification in MOSEL–2 more convenient and compact, or are additional features which are included as a replacement for the arbitrary C programming language constructs which were available for the definition of state-dependent properties in H. HEROLD's MOSEL system. Since in HEROLD's system the only available analysis tool SPNP 3.1 has an input language which is a superset of the programming language C, he designed MOSEL to be also a superset of C. Although this approach gave the modeller the full power of a general purpose programming language, it spoiled the idea of attaching more analysis tools to the evaluation environment, since most of them are not based on C as their input language. During the integration of the tool TimeNET into the MOSEL-2 evaluation environment [Beu03] the problems with the C-based input language became obvious, since TimeNET has a proprietary input language, which does not offer the flexibility of C. As a consequence, the MOSEL-2 modelling language is no more a superset of C. This, on the one hand, has reduced the flexibility in complex modelling situations, but on the other hand the further integration of other tools in the evaluation environment is not hampered any more.

4.2.1. The loop construct

This is the most valuable language construct from the "syntactic sugar" category and can be used to "fold" arbitrary identic parts of the system description into a single construct, leading to a compact but nevertheless intelligible description which would be very lengthy and incomprehensible otherwise. The loop constructs are interpreted by the *preprocessor* of the MOSEL-2 program (see Fig. 5.1) before the well-formedness of the description is checked

by the parser[2]. The following description of the loop construct syntax is adopted from [Beu03]:

An "@" character in a MOSEL-2 source text introduces a loop. A loop is a preprocessor expression and can be used anywhere in a MOSEL-2 source text. When the MOSEL-2 description is parsed, the loop is *expanded* and the expansion is re-fed into the input of the parser. In the simple case — if there is only one *range list* in the loop — the loop is expanded by repeating the *body* as many times as specified by the range list of *body*.

The replacement *body* can be any text with balanced curly brackets ("{ }"). That means, the number of opening brackets must match the number of closing brackets and no prefix of *body* may have more closing brackets than opening ones. The *body* may span multiple source lines. The optional concatenation link may contain any text, but no newline or "" . The link is inserted between repetitive expansions of the loop.

The *body* gets expanded for each loop range in the range list, as described for the following individual cases:

Numbers:

```
loop-range ::= loop-value
             | loop-value  ".."  loop-value .
```

A loop range may be a single number, or a sequence of integer values, given by its lower and the upper limits, which must be integer values.

```
loop-value ::= (number | constant | "#" [cardinal]) [("+"|"-") cardinal] .
cardinal ::= digits .
```

A loop value may be denoted as a literal, as the name of a constant, or as a loop index starting with "#" (see below). If the value is an integer, an integer displacement may be added to that value, or subtracted from that value. A constant used as a loop value must not depend on a parameter.

Enumerations:

```
loop-range ::= enum .
```

A loop range may be an enumeration name. In this case, the loop body will be expanded for each constant that belongs to enum.

Parameters:

```
loop-range ::= parameter .
```

A loop range may be a parameter name. In this case, the loop body will be expanded for each value of parameter.

[2] MOSEL did not have an own preprocessor, but used the C-preprocessor instead. The solution in MOSEL-2 is cleaner, since any manipulations of the input are performed by the MOSEL-2 program itself and no dependencies on external programs exist.

Evaluation time points:

```
loop-range ::=  TIME .
```

If transient evaluation of the model is selected, then a loop range may be "TIME". In this case, the loop body will be expanded for each time point for which the model will be evaluated.

Identifiers:

```
loop-range ::= identifier .
```

If the loop range is a single identifier and does not fit into any of the above categories, the loop body will be expanded once for this identifier. For example, "@<1..3>"*"{(a-b)}" would expand to "(a-b) * (a-b) * (a-b)".

The form of a loop with multiple range lists will be treated as multiple nested single loops. For example:

```
@<1..3>"+"<1..2>"*"{c}
```

is equivalent to

```
@<1..3>"+"{@<1..2>"*"{c}}
```

and expands to

```
@<1..2>"*"{c} + @<1..2>"*"{c} + @<1..2>"*"{c}
```

which in turn expands to

```
c * c + c * c + c * c
```

The value for which the loop body is expanded is called the *loop index*. It can be referenced anywhere in the loop body by the following construct:

```
loop-index ::= "#"|"##"|"<#"[cardinal] [("+"|"-") cardinal]">" .
```

A "#" or a "<#>" will be replaced by the current loop index. If the current loop index is an integer number, an integer displacement may be added or subtracted. Example: "<# + 2>" will be replaced by the current loop index "+ 2". If loops are nested, a "##" in a loop body will stand for the combined loop index, which consists of the indexes of all surrounding loops, concatenated by "_". To access the index of an individual loop, put the loop number (1 for the outmost loop) behind an "#" and put angle brackets around it. For example, to access the index of the 2nd loop, use "<#2>". An example: The loop

```
@<1..3><2,3,5>{CONST A## := <#2 + 2>;}
```

will be expanded to the definitions

```
CONST A1_2 := 4;
CONST A1_3 := 5;
CONST A1_5 := 7;
CONST A2_2 := 4;
CONST A2_3 := 5;
CONST A2_5 := 7;
CONST A3_2 := 4;
CONST A3_3 := 5;
CONST A3_5 := 7;
```

4.2.2. COND and FUNC constructs

These two elements of the MOSEL-2 language were introduced in [Beu03]. FUNCs and CONDs are both called *functions*. They can be used to define complex expressions that would look ugly in the place where that definition is needed. For example, if a firing rate of a rule depends on a node's value in a non-regular fashion, we could define the firing rate by a FUNC, as we do in the following example:

```
FUNC mue_io := IF io = 1 THEN 1 / 28
               ELIF io = 2 THEN 1 / 18.667
               ELIF io = 3 THEN 1 / 15.5553
               ELIF io = 4 THEN 1 / 13.998
               ELIF io = 5 THEN 1 / 13.0668
               ELIF io = 6 THEN 1 / 12.4443
               ELIF io = 7 THEN 1 / 11.99991
               ELIF io = 8 THEN 1 / 11.6669
               ELSE             0;
...
FROM io TO cpu RATE mue_io;
```

The "IF...THEN...ELSE" expression in this example is extended by several "ELIF..." parts. "ELIF" is an abbreviation for "ELSE IF". The value of the whole expression is the value of the first expression whose condition holds. If no condition holds, the expression after the keyword "ELSE" is used. A function may have explicit arguments which are placeholders in the function's definition for an expression that is passed to it when the function is called, as in the following example:

```
FUNC min(a,b) := IF a <= b THEN a ELSE b;
...
FROM n1, n2 TO n3 RATE mue * min(n1, n2);
```

This is equivalent to

```
FROM n1, n2 TO n3 RATE mue * (IF n1 <= n2 THEN n1 ELSE n2);
```

Notice that n1 and n2 (the *actual arguments*) take the positions of a and b (the *formal arguments*) in the definition of min. As one can also see, explicit parameters must be

enclosed in parameters, in the function definition as well as in the function call. Since arguments are always numeric values, we do not need to give an explicit type in the function definition, unlike in most programming languages.

As an example for the usefulness of the MOSEL-2 FUNC and COND constructs in a complex modelling situation, Fig. 4.4 shows a MOSEL-2 description of an UMTS cell which is divided into 4 *virtual zones* (see. [ABBBZ02]). In lines 18-39 of the model FUNCs and CONDs are used to express the topology of the virtual zones of the UMTS cell. An similar model for such a cell in H. HEROLD's MOSEL would have used C code to model the zones.

4.2.3. Reward measure definition

A kind of result specification that is frequently used in availability and performability evaluations are *reward measures*. As it was already mentioned in Sect. 2.2.1, three types of reward measures exist, namely state-, impuls- and path-based rewards. Due to the anonymity of the MOSEL-2 rules, currently only *state-based rewards* are supported, which can be specified in the result part of the MOSEL-2 description. Although reward measures are specified on the syntactic $\mathcal{L}_1$-level, they are functions defined on the states S of the PSLTS on the semantic $\mathcal{L}_2$ layer. Each state $i \in S$ is assigned a real-valued *reward rate* r_i. If the PSLTS is equivalent to a CTMC, the semantic model together with the reward function is called a Markov-Reward-Model (MRM). Two types of reward measures can be defined: A *cumulative* reward sums up the reward during a time interval, whereas an instantaneous reward is measand at a specific moment in time.

4.2.4. The rule preprocessor

At the end of Ch. 3 the principle of PH-approximation as an analysis technique for MOSEL-2 descriptions with *empirical distributed* transitions was mentioned. These type of delay distribution can be expressed in MOSEL-2 using the `EMP (mean, var)` construct, where `mean` or $\bar{t}$ is the mean of the firing time and `var` or σ^2 is the variance of the firing time distribution. The MOSEL-2 program contains a preprocessing component which, according to the values of $\bar{t}$ and σ^2 substitutes each rule of the type

```
FROM p1 TO p2 EMP (mean, var);
```

with one of the following three approximations:

Exponential Substitution ($c^2 = 1$) if the squared coefficient of variation c^2 is equal to 1, the EMP-distribution is simply replaced with an exponential one with rate $\frac{1}{\texttt{mean}}$.

General Exponential Substitution ($c^2 > 1$) if the squared coefficient of variation is greater than 1, a *general exponential* (GE) distribution is substituted. A detailled description of the GE-substitution applied by the MOSEL-2 preprocessor can be found in [Wüc04].

Phase Type Substitution ($c^2 < 1$) If the squared coefficient of variation of the empirical distribution is less than 1, a phase type substitution is applied by the rule preprocessor, where the number r of exponential phases needed is calculated as $r = \left\lceil \frac{1}{c^2} \right\rceil$ (see [Wüc04], pp 66).

```
 1  PARAMETER eps := 0.10, 0.30, 0.40, 0.45, 0.50;
 2  PARAMETER lambda := 0.25, 0.30, 0.35, 0.40, 0.45, 0.50, 0.55, 0.60, 0.65, 0.70;
 3  CONST Z := 4;   // number of virtual zones in the UMTS cell
 4  CONST SF := 32; // service factor
 5  CONST alpha1 := 0.25; CONST alpha2 := 0.25; CONST alpha3 := 0.25; CONST alpha4 := 0.25;
 6  CONST r := 4000; CONST ps := 125.0; CONST no := 0.00000001; // basic noise
 7  CONST pi := 3.14159265358979; CONST l := 0.15;
 8  CONST N  := SF * 2; // spreading factor
 9  @<1..Z>{ CONST N# := 2 * alpha# * N; }
10  CONST mue := 1.0 / 100;
11  @<1..Z>{CONST dd# := (SQRT(#)-SQRT(#-1))/SQRT(Z)*r;}
12  CONST d1 := @<1..1>"+"{dd#}; CONST d2 := @<1..2>"+"{dd#};
13  CONST d3 := @<1..3>"+"{dd#}; CONST d4 := @<1..4>"+"{dd#};
14
15  // Nodes
16  @<1..Z>{ NODE Zone#[N#]; }
17
18  // Functions and conditions
19  FUNC value(z) := (SF - eps * (z - 1));
20  // function to find the maximum distance
21  FUNC max_d(z) := IF value(z) < 0 THEN 0
22        ELSE l / (4 * pi) * SQRT (ps / no * value(z));
23  // the smallest index # for which max_d(x+1) < d<#+1>
24  FUNC find_index (x) :=    IF   max_d(x+1) < d1 THEN 0
25        @<2..Z>{ ELIF max_d(x+1) < d# THEN <#-1> }
26        ELSE  Z;
27  // sum of all zones of index > i
28  FUNC tsum(i) := @<1..Z>"+"{(IF #>i THEN Zone# ELSE 0)};
29
30  COND accept(w) := tsum(find_index(w)) = 0;
31
32  // sum of all zones
33  FUNC zone_sum := @<1..Z>"+"{Zone#};
34
35  @<1..Z>{ COND blocks_# := zone_sum >= N
36                            OR # > find_index (zone_sum)
37                            OR NOT accept (zone_sum); }
38  // Assertions
39  ASSERT zone_sum <= N;
40
41  // Rules
42  @<1..Z>{IF NOT blocks_# TO Zone# WITH lambda*alpha#;}
43  @<1..Z>{IF Zone# > 0 FROM Zone# WITH Zone# * mue; }
44
45  // Results
46  @<1..Z>{ RESULT MZ# := MEAN (Zone#); }
47  @<1..Z>{ RESULT blk# := PROB (blocks_#); }
48  PRINT MT := @<1..Z>"+"{MZ#};
49  PRINT util := MEAN (zone_sum);
50  PRINT blk := @<1..Z>"+"{alpha# * blk#};
51
52  // Pictures
53
54  PICTURE "Blocking"          PICTURE "Utilization"
55  PARAMETER lambda            PARAMETER lambda
56  CURVE blk                   CURVE util
57  XLABEL "call rate"          XLABEL "call rate"
58  YLABEL "blocking";          YLABEL "utilization";
```

Fig. 4.4: A MOSEL-2 description of an UMTS cell (from [ABBBZ02])

After the empirical distributions have been replaced by their substitutes, which contain only exponentially distributed delays, the solution algorithms for CTMC can be applied.

5. The MOSEL-2 Evaluation Environment

A fool with a tool is still a fool!

ANONYMOUS

This chapter is devoted to the MOSEL-2 *evaluation environment* as the second component of our lightweight formal method. The evaluation environment is responsible for the automated calculation of non-functional system properties after the modeller completed his task of providing a formal system specification using the MOSEL-2 FDT. In the following sections, we explain the architecture and the evaluation procedure of the MOSEL-2 evaluation environment schematically and then exemplify its application by means of a transient performance analysis study of a simplified real-world system.

5.1. Structure and Application Flow

The MOSEL-2 evaluation environment consists of an arbitrary text editor, the command line driven MOSEL-2 program and a set of external stochastic modelling packages which are invoked automatically by the MOSEL-2 program during the analysis process. As we have pointed out in section 3.4.3, the different performance evaluation packages provide different solution methods which can be applied to the analysis of diverse problems. In order to be operational, the MOSEL-2 evaluation environment relies on at least one of the external tools to be available. The full power of the MOSEL-2 LFWM is reached if all the stochastic modelling packages mentioned below are installed.

The flowchart in Fig. 5.1 illustrates the automated performance evaluation process with the MOSEL-2 modelling environment. It consists of the following steps:

1. Using his favourite text editor the modeller creates a formal system description in MOSEL-2 including the specification of the performance indices which he wants to be calculated. Recall from chapter 4 that the modeller may specify a variety of system parameter sets in the declaration part of the MOSEL-2 file using the keyword `PARAMETER`. This means that a single MOSEL-2 description may contain the specification of a whole family of stochastic models, which differ from each other only in the actual system parameters. A frequent interpretation in this situation is, that the set of stochastic models and their evaluation constitute an *experiment*, which is carried out in order to determine the influence of the variation in the system parameters on the performance indices. This feature brings about a gain of modelling convenience which should not be underestimated. The MOSEL-2 modelling environment is able to collect the calculated performance indices for all parameter sets in a single file and — if specified — actually creates a concise graphical representation of all the results in one diagram.

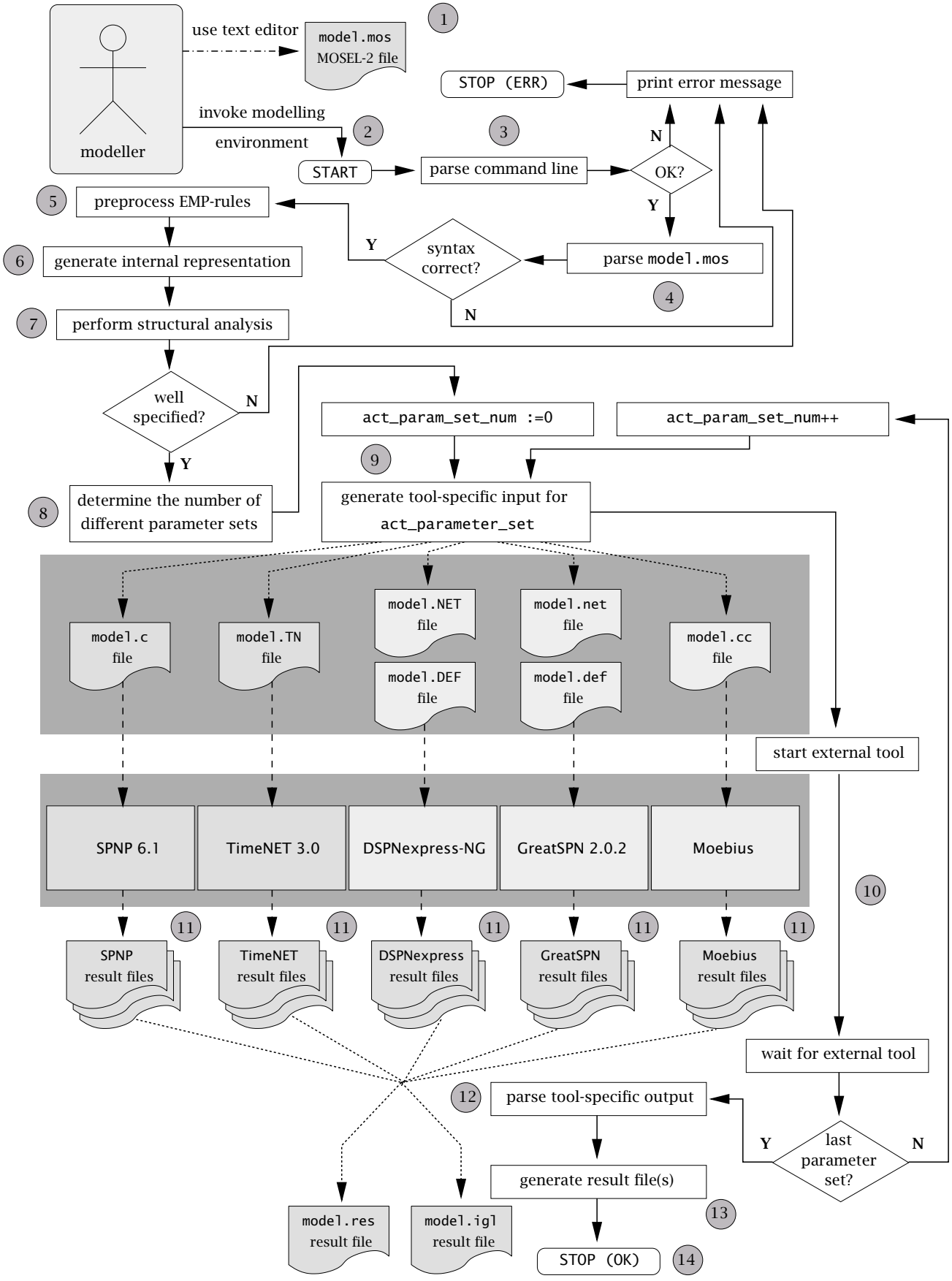

Fig. 5.1: Schematic view of the performance evaluation process using the MOSEL-2 evaluation environment

2. The modelling environment is invoked from the command line where various options can be specified by the user. For a detailed description of the MOSEL-2 command line options the reader is referred to appendix B. The automated performance evaluation process starts.

3. The MOSEL-2 program checks the command line options and opens the MOSEL-2 file (`model.mos`). If either the command line options are wrong or the file `model.mos` could not be opened, the program prints out an error message and stops.

4. A syntactical analysis of the specification is performed. If `model.mos` contains loop-expressions (see Sect. 4.2) they are expanded and re-fed into the parser on the fly. Should the file not conform to the context-free MOSEL-2 grammar (see appendix A) an error message is produced and the execution of the program stops.

5. The preprocessing of rules with empirical (`EMP`) distributions (see sect. 4.2) is performed.

6. The internal representation of the system is created.

7. Additional structural tests (context conditions) are performed on the internal representation. The MOSEL-2 program checks for the type of stochastic process underlying the specification and determines which external tools are able to provide a solution. If none of the installed tools is able to solve the model, an error message is printed out and the program halts.

8. The number of different parameter sets specified in the model is determined as the Cartesian product of all possible parameter values. The analysis of the system will be carried out iteratively for all parameter sets in the following loop.

9. Starting with the first set of parameters, the MOSEL-2 program generates an input file for the targeted external tool, i.e. another description of the system in a specific format as prescribed by the provider of the external package. The current version `2.10` of the MOSEL-2 program is able to generate input for the stochastic Petri net packages TimeNET 3.0 [Zim01] and SPNP 6.1 [Tri00]. Filters to other analysis tools, such as DSPNExpress-NG [LTK+00], GreatSPN [Ber01] or Möbius [CCD+01], are projected and may become available in future versions of MOSEL-2.

10. After the creation of the input file the MOSEL-2 program invokes the selected analysis tool and waits for the completion of the analysis. Note that command line options for the analysis tool can be specified on the MOSEL-2 command line. These options are passed on during the MOSEL-2 program's call of the external package.

11. Upon finishing the analysis for one parameter set, the external tool generates a result file in a customised format.

12. After the analysis for the last parameter set has terminated, the MOSEL-2 program parses *all* tool-specific result files that were created during the analysis loop.

13. The MOSEL-2 program generates the result file `model.res` and also the graphical representation of the results `model.igl` if the source file contained a graphics part.

14. The performance evaluation process with the MOSEL-2 evaluation environment terminates successfully.

After the application flow of the system analysis with the MOSEL-2 evaluation environment has been described, we continue with a discussion of the advantages and potential problems of the chosen architecture:

The design decision to rely on external stochastic modelling and analysis packages for the calculation of the quantitative system properties is motivated by the following considerations:

- A variety of tools exist which are based on extensions of the stochastic Petri net formalism and which offer different solution algorithms which are applicable in specific situations. The approach to integrate many of these tools into a single framework thus yields a versatile *"toolbox"*.

- The implementation of the multitude of analysis algorithms available would be a very time-consuming task. This is especially true for the numerical analysis methods for non-Markovian stochastic models, since many of these techniques are based on a sophisticated mathematical foundation and a sound knowledge of these foundations is needed in order to create a clean re-implementation.

- Some of the available tools have been in use for many years now and have reached a considerable level of maturity, at least with respect to academic software standards.

- The design and implementation of discrete event simulation engines from scratch is difficult and error-prone. Note that for example the development of the TimeNET simulation engine constituted the work of a whole Ph.D. research project [Kel95].

Of course, the advantages of the external tool integration do not come for free. The drawbacks of our approach have their roots in the inconsistencies between the model description file formats used by the existing modelling packages:

- Changes in the model description file format used by an external tool enforce an adaption of the associated translation procedure in the MOSEL-2 program. Fortunately, this problem occurs not frequently, but if it does an adoption of the MOSEL-2 program module which implements the specific translation routine is inevitable. Due to the clean encapsulation of the translation procedures in separate source code modules, a revision of one filter routine will not cause side effects to be corrected in the other modules of the MOSEL-2 program sources.

- To overcome the problems which are caused by the various proprietary model description formats, an XML-based standard for high-level Petri nets was adopted by the ISO/IEC recently [Int04]. Unfortunately, we are not aware of any widespread tool which supports this standard at the moment. On the other hand, a further development of the two packages SPNP 6.1 and TimeNET 3.0 which can be regarded as the analysis

"backbone" of the current MOSEL-2 evaluation environment is not very likely, since all the past developers of these packages have left their research groups in the meantime. This implies that the corresponding translation filters of the MOSEL-2 program have reached a stable state too.

All in all we are convinced that despite of the problems caused by the lack of standardisation in the area of stochastic modelling, the integrative approach used in the design of the MOSEL-2 evaluation environment is the right decision if the primary area of application of the method is the research on its potential for technology transfer. In an industrial setting however, an efficient, scalable and well-tested re-implementation of the most frequently used solution algorithms with a direct integration into the MOSEL-2 program may become desirable.

5.2. A Simple Analysis Example

In order to exemplify the performance evaluation process with the MOSEL-2 evaluation environment, we present a simple example of a transient performance evaluation of a border control station, which is described in [Beu03] informally as follows: Imagine a border control station at which trucks are arriving at an average rate of five trucks per hour, regardless of the time of day. The border is open 16 hours a day. We want to know how the queue length of waiting trucks evolves over time if the clearance rate at which the trucks are served at the station varies from 4 to 10 trucks per hour. We do not know the exact probability distribution of the truck inter-arrival times, so we assume the time span between two trucks arrivals is exponentially distributed. The same type of distribution is assumed also for the clearance process. The MOSEL-2 specification `border.mos` of this system is presented in Fig. 5.2.

Lines 1–5 constitute the parameter declaration part of the model. The two components of the system, namely the queue for the trucks and the node modelling the state of the border control station are declared in lines 6 and 7. The exponentially distributed truck arrival process is modelled by the rule in line 8. The departure process of trucks, which is enabled only if the the control station is open, is captured in line 9. The deterministically timed switching of the control station from state 'open' to state 'closed' and vice-versa is represented by the lines 10 and 11. Line 12 determines that the system should be evaluated at time instant 2 hours up to 72 hours every two hours. Lines 13 and 14 define two performance indices, namely the mean truck queue length and the probability for a full queue, for which the results will be stored in the MOSEL-2 result file `border.res`. Lines 15–24 contain the definition of two graphical representations of the results, from which plots will be generated by the MOSEL-2 program during the evaluation process and stored in the file `border.igl`.

Since the stochastic process underlying the border control system belongs to the class of continuous time semi-Markov processes (Sect. 3.4.4), it can be evaluated using the transient numerical solution algorithm of the tool TimeNET. The MOSEL-2 program is thus invoked by issuing

```
mosel2 -Ts border.mos
```

```
(1) CONST arrival_rate := 5;
(2) CONST hours_open := 16;
(3) CONST max_length := 100; // maximum length of the queue
(4) PARAMETER clearance_rate := 4..10;
(5) ENUM state := open, closed;
(6) NODE queue[max_length] := 0;
(7) NODE border[state] := open;
(8) FROM EXTERN TO queue RATE arrival_rate;
(9) IF border = open FROM queue TO EXTERN RATE clearance_rate;
(10) FROM border[open] TO border[closed] AFTER hours_open;
(11) FROM border[closed] TO border[open] AFTER 24 - hours_open;
(12) TIME 2..72 STEP 2;
(13) PRINT mean_queue_length := MEAN(queue);
(14) PRINT prob_queue_full := PROB(queue = max_length);
(15) PICTURE "mean queue length"
(16) PARAMETER TIME
(17) CURVE mean_queue_length "$clearance_rate trucks/h"
(18) XLABEL "time [h]"
(19) YLABEL "trucks";
(20) PICTURE "queue full"
(21) PARAMETER TIME
(22) CURVE prob_queue_full "$clearance_rate trucks/h"
(23) XLABEL "time [h]"
(24) YLABEL "P(queue full)";
```

constant and parameter decl. part

node part

rule part

time declaration

result part

Fig. 5.2: `border.mos` — a MOSEL-2 model for a border control station

on the command line. The option `-T` tells the MOSEL-2 program to translate the system description to the `.TN` input format of TimeNET and '`s`' causes the evaluation to be started automatically. According to the 7 clearance rate and 36 time parameters specified in the model, we get a total of $7 \cdot 36 = 252$ different parameter sets for which the system is evaluated by TimeNET in the main analysis loop of the MOSEL-2 program. After the last analysis has terminated, the MOSEL-2 program parses the 252 TimeNET result files and generates the files `border.res` and `border.igl`. The textual results in `border.res` are presented in a tabular form which we are not going to show here. Since for a transient evaluation the graphical representation which shows the evolution of the performance indices over time is more informative, we show the two plots for the mean truck queue length and the probability of having a full queue in Figs. 5.3 and 5.4. These plots have been generated from the values in the file `border.igl` by the graphics tool IGL which H. HEROLD created as part of his thesis [Her00].

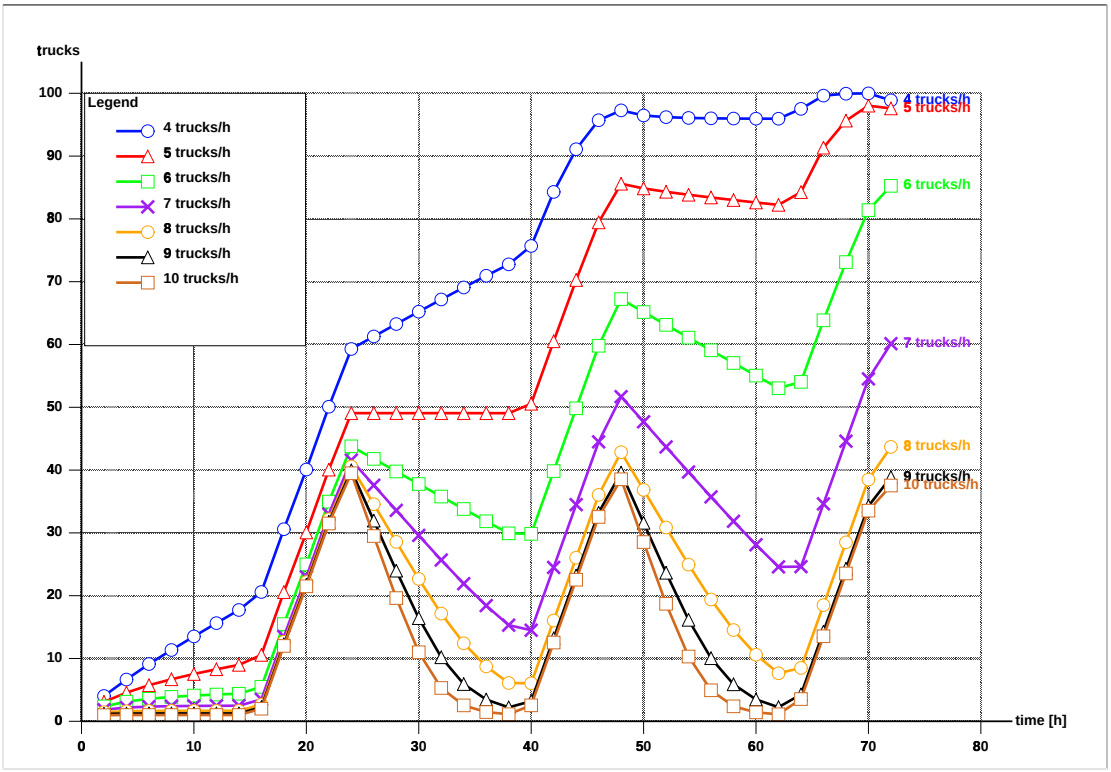

Fig. 5.3: Evolution of the truck waiting line length over time

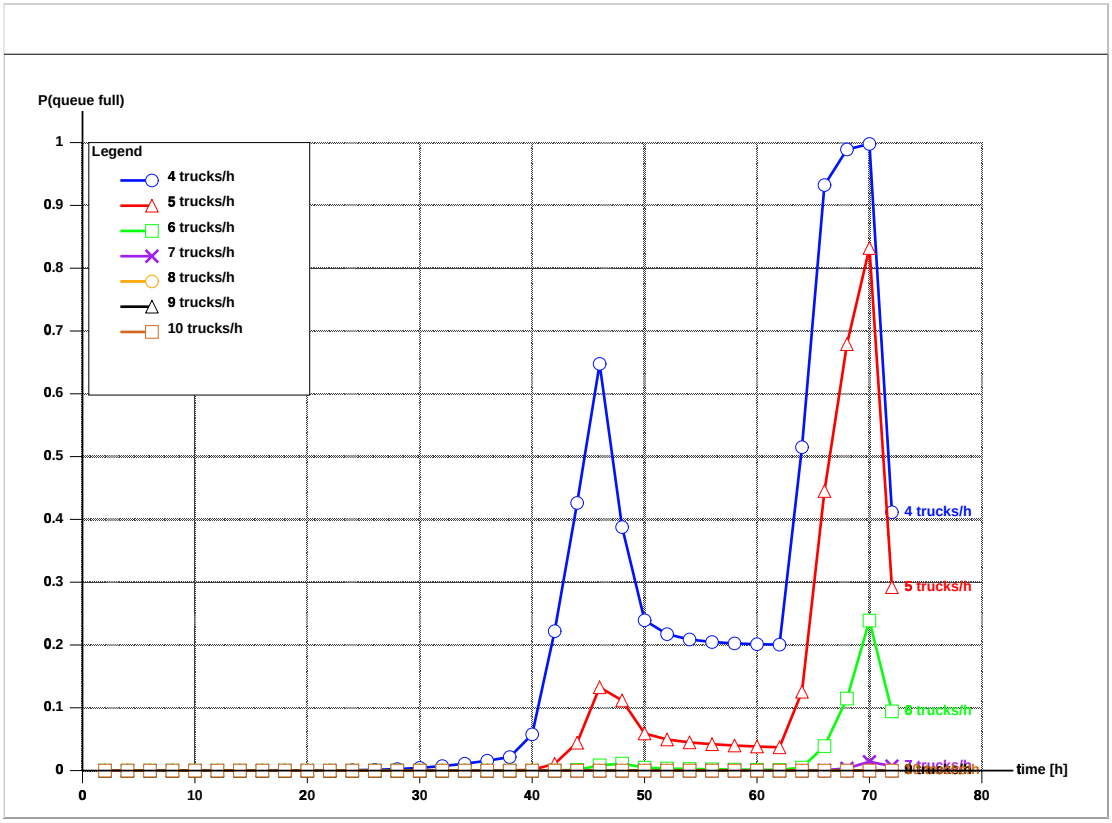

Fig. 5.4: Time progression of the probability for a full queue

6. Related Work

> *Promise of yet further benefit is obtained by accumulation of tools, libraries, theories, and case studies based on the work of scientists from many schools which were earlier considered as competitors.* C.A.R HOARE [BC98]

This chapter introduces a couple of languages and tools which are related to the MOSEL-2 method. Sect. 6.1 contains a synopsis of related work in a succinct form, whereas in Sect. 6.2 a comparative assessment of MOSEL-2 with the two most closely related approaches is presented.

6.1. Related Tools and Languages

The projects listed below are either stochastic FDT/tool combinations which feature dedicated solutions for specific problem classes or constitute larger frameworks in which stochastic modelling is combined with verification methods for other types of system properties:

MoDeST The MoDeST (Modelling and Description language for Stochastic and Timed systems) FDT [DHKK01] and it's associated analysis environment MoToR (Modest Tool EnviRonment) [BHKK03] was developed by the "Formal Methods and Tools Group" of the University of Twente. The MoDeST language supports the modular description and analysis of reactive systems covering both functional and nonfunctional such as hard and soft real-time, and stochastic performance and reliability aspects. MoDeST is based on the process-action based modelling paradigm of stochastic process algebras. The syntax of the language is similar to that of the programming language C and the process meta language PROMELA — the input language of the model checker SPIN [Hol97]. The formal semantics of MoDeST is given in terms of probabilistic and stochastic timed automata [DHKK01] which can be regarded as the union of a set of related semantic model types $\mathcal{U}_i$ in a model-theoretic formal system setup (see Sect. 3.2.1). The *semantic variation* is deliberately chosen by the MoDeST developers and is also reflected by the architecture of the analysis environment MoToR: It aims at supporting a variety of analysis algorithms tailored to the different kinds of semantic submodels of MoDeST. The basic idea of MoToR is to connect MoDeST to existing tools, rather than re-implementing existing analysis algorithms anew. Analysis takes place by extracting — possibly several — semantic submodels from a MoDeST system specification and let MoToR invoke appropriate external tools for the evaluation of different property types. For instance, for checking reachability properties, a possible strategy is to "distill" an automaton from the MoDeST specification and feed it into

an existing model checker such as provided by the CADP toolkit [Gar98]. On the other hand, for carrying out an evaluation of the stochastic process underlying a MoDeST specification, the discrete-event simulator of the Möbius tool can be used. The application areas of MoDeST/MoToR include modelling and analysis of embedded system software [KBKH04]), scheduling of tasks in production systems [BHK+04] or reliability modelling and analysis of train radio communication systems [HJU05].

Möbius: The Möbius Tool [CCD+01] was developed by G. CLARK et al. at the University of Illinois at Urbana-Champaign. It is inspired by previous work on UltraSAN, which was developed to model and solve performance and performability problems based on the stochastic activity network (SAN) [SM00] formalism (see below). Möbius is broader in scope than UltraSAN and aims to provide support for many different modelling formalisms. These include traditional performance modelling paradigms such as networks of queues and stochastic Petri nets, newer approaches such as stochastic process algebras (SPAs) and stochastic automata networks [PA91], and pure probabilistic approaches such as *fault trees* [Wat61] and combinatorial block diagrams. The Möbius design goal is extensibility, which means that it should be possible to add new modelling formalisms and, to a large degree, that they should be able to interact with existing formalisms and solvers without requiring that the tool undergoes any changes. Extensibility also means that it should be easy to add new model solvers so that, to the extent theoretically possible, they can be employed to solve models built with existing and future formalisms. The Möbius Tool was recently applied in the area of system security validation [SCS+04] and in a dependability analysis of a satellite-based communication network [ATS05].

EMPA$_{gr}$ and TwoTowers: This is a process algebra based language and tool for the combined/alternative analysis of performance and real-time properties developed by M. BERNARDO at the University of Bologna. It is based on a FDT with a precise definition of the syntactic and semantic layers [BCSS98, Ber02].

SMART: Which originally was named "Simulation and Markovian Analyzer for Reliability and Timing" now "Stochastic Model checking Analyzer for Reliability and Timing" was developed by G. CIARDO at the College of William and Mary and is based on the stochastic Petri net modelling formalism [CRMS01]. It contains sophisticated MTBDD-based storage techniques and solution methods for the generation and analysis of performance and reliability models with large state space.

ETMCC: The Erlangen-Twente Markov Chain Checker was developed by JOACHIM MEYER-KAYSER at the University of Erlangen-Nünberg [MK02]. It features stochastic model-checking, i.e. automatic combined verification of temporal and non-functional system properties. The system can give answers to questions of the type "does the system fulfil a performance requirement which is related to a particular functional behaviour?", e.g. "Is the probability that a `send` message is answered by an `ack` message within 50ms greater than 95% ?".

PRISM: This tool was developed by M. KWIATKOWSKA et al. at the University of Birmingham [KNP02]. Similar to ETMCC, it also features a combined verification of functional and nonfunctional system properties. Moreover, verification methods based on Markov decision processes are provided.

DSPNExpress-NG: This stochastic Petri net tool was developed by CH. LINDEMANN at the University of Dortmund [LRT99]. It includes sophisticated numerical solution algorithms for certain classes of GSMPs, for which usually only the discrete event simulation is applicable. Another feature of the DSPNEXPRESS-NG tool are the filters to the commercial UML design packages RationalRose and Together. Performance enhanced UML specifications can be interpreted as extended stochastic Petri nets and used as alternative modelling formalism. DSPNexpress–NG is a hot candidate for a forthcoming integration into the MOSEL-2 evaluation environment.

UltraSAN: Is a Stochastic Activity Network [SM00] based tool which was developed by W.H. SANDERS at the University of Illinois in Urbana-Champaign. An interesting feature of the SAN formalism are the replicate/join operators which allow to reduce the size of a model in a similar way as it is possible with the MOSEL-2 loop-construct.

GreatSPN: Is a SPN-based tool which was developed by S. BERNARDI et al. at the University of Torino [Ber01]. It features a compositional specification method based on Stochastic Well-Formed Nets (SWN) as a subclass of Petri Nets. GreatSPN is also an interesting candidate for an integration into the MOSEL-2 evaluation environment.

6.2. A Comparative Assessment with MOSEL-2

Among the projects mentioned above MoDeST/MoToR and Möbius are the most interesting candidates for a comparative assessment with the MOSEL-2 LWFM. Like the MOSEL-2 LWFM they are fully formal methods since they comprise a FDT with a formal syntax and semantics as well as a tool which performs the automated verification of system requirements. The similarities and differences of the MoDeST/MoToR, Möbius and MOSEL-2 approaches are discussed with respect to the following aspects: method scope, FDT expressiveness, abstraction and modularity and tool architecture. The section is concluded by a discussion to which degree the three approaches follow the recommendations of the LWFM definition (see Sect. 3.2.4, Def. 3.10).

Method scope: In contrast to MOSEL-2 which aims at the verification of non-functional, performance/reliability related system properties only (see Sect. 2.2.1), MoDeST/MoToR and Möbius can be also employed for the verification of other classes of system properties. A MoDeST specification, for example, can be evaluated by MoToR for real-time or combined non-functional/real-time properties. Möbius was originally designed as an extensible tool for performance and reliability modelling but recent developments aim also at the analysis of functional system properties.

FDT expressiveness: Due to the differences in the scope of the methods and the underlying modelling paradigm, the language constructs of the FDTs differ considerably.

MoDeST and MOSEL-2 are both textual FDTs but are based upon different conceptualisations. MoDeST, is based on the process/action modelling paradigm and has a pragmatic syntax which offers constructs to capture various aspects of concurrent process behaviour: Nondeterminism of actions can be expressed by the `alt`-construct, the keyword `palt` denotes probabilistic branching, and parallel composition of processes can be specified with the `par`-construct. Stochastic delay of a process action is specified via a random variable declaration associated with the action. Moreover, MoDeST contains language elements for the specification of real-time behaviour: *clock variables* are declared by the keyword `clock`, and process actions can be equipped with a *deadline guard*. An action guarded by `urgent(B)` must be executed as soon as the boolean expression `B` becomes true. MoDeST also offers constructs for exception handling and the definition of invariants, which can be used to express safety requirements. MoDeST supports basic data representation through `int` and `float` variables which can be used to model the state of a process component or to store a value representing a system requirement. In comparison to MoDeST the MOSEL-2 FDT (see Ch. 4) offers fewer language constructs. With the exception of the `NODE` declarations, data modelling aspects are missing in the MOSEL-2. A strong point of MOSEL-2 is the flexible and pragmatic representation of synchronisation. Complex synchronisation conditions can be modelled via `COND` and `FUNC` declarations and used subsequently in the `IF`-parts of the MOSEL-2 rules. Stochastic delay information can be included via a set of continuous and discrete probability distribution types (e.g. `RATE`, `AFTER`) and probabilistic branching is modelled with the `WEIGHT` construct. Moreover, the `ASSERT` keyword offers basic support for the specification of invariant conditions of the node population. Within the Möbius framework so-called *atomic* models can be specified using various FDTs, such as the *Buckets and Balls* [Ste94], stochastic activity network (SAN) [MM84], PEPA [Hil94a] and the MoDeST language.

For the specification of non-functional system requirements all approaches offer means to express the interesting properties as *reward measures* (see Sect. 2.2.1, page 24). Whereas in MOSEL-2 currently only state-based rewards can be specified, Möbius and MoDeST are capable to express state- and impulse-based rewards. Within the Möbius reward model also *path-based* reward measures (see [OS98, Voe02]) can be defined.

Modularity: At present MOSEL-2 specifications are monolithic, but using the MOSEL-2 loop-construct a concise and comprehensive specification of a complex system can be achieved (see Sect. 7.2). Identic parts of the specification are “folded” into a loop-construct (see Sect. 4.2.1). Since the MOSEL-2 loops are expanded before the evaluation, the advantages that the model symmetries offer with respect to more efficient solution methods are not exploited. MoDeST features the inherent compositionality of process algebraic languages in which complex systems can be modelled as the parallel composition of sub-process terms. A distinguished feature of the Möbius framework is the possibility to model a complex system as a so-called *composed* model, which is expressed as a set of *atomic* models. The properties of interest that have to be evaluated for the composed model are specified in a separate *reward model*.

Tool architecture A commonality in the tool architecture of all three approaches is that external solution algorithms which are integrated in the analysis environments can be invoked to perform the actual verification process. While MoToR and the MOSEL-2 evaluation environment exclusively rely on external solvers, the Möbius environment is also equipped with proprietary solution methods. These — in turn — can be exploited by other modelling frameworks, such as the distributed Möbius DES engine is used by MoToR to evaluate performance and dependability properties of systems modelled in MoDeST. The tool architectures differ in the design of the interface between the evaluation environment and the external tools. Möbius implements an *abstract functional interface* (AFI) [Doy00] as well as an *state level AFI* [DKSC02] which can be used by developers of other tools to connect their FDT to the Möbius system either at specification or state-space level. Also MoToR provides two programming interfaces, the *first next state* and the *abstract syntax tree* APIs, via which other software components can be connected to the MoDeST specification "frontend" (see [BHKK03]). In contrast to Möbius and MoToR the connections of the MOSEL-2 tool to external solvers are not implemented as a publicly available interface specification but instead as a set of internal filters of the MOSEL-2 tool binary. As a consequence thereof, the MOSEL-2 developers are responsible to reflect changes of the external tool input format in the internal filter modules, which was necessary, e.g., after the transition from version 3.1 to 6.0 of SPNP. With the interface-based approach of Möbius and MoToR this work is left to the developers of external tools.

Conclusions: the comparative assessment of the three projects shows that MOSEL-2 strictly follows the four recommendations of the lightweight formal method definition 3.10, whereas in MoDeST/MoToR and Möbius the broader scope of the methods result in a limited implementation of the LWFM ideas:

Partiality in language: The FDTs of all three approaches have been designed towards amenability to automated analysis by tools and therefore fulfil the "partiality in language" recommendation. The MoDeST and Möbius FDTs offer more modelling primitives which enables the solution of a wider range of problems. For this reason the set of language constructs is larger than recommended for a strict lightweight formal method.

Partiality in modelling: The application of a single atomic model of Möbius, which takes a *multi-formalism*, *multi-goal* approach, conforms well to the lightweight definition. As soon as a composed Möbius model is used, the "partiality in modelling" recommendation is violated, since the modeller has to get acquainted to more than one specification language. The learning curve required for the application of MoDeST and Möbius thus may be too steep for the average novice user. Instead, they are powerful methods for experts with a solid knowledge of the underlying theories of functional and nonfunctional system verification. In contrast, MOSEL-2 is based on three important modelling concepts only, namely *enabling condition*, *stochastic delay* and *probabilistic branching* which are expressed in a pragmatic way within the RULE modelling primitive. Although the descriptive power of MOSEL-2 is lower than that of MoDeST or

Möbius, the shallow learning curve is very attractive for an engineering practitioner if he is interested in the verification of nonfunctional system properties only.

Partiality in analysis: The structure of the underlying formal systems in all three approaches conforms to the model-theoretic style in which automated semantic reasoning about system properties is performed based on finite relational structures. None of the three methods follows a proof-theoretic approach of reasoning with the goal of proving the total correctness of the specified system. Consequently, MoDeST/MoToR, Möbius and MOSEL-2 all fulfil the "partiality in analysis" requirement.

Partiality in composition: All approaches conform to this requirement, since they can be applied side by side with other formal or semiformal description methods during the system development life cycle.

7. Case Studies

The best way to sell a mouse trap is to display some trapped mice. Trapping real mice also shows you how a trap can be improved. DAVID L. PARNAS *[Par96]*

7.1. Modelling a single GSM/GPRS Cell with Delay Voice Calls

In this section we will use MOSEL-2 to evaluate a non-Markovian model of a single GSM/GPRS cell with Delay Tolerant Voice Calls (DeTVoC). This case study, which uses many of the new language features of MOSEL-2 was first presented in [WABBB04].

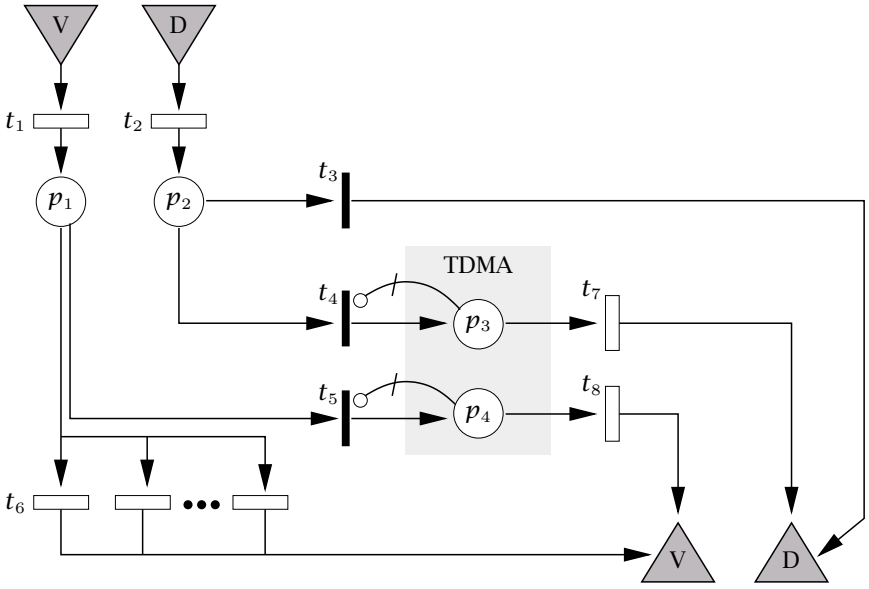

Fig. 7.1: DSPN model of a GSM/GPRS air interface

7.1.1. Informal real-world system description

We consider the air interface of a single GSM/GPRS cell with a maximum of 100 users and a single carrier frequency. This carrier frequency is split up into eight TDMA channels. Every single channel can be used for a single circuit switched voice or data over GSM connection or for one or several packet switched GPRS data connections. A single TDMA channel (without EDGE technology) can serve a gross data rate up to 21.4 Kbit/s [EMS+02]. A single GPRS user can use a maximum of eight TDMA channels simultaneously and therefore can achieve a maximum gross data rate of 171.2 Kbit/s. Every GPRS data connection is identified by a temporary flow identity (TFI) of 5 bit length which limits the number of simultaneous GPRS connections to 32. We assume that one TDMA channel is reserved for GPRS traffic. All

other TDMA channels can only be used by the GPRS system if they are not used by higher priority GSM voice (or data) connections. If a voice call cannot be served, the connection attempt is not rejected immediately but the user will wait up to a maximum of 10 seconds before giving up. This behaviour is called Delay Tolerant Voice Call (DeTVoC). If a data burst cannot be served, it gets rejected immediately and the transmission needs to be tried again later. We assume that voice calls arrive following a Poisson process with a rate of 1/100 per second and finish with a rate of 1/80 per second. For data traffic we assume an exponentially distributed burst rate of 0.1 to 2.5 bursts per second and a geometrically distributed burst size with a mean of 10 KB.

7.1.2. Conceptual Model

Figure 7.1 shows a Petri Net (DSPN) that describes the real system following the token-flow network-oriented modelling paradigm. In the following description times and rates refer to seconds:

t_1: Poisson voice call arrival process: exponential distributed transition with firing rate $\lambda_v = 1/100$.

t_2: Poisson data burst arrival process: exponential distributed transition with firing rate $\lambda_d = 0.1 \ldots 2.5$.

t_3: Packet loss: immediate transition with priority 1 (low).

t_4: Packet admission: immediate transition with priority 2 (high). and inhibitor arc (multiplicity 32, because of TFI).

t_5: Call admission: immediate transition with inhibitor arc (multiplicity 7, because one channel is reserved for packet service) and priority 2 (high).

t_6: DeTVoC: IS (infinite server) transitions with deterministic firing time of 10, marking dependent enabling functions and priority 1 (low).

t_7: Packet service: PS (processor sharing) using exponential distributed transition with firing rate $\mu_d = \frac{21.4}{8 \cdot 10} \cdot (8 - p4)$

t_8: Voice service: IS using exponential distributed transition with firing rate: $\mu_v = \frac{1}{80} \cdot p_4$.

p_1: place with capacity 100 for incoming (and waiting) voice connection requests.

p_2: place with capacity 1 for incoming data connection requests.

p_3: place with capacity 32 for bursts in service.

p_4: place with capacity 7 for voice connections in service.

7.1.3. MOSEL-2 description

Below, a description of the GSM/GPRS air interface in MOSEL-2 is given:

```
01 /* PARAMETER AND CONSTANTS *********************/
02 PARAMETER lambda_d := 0.1, 0.5 .. 2.5 STEP 0.25;
03 CONST lambda_v := 1/100;
04 CONST mue_single_v := 1/80;
05 CONST max_users_v := 100;
06 CONST chan_res_d := 1;
07 CONST waiting_time := 10;
08 CONST mue_single_d := (21.4/8) / 10;
09
10 /* NODES ***************************************/
11 NODE p1[max_users_v] := 0;
12 NODE p2[1] := 0;
13 NODE p3[32] := 0;
14 NODE p4[8-chan_res_d] := 0;
15
16 /* TRANSITIONS *********************************/
17 FROM EXTERN TO p1 RATE lambda_v;              //t1
18 FROM EXTERN TO p2 RATE lambda_d;              //t2
19 FROM p2 TO EXTERN PRIO 1;                    //t3
20 FROM p2 TO p3 PRIO 2;                        //t4
21 FROM p1 TO p4 PRIO 2;                        //t5
22 @<1..max_users_v>{                          //t6
23 IF p1 >= # FROM p1
24 TO EXTERN
25 AFTER waiting_time
26 PRIO 1;
27 }
28 FROM p3 TO EXTERN RATE mue_single_d * (8-p4); //t7
29 FROM p4 TO EXTERN RATE mue_single_v * p4;     //t8
30
31 /* RESULTS *************************************/
32 // TEXTUAL (.res)
33 PRINT d_loss := PROB (p3 == 32);
34 PRINT v_block := UTIL (p1);
35
36 // GRAPHICAL (.igl)
37 PICTURE "prob. of data loss and voice blocking"
38 PARAMETER lambda_d
39 XLABEL "incoming burst rate"
40 YLABEL "PROB"
41 CURVE d_loss
42 CURVE v_block
```

Lines 22 to 27 show the powerful loop construct of MOSEL-2. This preprocessor expression is introduced by the special character "@" followed by the range list(s) enclosed in angle brackets ("<>") and by the body in curly brackets ("{}"). The body may contain the special character "#" that will be exchanged with the current range list item. Loops can be used anywhere in a MOSEL-2 model description. The loop above is constructing the infinite server transitions of t_6 and will be expanded like this:

```
IF p1 >= 1 FROM p1 TO EXTERN AFTER waiting_time PRIO 1;
IF p1 >= 2 FROM p1 TO EXTERN AFTER waiting_time PRIO 1;
[...]
IF p1 >= 100 FROM p1 TO EXTERN AFTER waiting_time PRIO 1;
```

As one can see, loops are very useful to combine several code parts with similar structure. This keeps the model description concise and saves the user from excessive typing. The IF

parts of this rules describe the state dependent enabling function of each rule, i.e. for every single token in p_1 a deterministic transition gets enabled.

7.1.4. System evaluation

We invoke the MOSEL-2 evaluation environment on the command line using the option `+simulation` to evaluate the non-Markovian model with SPNP6.1's discrete event simulation:

```
mosel2 -cso +simulation gsmgprs.mos
```

Below a compressed form of the result file `gsmgprs.res` is given:

```
Simulation of "gsmgprs.mos" by SPNP
(length: 1000, runs: 1000)
Parameters:           | Parameters:
lambda_d = 0.1        | lambda_d = 0.5
Results:              | Results:
loss_prob_d = 0       | loss_prob_d = 0.001
block_prob_v = 0      | block_prob_v = 0.00200001
[...]
Parameters:           | Parameters:
lambda_d = 2.25       | lambda_d = 2.5
Results:              | Results:
loss_prob_d = 0.186   | loss_prob_d = 0.303
block_prob_v = 0.001 | block_prob_v = 0.00200001
```

In Fig. 7.2 an IGL-generated plot of the voice-blocking and data-loss probabilities is shown. One can see that the data loss probability increases fast with rising data arrival rate whereupon the probability of blocked calls of course is almost zero. Assuming, that a data loss of 5% can be tolerated, one can determine the maximum arrival rate of data bursts as 1.65 data bursts per second.

Now we will keep the arrival rate of data bursts constant at 1.5 bursts per second and raise the arrival rate of voice calls from 0.001 to 0.1 calls per second. The results are shown in Fig. 7.3.

The voice call blocking probability is rising fast and is allocating more and more TDMA channels. Therefore, the data-loss probability is rising too. Assuming that a blocking probability for voice calls of 1% is still convenient, we see that a reservation of further TDMA-channels for data transmissions would not be useful in this configuration, because the limit of the voice call blocking is exceeded earlier than the limit of the data loss. It would be even better to restrict the data traffic by using the reserved data channel for voice. This could be achieved with the "delay of voice end user" (DOVE) concept presented in [MEC01]: If only one channel is available, a new voice call is delayed for a certain amount of time before it is given permission to occupy the remaining channel. If another channel becomes available before the delay expires, the voice user is given permission to use this other channel. The effect of delaying the voice call is that the system stays in a state with an available channel for a greater proportion of time which — according to the simulation and Markov analysis presented in [MEC01] — reduces the bursty queue length of data packets in the queue.

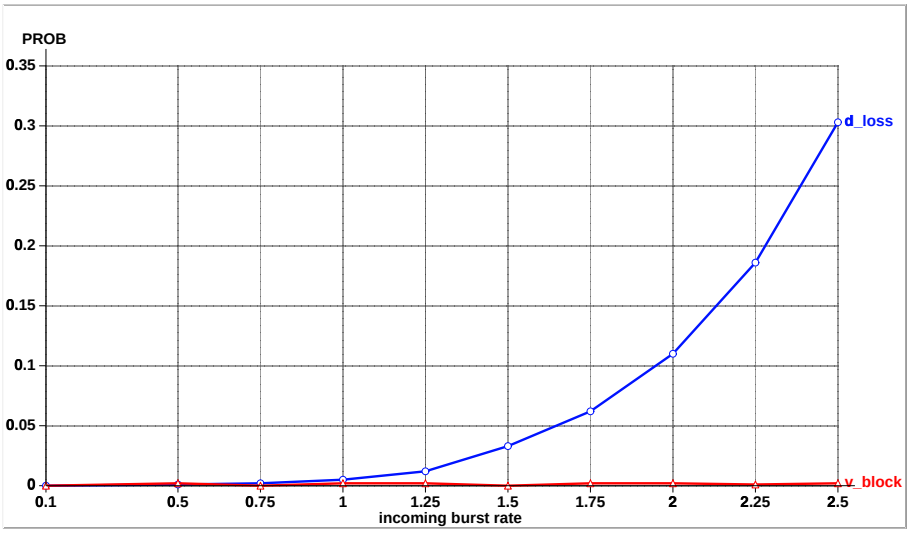

Fig. 7.2: Voice-blocking and data-loss over burst rate

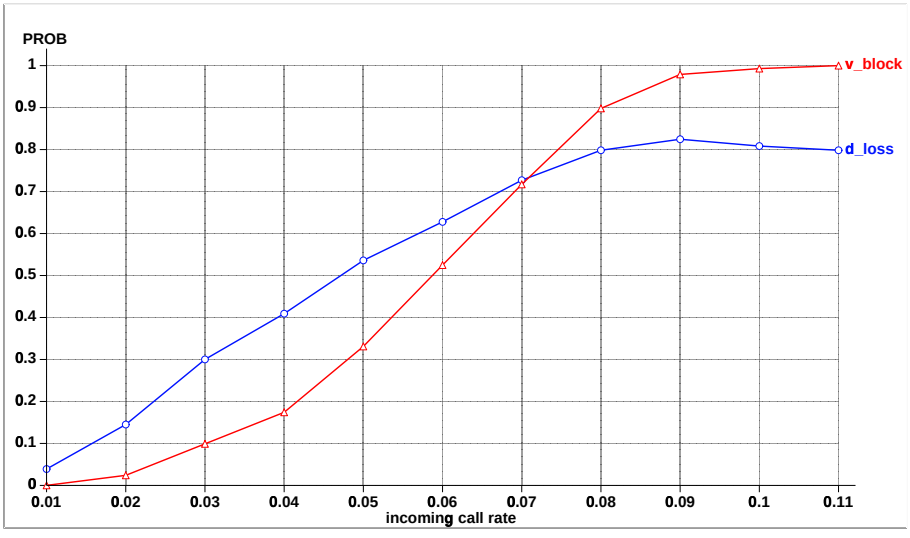

Fig. 7.3: Voice-blocking and data-loss over call rate

7.2. Performance Model of a WLAN-System

In this section we present a detailed performance study of a wireless local area network (WLAN) which is compliant to the IEEE 802.11 standard [Dep99]. In this 520 page document the media access control (MAC) and physical (PHY) protocol layers for WLANs operating in the 2.4-2.5 GHz band are specified in detail. A concise treatment of the main features of the IEEE 802.11 standard can be found in CROW [CWKS97].

The reasons for choosing WLAN systems as an application area of the MOSEL-2 method are manifold:

- Real existing system: The modelling of the algorithms needed for representing the MAC function exactly as specified in 802.11 makes use of many of the advanced MOSEL-2 features.
- No toy example: The level of detail used here indicates the maximum level at which a real system can be expressed in MOSEL-2, but this is exactly the level for which the applicability was promised in the introduction: MOSEL-2 should enable the modeller to obtain a feedback early in the system design process, thereby using as much information about the system as needed.
- Actuality: The (performance) analysis of wireless LAN systems is an actual field of research, as the task group E of the IEEE 802.11 working group is currently working on an extension of the IEEE 802.11 standard, called IEEE 802.11e. The goal of this extension is to enhance the access mechanisms of IEEE 802.11 and to provide a distributed access mechanism that can provide *service differentiation* [IEE99].
- Results can be compared: The analysis presented here is inspired by the studies of A. HEINDL and R. GERMAN in [HG01], [HG00], [GH99]. Many of the results in these contributions were calculated by the discrete event simulation engine of the Petri Net based tool TimeNET 3.0, which is also used as the solution engine for the MOSEL-2 model presented here.

7.2.1. System Architecture and Working Modes

The system architecture of a IEEE 802.11 compliant wireless LAN conforms to one of the two types shown in Fig. 7.4. The network on the left consists of several cells, called *basic service area* (BSA) in 802.11 terminology, in which a number of wireless stations communicate with an *access point* (AP) over the wireless channel. The APs of all BSAs are connected via an *interconnection network* – usually a wired LAN as Ethernet. Communication between stations belonging to different BSAs is achieved by routing packets via the APs and the interconnection network. This configuration is thus named an *extended basic service set* (EBSS) which operates in *infrastructure mode*. The small network on the right side of Fig. 7.4 consists of three stations which can communicate with each other via the radio channel, no access point and wired infrastructure are needed. This configuration is called an *independent basic service set* (IBSS) or *ad hoc* network, since networks of this type can be set up

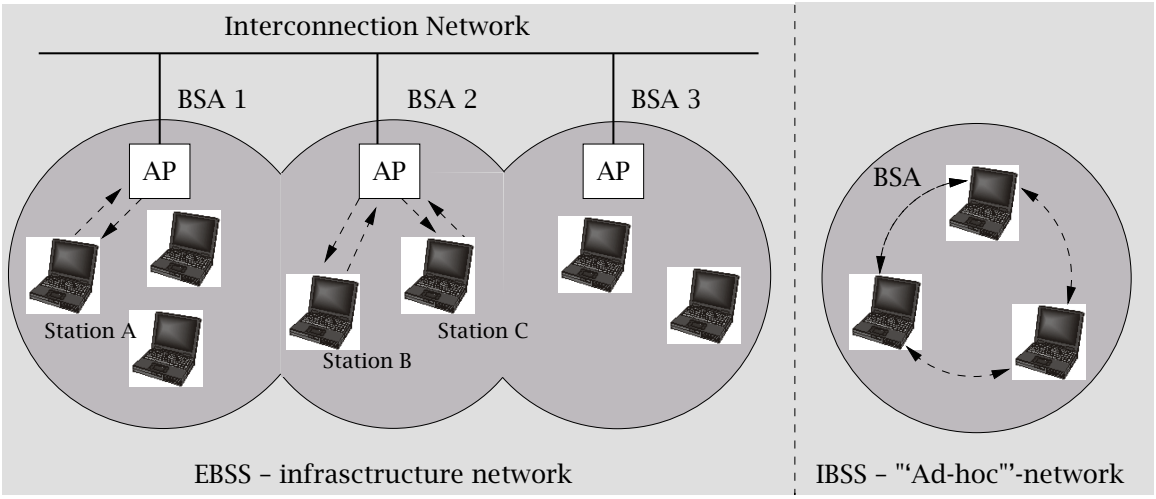

Fig. 7.4: Infrastructure mode vs. Ad-hoc mode

easily by placing stations within the coverage area of their antennas. A typical situation for the occurrence of an ad hoc network are conferences when several participants put their laptops on a table.

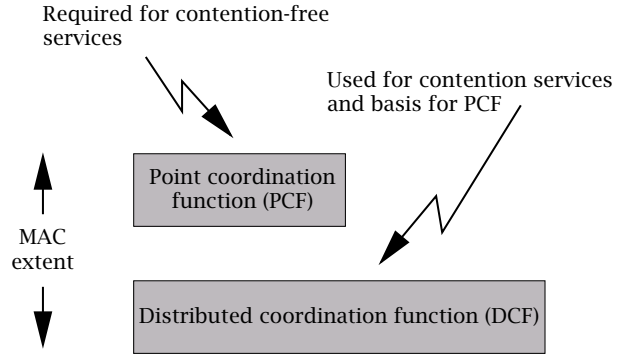

Fig. 7.5: MAC architecture

The 802.11 standard defines two access methods for the wireless medium at the MAC (media access control) sub-layer (see Fig. 7.5). The fundamental access method is the *distributed coordination function* (DCF), the second, optional *point coordination function* (PCF) can be used only in infrastructure mode since the centralised medium access has to be controlled by an access point. Within an ad hoc network only the DCF can be used.

Carrier-sense mechanism: A physical carrier-sense mechanism shall be provided by the physical layer (PHY). A *virtual carrier-sense* mechanism is included in the MAC-layer of each station. This mechanism is referred to as the *network allocation vector* (NAV). The NAV maintains a prediction of future traffic on the medium based on duration information that is announced in RTS/CTS frames prior to the actual exchange of data. The carrier-sense mechanism combines the NAV state and the STA's transmitter status with physical carrier sense to determine the busy/idle state of the medium. The NAV may be thought of a counter, which counts down to zero at a uniform rate.

The Medium access in DCF mode using the two-way handshaking of the basic access BA

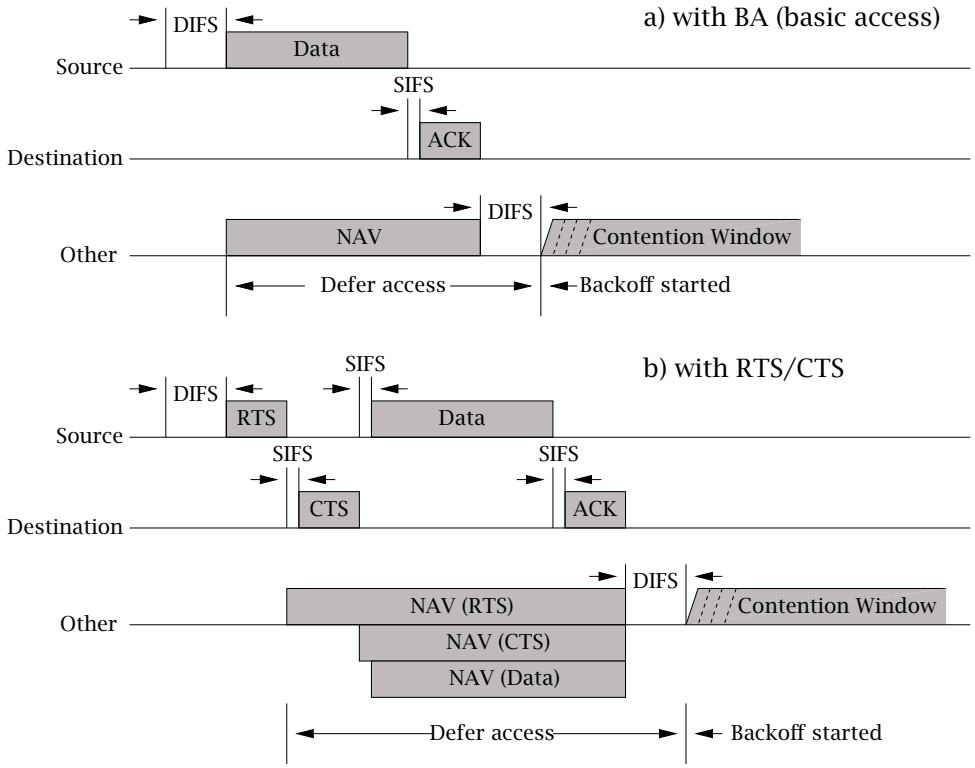

Fig. 7.6: Transmission of an MDPU using BA and RTS/CTS

mechanism works as follows:

1. A station which wants to transmit a data frame to another station senses the channel using the carrier-sense mechanism described above.

2. If the channel has been idle for longer than a DIFS, it transmits the frame and waits for a positive acknowledgement (ACK).

3. The station backoffs if:

   - No ACK has arrived in time.
   - The channel has not been free for a DIFS period.
   - The channel has not been free for a EIFS period after a failed reception.
   - The frame is consecutive to a previous transmission of the same station.

4. After the successful reception of the data frame, the receiving station waits for a period equal to SIFS and sends the ACK.

In the four-way handshaking of the RTS/CTS mechanism two more packets are exchanged. Although more management frames are sent in RTS/CTS, it has the big advantage to provide a solution to the *hidden node problem*: This is the situation when two stations want to send to a third station and cannot hear each other but only the station in the centre they want to send to. Since during the RTS/CTS handshake information to update the NAV is sent in every packet, both stations that cannot hear each other receive the NAV information via the

station in the middle, will update their NAVs accordingly and refrain from getting access to the channel. Figure 7.6 shows two successful transmissions of an MDPU using the BA and RTS/CTS mechanisms.

7.2.2. Conceptual Model

Figure 7.7 shows an extended deterministic and stochastic Petri net of a the MAC layer for a single station, which includes a set of places and transitions representing the DCF (middle block), an extension which models the performance relevant aspects of the *timing synchronisation function* (TSF, lower block) and an artificial "channel monitor" which does not exist in the real-world. It is needed here as a modelling artefact, because due to the untyped tokens of the EDSPN viz. MOSEL-2 modelling formalism the NAV mechanism cannot modelled more realistically. The large number of places and transitions needed to capture all the relevant details of the IEEE 802.11 specification shows, that a graphical representation of the system as a Petri net tends to get unintelligible.

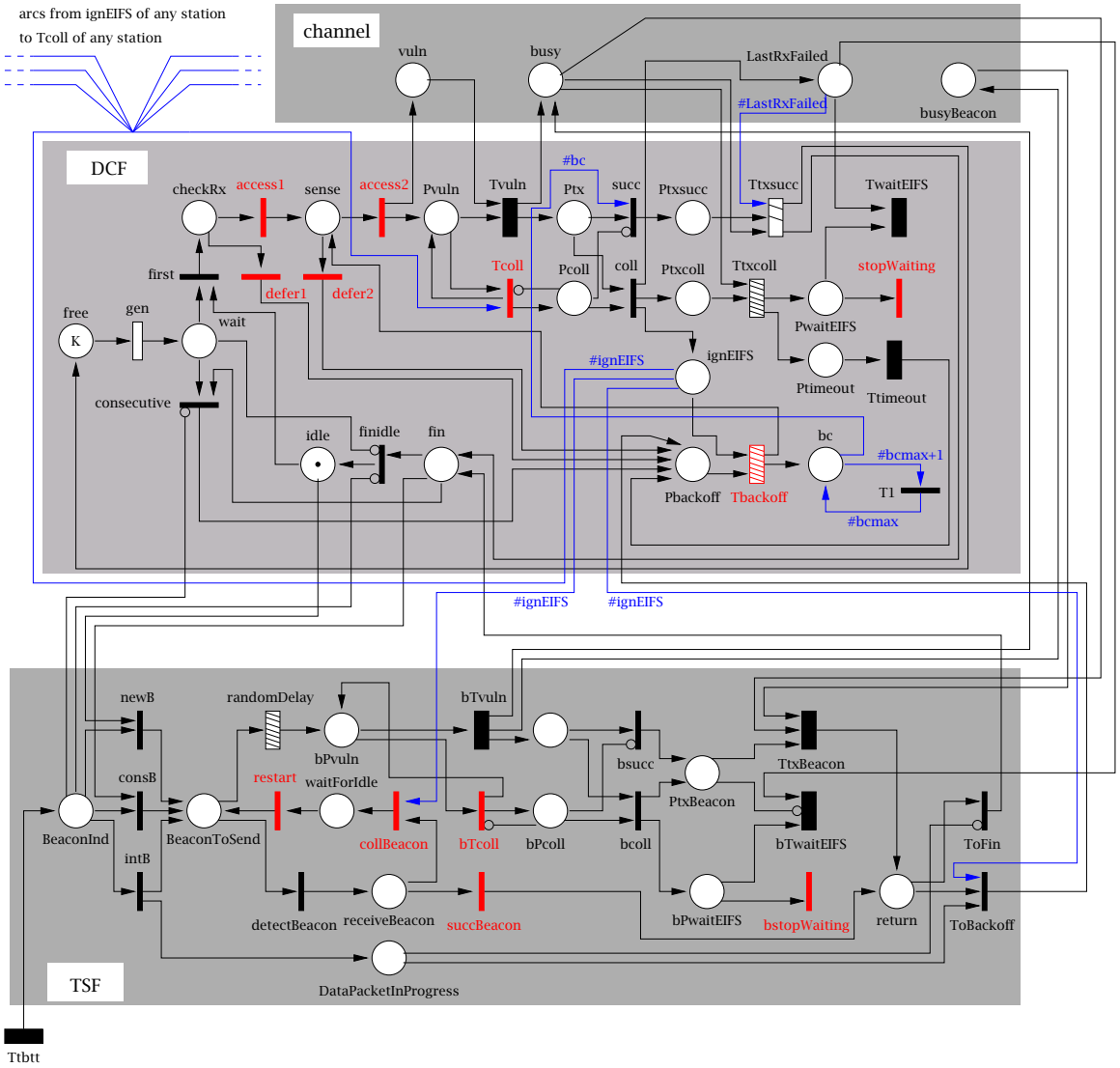

Fig. 7.7: A detailed EDSPN-Model of one station and the channel monitor

7.2.3. MOSEL-2 description

The MOSEL-2 description of the WLAN is also lengthy. But due to the fragmentation into several parts the description is still readable. Below, the constant and parameter part of the MOSEL-2 specification is listed:

```
// IEEE 802.11 wireless LAN model
// performance modeling of the DCF in the stations of an IBSS

// system parameters and constants, all times in microseconds
// all packet lengths in bits

CONST B = 2; CONST MinBitRate = 1; // Bit rates in MBps

// global parameters of the WLAN

PARAMETER V = 0.1, 0.3, 0.5, 0.7, 0.9, 1.0, 2.0, 4.0, 8.0, 10.0; // virtual load
PARAMETER K = 1, 2;   // max number of packets in MAC layer
CONST N = 3;  // number of stations

CONST MaxFrameBody = 8157*8;  CONST L = MaxFrameBody/2.0;
CONST lambda = V * B / (N * L);

// physical medium dependent: here values for DSSS
CONST aSlotTime = 20;  CONST aCCATime = 14;
CONST aRxTxTurnaroundTime = 4;
CONST SIFS = 10;       CONST DIFS = 50;
CONST EIFS = 1148;     CONST aCWmin = 31;
CONST bcmax = 5;       CONST aCWmax = 2^bcmax*(aCWmin+1) - 1;
CONST PHYHeader = 192; CONST BEACON = 808;

// physical medium independent
CONST aAirPropagationTime = 1;
CONST MACHeaderCRC = 272; CONST ACK = 112; // bits
CONST RTS = 160;          CONST CTS = 112; // bits
CONST TimeOut = 300;

// transmission times
CONST PHYHeaderTransTime = PHYHeader / MinBitRate;
// PHYHeader is always transmitted with min. bit rate
CONST MACHeaderCRCTransTime = MACHeaderCRC/B;
CONST ACKTransTime = ACK/B;
CONST RTSTransTime = RTS/B;
CONST CTSTransTime = CTS/B;

// transmission times for Basic Access in
// case of success/collision

CONST vulnerablePeriod = aAirPropagationTime + aCCATime + aRxTxTurnaroundTime;
CONST collConstTransTime = PHYHeaderTransTime + MACHeaderCRCTransTime + DIFS - aCCATime;
CONST collMaxBodyTransTime = MaxFrameBody/B;
CONST succConstTransTime = 2 * PHYHeaderTransTime + MACHeaderCRCTransTime + SIFS
                         + aAirPropagationTime + ACKTransTime + DIFS - aCCATime;
CONST succMaxBodyTransTime = MaxFrameBody/B;

//----------------- system component definition part-------------------------
// top level, the marking of the nodes models the state of the channel
NODE vuln[N]; // system is in vulnerable state;
NODE busy[N]; // indicates that medium is busy;
NODE lastRxFailed[N]; // last request for transmission failed
```

The node part continues on the next page:

```
// station core(s)
@<1..N>{ NODE free_#[K] = K; }  // free slots for arriving frames in buffer
@<1..N>{ NODE wait_#[1]; }      // station waits
@<1..N>{ NODE idle_#[1] = 1; }  // station is idle
@<1..N>{ NODE fin_#[1]; }       // station finished transmission
@<1..N>{ NODE checkRx_#[1]; }   // check if Rx was sent
@<1..N>{ NODE sense_#[1]; }     // station senses if channel is free
@<1..N>{ NODE Pvuln_#[1]; }     // in vulnerable period
@<1..N>{ NODE Pcoll_#[1]; }     // collision happened
@<1..N>{ NODE Ptx_#[1]; }       // ready to send Tx
@<1..N>{ NODE ignEIFS_#[N]; }   // ignore the EIFS-period
@<1..N>{ NODE Ptxsucc_#[1]; }   // sending Tx succeeded
@<1..N>{ NODE Ptxcoll_#[1]; }   // sending Tx collided
@<1..N>{ NODE PwaitEIFS_#[1]; } // wait for EIFS period
@<1..N>{ NODE Ptimeout_#[1]; }  // wait for timeout
@<1..N>{ NODE Pbackoff_#[1]; }  // in backoff mode
@<1..N>{ NODE bc_#[bcmax+1]; }  // backoff counter with upper limit

// Help places and conditions for simulation of discrete uniform distribution
@<1..N>{ NODE backoff_count_#[aCWmax];
         NODE backoff_now_#[1]; }
@<1..N>{ COND do_backoff_# := Pbackoff_# > 0
         AND ((ignEIFS_# = 0 AND lastRxFailed = 0 AND busy = 0)
              OR (ignEIFS_# > 0 AND busy = 0)); }

// End of help places and conditions

//---------------------- transition part----------------------------
// poisson-arrival of payload to MAC-layer (gen)
@<1..N>{ FROM free_# TO wait_# WITH lambda; }

// consecutive or first transmission ?
@<1..N>{ FROM wait_#, fin_#  TO Pbackoff_#; } // consecutive
@<1..N>{ FROM wait_#, idle_# TO checkRx_#; } // first

// proceed to access medium (access1)
@<1..N>{ IF (lastRxFailed = 0) FROM checkRx_# TO sense_#; }

// defer access, go into backoff (defer1, defer2)
@<1..N>{ IF (lastRxFailed > 0) FROM checkRx_# TO Pbackoff_#; } // defer1
@<1..N>{ IF (busy > 0) FROM sense_# TO Pbackoff_#; } // defer2

// start to access medium (access 2)
@<1..N>{ IF (busy == 0) FROM sense_# TO vuln, Pvuln_#; }

// deterministic transition Tvuln to model the fixed
// delay of the vulnerable period
@<1..N>{ FROM Pvuln_#, vuln TO Ptx_#, busy AFTER vulnerablePeriod; } // Tvuln

// after the delay of the vulnerable period the station transmits
// perceptibly for others, this transmission can collide or
// succeed (coll, succ)
@<1..N>{ IF (Pcoll_# > 0) FROM Ptx_# TO lastRxFailed, ignEIFS_#, Ptxcoll_#; }
// coll
@<1..N>{ IF (Pcoll_# = 0) FROM Ptx_# TO bc_#[0], Ptxsucc_#; }
// succ
```

On the following page, the rule part of the MOSEL-2 description is continued:

```
// a frame collision ocurred if there is more than one token in vuln
// memorize the occurrence of the collision in Pcoll_# (Tcoll)
@<1..N>{ IF (vuln > 1 AND Pcoll_# = 0) FROM Pvuln_# TO Pvuln_#, Pcoll_#; }
// and don't forget to flush the ignEIFS places of all stations!!!
@<1..N><1..N>{ IF (vuln > 1 AND Pcoll_<#1> = 0 AND ignEIFS_<#2> > 0) TO ignEIFS_<#2>[0]; }

// durations of collided and successful transmissions (uniformly distibuted)
// if transmission was successful, put a token to place fin which indicates
// end of medium access
@<1..N>{ FROM Ptxcoll_#, busy TO Ptimeout_#, PwaitEIFS_#
         AFTER collConstTransTime..collConstTransTime+collMaxBodyTransTime; }
// Txcoll
@<1..N>{ FROM Ptxsucc_#, busy TO lastRxFailed[0], fin_#, free_#
         AFTER succConstTransTime..succConstTransTime+succMaxBodyTransTime; }
// Txsucc

// after ACK or CTS timeout the stations falls into backoff
@<1..N>{ FROM Ptimeout_# TO Pbackoff_# AFTER TimeOut-DIFS; } // Ttimeout
@<1..N>{ FROM PwaitEIFS_#, lastRxFailed AFTER EIFS-DIFS PRD; }
// TwaitEIFS with policy

// immediate transition stopWaiting preempts TwaitEIFS
@<1..N>{ IF (lastRxFailed = 0) FROM PwaitEIFS_# TO lastRxFailed; }
// stopWaiting

// discrete uniform transition to model the backoff procedure Tbackoff
// note that the arc from ignEIFS is marking dependent
//@<1..N>{ IF ((ignEIFS_# == 0 AND lastRxFailed == 0 AND busy == 0) OR (ignEIFS_# > 0 AND busy = 0))
//         FROM Pbackoff_# TO ignEIFS_#[0], bc_#, sense_#
//         AFTER 0..aCWmax*aSlotTime STEP aSlotTime PRS; }

// Simulation of discrete uniform distribution in the above rule
@<1..N>{ IF do_backoff_# AND backoff_count_# = 0
         THEN { TO backoff_count_#[aCWmax] WEIGHT aCWmax;
                TO backoff_now_#           WEIGHT 1; }
       }
@<1..N>{ IF do_backoff_# FROM backoff_count_# AFTER aSlotTime PRS
         THEN { /* do nothing */ WEIGHT backoff_count_#;
                TO backoff_now_#, backoff_count_#[0] WEIGHT 1; }
       }
@<1..N>{ FROM backoff_now_#, Pbackoff_# TO ignEIFS_#[0], bc_#, sense_#; }
// End of simulation

// backoff counter has an upper limit
@<1..N>{ IF (bc_# = bcmax+1) FROM bc_#; } // T1
// no consecutive transmission
@<1..N>{ IF wait_# = 0 FROM fin_# TO idle_#; }

// Result part
@<1..N>{ PRINT throughput_# := (UTIL (free_#) * lambda * L)/B; }
PRINT throughput := @<1..N>"+"{ throughput_# };

PICTURE "throughput for varying virtual load"
PARAMETER V
CURVE throughput
```

During the generation and experimentation with the large MOSEL-2 description some problems in modelling the backoff procedure occurred: A condition (guard) that has to be true for enabling the backoff procedure:

```
@<1..N>{ COND do_backoff_# := Pbackoff_# > 0
         AND ((ignEIFS_# = 0 AND lastRxFailed = 0 AND busy = 0)
            OR (ignEIFS_# > 0 AND busy = 0)); }
```

The selection of a random backoff time is implemented by the following MOSEL-2 rule:

```
@<1..N>{ IF (do_backoff_#)
         FROM Pbackoff_# TO ignEIFS_#[0], bc_#, sense_#
         AFTER 0..aCWmax*aSlotTime STEP aSlotTime PRS; }
```

Although the above rule can be translated into syntactic correct TimeNET input format it turned out that the representation of a discrete uniform distribution in the .TN-format cannot be recognised by the TimeNET discrete-event simulator if the distribution contains more than about 40 discrete values (1024 discrete values here). In order to enable analysis, the above rule with discrete uniform timing is replaced following set of of auxiliary nodes and transitions:

```
// Help places and conditions for simulation of discrete uniform distribution
@<1..N>{ NODE backoff_count_#[aCWmax];
         NODE finish_backoff_#[1]; }
// Simulation of discrete uniform distribution in the above rule
@<1..N>{ IF do_backoff_# AND backoff_count_# = 0
         THEN { TO backoff_count_#[aCWmax] WEIGHT aCWmax;
                TO finish_backoff_#           WEIGHT 1; }
       }
@<1..N>{ IF do_backoff_# FROM backoff_count_# AFTER aSlotTime PRS
         THEN { /* do nothing */ WEIGHT backoff_count_#;
                TO finish_backoff_#, backoff_count_#[0] WEIGHT 1; }
       }
@<1..N>{ FROM finish_backoff_#, Pbackoff_# TO ignEIFS_#[0], bc_#, sense_#; }
```

The first rule initialises the auxiliary node backoff_count_# with the actual maximum contention window length (aCWmax).

7.2.4. System evaluation

Different variants of the WLAN system were analysed with the MOSEL-2 evaluation environment using the simulation component of TimeNET. Due to the size and detailedness of the descriptions the evaluation by the TimeNET simulation engine with selected 99% confidence interval and a maximum relative error of 1% took up to 8 hours on a modern COTS PC running the Linux operating system. A comparison of the throughputs and waiting times calculated during the MOSEL-2 evaluation with the results obtained by HEINDL and GERMAN in [HG01] shows a high conformity of the predicted values. These findings indicate that the integration of the TimeNET tool into the MOSEL-2 evaluation environment was successful.

The IGL-generated plot in Fig. 7.8 shows the results of one experiment which was carried out using the MOSEL-2 evaluation environment. The packet throughput of the stations in a configuration of the "ad-hoc" network with $N = 10$ nodes was estimated for various load parameters. The figure contains two plots. The upper one shows the results for the network in which the stations obey the EIFS periods as demanded in the IEEE 802.11 specification. The lower plot represents the results for the same network but with the EIFS functionality of the DCF switched off which — of course — does not conform to the standard. The positive effect of the EIFS periods on the throughput for increasing network load can be seen clearly.

7.2.5. Discussion

The EDSPN model presented in [HG01] takes care of many performance relevant aspects and details described in the IEEE 802.11 standard. An interesting point to notice is the

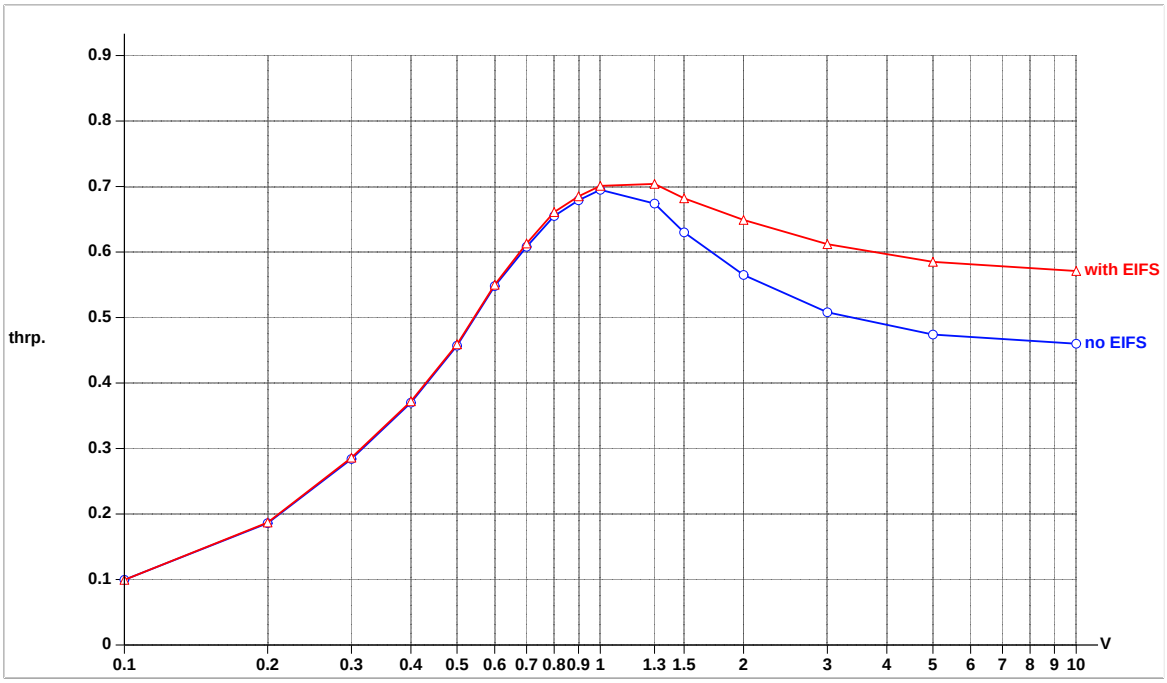

Fig. 7.8: The influence of the EIFS on the throughput for increasing virtual load (DSSS, BA, $N = 3$, $K = 10$)

way how the authors choose to represent the status of the wireless communication channel in their model: The local network allocation vectors (NAVs) of the stations are modelled as a global variable, a *modelling artefact* which acts like a semaphore. It is set by every station in order to block other stations from trying to access the medium. This artefact is an illustrative case of a general phenomenon in performance modelling based on untyped token formalisms. All these formalisms are very descriptive concerning control aspects but are weak concerning the description of typed data that is needed, e.g. to represent the values of the NAVs realistically. In case of the performance study in [HG00] the chosen artefact does not render the results of the analysis useless, since the effect on the time behaviour is almost the same, if a semaphore for controlling medium access is used instead of updating a set of local (NAV) variables.

The MOSEL-2 model of the WLAN DCF demonstrates that the lightweight approach taken in this thesis provides a useful tool for the software engineer in a real-world system development project. The level of detail in which the performance relevant functionality of the DCF and its extensions are represented in the compact MOSEL-2 description corresponds to the abstraction level needed during the early conceptual design phases of the SDLC. The application of MOSEL-2 in current projects to extend the 802.11 MAC layer specification with QoS capabilities is straightforward: [1] Different proposals for new backoff algorithms and priority schemes to support QoS could be easily described and evaluated with the MOSEL-2 LWFM.

[1] see [NRT04] for a survey on current research on QoS and WLANs

8. Conclusion

> *In a word, the computer scientist is a toolsmith — no more, but no less. It is an honorable calling.*
>
> FREDERICK P. BROOKS, JR. *[BJ96]*

In this concluding chapter we recapitulate the achievements of the presented research and discuss possible directions for future work. Since the quotation by F.P. BROOKS above served the author of this thesis as a kind of guiding idea throughout his PhD research, the following retrospective view on this dissertation is taken from a toolsmith's perspective:

8.1. A Retrospect

The main objective of the research presented in this thesis was to contribute to the construction of a technology transfer bridge which allows for a better integration of Performance Evaluation methods into industrial software development projects. The usefulness of this project was motivated by the identification of four deficiencies which have been hampering the interplay of PE and SE for several decades now. While the SE practitioners have to be accused for their notorious "fix-it-later" mentality [Smi03] towards performance evaluation, a large portion of the academic PE community has to be blamed for the "insularity problem" [Fer86] and for "putting the cart before the horse" too often [Hol96]. Both parties are responsible for the downbreak of the "industry-academia gap" concerning the usefulness of formal methods. In this situation, BROOKS analogy of the computer scientist as a toolsmith turns out to provide a very helpful "third way" to the solution of the problems: The toolsmith is neither a pure practitioner, who may not be interested in the the details of a formal methods theoretical foundation, nor is he a pure theoretician who may not be aware that many case-studies which come from the ivory-towers have little in common with reality. The toolsmith knows that the problems for which a solution has to be provided are located on the industrial side of the industry-academia gap and that these problems have to be solved by the practitioners. But he also knows, that without a solid knowledge of the theoretical foundations of his tool he will not be able to provide a useful "instruction manual" to his engineering customers.

Chapter 2 set the context of this research: Software Engineering provides phase-structured SDLC models and Performance Evaluation contributes formalisms for the model-based prediction of five classes of nonfunctional system properties, namely *performance*, *reliability*, *availability*, *dependability* and *performability*. The advantages of an integration of PE activities in the SDLC was motivated economically: Performance Evaluation reduces the overall development cost most effectively if it is applied during the early requirements engineering and conceptual design phases. The basic idea of formal method application was

exemplified by presenting some description techniques and associated tools for the verification of different functional system properties. Afterwards, the evolution of modelling formalisms in computer science was surveyed. The excursus started with the computability models of the 1930s, moved on to the concurrency models of the 1960s and 70s, and finally arrived at the stochastic modelling formalisms which have been in use since 1981. During all periods, the formalisms were based on different notational styles and modelling paradigms which emphasize different aspects of the system and its formal representation. The conclusion was drawn, that for engineering practitioners who want to reach the *single goal* of nonfunctional system property prediction, the token-flow network-oriented modelling paradigm combined with a pure textual notation is the most *pragmatic choice*.

The core ideas and concepts of the thesis were introduced in Chapter 3. A toolsmith should describe the functionality, structure and application of his tool in a "user manual", or *formal method description*, which contains also *methodological information* to guide the engineer through the development process. W.L. SCHERLIS remarked [Sch89] that the essence of requirements engineering consists of mastering the *formalization problem*, i.e. the transition of the informal reality to the world of formal system descriptions. Before a tool can be employed for the evaluation of system properties, the input for that tool has to be created by the software engineer. In order to support him in this formalization task, the toolsmith should not only state precisely how well-formed input for the tool looks like. He should also explain how the engineer has to *conceptualize* the real-world during the creation of the system description in the tool's formal input language. In other words, the toolsmith is also an *ontology maker* who provides an explicit specification of the conceptualization of reality that he had in mind when he designed the tool's input language. The existence of an ontology in the "user manual" ensures that software engineer and toolsmith share a common view of the application domain, which is an important prerequisite to prevent the software engineer from misusing the tool. Section 3.2 elaborated on the proof-theoretic and the model-theoretic approach in the definition of a formal foundation for automated reasoning about arbitrary system properties. Moreover, the principles of the *lightweight* formal method approach [JW96] as an application-oriented and user-friendly formal methods framework were presented. The methodological and theoretical insights gained in the two preceeding sections were used in Sect. 3.3 to define the novel MOSEL-2 lightweight formal method for the prediction of non-functional system properties. The following components of the approach were described in detail:

- The *ontology* which includes the purely textual network-token-flow based modelling paradigm;
- The model-theoretic *formal system* which enables automated semantic reasoning about nonfunctional system properties;
- The transfer of the LWFM approach from functional verification to PE including the presentation of arguments which show that MOSEL-2 fulfils the four main LWFM criteria of *partiality* in language, modeling, analysis and composition.
- The presentation of the *four-layered architecture* of the method.

Chapters 4 and 5 elaborated on the modelling language and the associated evaluation environment as the user interface and tool of the MOSEL-2 LWFM and provided the user with all the details which are needed in order to apply the method in his own development projects. Compared to its predecessor MOSEL [Her00], the MOSEL-2 modelling language as well as the evaluation environment have been extended substantially concerning their expressiveness and the offered analysis algorithms [Beu03, Wüc03, Wüc04]. The MOSEL-2 language allows for the definition of a wide range of non-exponential delay distributions, which enables the modeller to capture many time-related real-world phenomena much more accurately than it would be possible with MOSEL. The philosophy of the evaluation environment was to integrate various existing PE tools which offer solution methods for different analysis problems. In this way the MOSEL-2 evaluation environment became a versatile toolbox for the solution of a wide range of evaluation problems.

Related work on languages and tools for nonfuncional system property evaluation was presented and compared to the MOSEL-2 approach in chapter 6. In chapter 7 some new features of the MOSEL-2 LWFM were demonstrated via modelling examples from the domain of wireless communication systems. The larger performance evaluation study of the wireless ad-hoc network was based on a MOSEL-2 description in which many of the structural and behavioural details specified in the IEEE 802.11 protocol were incorporated. Since the detailedness of the informal protocol standard document is close to reality, this case study can thus be considered as a representative application of the MOSEL-2 LWFM in a software development project.

8.2. Has the Goal been reached?

Figure 8.1 illustrates the contributions of this research, and shows how the MOSEL-2 LWFM spans the industry academia gap to transfer PE technology from academia to the area of industrial software engineering. A “round-trip-engineering” cycle [HL03] which integrates the MOSEL-2 LWFM into an industrial SDLC can be described as follows: During the requirements engineering or conceptual design phase of the SDLC the software engineer wishes to check some nonfunctional requirements of his incomplete system specification. The transition from the industrial to the academic side is facilitated by the explicit methodological foundation of MOSEL-2: the ontology defines a pragmatic modelling paradigm in which the real-world systems are expressed in a user-friendly, purely textual modelling language. This ensures that the software engineer is guided safely through the difficult formalization process. The next step is the prediction of nonfunctional system properties which is performed automatically by the MOSEL-2 evaluation environment. Following an integrative philosophy, the analysis makes use of a lot of well-tested PE technology implemented in the embedded tools. The results of the evaluation are formatted in a pragmatic and user-friendly way by the MOSEL-2 evaluation enviroment. Therefore, it is easy for the software engineer to retransform the formally derived statements back to his application domain and to extract useful statements about the system under development. If the system specification fulfils the nonfunctional requirements, the software engineer moves to the next phase of his SDLC.

Otherwise, the system specification has to be changed in order to fix the problem and the "round-trip-engineering" cycle has to be traversed again.

The final remark on the contributions of this thesis refers to the following objection stated by DAVID L. PARNAS in [Par96]:

> *Much on the work on "formal methods" is misguided and useless, because it continues to search for new foundations although the ground is littered with sound foundations on which nobody has erected a useful edifice.*

We conclude that the approach presented in this thesis does not fall within this category of useless and misguided formal methods research, since the the MOSEL-2 LWFM sucessfully *"erected an useful edifice"* for practical Performance Evaluation on the ground of stochastic process theory, model-theory and multiset rewriting theory.

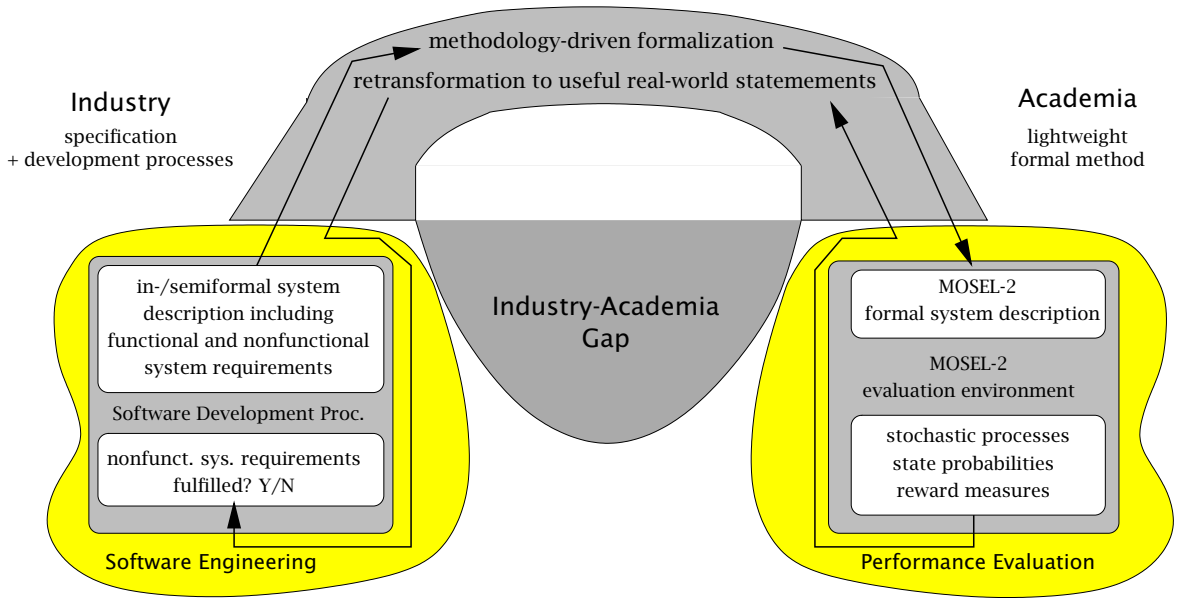

Fig. 8.1: Spanning the industry-academia gap via the technology transfer bridge

8.3. Future Work

Future work based on the MOSEL-2 LWFM could be conducted in the three areas of language extensions, enhancements of the evaluation environment and method application.

Language Extensions Possible extensions of the language include:

- An introduction of *named* rules, which would enable the definition of *impulse reward measures* [HRTT04].
- The introcduction of a module concept, in order to reuse parts of MOSEL-2 descriptions in other models. This is closely related to an inclusion of compositionality in the modelling language. A possible solution which is based on ideas for Stochastic Petri Nets is presented in [BDH01] and could probably be adapted to MOSEL-2.

- An enhancement of language with typed (colored) tokens would enable the representation of control decisions that are based on the actual values of tokens in a marking. As a consequence, a more realistic representation of real-world phenomena would be possible, e.g. for the control of flexible manufacturing systems or for the modelling of intelligent mobile stations in a cellular wireless network.

Enhancements of the Evaluation Environment The emphasis of future work on the MOSEL-2 evaluation enviroment lies on the integration of additional PE tools which would increase the versatility of the "toolbox" via the availability of new solution methods for specific clasess of stochastic processes. Hot candidates for a future integration are:

- DSPNExpressNG [LTK+00] for example, provides numerical solution methods for the transient and stationary analysis of a subclass of GSMPs.

- The multi-paradigm modelling tool Möbius [CCD+01] contains a number of well-tested DES simulators, and also numerical solution methods for CTMCs which employ memory-efficent, MTBDD-based storage techniques which enable the analysis of models with large state-space.

- With an inclusion of the tool GreatSPN [Ber01] the MOSEL-2 user would get access to the solution methods of another well-tested, stochastic Petri net based package.

- The possibility to integrate analysis packages based on Queueing Networks, such as PEPSY [BBG04] should be examined, since the analytical solution methods for Queueing Networks would substantially increase the versatility of the MOSEL-2 evaluation environment.

Other possible directions to enhance the evaluation environment are:

- Check, whether the calculation of complex results should be shifted from the embedded tools into the evaluation environment itself.

- The provision of a link between UML-based, industrial CASE tools to the level $\mathcal{L}_1$ of the MOSEL-2 LWFM. Then, the MOSEL-2 evaluation environment could be used as the formal reasoning engine of a UML-based description technique. The main problem which has to be solved here is how to represent the MOSEL-2 loop construct within the graphically oriented UML descriptions.

- Explore, whether the MOSEL-2 LWFM approach is suited to be integrated in an *agile model driven development* (AMDD) process, as it is described by A. Uhl and S.W. Ambler in [UA03].

- The inclusion of structral analysis of the MOSEL-2 models prior to the translation into the tool-specific modelling languages is also interesting. Checks for *extended conflict sets* and other *well-formedness* tests can help to determine the type of the underlying stochastic process before the semantic model is constructed.

Applications Last but not least, future work should aim at the application of the MOSEL-2 LWFM in a larger, interdisciplinary software development project together with researchers and domain experts from other engineering disciplines, e.g. mobile telecommunications engineering.

Eine leichtgewichtige formale Methode für die Vorhersage nichtfunktionaler Systemeigenschaften

Kurzfassung

Die Leistungs- und Zuverlässigkeitsbewertung während der Entwicklung von computerbasierten Systemen hat zum Ziel, die Einhaltung nichtfunktionaler Systemanforderungen vorherzusagen. So kann z.B. die zu erwartende mittlere Übertragungsdauer von Dateneinheiten in einem geplanten Kommunikationsnetzwerk, auf der Basis stochastischer Modelle bereits in der Phase des konzeptionellen Systementwurfs ermittelt werden. In der Praxis zeigt sich jedoch, daß die im akademischen Umfeld entwickelten Methoden in der Softwareindustrie trotz ihres hohen finanziellen Einsparungspotentials nicht in größerem Maße zum Einsatz kommen. Dies liegt zum großen Teil daran, daß ihre erfolgreiche Anwendung wegen der umständlichen Notation einiger stochastischer Beschreibungsverfahren sowie der praxisfernen Aufbereitung des theoretischen Unterbaus in der Regel nur einigen Experten mit umfangreichem theoretischen Vorwissen möglich ist. Dieses zuweilen als *Abgeschiedenheitsproblem* der Leistungs- und Zuverlässigkeitbewertung bezeichnete Phänomen ist eine Ausprägung des allgemeineren Problems der *Akzeptanz formaler Methoden* in der industriellen Softwareentwicklung. Die ersten formalen Methoden wurden vor ca. 40 Jahren mit dem Ziel entwickelt, nachweisbar korrekte Programme zu konstruieren, konnten sich aber in der industriellen Praxis wegen ihrer Unhandlichkeit und mangelnden Skalierbarkeit nicht allgemein durchsetzen. Ein Ansatz zur Überbrückung der entstandenen „Kluft“ zwischen Industrie und Wissenschaft gelang erst Mitte der neunziger Jahre durch das Aufkommen „leichtgewichtiger“ formaler Methoden zur Verifikation von funktionalen Systemanforderungen, die in verschiedenen Industrieprojekten erfolgreich eingesetzt wurden.

Den Ausgangspunkt der vorliegenden Arbeit bildet der Ansatz, die Leistungs- und Zuverlässigkeitsbewertung als ein Teilgebiet der Anwendung formaler Verifikationsverfahren in der Softwareentwicklung aufzufassen. Aufgrund der Ähnlichkeiten in Aufbau und Zielsetzung der Methoden in den beiden Bereichen wird angenommen, daß die Übertragung des Konzepts der leichtgewichtigen formalen Methode auf die Vorhersage nichtfunktionaler Systemanforderungen einen positiven Beitrag zur Lösung des Abgeschiedenheitsproblems leisten kann. Um diese Annahme zu stützen wird im Rahmen dieser Arbeit eine neue leichtgewichtige formale Methode zur Vorhersage nichtfunktionaler Systemeigenschaften entwickelt. Sie besteht aus einer formalen Beschreibungstechnik mit wohldefinierter Syntax und Semantik, in der Systemstruktur und -verhalten, die Beeinflussung des Systems durch seine Umgebung, sowie die interessierenden nichtfunktionalen Systemeigenschaften auf hohem Abstraktionsniveau als stochastisches Modell spezifiziert werden. Die Syntax der vorgestellten Beschreibungssprache kombiniert die Vorteile des pragmatischen, netzwerkorientierten Modellierungsparadigmas mit der Knappheit einer rein textuellen Notation. Die Bewertung der Systemeigenschaften wird in anwenderfreundlicher Weise durch die automatische Analyse der Modelle in der zur Methode gehörenden Analyseumgebung vorgenommen. Die Anwendbarkeit der Methode wird exemplarisch durch die Modellierung und Analyse von softwarebasierten Systemen aus dem Bereich der drahtlosen Kommunikation (WLAN, GSM) gezeigt. Der Grundgedanke dieser Arbeit besteht darin, den Transfer wissenschaftlicher Technologien in das industrielle Umfeld zu fördern. Insbesondere trägt sie zu einer verbesserten Integration formaler Methoden zur Leistungsbewertung in die frühen Phasen des Softwareentwicklungsprozesses bei.

Inhaltsverzeichnis

Einleitung

"Mind the gap!"

LONDON UNDERGROUND Sicherheitshinweis, seit ca. 1960

Motivation

Die Komplexität von modernen technischen Systemen, wie z.B. Telekommunikationsnetzwerken, Rechner- und Fertigungssystemen oder auch Kraft- und Luftfahrzeugen hat in den letzten Jahrzehnten kontinuierlich zugenommen. Diese Komplexitätssteigerung ist auf die große Anzahl der verwendeten Systemkomponenten, die Vielschichtigkeit der beim Entwurf eingesetzten Architekturen, sowie auf die unvorhersehbaren Interaktionen der Systeme mit ihrer Umgebung zurückzuführen.

Die überwiegende Mehrheit moderner Systeme besteht sowohl aus Hardware- als auch aus Softwarekomponenten, welche den Ablauf des Systems steuern. Unglücklicherweise konnte die Qualität der Softwarekomponenten sowohl im Hinblick auf die Einhaltung funktionaler Anforderungen, wie z.B. Verklemmungs- oder Behinderungsfreiheit sowie ordnungsgemäßer Reaktion in Ausnahmesituationen, als auch bezüglich der gewünschten Leistungsfähigkeit, wie z.B. mittlerer Systemantwortzeit, Durchsatz und Betriebsmittelauslastung oft nicht mit der von der Hardware zur Verfügung gestellten Rechenleistung Schritt halten. Diese Beobachtung gipfelte Anfang der 70er Jahre des letzten Jahrhunderts in die Identifizierung einer *Software-Krise*[1] durch EDSGER W. DIJKSTRA [Dij72].

Um auf diese Krise zu reagieren, entwickelten Wissenschaftler und Praktiker auf dem Gebiet des *Software Engineering* (SWE), welches sich seit 1968 unter diesem Namen herausbildete (siehe NAUR u.a. [NR68]), zahlreiche Methoden für die systematische Erstellung komplexer Softwaresysteme. Eines der Grundprinzipien des Software Engineering ist die Trennung von Systementwurf und Programmierung. Der Entwicklungsprozess sollte in mehrere Phasen unterteilt werden. In jeder der Phasen werden Systembeschreibungen auf angemessenem Abstraktionsniveau erstellt, die dem Entwickler helfen, die Struktur und das Verhalten des in Entwicklung befindlichen Systems zu spezifizieren.

Darüber hinaus können Systembeschreibungen auch für eine automatisierte, werkzeugbasierte Analyse von funktionalen und nicht-funktionalen Eigenschaften eingesetzt werden, vorausgesetzt, dass sie auf einer präzisen bzw. *formalen* mathematischen Grundlage aufbauen.

[1] Im Jahre 1979 regte ROBERT W. FLOYD an, angesichts der anhaltenden Probleme in der Softwareentwicklung besser von einer *"Software-Depression"* zu sprechen [Flo79].

Für die *modellbasierte* Vorhersage nicht-funktionaler Systemeigenschaften können klassische Formalismen der *Leistungsbewertung*, wie z.B. Markov-Modelle [MS06], Warteschlangennetzwerke [Jac54], verschiedene Arten stochastischer Petrinetze [Pet62] [Mol81] oder die später eingeführten stochastischen Prozeßalgebren [NY85] verwendet werden. Einige dieser Formalismen sind wesentlich älter als das Software Engineering: Markov-Modelle werden für die Kapazitätsplanung von Telefonvermittlungsstellen seit 1917 (ERLANG [Erl17]) eingesetzt; Warteschlangennetzwerke wurden in den 1950er Jahren eingeführt, um Probleme im Bereich des Operations Research (JACKSON [Jac57]) zu lösen. Da die oben erwähnten Modellierungsverfahren Systembeschreibungen auf *Stochastische Prozesse* [Ros83] als zugrundeliegendes mathematisches Rahmenwerk abbilden, wird der modellbasierte Zweig der Leistungsbewertung häufig auch unter dem Oberbegriff *Stochastische Modellierung* zusammengefasst.

Obwohl es naheliegend erscheint, die Methoden des Software Engineering und der Leistungsbewertung gemeinsam zu verwenden, um die Qualität komplexer Softwaresysteme zu verbessern, zeigt sich in der Praxis, dass das Zusammenspiel der beiden Subdisziplinen häufig von den folgenden Unzulänglichkeiten behindert wird:

- In [Smi03] erkennt CONNIE SMITH eine unter Softwareentwicklern weit verbreitete *„Repariere-es-später“*-Einstellung (engl.: *fix-it-later* mentality) bezüglich der Bedeutung nicht-funktionaler Systemeigenschaften. Das Hauptaugenmerk liegt während des Entwicklungsprozesses auf funktionalen Eigenschaften. Die Überprüfung von Leis- tungsmerkmalen spielt — wenn überhaupt — erst eine Rolle in der späten Systemtest-Phase. Die Tatsache, dass die allermeisten leistungsbezogenen Mängel eines Systems auf Fehler in der frühen Entwurfsphase des Entwicklungsprozesses zurückzuführen sind, wird ignoriert. Wenn die Leistungsanforderungen vom fertigen System nicht erfüllt werden, ist eine kostspielige Überarbeitung des Entwurfs vonnöten (siehe Abschn. 2.1.3). Nichtsdestoweniger ist die ablehnende Haltung gegenüber der frühzeitigen Leistungsbewertung eine für Softwareentwickler übliche Einstellung und darüber hinaus auch an den Hochschulen weit verbreitet . Ein bevorzugtes und häufig von hartnäckigen Gegnern der Leistungsbewertung angeführtes Zitat ist die folgende Aussage von D.E. KNUTH ([Knu74]) [2]:

  > *. . . Wir sollten uns nicht um geringe Wirkungsgrade scheren, sagen wir mal bis 97% der Entwicklungszeit abgelaufen sind: frühzeitige Optimierung ist die Wurzel allen Übels. . . .*

- Laut DOMENICO FERRARI leidet die akademische Gemeinschaft der Leistungsbewerter seit langem unter einem *„Abgeschiedenheitsproblem“* (engl.: *insularity problem*), wie er in seinem selbstkritischen und kontrovers diskutierten Beitrag [Fer86] ausführt. Er belegt, dass viele der Wissenschaftler Lösungen für veraltete Probleme anbieten, wie z.B. Bewertungen von Monoprozessorsystemen, während der Rest der Welt sich längst auf die Entwicklung von Mehrprozessorsystemen konzentriert. Diese Einstellung wird

[2] Obwohl nicht ausdrücklich in [Knu74] erwähnt, geht dieser Ausspruch höchstwahrscheinlich nicht auf KNUTH selbst zurück, sondern ist vielmehr seinem Kollegen und Freund R.W. FLOYD zuzuschreiben.

gefördert von einem Mangel an interdisziplinärem Geist sowie von dem Unwillen, theoretische Konzepte in einer Weise zu präsentieren, dass diese den Praktikern leichter zugänglich werden. In einer neueren Publikation [Fer03] nimmt FERRARI eine Neubewertung der Situation vor. Er zieht hierin das Fazit, dass einerseits viele Leistungsbewerter während der letzten 20 Jahre der selbstgewählten Isolation den Rücken gekehrt haben. Andererseits warte aber noch immer eine Menge Arbeit auf die Gemeinschaft, welche ihre Forschungen noch besser in die Anwendungsgebiete integrieren und sie mit ihnen synchronisieren sollte.

- Ein schwerwiegendes Problem ist die *„Kluft"* die sich zwischen Industrie und Wissenschaft bezüglich der Einschätzung von Wichtigkeit und Rolle formaler Methoden in der Softwareentwicklung aufgetan hat. Wissenschaftler sind häufig zu enthusiastisch über die Erfolge der Formalisierung. Aus ihrer idealistischen Perspektive gesehen, führt eine vollständige formale Beschreibung aller Systemaspekte während aller Phasen des Entwicklungsprozesses zwangsläufig zu einem fast vollständig korrekten Endprodukt. Auf der anderen Seite werden viele Praktiker wegen der abstrakten und beschwerlichen Syntax vieler Beschreibungstechniken von der Verwendung formaler Methoden abgeschreckt. Selbst in den Fällen, in denen Praktiker gewillt sind eine neue formale Methode in ihrem Projekt anzuwenden, müssen sie häufig feststellen, dass die Theorie ungeeignet ist ihre Probleme zu lösen. In [Hol96] beschreibt GERALD HOLZMANN die Situation sehr treffend:

  > *... Viele Praktiker sind interessiert an neuen Lösungen. Sie sind bereit, zuzuhören und sie auszuprobieren. Dennoch passt der Schlüssel, den sie vom Wissenschaftler zur Lösung ihres Problems erhalten haben, häufig nicht ins Schloss. Wenn der Praktiker dann zurückkommt um sich zu beschweren, so wird ihm gesagt, dass nicht der Schlüssel falsch ist, sondern das Schloss und die Tür und die Wand ...*

- Wissenschaftliche Forschung im allgemeinen, und im besonderen in der Leistungsbewertung, wird oft nach dem Motto *„das Pferd von hinten aufzäumen"* durchgeführt, d.h. eine *Lösung* wird entwickelt *bevor* ein konkretes *Problem* identifiziert wurde. Nach der Entwicklung neuer theoretischer Grundlagen, Methoden und Algorithmen wird ein Fallbeispiel — das „Problem"— *konstruiert* um zu zeigen, dass die Anwendung der neuen Theorie geeignet ist, dieses Problem zu lösen. Diese Grundhaltung dominiert ebenso die akademische Perspektive bezüglich der Rolle der Modellierung, welche sich stark von der Sicht des Ingenieurs unterscheidet: In [BJ96] trifft FREDERICK P. BROOKS den Nagel auf den Kopf, als er sagt:

  > *Der Wissenschaftler konstruiert um zu studieren — der Ingenieur studiert um zu konstruieren.*

  Unglücklicherweise sind die Modelle in vielen „realitätsnahen" Fallbeispielen aus wissenschaftlichen Beiträgen verzerrt durch zu starke Vereinfachung, unrealistische Annahmen und Modellierungsartefakte. In der Konsequenz können die Aussagen, die durch Schlussfolgern auf der formalen Modellebene gewonnen wurden, nicht in eine

„nützliche“ Aussage über das reale System in der informellen Sprache des Anwendungsgebietes transformiert werden.

Zusammengefaßt münden die oben erwähnten Punkte in ein *Technologietransfer-* und *Methodenintegrations*-Problem [Kro92]: Die Leistungsbewertung hat eine Fülle von grundlegenden Theorien und Lösungsverfahren zur modellbasierten *Analyse* und Bewertung von quantitativen Systemeigenschaften entwickelt, die auf *formalen Methoden* beruhen. Auf dem Gebiet des Software Engineering hingegen wurden etliche Anstrengungen unternommen, um *Software-Entwicklungsprozesse* und *Spezifikations- und Beschreibungsstandards* zu etablieren, die den Praktiker in die Lage versetzen, die Probleme beim Entwurf und der Implementierung komplexer Softwaresysteme zu bewältigen. Die Herausforderung des Technologietransfers besteht nun darin, die formalen Methoden der Leistungsbewertung so aufzubereiten, dass sie der Gemeinschaft der Softwareentwickler als ein zusätzliches, leicht bedienbares Werkzeug zugänglich werden.

Abbildung 1.1 verdeutlicht die Situation: Die Kluft zwischen Industrie und Wissenschaft, welche das Hindernis für die Verwendung formaler Methoden in Softwareprojekten darstellt, befindet sich im Zentrum. Links davon, auf dem industriellen Ufer des Software Engineering, nutzen die Praktiker, welche an einem Projekt aus einem konkreten Anwendungsgebiet arbeiten, standardisierte Beschreibungstechniken, wie z.B. die *Unified Modelling Language* (UML, [OMG02b]) oder die *Specification and Description Language* (SDL, [Uni00]). Unter Verwendung von den auf den Beschreibungstechniken aufsetzenden CASE-Werkzeugen, erfassen die Entwickler die verschiedensten Aspekte des geplanten Systems.

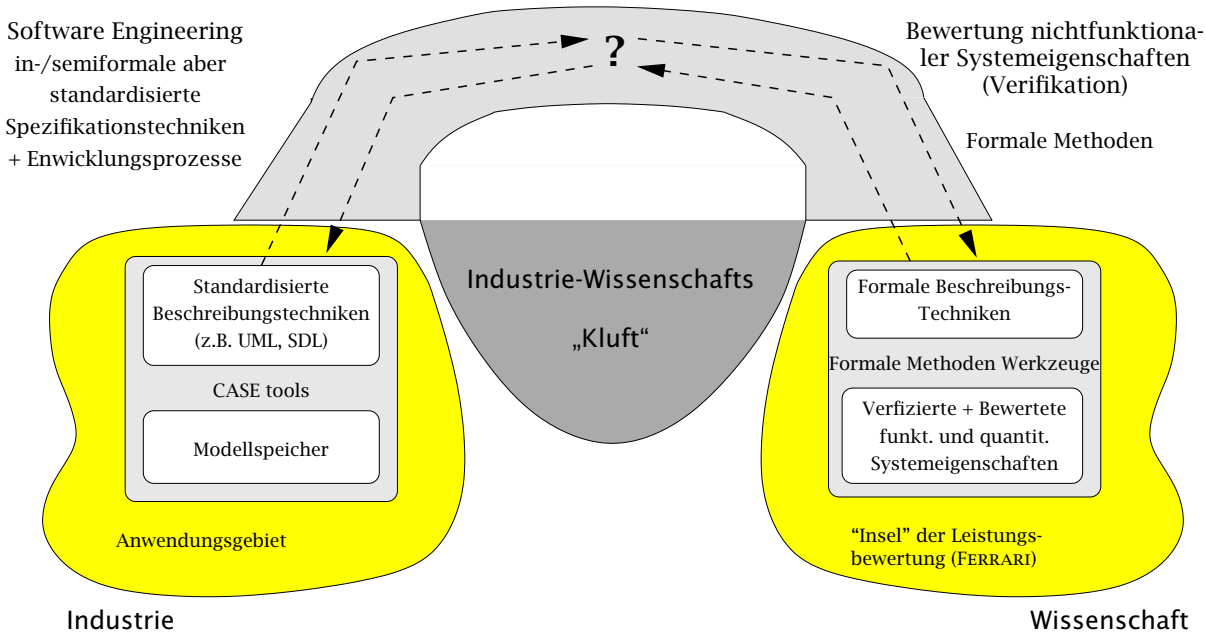

Abbildung 1.1: Die “Kluft” zwischen Industrie und Wissenschaft

Auf der rechten Seite des Grabens, dem wissenschaftlichen Ufer, haben die Forscher auf dem Gebiet der Leistungsbewertung eine Vielzahl von Werkzeugen erstellt, mit der sich nicht-funktionale Eigenschaften von Systemen bestimmen lassen. Diese Werkzeuge benötigen als Eingabe ein *stochastisches Modell* des zu untersuchenden Systems, welches unter

Verwendung einer von vielen verfügbaren, nicht-standardisierten, (semi-)formalen Beschreibungstechniken erstellt wird. Die Brücke über der Kluft symbolisiert die wünschenswerte Integration von formalen Methoden zur Leistungsbewertung in die industrielle Systementwicklung, mit der einerseits die Probleme der mangelnden Leistungsfähigkeit der industriellen Produkte verringert würden und die andererseits den Wissenschaftlern eine Möglichkeit eröffnet, ihr Abgeschiedenheitsproblem zu lösen.

Ziele der Arbeit

Das Ziel dieser Arbeit ist, zur Konstruktion der Technologietransfer-Brücke aus Abb. 1.1 beizutragen. Der Hauptbeitrag ist auf der akademischen Seite des Grabens angesiedelt und besteht aus der Entwicklung einer neuen *Leichtgewichtigen Formalen Methode* (LWFM) [JW96] zur quantitativen Bewertung nicht-funktionaler Systemeigenschaften. Die MOSEL-2 (MOdelling, Specification and Evaluation Language, 2nd revision) LWFM dient als neue Integrationsplattform für viele existierende Formale Methoden zur stochastischen Modellierung und Leistungsbewertung. Der LWFM Ansatz hat seinen Ursprung im Bereich der formalen Systemverifikation (siehe Abschnitt 2.1.4) und basiert auf einer Reihe von Empfehlungen, die von einer Gruppe von Wissenschaftlern des Software Engineering in der Mitte der 1990er Jahre erarbeitet wurden. Sie dienen als Richtlinien wie eine Formale Methode aufgebaut sein sollte, um möglichst leicht von den Praktikern verwendet werden zu können. Die MOSEL-2 LWFM besteht aus einer Beschreibungssprache zur stochastischen Modellierung und einer dazugehörigen Auswerteumgebung, welche unter der Federführung des Autors während der letzten Jahre entwickelt wurden ([ABBBZ02], [Beu03], [Wüc03], [Wüc04], [WABBB04]). Die Modellierungssprache basiert auf ihrer Vorgängerin MOSEL welche von HELMUT HEROLD im Rahmen seiner Dissertation [Her00] in den Jahren 1995-2000 entwickelt wurde. Im Gegensatz zu HEROLD, der die Anwendung der MOSEL Sprache mittels des direkten aber informalen *„Erklären am Beispiel"*-Verfahrens anhand zahlreicher, zumeist kleinerer Fallstudien einführt, wird in dieser Arbeit der formale Charakter der gesamten *Methode* herausgearbeitet: Die MOSEL-2 LWFM ruht auf einem soliden methodologischen Fundament, welches eine *Ontologie* als explizite Spezifikation der Konzeptualisierung umfasst, die sich der Modellierer beim Anwenden der Methode zu Eigen machen muss. Darüber hinaus erfolgt eine präzise Definition der von der Methode zur Verfügung gestellten formalen Beschreibungstechnik. Diese besteht aus einer formalen Syntax, der formalen Semantik und dem zugrundeliegenden formalen logischen System besteht.

Neben diesem methodologischen Fortschritt liegt ein weiterer Schwerpunkt der Arbeit auf der Erweiterung der Ausdrucksstärke der Modellierungssprache. Die Klasse realer Phänomene, die in MOSEL-2 adäquat beschrieben werden können, ist wesentlich größer als die Menge der Systeme, die von der Vorgängersprache erfasst werden konnte. Die gesteigerte Ausdrucksstärke der Modellierungssprache erfordert eine Anpassung der zugehörigen Bewertungsumgebung, welche die automatische Analyse der MOSEL-2 Beschreibungen durchführt. Die notwendige Erweiterung wird durch die Integration des Petrinetz Analysetools TimeNET 3.0 sowie der Simulationskomponente des stochastischen Petrinetz Pakets SPNP 6.1 erreicht. Diese wurden im Rahmen zweier Diplom- bzw. Studienarbeiten in die MOSEL-2

Bewertungsumgebung eingebunden (siehe [Beu03], [Wüc03]).

Ein in der Arbeit verfolgtes Ziel ist es zu untersuchen, wie der mittlere Teil der Brücke in Abb. 1.1 konstruiert werden könnte. Die Grundidee besteht darin, eine für die Zwecke der Leistungsbewertung erweiterte UML-Systembeschreibung auf MOSEL-2 Modelle abzubilden. Erfreulicherweise sind die dazu auf der industriellen Seite des Grabens nötigen Vorarbeiten zum Teil schon durchgeführt worden: Das *"UML Profile for Schedulability, Performance, and Time Specification* [OMG02a] definiert einen Satz von notationellen Erweiterungen der UML, die auch für die Zwecke der modellbasierten Leistungsbewertung geeignet sind.

Aufbau der Arbeit

Die Arbeit ist folgendermaßen gegliedert: Kapitel 2 gibt eine Einführung in die zentralen Konzepte und Resultate des Software Engineering und der Leistungsbewertung, auf denen die folgenden Kapitel aufbauen. Abschnitt 2.1 erinnert an die Entstehung des Software Engineering als ein separates Forschungsgebiet innerhalb der Informatik. Anschließend werden einige Schlüsselkonzepte, wie z.B. Software Lebenszyklus-Modelle, PARNAS Modulkonzept [Par72a] und der Einfluss des Prinzips der strukturierten Programmierung [Dij69], [DDH72] auf die Entwicklung von Programmiersprachen und Software Engineering betrachtet. Abschnitt 2.1.2 erläutert den Ursprung und einige Grundideen der standardisierten Modellierungssprache UML, die zur Zeit als *die* Standard-Notation für Systembeschreibungen in der industriellen Softwareentwicklung angesehen wird. Im Abschnitt 2.1.4 werden einige Methoden für die Verifikation verschiedener Systemeigenschaften betrachtet, die auf formalen Beschreibungen des Systems basieren. Die finanzielle Bedeutung einer möglichst frühzeitigen Bewertung nicht-funktionaler Systemeigenschaften wird in Abschnitt 2.1.3 anhand ökonomischer Betrachtungen hervorgehoben. In Abschnitt 2.2 werden die Ursprünge der modernen Leistungsbewertung erwähnt, sowie Standard-Definitionen für einige Arten von nicht-funktionalen Systemeigenschaften eingeführt. Die drei am häufigsten verwendeten Modellierungsformalismen der Leistungsbewertung, nämlich Warteschlangennetzwerke, stochastische Petrinetze und stochastische Prozess-Algebren, werden kurz erläutert. Abschnitt 2.3 beschließt das Kapitel mit einer Untersuchung der Entwicklung formaler Modelle in der Informatik. Die gründliche historische Abhandlung dient dem Zweck, klar und deutlich herauszustellen, dass heute zwei unterschiedliche stochastische Modellierungs-Paradigmen existieren, welche nicht nur die notationelle Form, sondern auch Zielsetzungen und Methodologie von ihren nicht-stochastischen Vorläufern übernommen haben. Diese Erkenntnis hat einen wesentlichen Einfluss auf die Entwicklung des Kern-Konzepts dieser Arbeit, welches in Kapitel 3 präsentiert wird.

Um die nachfolgenden Überlegungen auf eine präzise terminologische Grundlage zu stellen, werden am Anfang von Abschn. 3.2 zentrale Begriffe zu Methoden und Modellierung eingeführt. Insbesondere wird der Begriff der *Leicht-gewichtigen Formalen Methode*, in Anlehnung an die Definition von JACKSON, WING ET AL. [JW96] konkretisiert. Die LWFM gilt als *die* Lösung der Akzeptanzprobleme formaler Methoden zur Verifikation funktionaler Systemeigenschaften. Die Betrachtung der Hauptargumente in der Debatte über formale Verifikationsmethoden lässt den Schluss zu, dass angesichts der zahlreichen methodologi-

schen Parallelen zwischen Verifikation und Leistungsbewertung, die Übertragung des leichtgewichtigen Prinzips gleichermaßen geeignet ist, die frühzeitige Integration der Leistungsbewertung in den Software-Entwicklungsprozess zu ermöglichen.

In Abschn. 3.4 wird die Architektur der MOSEL-2 LWFM vorgestellt und einige Details anhand eines überschaubaren Beispiels beschrieben. Die informale Realität, die syntaktischen und semantischen Bereiche, sowie die Theorie der stochastischen Prozesse bilden die vier Ebenen der MOSEL-2 LWFM.

Kapitel 4 enthält eine detaillierte Beschreibung der konkreten MOSEL-2 Syntax. Es wird zwischen den Kernkonstrukten, die die Ausdrucksstärke von MOSEL-2 festlegen, und zusätzlichen Sprachelementen, die der Verbesserung des Komforts bei der Modellierung dienen, unterschieden. Der Aufbau einer MOSEL-2 Beschreibung als eine festgelegte Folge von Abschnitten sowie die wichtigsten der benötigten Kernkonstrukte werden vorgestellt. Das wichtigste zusätzliche Sprachelement zur Erhöhung des Modellierungskomforts ist das *loop*-Konstrukt, welches die „Faltung“ identischer Modellteile in einen einzelnen, parametrisierbaren Block erlaubt und so die Größe der Beschreibung im Einzelfall drastisch reduzieren.

Kapitel 5 ist der Beschreibung der MOSEL-2 Bewertungsumgebung gewidmet. Im Gegensatz zu der geschichteten logischen Sicht der MOSEL-2 Architektur, steht nun die tatsächliche Implementierung der Transformationen innerhalb der drei unteren Architekturebenen im Vordergrund. Das *integrative* Grundprinzip der MOSEL-2 Bewertungsumgebung wird anhand der Einbettung existierender stochastischer Analysewerkzeuge beschrieben.

In Kapitel 6 finden einige verwandte Ansätze, Methoden und Werkzeuge Erwähnung und werden mit der in dieser Arbeit vorgestellten Methode verglichen.

Um den Nachweis der Anwendbarkeit der MOSEL-2 LWFM zu erbringen, werden in Kapitel 7 Untersuchungen an größeren Fallbeispielen durchgeführt.

Kapitel 8 beschließt die Arbeit mit einer Zusammenfassung der wichtigsten gewonnenen Erkenntnisse, sowie einer Diskussion von möglichen weiterführenden Arbeiten.

Der Anhang enthält eine Auflistung der MOSEL-2 Syntax in erweiterter Backus-Naur Form (App. A) sowie eine Liste der Kommandozeilen-Optionen der MOSEL-2 Bewertungs-Umgebung (App. B).

Zusammenfassung

> *In a word, the computer scientist is a toolsmith — no more, but no less. It is an honorable calling.*
>
> FREDERICK P. BROOKS, JR. *[BJ96]*

Dieses Schlusskapitel fasst das in dieser Arbeit Erreichte zusammen und gibt Anregungen für weiterführende Forschungen. Da das obige Zitat von F.P. BROOKS dem Autor der vorliegenden Dissertation als eine Art Leitgedanke während der Anfertigung der Arbeit gedient hat, wird die folgende Rückschau auf die erzielten Ergebnisse aus der Sichtweise des „Werkzeugherstellers“ (engl. toolsmith) geschildert.

Eine Rückschau

Das Hauptanliegen dieser Dissertation war, einen Beitrag zur Konstruktion einer Technologie-Transfer-Brücke zu leisten, die eine verbesserte Integration von Methoden der Leistungsbewertung in die industrielle Softwareentwicklung ermöglicht. Die Nützlichkeit dieses Projektes wurde durch die Identifikation von vier Unzulänglichkeiten motiviert, die seit geraumer Zeit das Zusammenspiel von Leistungsbewertung und Software-Engineering behindern: Während den Praktikern aus der Industrie die notorische „repariere-es-später“ Einstellung (engl. “fix-it-later” mentality) [Smi03] gegenüber der Berücksichtigung nichtfunktionaler Systemeigenschaften vorzuwerfen ist, muß ein großer Teil der akademischen Gemeinschaft der Leistungsbewerter wegen der Existenz des „Abgeschiedenheitsproblems“ (engl. “insularity problem”) [Fer86], sowie für zu häufiges „Aufzäumen des Pferdes von hinten“ [Hol96] getadelt werden. Beide Seiten sind verantwortlich für das Entstehen der „Industrie-Wissenschafts Kluft“ bezüglich der Bewertung der Nützlichkeit formaler Methoden. In dieser Situation erweist sich BROOKS Analogie des Computerwissenschaftlers als Werkzeughersteller als äußerst hilfreicher „dritter Weg“ zur Lösung der Probleme: Der Werkzeughersteller ist weder ein reiner Praktiker, der möglicherweise nicht an den Details der theoretischen Grundlagen einer formalen Methode interessiert ist, noch ist er ein reiner Theoretiker, der sich nicht darüber im Klaren ist, dass viele Fallstudien aus den Elfenbeintürmen mit der Realität nur wenig gemeinsam haben. Der Werkzeughersteller ist sich bewußt, dass die Probleme, für die eine Lösung gefunden werden soll, auf der industriellen Seite der „Kluft“ zu finden sind und dass diese Probleme von den Praktikern gelöst werden sollen. Er weiß aber auch, dass er ohne solides Wissen über die theoretischen Grundlagen seines Werkzeugs seine Kundschaft aus der Praxis nicht mit einer brauchbaren „Bedienungsanleitung“ versorgen kann.

Kapitel 2 zeigte den Rahmen dieser Arbeit auf: Das Software-Engineering stellt phasenbasierte Software-Lebenszyklusmodelle zur Verfügung, während die Leistungsbewertung Formalismen zur modellbasierten Vorhersage von fünf Klassen nicht-funktionaler Systemeigenschaften beiträgt, namentlich *Leistung*,*Zuverlässigkeit*, *Verfügbarkeit*, *Verläßlichkeit* und *Performability*[3].

Die Vorteile der Integration von Verfahren zur Vorhersage nichtfunktionaler Systemeigenschaften in den Software-Entwicklungsprozess wurde ökonomisch motiviert: Die Leistungsbewertung reduziert die Gesamt-Entwicklungskosten am effektivsten, wenn sie bereits während der frühen Phasen der Anforderungsbestimmung und des konzeptionellen Systementwurfs angewandt wird. Das Grundprinzip des Einsatzes formaler Methoden wurde anhand einiger Beschreibungstechniken und Werkzeuge zur Verifikation funktionaler Systemeigenschaften erläutert. Anschließend wurde ein Überblick über die Evolution der Modellierungs-Formalismen in der Informatik gegeben. Der Exkurs begann mit den Modellen zur Berechenbarkeit aus den 30er Jahren, ging über zu den Nebenläufigkeitsmodellen der 60er und 70er Jahre und endete bei den stochastischen Modellierungformalismen, die seit Anfang der 80er Jahre[4] gebräuchlich sind. Während all dieser Perioden basierten die Formalismen auf unterschiedlichen notationellen Stilformen und Modellierungsparadigmen, die jeweils unterschiedliche Aspekte des modellierten Systems und seiner formalen Repräsentation hervorheben. Aus dem historischen Abriss wurde die Schlußfolgerung gezogen, dass für praxisorientierte Entwickler, die mit der Modellierung das *alleinige Ziel* der Vorhersage nichtfunktionaler Systemeigenschaften verfolgen, das Markenfluß, Netzwerkorientierte Modellierungsparadigma in Kombination mit einer rein textuellen Notation die *pragmatischste* Wahl ist.

Die Kerngedanken und Konzepte dieser Arbeit wurden in Kapitel 3 eingeführt. Der Werkzeughersteller sollte die Funktionalität und Struktur, sowie die Anwendung seines Werkzeuges in einer formalen „Gebrauchsanweisung“ beschreiben, die unter anderem auch *methodologische Information* enthält, um den Anwender sicher durch den Entwicklungsprozess zu geleiten. W.L. SCHERLIS bemerkte in [Sch89], dass das *Formalisierungproblem*, d.h. der Übergang von der Realität in die Welt der formalen Systembeschreibung, die wesentliche Hürde der Anforderungsbestimmung darstellt. Bevor ein Werkzeug zur Bestimmung von Systemeigenschaften eingesetzt werden kann, muß der Systementwickler die Eingabe in Gestalt einer formalen Systembeschreibung erstellen. Um den Entwickler in dieser Formalisierungstätigkeit zu unterstützen, muss der Werkzeughersteller nicht nur genau angeben, wie wohlgeformte Beschreibungen auszusehen haben, sondern sollte auch beschreiben, wie der Anwender den relevanten Ausschnitt der realen Welt während der Erstellung der formalen Beschreibung begrifflich fassen soll. Anders ausgedrückt ist der Werkzeughersteller also auch ein *Ontologie-Ersteller*, der eine explizite Spezifikation der Konzeptualisierung der Realität zur Verfügung stellt, welche er beim Entwurf der Eingabesprache seines Werkzeuges im Sinn hatte. Die Existenz einer Ontologie in der formalen „Gebrauchsanweisung“ stellt sicher, dass der Entwickler und der Werkzeughersteller dieselbe Sicht auf das Anwendungsgebiet teilen, welches eine wichtige Voraussetzung für die Verhinderung eines mißbräuch-

[3] Eine deutsche Bezeichnung für den Begriff Performability existiert nicht.
[4] mit Ausnahme der älteren Warteschlangennetzwerke

lichen Einsatzes des Werkzeugs darstellt. In Abschnitt 3.2 wurden das beweistheoretische und das modelltheoretische Grundpinzip als die zwei Arten von mathematischen Theorien beschrieben, die zur Definition eines formalen Fundaments für Werkzeuge zur Bestimmung beliebiger Systemeigenschaften herangezogen werden können. Darüber hinaus wurden die Grundideen der *leichtgewichtigen* formalen Methoden [JW96], welche eine anwendungsorientierte und benutzerfreundliche Ausprägung formaler Methoden darstellen, eingeführt. Die in den beiden vorhergehenden Abschnitten gewonnenen methodologischen und theoretischen Erkenntnisse wurden in Abschn. 3.3 dazu verwendet, die neuartige, leichtgewichtige formale Methode MOSEL-2, die zur Vorhersage nichtfunktionaler Systemeigenschaften geeignet ist, zu definieren. Die folgenden Komponenten des Ansatzes wurden im Detail beschrieben:

- Die *Ontologie*, welche um das rein textbasierte, Marken-Fluß, Netzwerk-orientierte Modellierungsparadigma aufgebaut ist.
- Das modelltheoretische *formale System*, welches das automatisierte Schlussfolgern von Aussagen über nichtfunktionale Systemeigenschaften auf der semantischen Ebene ermöglicht.
- Die Übertragung der LWFM Grundprinzipien von der Verifikation funktionaler Systemeigenschaften auf das Gebiet der Leistungsbewertung, einschließlich der Angabe von Argumenten die belegen, dass MOSEL-2 die vier Hauptkriterien der *Eingeschränktheit* in Sprache, Modellierung, Analyse and Komposition erfüllt.
- Die Präsentation der *vierschichtigen Architektur* der Methode.

Kapitel 4 und 5 beschäftigten sich mit der Modellierungssprache und der dazugehörgen Bewertungsumgebung, welche die Benutzerschnittstelle bzw. das Werkzeug der MOSEL-2 LWFM bilden. Sie versorgten den Anwender mit allen Details, die zur Anwendung der Methode in eigenen Entwicklungsprojekten vonnöten sind. Im Vergleich zum Vorgänger MOSEL [Her00], sind in MOSEL-2 sowohl die Modellierungssprache als auch die Bewertungsumgebung wesentlich erweitert worden, was die zur Verfügung gestellten Sprachmittel und Analyseverfahren angeht [Beu03, Wüc03, Wüc04]. Die MOSEL-2 Modellierungssprache erlaubt die Verwendung etlicher, nicht-exponentiell verteilter Verzögerungszeiten, was den Modellierer in die Lage versetzt, viele zeitbezogene Phänomene aus dem realen Anwendungsgebiet viel akkurater zu beschreiben, als es ihm mit MOSEL möglich ist.

Die Grundidee der Bewertungsumgebung ist, bestehende Werkzeuge der Leistungsbewertung, die verschiedenartigste Lösungsverfahren implementieren, zu integrieren. Auf diese Weise wird die MOSEL-2 Entwicklungsumgebung zu einer vielseitigen „Werkzeugkiste“(engl. toolbox), die zur Lösung vielfältigster Probleme der Leistungsbewertung geeignet ist.

Verwandte Arbeiten zu Sprachen und Werkzeugen für die Bestimmung nichtfunktionaler Systemeigenschaften wurden in Kapitel 6 behandelt und mit dem MOSEL-2-Ansatz verglichen. Im Kapitel 7 wurden einige der Fähigkeiten der MOSEL-2 LWFM an Fallbeispielen aus dem Anwendungsbereich der drahtlosen Kommunikation demonstriert. Die umfangreichere Leistungsbewertung eines drahtlosen „ad-hoc“ Netzwerkes basierte auf einer MOSEL-2 Beschreibung, in der viele der strukturellen und verhaltensbezogenen Details aus dem

IEEE 802.11 Protokollstandard berücksichtigt sind. Da der Detailliertheitsgrad dieses Dokuments recht genau ist, kann die Fallstudie als repräsentative Anwendung der MOSEL-2 Methode in einem Software-Entwicklungsprozess angesehen werden.

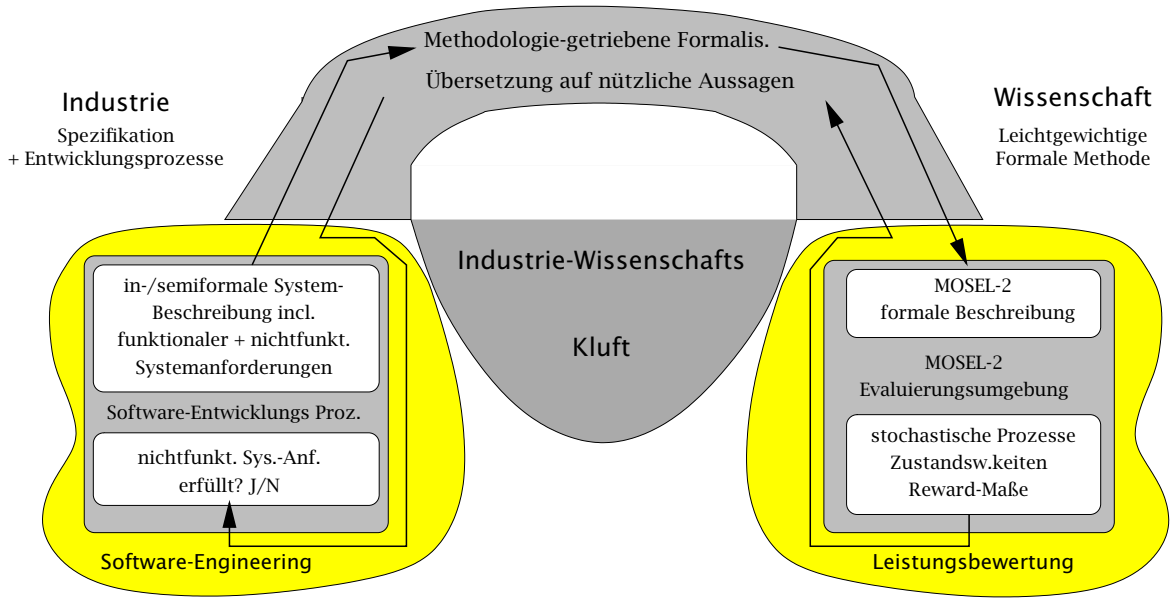

Abbildung 8.1: Die Überbrückung der Industrie-Wissenschafts Kluft mittels der Technologietransfer-Brücke

Wurde das Ziel erreicht?

Abbildung 8.1 veranschaulicht den Beitrag dieser Arbeit und zeigt, wie mithilfe der MOSEL-2 LWFM die „Industrie-Wissenschafts-Kluft“ durch den Transfer von Methoden zur Leistungsbewertung in die Software-Entwicklung überbrückt werden kann. Der “round-trip-engineering” Zyklus [HL03], welcher die MOSEL-2 Methode in den Software-Entwicklungsprozess integriert, kann wie folgt beschrieben werden: Während der Phasen der Anforderungsermittlung oder des konzeptionellen Entwurfs möchte der Systementwickler Aussagen über nichtfunktionale Eigenschaften seiner Systemspezifikation gewinnen. Der Übergang von der industriellen zur akademischen Seite der Kluft wird durch die explizite methodologische Basis der MOSEL-2 Methode ermöglicht: Die Ontologie definiert ein pragmatisches Modellierungsparadigma, in welchem das reale System in einer benutzerfreundlichen, rein textuellen Modellierungssprache beschrieben wird. Dies gewährleistet, dass der Software-Entwickler sicher durch den schwierigen Formalisierungprozess geleitet wird. Der nächste Schritt besteht aus der automatischen Vorhersage der nichtfunktionalen Systemeigenschaften, die von der MOSEL-2 Entwicklungsumgebung auf Basis der formalen Beschreibung durchgeführt wird. Der integrativen Philosophie folgend, bedient sich die Analyse einer Reihe von ausgetesteten Leistungsbewertungs-Technologien, welche in den in die Evaluierungsumgebung eingebetteten Werkzeugen implementiert sind. Die Ergebnisse der Bewertung werden in pragmatischer und benutzerfreundlicher Weise von der MOSEL-2 Evaluierungsumgebung aufbereitet. Dadurch fällt es dem Software-Entwickler leicht, die auf

der formalen Ebene gewonnenen Resultate in das reale Anwendungsgebiet zu übertragen und dort sinnvolle Aussagen über das zu entwickelnde System zu gewinnen. Sofern die nichtfunktionalen Systemanforderungen als erfüllt bewertet wurden, kann der Software-Entwickler zur nächsten Phase des Software-Entwicklungsprozesses übergehen. Anderenfalls muss die Systemspezifikation abgeändert werden um das Problem zu beheben und anschliessend der "round-trip-engineering" Zyklus erneut durchlaufen werden.

Die Schlussbemerkung zu dem in dieser Arbeit geleisteten Beitrag nimmt Bezug auf den folgenden Einwand von DAVID L. PARNAS aus [Par96]:

> *Viele der Arbeiten über „Formale Methoden" sind fehlgeleitet und nutzlos, da sie fortwährend nach neuen Fundamenten suchen, obwohl der Boden mit soliden Fundamenten übersät ist, auf denen niemand ein sinnvolles Gebäude errichtet.*

Wir schließen, dass der in dieser Arbeit entwickelte Ansatz nicht in diese Kategorie nutzloser und fehlgeleiteter Forschung zu formalen Methoden fällt, da die MOSEL-2 LWFM mit Erfolg ein *"sinnvolles Gebäude"* für die praxistaugliche Leistungsbewertung auf dem Fundament stochastischer Prozesse, Modelltheorie und Multimengen-Theorie errichtet hat.

Weiterführende Arbeiten

Mögliche Fortführungen der Arbeiten auf der Basis der MOSEL-2 LWFM können auf den drei Gebieten Spracherweiterung, Weiterentwicklung der Evaluierungumgebung, sowie der Methodenanwendung erfolgen.

Spracherweiterungen Mögliche Erweiterungen der Modellierungssprache sind:

- Eine Einführung *benannter* Regeln, welches die Definition von *Impuls Reward-Maßen* [HRTT04] ermöglicht.
- Die Erarbeitung eines Modulkonzepts, um Teile von MOSEL-2 Modellbeschreibungen in anderen Untersuchungen wiederzuverwenden. Diese Idee ist eng verknüpft mit der Unterstützung von Kompositionalität in der Modellierungssprache. Eine mögliche Lösung, die auf Ideen für stochastische Petrinetze basiert, ist in [BDH01] beschrieben und könnte an MOSEL-2 angepasst werden.
- Eine Erweiterung der Sprache mit getypten (farbigen) Marken würde die Darstellung von Kontrollentscheidungen ermöglichen, die auf den Werten der Marken basieren. Als Folge daraus wäre eine realistischere Modellierung einiger Phänomene, wie z.B. die Kontrolle von flexiblen Herstellungssystemen oder die Darstellung intelligenter Mobilstationen in einem drahtlosen Funknetz möglich.

Weiterentwicklung der Evaluierungsumgebung Der Schwerpunkt weiterführender Arbeiten an der MOSEL-2 Evaluierungsumgebung sollte auf der Integration weiterer Leistungsbewertungswerkzeuge liegen. Dadurch wird die Vielseitigkeit der „Werkzeugkiste" aufgrund der Verfügbarkeit neuer Lösungsverfahren für spezielle Klassen von stochastischen Prozessen weiter erhöht. Im einzelnen bieten sich dazu die folgenden Werkzeuge an:

- DSPNExpressNG [LTK+00] stellt zum Beispiel numerische Lösungsverfahren für die transiente und stationäre Analyse von GSMP-Unterklassen zur Verfügung.
- Das Multiparadigmen Modellierungswerkzeug Möbius [CCD+01] enthält eine Reihe von ausgetesteten Simulationsverfahren, und desweiteren numerische Lösungsverfahren, mit der große zeitkontinuierliche Markovketten durch die Zuhilfenahme von Multi-terminalen Entscheidungsdiagrammen (MTBDD) als zugrundeliegender Speichertechnik analysiert werden können.
- Mit einer Anbindung des Tools GREATSPN [Ber01] würde der MOSEL-2 Benutzer Zugriff auf die ausgetesteten Lösungsverfahren eines weiteren stochastischen Petrinetz Werkzeuges erhalten.
- Die Möglichkeit der Integration Warteschlangennetz-basierter Werkzeuge wie z.B. PEPSY [BBG04] sollte ebenfalls in Erwägung gezogen werden, da die dort angebotenen analytischen Lösungsverfahren die Vielseitigkeit der Evaluierungsumgebung ebenfalls stark erhöhen würde.

Andere mögliche Richtungen zur Erweiterung der Evaluationsumgebung sind:

- Eine Überprüfung, ob die Berechnung komplexer Resultate von den angebundenen Werkzeugen in das MOSEL-2 Programm selbst verlagert werden sollte.
- Die Erstellung einer Verbindung zwischen UML-basierten, industriellen CASE Tools und der Ebene $\mathcal{L}_1$ der MOSEL-2 LWFM. Dadurch könnte die MOSEL-2 Evaluierungsumgebung als formale Bewertungsmaschine einer UML-basierten Beschreibungstechnik eingesetzt werden. Das Hauptproblem, welches hierbei zu lösen wäre, besteht in der Abbildung der MOSEL-2 loop Konstrukte innerhalb der graphischen UML-Beschreibung.
- Eine Überprüfung, ob der MOSEL-2 LWFM Ansatz geeignet ist in einen *Agilen Modell-getriebenen Entwicklungsprozess* (AMDD) integriert zu werden, wie er von A. UHL und S.W. AMBLER in [UA03] beschrieben wird.
- Der Einbau von Verfahren zur statischen, strukturellen Analyse der MOSEL-2 Beschreibungen vor der Übersetzung in die tool-spezifischen Modellierungssprachen wäre ebenfalls interessant. Tests auf *erweiterte Konfliktmengen* und andere *Wohlgeformtheitsüberprüfungen* können zur Feststellung des Typs des zugrundeliegenden stochastischen Prozesses verwendet werden, bevor das semantische Modell erzeugt wird.

Anwendungen Zu guter Letzt sollten die weiterführenden Arbeiten auf die Anwendung der MOSEL-2 Methode in größeren, interdisziplinär ausgerichteten Software-Entwicklungsprojekten abzielen. So könnte die Leistungsfähigkeit der Methode etwa in Zusammenarbeit mit Wissenschaftlern und Domänenexperten auf dem Gebiet der mobilen Telekommunikationssysteme getestet werden.

A. MOSEL-2 syntax reference

The context-free syntax rules of the MOSEL-2 language are presented below in EBNF notation. Definitions of lexical items are marked by an asterisk (*). The start symbol is `mosel2_file`. Provide a complete, annotated reference for the MOSEL-2 specification language.

```
mosel2_file     ::= const_part node_part funcs_and_conds assert_part rule_part
                    result_part pictures
const_part      ::= { const_def }
identif(*)      ::= ( letter | "_") { letter | digit | "_" }
letter(*)       ::= "A" | ... | "Z" | "a" | ... | "z"
number(*)       ::= digits [ "." digits ] [ ( "e" | "E" ) [ "+" | "-" ] digits ]
digit(*)        ::= "0" | ... | "9"
digits(*)       ::= digit { digit }
string(*)       ::= ‘‘"’’ sequence of printable chars ‘‘"’’
const_def       ::= ( CONST identif ":=" const_expr | PARAMETER identif ":="
                    expr_list | ENUM identif ":=" "{" enum_list "}" ) ";"
expr_list       ::= range_expr { "," range_expr }
range_expr      ::= const_expr [ ".." const_expr [ STEP const_expr ] ]
enum_list       ::= identif { "," identif }
node_part       ::= node { node }
node            ::= NODE identif opt_capacity opt_init ";"
opt_capacity    ::= [ "[" const_expr "]" ]
opt_init        ::= [ ":=" const_expr ]
funcs_and_conds ::= { func_or_cond }
func_or_cond    ::= ( FUNC identif opt_formal_args ":=" state_expr | COND ID
                    opt_formal_args ":=" condition ) ";"
opt_formal_args ::= [ "(" formal_args ")" ]
formal_args     ::= { identif "," } ID
assert_part     ::= { assert }
assert          ::= ASSERT condition ";"
rule_part       ::= rule { rule }
rule            ::= rule_elements rule_end
rule_end        ::= opt_then "{" local_rules "}"
                  | ";"
opt_then        ::= [ THEN ]
local_rules     ::= { rule_elements ";" }
rule_elements   ::= rule_element { rule_element }
rule_element    ::= FROM fromto_names
                  | TO fromto_names
                  | IF condition
                  | PRIO const_expr
                  | WEIGHT state_expr
```

```
                  | RATE state_expr
                  | AFTER const_expr
                  | AFTER const_expr ".." const_expr
                  | AFTER const_expr ".." const_expr STEP const_expr
                  | GEOM "(" const_expr "," const_expr ")"
                  | NORM "(" const_expr "," const_expr ")"
                  | LOGN "(" const_expr "," const_expr ")"
                  | ERLANG "(" const_expr "," const_expr ")"
                  | GAMMA "(" const_expr "," const_expr ")"
                  | BETA "(" const_expr "," const_expr ")"
                  | CAUCHY "(" const_expr "," const_expr ")"
                  | POIS "(" const_expr "," const_expr ")"
                  | PARETO "(" const_expr "," const_expr ")"
                  | BINOM "(" const_expr "," const_expr "," const_expr ")"
                  | HYPER "(" const_expr "," const_expr "," const_expr ")"
                  | HYPO "(" const_expr "," const_expr "," const_expr ","
                    const_expr ")"
                  | NEGB "(" const_expr "," const_expr "," const_expr ")"
                  | WEIB "(" const_expr "," const_expr ")"
                  | PRI
                  | PRD
                  | PRS
                  | EMP "(" const_expr "," const_expr ")"
fromto_names    ::= fromto_name { "," fromto_name }
fromto_name     ::= ID
                  | identif "(" state_expr ")"
                  | identif "[" const_expr "]"
                  | EXTERN
result_part     ::= time results
time            ::= [ TIME number ( ";" | ".." number STEP number ";" ) ]
results         ::= { result }
result          ::= ( PRINT identif ":=" p_expr | RESULT identif ":=" p_expr | PRINT
                    identif ":=" TIME TO condition | PRINT prob_type DIST identif ) ";"
pictures        ::= { picture }
picture         ::= PICTURE opt_string picture_parts opt_semicolon
picture_parts   ::= picture_part { picture_part }
picture_part    ::= FIXED fixed_params
                  | PARAMETER TIME
                  | PARAMETER ID
                  | CURVE prob_type DIST identif opt_string
                  | CURVE curves
                  | XLABEL string
                  | YLABEL string
fixed_params    ::= fixed_param { "," fixed_param }
fixed_param     ::= ( TIME | identif ) equal const_expr
curves          ::= curve { "," curve }
curve           ::= identif opt_string
opt_string      ::= [ string ]
opt_semicolon   ::= [ ";" ]
```

```
const_expr      ::= if_expr
state_expr      ::= if_expr
p_expr          ::= if_expr
condition       ::= cond
if_expr         ::= IF cond THEN expr elif_expr
                  | expr
elif_expr       ::= ELIF cond THEN expr elif_expr
                  | ELSE expr
expr            ::= { term ( "+" | "-" ) } term
term            ::= { factor ( "*" | "/" ) } factor
factor          ::= atom { "^" atom }
atom            ::= "(" cond ")"
                  | prob_type PROB "(" cond ")"
                  | prob_type UTIL "(" identif ")"
                  | mean_type MEAN "(" if_expr ")"
                  | SIN "(" if_expr ")"
                  | SQRT "(" if_expr ")"
                  | FLOOR "(" if_expr ")"
                  | CEIL "(" if_expr ")"
                  | number
                  | identif opt_actual_args
opt_actual_args ::= [ "(" actual_args ")" ]
actual_args     ::= state_expr { "," state_expr }
mean_type       ::= [ AVG | CUM ]
prob_type       ::= [ AVG ]
cond            ::= { and_cond OR } and_cond
and_cond        ::= { not_cond AND } not_cond
not_cond        ::= [ NOT ] comparison
comparison      ::= if_expr [ compare_op if_expr ]
compare_op      ::= equal
                  | not_equal
                  | "<="
                  | ">="
                  | "<"
                  | ">"
equal           ::= "="
                  | "=="
define          ::= ":="
                  | "="
not_equal       ::= "/="
                  | "!="
```

B. Using the MOSEL-2 environment

The MOSEL-2 evaluation environment is invoked from a shell using the following command line syntax:

`mosel2` *options input-file*

The parameter *input-file* is the name of a MOSEL-2 description file, which usually has the suffix `.mos`. This MOSEL-2 file is read in, parsed and checked for errors. The options are prefixed by a dash `-`. Each option is denoted by a single letter, like `-s` . Multiple options can follow a single dash, like -Ts , which has the same meaning as -T -s . Some options need an argument; this argument must immediately follow the option, separated by whitespace. MOSEL-2 knows the following options:

```
This is MOSEL-2, version 2.10 (2004-05-28).
Usage: mosel2 [OPTIONS] MOSELFILE
OPTIONS:
  -m            Create a MOSLANG model for MOSES.
  -mo METH[,N]  METH must be a MOSES method (power, power2, lpu,
                crout, grassmann, multilevel). Default is lpu.
                N is the iteration count for power/power2.
  -c            Create a CSPL model for SPNP.
  -co OPTION    OPTION can be +OPT, -OPT or OPT=VALUE, where OPT is
                a CSPL option name (+OPT or -OPT or OPT=VALUE)
  -T            Create a TN model for TimeNET.
  -To OPTION    Set TimeNET OPTION.
  -s            Run selected tool, don't keep created model file.
  -r RESFILE    Write analysis results to RESFILE.
                Default is input file name with extension ".res".
  -i IGLFILE    Write picture descriptions to IGLFILE.
                Default is input file name with extension ".igl".
  -tTIME        Do transient analysis at TIME.
  -tS,E,SW      Do trans. anal. in the interval S..E with stepwidth SW.
  -v            Be verbose: show messages of analysis tool.
  -k            Keep intermediate files of analysis tool.
  -d            Dump model description (for debugging).
  -h, -?        Print the information you're currently reading.
```

Bibliography

[Aan66] Stål Aanderaa. On the algebra of regular expressions. *Applied Mathematics*, pages 1–18, January 1966. 33, 36

[AB93] Khalid Al-Begain. Using subclasses of PH distributions for the modeling of empirical activities. In *Proc. 18th Int. Conf. on Statistics and Computer Sciences (ICSCS'93)*, pages 141–152, 1993. 76

[ABAK03] K. Al-Begain, I. Awan, and D.D. Kouvatsos. Analysis of GSM/GPRS cell with multiple data service classes. *Wireless Personal Communications*, 25(1):41–57, April 2003. 22

[ABBBZ02] Khalid Al-Begain, Jörg Barner, Gunter Bolch, and A. Zreikat. The Performance and Reliability Modelling Language MOSEL-2 and its Applications. *International Journal on Simulation: Systems, Science, and Technology*, 3(3-4):69–79, 2002. v, 4, 18, 87, 88, 133

[ABBT00] Khalid Al-Begain, Gunter Bolch, and Miklós Telek. Scalable schemes for call admission and handover in cellular networks with multiple services. *Wireless Personal Communications*, 15(2):125–144, November 2000. 22

[AD94] Rajev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994. 17, 19

[AL98] Sten Agerholm and Peter Gorm Larsen. A lightweight approach to formal methods. In Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullmann, editors, *Proc. of the Int. Workshop on Current Trends in Applied Formal Methods*, volume 1641 of *LNCS*, pages 168–183. Springer, October 1998. 51

[ALR98] Sten Agerholm, Pierre-Jean Lecoeur, and Etienne Reichert. Formal specification and validation at work: A case study using VDM-SL. In *Formal Methods in Software Practice — Proceedings of the second workshop on Formal methods in software practice*, pages 78–84, 1998. 52

[ALR01] Algirdas Avižienis, Jean-Claude Laprie, and Brian Randell. Fundamental concepts of computer system dependability. Technical Report UCLA CSD Report 010028, University of California in Los Angeles, Computer Science Department, 2001. 23

[ALRL04] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January-March 2004. 23

[AMBB+89] Marco Ajmone Marsan, Gianfranco Balbo, Andrea Bobbio, Giovanni Chiola, Gianni Conte, and Aldo Cumani. The effect of execution policies on the semantics and analysis of stochastic Petri nets. *IEEE Trans. on Software Engineering*, 15(7):832–846, July 1989. 67

[AMC87] M. Ajmone Marsan and G. Chiola. On Petri nets with deterministic and exponentially distributed firing times. In G. Rozenberg, editor, *Advances in Petri Nets 1987; Proc. 7th Europ. Workshop on Appl. and Theor. of Petri Nets*, volume 266 of *LNCS*, pages 132–145. Springer, 1987. 28, 33

[AS85] B. Alpern and F.B. Schneider. Defining liveness. *Information Processing Letters*, 21(4), October 1985. 16

[ATS05] Eleftheria Athanasopoulou, Purvesh Thakker, and William H. Sanders. Evaluating the dependability of a leo satellite network for scientific applications. In *Proc. 2nd Int. Conf. on the Quantitative Evaluation of Systems (QEST 2005)*, page 10, September 2005. 100

[AWG00] Architecture Working Group AWG. IEEE Std 1471-2000: IEEE recommended practice for architectural description of software-intensive systems. Technical report, IEEE Standards Association, 2000. 54

[Bae04] Jos C.M. Baeten. A brief history of process algebra. Technical Report Rapport CSR04-02, Vakgroep Informatica, Technische Universiteit Eindhoven, The Netherlands, 2004. 29, 37

[Bat98] Iain John Bate. *Scheduling and Timing Analysis for Safety Critical Real-Time Systems*. PhD thesis, Department of Computer Science, University of York, November 1998. 16

[BB87] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987. 33, 38

[BB01] Jos. C.M. Baeten and T. Basten. Partial-order process algebra and its relation to Petri nets. In *Handbook of Process Algebra*, chapter 13, pages 769–872. Elsevier, Amsterdam, The Netherlands, 2001. 37

[BBG04] Peter Bazan, Gunter Bolch, and Reinhard German. WinPEPSY-QNS performance evaluation and prediction system for queueing networks. In Khalid Al-Begain and Gunter Bolch, editors, *Proc. of the 11th Int. Conf. on Analytical and Stochastic Modelling Techniques and Applications (ASMTA'04)*, pages 147–150, June 2004. 123, 142

[BC98] Dines Bjørner and Jorge R. Cuéllar. Software engineering education: Rôles of formal specification and design calculi. *Annals of Software Engineering*, 6:365–410, 1998. 99

[BCMP75] F. Baskett, K.M. Chandy, R.R. Muntz, and F.G Palacios. Open, closed and mixed networks of queues with different classes of customers. *Journal of the ACM*, 22(2):248–260, 1975. 25

[BCNN01] Jerry Banks, John S. Carson, Barry L. Nelson, and David Nicol. *Discrete Event Simulation*. Prentice-Hall, Upper Saddle River, NJ, 3rd edition, 2001. 74

[BCS69] W.G. Bouricius, W.C. Carter, and P.R. Schneider. Reliability modeling techniques for self-repairing computer systems. In *Proc. of the 1969 24th national ACM conference*, pages 295–309, August 1969. 22

[BCSS98] M. Bernardo, R. Cleaveland, S. Sims, and W. Stewart. TwoTowers: a tool integrating functional and performance analysis of concurrent systems. In *Formal Description Techniques*, pages 457–467, 1998. 100

[BD02] Mario Bravetti and Pedro R. D'Argenio. Tutte le algebre insieme: Concepts, discussions and relations of stochastic process algebras with general distributions. In *GI Dagstuhl Research Seminar: Validation of Stochastic Systems*, 2002. 31

[BDH01] S. Bernardi, S. Donatelli, and A. Horváth. Implementing compositionality for stochastic Petri nets. *Software Tools for Technology Transfer*, 3(4):417–430, 2001. 39, 122, 141

[BDTGN84] Joanne Bechta Dugan, Kishor S. Trivedi, Robert M. Geist, and Victor F. Nicola. Extended Stochastic Petri Nets: application and analysis. In E. Gelenbe, editor, *Performance '84*, pages 507–519. North-Holland, December 1984. 28, 33

[Bea77] M. Danielle Beaudry. Performance related reliability measures for computing systems. In *Proc. 7th Annual Int. Conf. on Fault Tolerant Computing Systems*, pages 16–21. IEEE, June 1977. 23

[Bec99] Kent Beck. *Extreme Programming Explained*. Addison-Wesley, Reading, 1999. 10

[Bek71] Hans Bekič. Towards a mathematical theory of processes. Technical Report TR 25.125, IBM Laboratory Vienna, 1971. 33, 37

[Ber01] Susanna Bernardi. GreatSPN Manual — Version 2.0.2. Technical report, Performance Evaluation Group, Dipartimento di Informatica, Università di Torino, Italy, 2001. 93, 101, 123, 142

[Ber02] Marco Bernardo. TwoTowers 2.0 User Manual. Technical report, Department of Computer Science, University of Bologna, November 2002. 100

[Beu03] Björn Beutel. Integration of the Petri Net Analysator TimeNET into the Model Analysis Evironment MOSEL. Master's thesis, Dept. of Computer Science, University of Erlangen-Nürnberg, April 2003. 4, 5, 83, 84, 86, 95, 121, 133, 134, 139

[BG94] Lorenzo Bernardo, Marco Donatiello and Roberto Gorrieri. MPA: A stochastic process algebra. Technical Report UBLCS-94-10, Department of Computer Science, University of Bologna, 1994. 33

[BG95] Carolyn Brown and Doug Gurr. A categorical linear framework for Petri nets. *Information and Computation*, 122:268–285, 1995. 58

[BG97] Marco Bernardo and Roberto Gorrieri. A tutorial on EMPA: A theory of concurrent processes with nondeterminism, priorities, probabilities and time. Technical Report UBLCS-96-17, Department of Computer Science, University of Bologna, January 1997. 30, 31

[BG02] Mario Bravetti and Roberto Gorrieri. The theory of interactive generalized semi-Markov processes. *Theoretical Computer Science*, 282:5–32, 2002. 33

[BGdMT98] Gunter Bolch, Stefan Greiner, Hermann de Meer, and Kishor S. Trivedi. *Queueing Networks and Markov Chains*. Wiley Interscience, New York, 1998. 24, 25

[BH95] Jonathan P. Bowen and Michael G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(3):34–41, 1995. 52

[BHK+04] Henrik Bohnenkamp, Holger Hermanns, Ric Klaren, Angelika Mader, and Y.S. Usenko. Synthesis and stochastic assessment of schedules for lacquer production. In *First Int. Conf. ond the Quantitative Evaluation of Systems (QUEST'04)*, pages 28–37, September 2004. 100

[BHKK03] Henrik Bohnenkamp, Holger Hermanns, Joost-Pieter Katoen, and Ric Klaren. The MoDeST modelling tool and its implementation. In *Computer Performance Evaluation, Modelling Techniques and Tools, Proc. 13th Int. Conf., TOOLS 2003*, volume 2794 of *LNCS*, pages 116–133. Springer, September 2003. 99, 103

[BJ96] Frederick P. Brooks Jr. The computer scientist as toolsmith II. *Communications of the ACM*, 39(3):61–68, March 1996. 3, 119, 131, 137

[BJR96] G. Booch, I. Jacobson, and J. Rumbaugh. The Unified Modeling Language for Object-Oriented Development v. 0.9. Technical report, Rational Software Corp., June 1996. 11

[BK82] Jan Bergstra and Jan Willem Klop. Fixed point semantics in process algebra. Technical Report IW 208, Mathematical Centre, Amsterdam, 1982. 37

[BK84] Jan Bergstra and Jan Willem Klop. Process algebra for synchronous communication. *Information and Contol*, 60(1/3):109–137, 1984. 33, 37

[BKLL95] Ed Brinksma, Joost-Pieter Katoen, Rom Langerak, and Diego Latella. A stochastic causality-based process algebra. *The Computer Journal*, 38(7):552–565, 1995. 29

[Bli89] Wayne D. Blizard. Multiset theory. *Notre Dame Journal of Formal Logic*, 30(1):36–66, Winter 1989. 57, 60

[Boe84] Barry W. Boehm. Verifying and validating software requirements and design specifications. *IEEE Software*, 1(1):75–88, 1984. 13

[Boe88] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988. 10

[Bör02] Egon Börger. The origins and the development of the ASM method for high level system design and analysis. *Journal of Universal Computer Science*, 8(1):2–74, 2002. 49

[Box79] George E.P. Box. Robustness in the strategy of scientific model building. In R. Launer and G. Wilkinson, editors, *Robustness in Statistics, Proc. of a Workshop spons. by the Mathematics Division, Army Research Office*, page 202, 1979. 9

[BR69] J.N. Buxton and Brian Randell, editors. *Software Engineering Techniques*, Rome, Italy, October 1969. NATO Science Committee, NATO, Scientific Affairs Division. 10

[Bra02] Mario Bravetti. *Specification and Analysis of Stochastic Real-Time Systems*. PhD thesis, Università di Bologna, Padova, Venezia, 2002. 30

[Bro96] Manfred Broy. Formal description techniques — how formal and descriptive are they? In R. Gotzhein and J. Bredereke, editors, *FORTE IX*, pages 95 – 112. Chapman & Hall, 1996. 44

[Bro97] Manfred Broy. Mathematical methods in system and software engineering. In M. Broy and B. Schneider, editors, *Mathematical Methods in Program Development*, volume F 158 of *NATO ASI series*, pages 1–42. Springer, 1997. 41

[Bry92] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992. 19

[BS93] Jonathan Bowen and Victoria Stavridou. The industrial take-up of formal methods in safety-critical and other areas: A perspective. In J.C.P. Woodcock and P.G. Larsen, editors, *Proc. FME'93: Industrial-Strength Formal Methods*, volume 670 of *LNCS*, pages 183–195. Springer, 1993. 52

[BS98] Manfred Broy and Oscar Slotosch. Enriching the software development process by formal methods. In Werner Hutter, Werner Stephan, Paolo Traverso, and Markus Ullmann, editors, *Applied Formal Methods – FM Trends 98*, volume 1641 of *LNCS*, pages 44–61. Springer, 1998. 44, 45

[BT98] Andrea Bobbio and Miklós Telek. Non-exponential stochastic Petri nets: an overview of methods and techniques. *Computer Systems Science and Engineering*, 13(6):339–351, 1998. 76

[Buc94] Peter Buchholz. Markovian process algebra: Composition and equivalence. In Ulrich Herzog and Michael Rettelbach, editors, *Proc. of the 2nd Workshop on Process Algebras and Performance Modelling*, pages 11–30, July 1994. 30

[Buz73] J.P. Buzen. Computational algorithms for closed queueing networks with exponential servers. *Communications of the ACM*, 16:527–531, 1973. 25

[BvW98] R.-J. Back and J. von Wright. *Refinement Calculus: a systematic introduction*. Graduate texts in computer science. Springer, New York, 1998. 50

[BW99] Michael Barr and Charles Wells. *Category Theory for Computing Science*. Prentice Hall, Upper Saddle River, NJ, third edition, 1999. 58

[CC04] David Crocker and Judith Carlton. Perfect developer: what it is and what it does. *Newsletter of the BCS Formal Aspects of Computer Science special group*, pages 1–8, November 2004. 18, 19

[CCD+01] Graham Clark, Tod Courtney, David Daly, Dan Deavours, Salem Derisavi, Jay M. Doyle, William H. Sanders, and Patrick Webster. The möbius modelling tool. In Reinhard German and Boudewijn Haverkort, editors, *Proc. 9th Int. Workshop on Petri Nets and Performance Models (PNPM'01)*, pages 241–250, September 2001. 93, 100, 123, 142

[CDFH93] Giovanni Chiola, Claude Dutheillet, Guiliana Franceschinis, and Serge Haddard. Stochastic well-formed colored nets and symmetric modeling applications. *IEEE Transactions on Computers*, 42(11):1343–1360, 1993. 29

[Cer94] Iliano Cervesato. Petri nets as multiset rewriting systems in a linear framework. online: `http://www.cs.cmu.edu/~iliano/ON-GOING/deMich.ps.gz`, December 1994. 57, 60

[CGR95] Dan Craigen, Susan Gerhart, and Ted Ralston. Formal methods reality check: Industrial usage. *IEEE Transactions on Software Engineering*, 21(2):90–98, February 1995. 51, 77

[Cho56] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956. 33, 35

[Cho62] Noam Chomsky. Context-free grammars and pushdown storage. Technical Report Quarterly Progress Report 65, MIT Research Laboratory in Electronics, Cambridge, Massachusetts, USA, 1962. 33, 35

[Chu36] Alonso Church. A note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1:101–102, 1936. 33, 34

[Chu41] Alonso Church. *The calculi of lambda conversion*, volume 6 of *Annals of Mathematics Studies*. Princeton University Press, 1941. 35

[Cia87] Gianfranco Ciardo. Toward a definition of modeling power for stochastic Petri nets. In *Proc. Int. Workshop on Petri Nets and Performance Models*, pages 54–62. IEEE CS Press, 1987. 28

[Cin69] Erhan Cinlar. Markov renewal theory. *Advanced Applied Probability*, 1(123-187), 1969. 33, 70

[CL99] Ch. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Kluver Academic Publishers, 2 edition, 1999. 62

[CLR00] Mark Crovella, Christoph Lindemann, and Martin Reiser. Internet performance modelling: The state of the art at the turn of the century. *Performance Evaluation*, 42(2-3):91–108, September 2000. 80

[CMBC93] Giovanni Chiola, Marco Ajmone Marsan, Gianfranco Balbo, and Gianni Conte. Generalized stochastic Petri nets: A definition at the net level and its implications. *IEEE Transactions on Software Engineering*, 19(2):89–107, February 1993. 28

[CMT89] Gianfranco Ciardo, Jogesh Muppala, and Kishor S. Trivedi. SPNP: Stochastic Petri net package. In *Proc. 3rd Int. Workshop on Petri Nets and Performance Models*, pages 142–151, December 1989. 24

[Coc04] Alistair Cockburn. *Crystal Clear: A Human-Powered Methodology for Small Teams*. Addison-Wesley Professional, Boston, 2004. 10

[Coo83] Stephen A. Cook. An overview of computational complexity. *Communications of the ACM*, 26(8):401–408, 1983. 34

[Cox55a] D.R. Cox. The analysis of non-Markov stochastic processes by the inclusion of supplementary variables. *Proc. Camb. Phil. Soc. (Math. and Phys. Sciences)*, 51:433–441, 1955. 28, 33, 70

[Cox55b] D.R. Cox. A use of complex probabilities in the theory of stochastic processes. *Proc. Camb. Phil. Soc. (Math. and Phys. Sciences)*, 51:313–319, 1955. 75

[Cra91] Dan Craigen. Tool support for formal methods. In *Proc. 13th Int. Conf. on Software engineering*, pages 184–185, 1991. 51

[CRMS01] G. Ciardo, Jones R.L., A.S. Miner, and R. Siminiceanu. SMART - Stochastic Model Analyzer for Reliability and Timing. In *Proc. Tools of Aachen 2001 Int. Multiconference on Measurement, Modeling and Evaluation of Computer-Communication Systems*, pages 29–34, September 2001. 100

[CS96] Rance Cleaveland and Scott A. Smolka. Strategic directions in concurrency research. *ACM Computing Surveys*, 28(4):607–625, December 1996. 38

[CWKS97] Brian P. Crow, Indra Widjaja, Jeong Geun Kim, and Prescott T. Sakai. IEEE 802.11, wireless local area networks. *IEEE Communications Magazine*, pages 116–126, September 1997. 110

[D'A99] Pedro R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. PhD thesis, University of Twente, November 1999. 30, 33

[Dal01] Stefan Dalibor. *Erstellung von Testplänen für verteilte Systeme durch stochastische Modellierung*. PhD thesis, University of Erlangen-Nuremberg, September 2001. 15

[DDH72] Ole-Johan Dahl, Edsger W. Dijkstra, and Charles A.R. Hoare. *Structured Programming*. Number 8 in A.P.I.C. Strudies in Data Processing. Academic Press, 1972. 5, 11, 134

[Dep99] IEEE I.S. Department. 802.11: IEEE Standard for Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, 1999 edition. Technical report, IEEE Standards Association, 1999. 110

[DHKK01] Pedro D'Argenio, Holger Hermanns, Joost-Pieter Katoen, and Ric Klaren. MoDeST – a modelling and description language for stochastic timed systems. In Luca de Alfaro and Stephen Gilmore, editors, *Proc. PAPM-ProbMiV 2001*, LNCS, pages 87–104. Springer, 2001. 99

[Dij68] Edsger W. Dijkstra. A constructive approach to the problem of program correctness. *BIT*, 8(3):174–186, 1968. 50

[Dij69] Edsger W. Dijkstra. Notes on structured programming. Technical Report T.H.-Report 70-WSK-03, Department of Mathematics, Technological University Eindhoven, The Netherlands, August 1969. 5, 13, 134

[Dij72] Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972. 1, 129

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications if the ACM*, 18(8):453–457, August 1975. 50

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. 16, 47

[Dij89] Edsger W. Dijkstra. On the cruelty of really teaching computing science. *Communications of the ACM*, 32(12):1398–1404, December 1989. 51

[Dij00] Edsger W. Dijkstra. Answers to questions from students of software engineering. available online from: `http://www.cs.utexas.edu/users/EWD/transcriptions/EWD13xx/EWD1305.html`, November 2000. 51

[Din03] Juergen Dingel. Automatic transition trace analysis of parallel programs using VeriSoft. Technical Report 2003-467, School of Computing, Queen's University, Kingston, Canada, June 2003. 53

[DKSC02] Salem Derisavi, Peter Kemper, William H. Sanders, and Tod Courtney. The möbius state-level abstract functional interface. In T. Field, P.G. Harrison, J. Bradley, and U. Harder, editors, *Proc. TOOLS 2002, 12th Int. Conf. on Computer Performance Evaluation, Modelling Techniques and Tools*, volume 2324 of *LNCS*, pages 31–50. Springer, 2002. 103

[DLM02] Vieri Del Bianco, Luigi Lavazza, and Marco Mauri. A formalization of UML statecharts for real-time software modeling. In *Proc. 6th Biennial World Conf. on Integrated Design Process Technology (IDPT 2002)*, pages 1–8, June 2002. 52

[dM92] Hermann de Meer. *Transiente Leistungsbewertung und Optimierung Rekonfigurierbarer Fehlertoleranter Rechensysteme*. PhD thesis, Universität Erlangen-Nürnberg, October 1992. 19

[DOS03] Michael C. Daconta, Leo J. Obrst, and Kevin T. Smith. *The Semantic Web : A Guide to the Future of XML, Web Services, and Knowledge Management*. Wiley Publishing Inc., Indianapolis, 2003. 43

[Doy00] J.M. Doyle. Abstract model specification using the Möbius modeling tool. Master's thesis, University of Illinois at Urbana Champaign, January 2000. 103

[EMS+02] M. Ermel, T. Müller, J. Schüler, M. Schweigel, and K. Begain. Performance of GSM networks with general packet radio services. *Performance evaluation*, 48:285–310, 2002. 105

[Erl17] Agner Krarup Erlang. Solution of some problems in the theory of probabilities of significance in automatic telephone exchanges. *Elektrotkeknikeren*, 13, 1917. 1, 20, 33, 130

[ETS00] ETSI. Digital cellular telecommunications system (Phase 2+); Radio Link Protocol (RLP) for data and telematic services on the Mobile Station - Base Station System (MS - BSS) interface and the Base Station System - Mobile-services Switching Centre (BSS - MSC) interface (GSM 04.22 version 7.1.0 Release 1998). Technical Report ETSI TS 100 946 V7.1.0, European Telecommunications Standards Institute, January 2000. 22

[EW90] Uffe Engberg and Glynn Winskel. Petri nets as models of linear logic. In Andre Arnold, editor, *Proc. 15th colloquium on Trees in Algebra and Programming (CAAP'90)*, volume 431 of *LNCS*, pages 147–161. Springer, May 1990. 58

[FDR98] Martin S. Feather, Julia Dunphy, and Nicolas Rouquette. "Pushbutton" analysis via integration of industrial tools with formal validation. In *Proc. 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*, pages 140–141, October 1998. 18

[Fea98] Martin S. Feather. Rapid application of lightweight formal methods for consistency analyses. *IEEE Trans. in Software Engineering*, 24(11):949–959, November 1998. 53

[Fer86] Domenico Ferrari. Considerations on the insularity of performance evaluation. *IEEE Transactions on Software Engineering*, SE-12(6):678–683, June 1986. 2, 119, 130, 137

[Fer03] Domenico Ferrari. Insularity revisited. In Gabriele Kotsis, editor, *Performance Evaluation – Stories and Perspectives*, pages 1–9. Austrian Computer Society, December 2003. 2, 131

[Fet88] James H. Fetzer. Program verification: The very idea. *Communications of the ACM*, 31(9):1048–1063, September 1988. 51

[FG88] B.L. Fox and P.W. Glynn. Computing poisson probabilities. *Communications of the ACM*, 31(4):440–445, 1988. 69

[FKN+92] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):31–58, March 1992. 54

[Flo67] Robert W. Floyd. Assigning meanings to programs. In *American Mathematical Society Symposium in Applied Mathematics*, volume 19, pages 19–31, 1967. 16, 50

[Flo79] Robert W. Floyd. The paradigms of programming. *Communications of the ACM*, 22(8):455–460, August 1979. 1, 129

[FPTT98] Ricardo M. Fricks, Antonio Puliafito, Miklós Telek, and Kishor S. Trivedi. Applications of Non-Markovian stochastic Petri nets. *Performance Evaluation Review*, 26(2):15–27, August 1998. 81

[GA04] Erek Göktürk and M. Naci Akkøk. Paradigm and software engineering. In *Proc. Workshop on the Impact of Software Process on Quality (IMPROQ 2004)*, pages 10–17, May 2004. 32, 44

[Gar98] H. Garavel. OPEN/CÆSAR: An open software architecture for verification, simulation, and testing. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *LNCS*, pages 68–84. Springer, 1998. 100

[GB92] Joseph A. Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the ACM*, 39(1):95–146, January 1992. 49

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schliessen. *Mathematische Zeitschrift*, 39:176–210 and 405–431, 1935. 47

[Ger97] Rob Gerth. Concise PROMELA reference. Available from: `http://spinroot.com/spin/Man/Quick.html`, June 1997. 52

[Ger00] Reinhard German. *Performance Analysis of Communication Systems – Modeling with Non-Markovian Stochastic Petri Nets*. Wiley-Interscience Series in Systems and Optimization. Wiley, 2000. 69, 70, 71, 72

[GGM98] J.H. Gennari, W.E. Grosso, and M.A. Musen. A method-description language: An initial ontology with examples. In *Proc. 11th Workshop on Knowledge Acquisition, Modeling, and Management*, November 1998. 43

[GH99] Reinhard German and Armin Heindl. Performance evaluation of IEEE 802.11 wireless LANs with stochastic Petri nets. In *Proc. 8th Int. Workshop on Petri Nets and Performance Models*, pages 44–53, September 1999. 110

[GHR92] Norbert Götz, Ulrich Herzog, and Michael Rettelbach. TIPP — a language for Timed Processes and Performance evaluation. Technical Report TR 4/92, Computer Science Dept. 7, University of Erlangen-Nürnberg, 1992. 30, 31, 33

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987. 48, 58

[GJM02] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Upper Saddle River, New Jersey, 2nd edition, 2002. 11

[GL93] Reinhard German and Christoph Lindemann. Analysis of stochastic Petri nets by the method of supplementary variables. In G. Iazeolla and S.S. Lavenberg, editors, *Proc. 16th Int. Symposium on Computer Performance Modeling, Measurement, and Evaluation (PERFORMANCE '93)*, pages 320–338, September 1993. 28

[Gla96] Robert L. Glass. Formal methods are a surrogate for a more serious software concern. *IEEE Computer*, 29(4):19, April 1996. 51

[GLFH03] Carmelita Görg, Eugen Lamers, Oliver Fuß, and Poul Heegaard. Rare event simulation. Technical Report Final Report COST project 257, Communication Networks, Univ. Bremen, Germany and Telenor, Norway, 2003. 75

[Gly83] Peter W. Glynn. On the role of generalized semi-Markov processes in simulation output analysis. In S. Roberts, J. Banks, and B. Schmeiser, editors, *Proc. of the 1983 Winter Simulation Conference*, pages 39–42. IEEE, 1983. 33, 65, 72

[GMo97] Software life-cycle process model (v-model 97). Technical Report GP 650, German Ministry of Defense, June 1997. 10

[Göd29] Kurt Gödel. *Über die Vollständigkeit des Logikkalküls*. PhD thesis, University Of Vienna, 1929. 47

[Gor02] Jaap Gordijn. *Value-based Requirements Engineering: Exploring Innovative e-Commerce Ideas*. PhD thesis, Vrije Universiteit Amsterdam, 2002. 54

[Gra77] Winfried K. Grassmann. Transient solutions in Markovian queues. *European Journal of Operational Research*, 1:396–402, 1977. 69

[Gre81] Sheila A. Greibach. Formal languages: Origins and directions. *Annals of the History of Computing*, 3(1):14–41, January 1981. 32

[Gri81] David Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer, New York, 1981. 47

[Gru93] Thomas R. Gruber. A translation approach to portable ontology specification. *Knowledge Acquisition*, 5:199–220, 1993. 42

[GT99] Reinhard German and Miklós Telek. Formal relation of Markov renewal theory and supplementary variables in the analysis of stochastic Petri nets. In *Proc. 8th Int. Workshop on Petri Nets and Performance Models*, pages 64–73, September 1999. 71, 72

[Gur84] Yuri Gurevich. Reconsidering turing's thesis: Toward more realistic semantics of programs. Technical Report CRL-TR-38-84, EECS Department, University of Michigan, September 1984. 49

[Hal90] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, 1990. 52

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987. 11

[Hei98] Constance Heitmeyer. On the need for practical formal methods. In *Proc. of the Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, volume 1486 of *LNCS*, pages 18–26. Springer, 1998. 51

[Her98] Holger Hermanns. *Interactive Markov Chains*. PhD thesis, University of Erlangen-Nürnberg, 1998. 33

[Her00] Helmut Herold. *MOSEL – An Universal Language for Modeling Computer, Communication and Manufacturing Systems*. PhD thesis, Faculty of Technical Sciences, University of Erlangen-Nürnberg, July 2000. 4, 33, 42, 80, 96, 121, 133, 139

[Her01] Ulrich Herzog. Formal methods for performance evaluation. In Ed Brinksma and Joost-Pieter Katoen, editors, *FMPA 2000*, volume 2090 of *LNCS*, pages 1–37. Springer, 2001. 31, 44

[Her03] Ulrich Herzog. From Norton to performance enhanced specifications. In *Performance Evaluation – Stories and Perspectives*, pages 165–177. Austrian Computer Society, December 2003. 15

[HG00] Armin Heindl and Reinhard German. The impact of backoff, EIFS, and beacons on the performance of IEEE 802.11 wireless LANs. In *Proc. 4th Int. Computer*

Performance and Dependability Symposium, pages 103–112, March 2000. 110, 118

[HG01] Armin Heindl and Reinhard German. Performance modeling of IEEE 802.11 wireless LANs with stochastic Petri nets. *Performance Evaluation*, 44:139–164, 2001. 110, 117

[HH79] Ulrich Herzog and W. Hoffmann. Synchronization problems in hierarchically organized multiprocessor computer systems. In *Proc. 3rd Int. Symp. on Modelling and Performance Evaluation of Computer Systems: Performance of Computer Systems*, pages 29–48. North Holland, February 1979. 25

[HHK+00] Holger Hermanns, Ulrich Herzog, Ulrich Klehmet, Vassilios Mertsiotakis, and Markus Siegle. Compositional performance modelling with the TIPPtool. *Performance Evaluation*, 39:5–35, 2000. 38

[HHK02] Holger Hermanns, Ulrich Herzog, and Joost-Pieter Katoen. Process algebra for performance evaluation. *Theoretical Computer Science*, 274(1-2):43–87, 2002. 29

[Hil94a] Jane Hillston. *A Compositional Approach to Performance Modelling*. PhD thesis, Department of Computer Science, University of Edinburgh, April 1994. 30, 31, 33, 102

[Hil94b] Jane Hillston. The nature of synchronisation. In *Proc. PAPM'94*, pages 51–70, 1994. 31

[HJU05] Holger Hermanns, David N. Jansen, and Yaroslav S. Usenko. From StoCharts to MoDeST: a comparative reliability analysis of train radio communications. In *Proc. 5th Int. Workshop on Software and Performance (WOSP'05)*, pages 13–23, 2005. 100

[HK00] Holger Hermanns and Joost-Pieter Katoen. Automated compositional Markov chain generation for a plain old telephone system. *Science of Computer Programming*, 36:97–127, 2000. 39

[HL03] Anders Henriksson and Henrik Larsson. A definition of round-trip engineering. Technical report, Department of Computer and Information Science, Linköpings Universitet, Sweden, 2003. 121, 140

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969. 16, 47, 50

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978. 33, 37

[Hoa90] C.A.R. Hoare. Let's make models. In J.C.M. Baeten and J.W. Klop, editors, *Proc. CONCUR '90: Theories of Concurrency: Unification and Extension*, volume 458 of *LNCS*, page 32. Springer, August 1990. 52

[Hod93] Wilfrid Hodges. *Model Theory*. Cambridge University Press, Cambridge, 1993. 46, 49

[Hol96] Gerald J. Holzmann. Early fault detection tools. In Tiziana Margaria and Bernhard Steffen, editors, *Proc. 2nd Int. Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *LNCS*, pages 1–13. Springer, 1996. 3, 51, 119, 131, 137

[Hol97] Gerald J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 25(5):279–295, May 1997. 18, 19, 52, 99

[How71] Ronald A. Howard. *Dynamic Probabilistic Systems, volume II: Semi-Markov and Decision Processes*. Series in decision and control. John Wiley & Sons, New York, 1971. 33

[HR98] Holger Hermanns and Marina Ribaudo. Exploiting symmetries in stochastic process algebra. In *Proc. 12th European Simulation Multiconference*. SCS Europe, 1998. 30

[HR04] David Harel and Bernhard Rumpe. Meaningful modeling: What's the semantics of "semantics"? *IEEE Computer*, 37(10):64–71, October 2004. 12

[HRTT04] Gábor Horváth, Sándor Rácz, Árpád Tari, and Miklós Telek. Evaluation of reward analysis methods with MRMSolve 2.0. In *Proc. 1st Int. Conf. on the Quantitative Evaluation of Systems (QUEST'04)*, pages 165–174, September 2004. 122, 141

[HS87] P.H. Haas and G.S. Shedler. Regenerative generalized semi-Markov processes. *Communications in Statistics — Stochastic Models*, 3(3):409–438, 1987. 33

[HS89] Peter J. Haas and Gerald S. Shedler. Stochastic Petri net representations of discrete event simulations. *IEEE Trans. on Software Engineering*, 15(4):381–393, April 1989. 65

[HS99] Holger Hermanns and Markus Siegle. Bisimulation algorithms for stochastic process algebras and their BDD-based implementation. In Joost-Pieter Katoen, editor, *Proceedings of the 5th International AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems*, volume 1601 of *LNCS*, pages 244–264. Springer, 1999. 39

[Huf54] David A. Huffman. The synthesis of sequential switching circuits. *Journal of the Franklin Institute*, 257(3):161–190, March 1954. 35

[IEE90] IEEE. IEEE Std. 610.12-1990 – IEEE standard glossary of software engineering terminology, February 1990. 20, 21, 45

[IEE99] IEEE. 802.11: IEEE Standard for wireless LAN medium access control (MAC) and physical layer (PHY) specifications, 1999 edition. Technical report, IEEE Standards Association, 1999. 110

[Int04] International Organization for Standardization/International Electotechnical Commission. Software and system engineering – high-level Petri nets – part 1: Concepts, definitions and graphical notation, 2004. 94

[Jac54] Rex Raymond Phillip Jackson. Queueing systems with phase type service. *Operations Research Quarterly*, 5(2):109–120, 1954. 1, 20, 24, 130

[Jac57] James R. Jackson. Networks of waiting lines. *Operations Research*, 5:518–521, 1957. 1, 24, 33, 130

[Jac63] James R. Jackson. Jobshop-like queueing systems. *Management Science*, 10(1):131–42, 1963. 24, 25

[Jac02] James R. Jackson. How Networks of Queues came about. *Operations Research*, 50(1):112–113, January-February 2002. 26

[Jan01] Peter Janich. Wozu Ontologie für Informatiker? Objektbezug durch Sprachkritik. In K. Bauknecht, editor, *Informatik 2001 — Tagungsband der GI/OCG Jahrestagung*, volume II, pages 765–769. Österr. Computer Gesellschaft, 2001. 42

[Jon96] Cliff B. Jones. A rigorous approach to formal methods. *IEEE Computer*, 29(4):20–21, April 1996. 52

[Jon03] Cliff B. Jones. The early search for tractable ways of reasoning about programs. *IEEE Annals of the History of Computing*, 25(2):26–49, April-June 2003. 50, 51

[JR00] Daniel Jackson and Martin Rinard. Software analysis: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, pages 133–145, June 2000. 53

[JW96] Daniel Jackson and Jeannette Wing. Lightweight formal methods. *IEEE Computer*, 29(4):21–22, April 1996. 4, 6, 52, 54, 120, 133, 134, 139

[Kat99] Joost-Pieter Katoen. *Concepts, Algorithms and Tools for Model Checking*, volume 32/1 of *Arbeitsberichte des Instituts für Informatik*. Institute of Computer Science, Erlangen, Germany, June 1999. 19

[KBKH04] Joost-Pieter Katoen, Henrik Bohnenkamp, Ric Klaren, and Holger Hermanns. Embedded software analysis with motor. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: International School on Formal Methods for the Design of Computer, Communication, and Software Systems*, volume 3185 of *LNCS*, page 286. Springer, September 2004. 100

[Kel76] Robert M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, July 1976. 6, 57

[Kel95] Christian Kelling. *Simulationsverfahren für zeiterweiterte Petri-Netze*. PhD thesis, Technische Universität Berlin, November 1995. 74, 75, 94

[Ken51] David George Kendall. Some problems in the theory of queues. *J. Royal Statist. Soc.*, 13:151–173, 1951. 24

[Ken53] David George Kendall. Stochastic Processes occurring in the theory of queues and their analysis by the Method of Imbedded Markov Chains. *Annals of Mathematical Statistics*, 24:338–354, 1953. 24, 50

[Kle56] Stephen Cole Kleene. Representation of events in nerve nets and finite automata. In Claude Elwood Shannon and William Ross Ashby, editors, *Automata Studies*, volume 34 of *Annals of mathematics studies*, pages 3–41. Princeton University Press, 1956. 33, 35

[Kle64] Leonard Kleinrock. *Communication nets — Stochastic message flow and delay*. Lincoln laboratory publications. McGraw-Hill, New York, 1964. 25

[KNP02] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM: Probabilistic symbolic model checker. In *Proc 12th Int. Conf. on Computer Performance Evaluation, Modelling Techniques and Tools (TOOLS'02)*, volume 2324 of *LNCS*, pages 200–204. Springer, April 2002. 101

[Knu74] Donald E. Knuth. Structured programming with go to statements. *Computing Surveys*, 6(4):261–301, December 1974. 2, 130

[Kro92] Klaus Kronlöf, editor. *Method Integration: Concepts and Case Studies*. Wiley Series in Software-Based Systems. John Wiley & Sons, Chichester, 1992. 3, 132

[Kru00] Philippe Kruchten. *The Rational Unified Process — An Introduction*. Addison-Wesley-Longman, Reading, 2 edition, 2000. 10

[Kuh96] Thomas S. Kuhn. *The Structure of Scientific Revolutions*. University Of Chicago Press, 3rd edition, 1996. 32

[Kur64] S.Y. Kuroda. Classes of languages and linear-bounded automata. *Information and Control*, 7:207–223, 1964. 33, 35

[LAK92] J.C.C. Laprie, A. Avizienis, and H. Kopetz, editors. *Dependability: Basic Concepts and Terminology*. Springer, 1992. 23

[Lan66] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966. 77

[LL91] Leslie Lamport and Nancy Lynch. Distributed computing: models and methods. In Jan van Leeuwen, editor, *Handbook of theoretical computer science (vol. B): formal models and semantics*, pages 1157–1199. MIT Press, Cambridge, MA, USA, 1991. 36

[LLT04] Christoph Lindemann, Marco Lohmann, and Axel Thümmler. Adaptive call admission control for QoS/revenue optimization in cdma cellular networks. *Wireless Networks*, 10(4):457–472, July 2004. 22

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, December 1997. 18, 19

[LRRA98] Peter Liggesmeyer, Martin Rothfelder, Michael Rettelbach, and Thomas Ackermann. Qualitätssicherung Software-basierter technischer Systeme – Problembereiche und Lösungsansätze. *Informatik-Spektrum*, 21:249–258, 1998. 13

[LRT99] Christoph Lindemann, Andreas Reuys, and Axel Thümmler. The DSPNexpress 2000 performance and dependability modelling environment. In *Proc. 29th Int. Symposium on Fault-Tolerant Computing Systems (FTCS)*, pages 228–231, June 1999. 101

[LS96] Christoph Lindemann and Gerald S. Shedler. Numerical analysis of deterministic and stochastic Petri nets with concurrent deterministic transitions. *Performance Evaluation*, 27&28:565–582, 1996. 72

[LT99] Christoph Lindemann and Axel Thümmler. Transient analysis of deterministic and stochastic Petri nets with concurrent deterministic transitions. *Performance Evaluation*, 36&37:35–54, 1999. 72

[LTK+00] Christoph Lindemann, Axel Thümmler, Alexander Klemm, Marco Lohmann, and Oliver P. Waldhorst. Quantitative system evaluation with DSPNexpress 2000. In *Proceedings of the second international workshop on Software and performance WOSP*, pages 12–17, 2000. 93, 123, 142

[Lut92] Robyn R. Lutz. Analyzing software requirements errors in safety-critical, embedded systems. Technical Report TR 92-27, Department of Computer Science, Iowa State University of Science and Technology, August 1992. 14

[Man01] Agile Manifesto. Manifesto for agile software development. online: `http://agilemanifesto.org/`, 2001. 10

[Mat62] K. Matthes. Zur Theorie der Bedienungsprozesse. In *Trans. 3rd Prague on Information Theory, Statistical Decision Functions, Random Processes*, pages 513–528, 1962. 33

[Maz96] Antoni Mazurkiewicz. *Introduction to Trace Theory*. Institute of Computer Science, Polish Academy of Sciences, Warszawa, Poland, 1996. 62

[MBC+03] F. Martinelli, S. Bistarelli, I. Cervesato, G. Lenzini, and R. Marangoni. Representing biological systems through multiset rewriting. In *Workshop on Computational Methods in Biomathematics (CMB'03) — 9th Int. Conf. on Computer Aided Systems Theory (EUROCAST 2003)*, volume 2809 of *LNCS*, pages 415–426. Springer, February 2003. 60

[Mea55] G.H. Mealy. A method for synthesizing sequential cicuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955. 33, 35

[MEC01] M. Mahdavi, R.M. Edwards, and S.R. Cvetkovic. Policy for enhancement of traffic in TDMA hybrid switched integrated voice/data cellular mobile communications systems. *IEEE Communication Letters*, 5(6):242–244, 2001. 108

[Mey78] John F. Meyer. On evaluating the performability of degradable computing systems. In *Proc. 8th Int. Symp. on Fault-Tolerant Computing*, pages 44–49, June 1978. 20, 23

[Mey01] John F. Meyer. Performability evaluation: Putting teeth in the mouth of QoS assessment. In Reinhard German, Johannes Lüthi, and Miklós Telek, editors, *Proc. 5th Int. Workshop on Performability Modeling of Computer and Communication Systems*, volume 34, number 13 of *Arbeitsberichte des Instituts für Informatik, Univ. Erlangen-Nürnberg*, September 2001. 24

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science. Springer, New York, 1980. 33, 37

[MJ54] Andreij A. Markov Jr. *Theory of Algorithms*. Academy of Sciences of the USSR, 1954. 35

[MK02] Joachim Meyer-Kayser. ETMCC Erlangen-Twente Markov Chain Checker, user's guide — version 1.4. Technical report, Institute of Computer Science 7, University of Erlangen-Nürnberg, Germany, June 2002. 100

[ML98] Saunders Mac Lane. *Category Theory for the Working Mathematician*. Graduate Texts in Mathematics. Springer, New York, 2nd edition, 1998. 58

[MM84] Ali Movaghar and John F. Meyer. Performability modeling with stochastic activity networks. In *Proc. 1984 Real-Time Systems Symposium*, December 1984. 29, 33, 39, 102

[MM90] José Meseguer and Ugo Montanari. Petri nets are monoids. *Information and Computation*, 88(2):105–155, October 1990. 58

[MN98] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans. on Modeling and Computer Simulation*, 8(1):3–30, January 1998. 75

[Mol81] Michael Karl Molloy. *On the integration of delay and throughput measures in distributed processing models*. PhD thesis, University of California, Los Angeles, USA, 1981. 1, 27, 33, 130

[Möl96] K.-H. Möller. Ausgangsdaten für Qualitätsmetriken – eine Fundgrube für Analysen. In C. Ebert and R. Dumke, editors, *Software-Metriken in der Praxis*. Springer, Berlin, 1996. 14

[MOM91] Narciso Martí-Oliet and José Meseguer. From Petri nets to linear logic through categories: a survey. *International Journal of Foundations of Computer Science*, 2(4):297–399, 1991. 58

[Moo56] E.F. Moore. Gedanken experiments on sequential machines. In C.E. Shannon and J. McCarthy, editors, *Automata Studies*, Annals of Mathematics Studies, pages 129–153. Princeton University Press, Princeton, NJ, 1956. 35

[Mor87] J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987. 10, 50

[MP69] Zohar Manna and Amir Pnueli. Formalization of properties of recursively defined functions. In *Proc. 1st annual ACM Symposium on the Theory of Computing*, pages 201–210, May 1969. 16

[MPW89] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, Part I. Technical Report ECS-LFCS-89-85, Computer Science Department, University of Edinburgh, UK, 1989. 33

[MS06] Andreij A. Markov Sr. Rasprostranenie zakona bol'shih chisel na velichiny, zavisyaschie drug ot druga. *Izvestiya Fiziko-matematicheskogo obschestva pri Kazanskom universitete*, 2(15):135–156, 1906. 1, 130

[MSSS00] A. Mädche, H.-P. Schnurr, S. Staab, and R. Studer. Representation language-neutral modeling of ontologies. In U. Frank, editor, *Proc. German Workshop Modellierung 2000*, pages 128–144, April 2000. 43

[MT04] Jelena Mišić and Yik Bun Tam. Call level QoS performance under variable user mobilities in wireless networks. *Mobile Networks and Applications*, 9(3):207–218, June 2004. 22

[Mur89] Tadao Murata. Petri nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989. 26, 27

[Nat80] Stéphane Natkin. *Réseaux de Petri Stochastiques*. PhD thesis, Conservatoire National des Arts et Métiers, Paris, June 1980. 27

[Nau66] Peter Naur. Proof of algorithms by general snapshots. *BIT*, 6(4):310–316, 1966. 50

[NE00] Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: a roadmap. In A. Finkelstein, editor, *Proc. ICSE 2000 — The Future of Software Engineering*, pages 35–46, 2000. 10, 18

[Nel85] W. Nelson. Weibull analysis of reliability data with few or no failures. *Journal of Quality Technology*, 17(3):140–146, 1985. 21, 80

[Neu75] Marcel F. Neuts. Probability distributions of phase type. In *Liber Amicorum Prof. Emeritus H. Florin*, pages 173–156. University of Louvain, Belgium, 1975. 33, 75

[Neu95] Marcel F. Neuts. *Matrix-Geometric Solutions in Stochastic Models: An Algorithmic Approach*. Dover Publications, New York, 2nd edition, 1995. 75

[NR68] Peter Naur and Brian Randell, editors. *Software Engineering*, Garmisch, Germany, October 1968. NATO Science Committee, NATO, Scientific Affairs Division. reprinted in 2001. 1, 9, 50, 129

[NRT04] Qiang Ni, Lamia Romdhani, and Thierry Turletti. A survey of QoS enhancements for IEEE 802.11 wireless LAN. *Journal of Wireless Communications and Mobile Computing*, 4(5):547–566, 2004. 118

[NS73] Isaac Nassi and Ben Shneiderman. Flowchart techniques for structured programming. *ACM Sigplan Notices*, 8(8):12–26, August 1973. 72

[NST04] David M. Nicol, H. Sanders, William, and Kishor S. Trivedi. Model-based evaluation: From dependability to security. *IEEE Transactions on Dependable and Secure Computing*, 1(1):48–65, January-March 2004. 23

[NWvW+60] Peter Naur, J.H. Wegstein, A. van Wijngaarden, M. Woodger, F.L. Bauer, J. Green, C. Katz, J. McCarthy, A.J. Perlis, H. Rutishauser, K. Samelson, and B. Vauquois. Report on the Algorithmic Language ALGOL 60. *Communications of the ACM*, 3(5):299–314, May 1960. 33, 35

[NY85] N.M. Nounou and Y. Yemini. Algebraic specification-based performance analysis of communication protocols. In Y. Yemini, R. Strom, and S. Yemini, editors, *Proc. 5th Int. Conf. on Protocol Specification, Testing and Verification (PSTV V)*, pages 541–560. North Holland, 1985. 1, 33, 130

[Nyg86] Kristen Nygaard. Basic concepts in object oriented programming. *ACM SIGPLAN notices*, 21(10):128–132, October 1986. 11

[Obj97] Object Management Group. What is OMG-UML and why is it important? online: `http://www.omg.org/news/pr97/umlprimer.html`, 1997. 11

[OG76] Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976. 16

[OL82] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Trans. on Programming Languages and Systems*, 4(3):455–495, July 1982. 16

[OMG01] Unified modeling language 2.0 infrastructure proposal — version 0.61. Technical Report ad/2001-09-02, Object Management Group, September 2001. 54

[OMG02a] UML profile for schedulability, performance, and time specification. Technical Report ptc/02-3-02, Object Management Group, March 2002. 5, 13, 134

[OMG02b] Unified modeling language: Superstructure — version 2.0. Technical Report ptc/03-08-02, Object Management Group, August 2002. 3, 132

[OS98] W.D. Obal and W.H. Sanders. State-space support for path-baser reward variables. In *Proc. 3rd IEEE Annual Int. Computer Performance and Dependability Symposium (IPDS'98)*, pages 228–237, September 1998. 24, 102

[PA91] B. Plateau and K. Atif. A methodology for solving markov models of parallel systems. *IEEE Journal on Software Engineering*, 17(10):1093–1108, 1991. 100

[Par72a] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972. 5, 11, 134

[Par72b] David L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, 1972. 11

[Par81] David M.R. Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Proc. 5th GI-Conference on Theoretical Computer Science*, volume 104 of *LNCS*, pages 167–183. Springer, March 1981. 36

[Par96] David L. Parnas. Mathematical methods: What we need and don't need. *IEEE Computer*, 29(4):28–29, April 1996. 51, 105, 122, 141

[Paw03] Krzysztof Pawlikowski. Do not trust all simulation studies of telecommunication networks. In *Proc. International Conf. on Information Networking ICOIN'03*, pages 3–12, February 2003. 75

[Per94] Harry G. Perros. *Queueing Networks with blocking*. Oxford University Press, 1994. 25

[Pet62] Carl-Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Universität Bonn, Institut für Instrumentelle Mathematik, 1962. 1, 26, 33, 35, 130

[Pet67] Carl-Adam Petri. Grundsätzliches zur Beschreibung diskreter Prozesse. In Wolfgang Händler, editor, *3. Colloquium über Automatentheorie vom 19. bis 22. Oktober 1965*, pages 121–140, Basel, 1967. Birkhäuser. 36

[Plo81] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981. 30

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proc. of the 18th Annual IEEE Symposium on Foundations of Computer Science (FOCS'77)*, pages 46–57, October/November 1977. 19

[Pnu86] Amir Pnueli. Specification and development of reactive systems. In Hans-Jürgen Kugler, editor, *Information Processing 86, Proceedings of the IFIP 10th World Computer Congress*, pages 845–858. North-Holland/IFIP, 1986. 36, 60

[Pos43] Emil Leon Post. Formal reductions of the general combinatorial decision problem. *American J. Mathematics*, 65:197–215, 1943. 33, 35

[Pos44] Emil Leon Post. Recursively enumerable sets of positive integers and their decision problems. *Bulletin of the American Mathematical Society*, 50:284–316, 1944. 33

[Ric53] Henry G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. AMS*, (89):25–29, 1953. 53

[RL80] M. Reiser and S. Lavenberg. Mean value analysis of closed multichain queueing networks. *Journal of the ACM*, 27(2):313–322, 1980. 25

[Roj97] Isabel C. Rojas. *Compositional construction and analysis of Petri net systems*. PhD thesis, Institute for Computing Systems Architecture, University of Edinburgh, 1997. 39

[Ros83] Sheldon M. Ross. *Stochastic Processes*. Wiley series in probability and mathematical statistics. John Wiley & Sons, New York, 1983. 2, 62, 130

[Roy70] Winston W. Royce. Managing the development of large software systems. In *Proc. IEEE WESCON*, pages 1–9, August 1970. 10

[Roz91] Grzegrorz Rozenberg. Carl Adam Petri und die Informatik. Laudatio on the occasion of the retirement of Carl Adam Petri, July 1991. 35

[RS59] Michael Oser Rabin and Dana Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, April 1959. 33, 35

[Rus01] John Rushby. Theorem proving for verification. In *Modeling and verification of parallel processes*, volume 2067 of *LNCS*, pages 39–57. Springer, 2001. 19

[RW87] Peter. J.G. Ramadge and W. Murray Wonham. Supervisory control theory of a class of discrete event systems. *SIAM J. Control and Optimization*, 25(3):206–230, 1987. 19

[RW89] Peter. J.G. Ramadge and W. Murray Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1), January 1989. 19, 54

[Sal66] Arto Salomaa. Two complete axiom systems for the algebra of regular events. *Journal of the ACM*, 13(1):158–169, January 1966. 33, 36

[SB59] Klaus Samelson and Friedrich Bauer. Sequentielle Formelübersetzung. *Elektronische Rechenanlagen*, 1:176–182, 1959. 35

[Sch89] W.L. Scherlis. Responding to E.W. Dijkstra: On the cruelty of really teaching computer science. *Communications of the ACM*, 32(12):1407, 1989. 41, 120, 138

[SCS+04] Fabrice Stevens, Todd Courtney, Sankalp Singh, Adnan Agbaria, John F. Meyer, William H. Sanders, and Partha Pal. Model-based validation of an intrusion-tolerant information system. In *Proc. 23rd Symposium on Reliable Distributed Systems (SRDS 2004)*, pages 184–194, October 2004. 100

[Sha90] Mary Shaw. Prospects for an engineering discipline of software. *IEEE Software*, 7(6):15–24, November 1990. 13

[Sie02] Markus Siegle. *Behaviour analysis of communication systems: Compositional modelling, compact representation and analysis of performability problems*. Habilitation Thesis. Faculty of Technical Sciences, University of Erlangen-Nürnberg, 2002. 57

[SK04] H.R. Schwarz and N. Köckler. *Numerische Mathematik*. B.G. Teubner, 5 edition, 2004. 69

[SM00] William H. Sanders and John F. Meyer. Stochastic activity networks: Formal definitions and concepts. In Ed. Brinksma, Holger Herrmanns, and Joost-Pieter Katoen, editors, *Lectures Formal Methods and Performance Analysis (FMPA 2000)*, volume 2090 of *Lecture Notes in Computer Science*, pages 315–343. Springer, 2000. 23, 100, 101

[Smi03] Connie U. Smith. Software performance engineering. In Gabriele Kotsis, editor, *Performance Evaluation – Stories and Perspectives*, pages 193–202. Austrian Computer Society, December 2003. 2, 15, 119, 130, 137

[Spi88] J. M. Spivey. *Understanding Z: a specification language and its formal semantics*, volume 3 of *Cambridge Tracts In Theoretical Computer Science*. Cambrigde University Press, 1988. 53

[Ste94] William J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, Princeton, NJ, 1994. 69, 102

[Str88] Bjarne Stroustrup. What is object-oriented programming? *IEEE Software*, 5(3):10–20, May/June 1988. 11

[Sup60] Patrick Suppes. A comparison of the meaning and uses of models in mathematics and the empirical sciences. *Synthese*, 12(29):287–301, 1960. 49

[Tar44] Alfred Tarski. The semantic conception of truth. *Philosophy and Phenomenlological Research*, 4:13–47, 1944. 49

[Tar53] Alfred Tarski. A general method in proofs of undecidability. In A. Tarski, A. Mostowski, and R.M. Robinson, editors, *Undecidable Theories*, chapter 1. North-Holland, Amsterdam, 1953. 49

[Tay97] John C. Taylor. *Measure and Probability*. Springer, New-York, 1997. 65

[TB03] J. Tretmans and H. Brinksma. Torx: Automated model based testing. In A. Hartman and K. Dussa-Ziegler, editors, *Proc. 1st European Conf. on Model-Driven Software Engineering*, 2003. 15

[TH98] Miklós Telek and András Horváth. Supplementary variable approach applied to the transient analysis of Age-MRSPNs. In *Proc. IEEE Int. Computer Performance and Dependability Symposium (IPDS'98)*, pages 44–51. IEEE Computer Society Press, September 1998. 72

[Thü03] Axel Thümmler. *Stochastic Modeling and Analysis of 3G Mobile Communication Systems*. PhD thesis, Universität Dortmund, 2003. 68, 72

[Tri00] Kishor S. Trivedi. SPNP user's manual, version 6.0. Technical report, Department of Electrical Engineering, Duke University, Durham, North Carolina, USA, 2000. 93

[Tri02] Kishor S. Trivedi. *Probability and Statistics with Reliability, Queueing and Computer Science Applications*. John Wiley, New York, 2 edition, 2002. 22

[TS00] A.S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 2nd edition, 2000. 46

[Tur36] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–255, November 1936. 33, 34

[Tur50] Alan Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950. 35

[TWC01] Jan Tretmans, Klaas Wijbrans, and Michel Chaudron. Software engineering with formal methods: The development of a storm surge barrier control system — revisiting seven myths of formal methods. *Formal Methods in System Design*, 19:195–215, 2001. 52

[UA03] Axel Uhl and Scott W. Ambler. Point/counterpoint: Model driven architecture is ready for prime time / agile model driven development is good enough. *IEEE Software*, 20(5):70–74, September/October 2003. 123, 142

[Uni00] International Telecommunication Union. ITU-T recommendation Z.100 (11/99) — languages for telecommunications applications–specification and description language (SDL), 2000. 3, 132

[Voe02] Jeroen P.M. Voeten. Performance evaluation with temporal rewards. *Performance Evaluation*, 50:189–218, 2002. 102

[W3C04] W3C. Owl web ontology language overview. Technical report, World Wide Web Consortium, February 2004. 43

[WABBB04] Patrick Wüchner, Khalid Al-Begain, Jörg Barner, and Gunter Bolch. Modelling a single GSM/GPRS cell with delay tolerant voice calls using MOSEL-2. In D. Al-Dabass, editor, *Proc. of the UK Simulation Conference*, pages 88–94, March 2004. 4, 105, 133

[Wad93] Philip Wadler. A taste of linear logic. In Andrzej M. Borzyszkowski and Stefan Sokolowski, editors, *Proc. 18th Mathematical Foundations of Computer Science (MFCS 1993)*, volume 711 of *LNCS*, pages 185–210. Springer, 1993. 58

[Wat61] H.A. Watson. Launch control safety study. Technical report, Bell Telephone Laboratories, Murray Hill, NJ, USA, 1961. 100

[WC99] Andre Wong and Marsha Chechik. Formal modeling in a commercial setting: A case study. In *Proc. World Congress on Formal Methods (FM '99)*, pages 590–607, September 1999. 51

[Whi00] James A. Whittaker. What is software testing? And why is it so hard? *IEEE Software*, 17(1):70–79, January 2000. 10

[Who56] Benjamin Lee Whorf. Science and linguistics. In John B. Carroll, editor, *Language, Thought and Reality*, pages 207–219. MIT Press, Cambridge, MA, 1956. 32

[Win01] Jeanette M. Wing. Formal methods. In John J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 547–559. John Wiley & Sons, New York, 2nd edition, 2001. 50

[Wir71] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, April 1971. 10

[Wüc03] Patrick Wüchner. Extending the Interface between the Modeling Languages MOSEL and CSPL by Adding Simulation Constructs. Master's thesis, Dept. of Computer Science, University of Erlangen-Nürnberg, June 2003. 4, 5, 121, 133, 134, 139

[Wüc04] Patrick Wüchner. Performance Modeling of Mobile networks using MOSEL-2. Master's thesis, Dept. of Computer Science, University of Erlangen-Nürnberg, May 2004. 4, 87, 121, 133, 139

[Zem66] Heinz Zemanek. Semiotics and programming languages. *Communications of the ACM*, 9(3):139–143, March 1966. 40

[Zim01] Armin Zimmermann. TimeNET 3.0 user manual. a software tool for the performability evaluation with stochastic Petri nets. Technical report, Technische Universität Berlin, 2001. 93

Index