Behaviour analysis of communication systems: Compositional modelling, compact representation and analysis of performability properties

Habilitation Thesis

Markus Siegle
Friedrich-Alexander-Universität Erlangen-Nürnberg

2002

Abstract

In this habilitation thesis ("Habilitationsschrift"), we describe novel techniques for analysing the behaviour of complex communication systems by means of advanced stochastic modelling. We are interested in performance and dependability properties (which aspects are subsumed by the term "performability") and focus on Markovian models derived from high-level specifications. It is well known that the notorious problem of state space explosion can make the storing and analysis of large models difficult or even infeasible in practice. We approach this problem by exploiting the structure of the system to be modelled. For compositional model specification we advocate the use of stochastic process algebras (SPA), which offer composition operators for building large models from small components and enable a compositional reduction of the state space on the basis of bisimulation equivalences. For compact model representation we use binary decision diagrams (BDD) and extensions thereof. It is shown that such symbolic representations can be extremely space-efficient, provided that they are built according to the compositional structure as given by the high-level model specification. We describe a comprehensive set of model construction and analysis techniques which rely completely on the symbolic data structures. These include the BDD construction from explicit representations, symbolic parallel composition, reachability analysis, elimination of instable states (which are caused by immediate transitions), computation of bisimilar states and numerical analysis, which latter is needed for determining the performability measures of the system. It turns out that the runtime of numerical computations, in particular of linear algebraic operations, is currently the bottleneck of the symbolic approach. We provide a systematic analysis of this problem and propose possible solutions for speeding up BDD-based numerical analysis. Then we focus our attention on the type of measure that can be specified and analysed. Traditionally, the purpose of modelling has been to derive state, throughput or more general reward measures. It is shown that such classical measures are often not sufficient when studying complex behavioural properties of interest. Therefore we develop a temporal logic for specifying a more general class of performability properties. This logic is based on actions rather than on elementary state properties and therefore ideally suited to be used in conjunction with SPA models. We describe model checking for this logic, i.e. algorithms for checking whether a certain property actually holds for a given model, and point out that this type of analysis fits in well with our symbolic approach to model representation. Throughout this thesis we refer to software tools that support the described techniques, discussing special features of the tools that were developed during the course of this work. These tools are also employed to carry out the application case studies which are described towards the end of the thesis.

Acknowledgements

This habilitation thesis is the result of several years of research, and many people have, in one way or another, influenced this work. First and foremost, I would like to express my deepest thanks to Prof. Dr.-Ing. Ulrich Herzog, at whose "Lehrstuhl" I have been working since October 1990, and whose support and encouragement in all phases of my academic career have been invaluable to me. I have always profited greatly from the stimulating discussions with Prof. Herzog, and his ideas, advice, and critical reading have boosted the present work enormously.

Secondly, I would like to thank Prof. Dr. Marta Kwiatkowska from the University of Birmingham who graciously agreed to serve as second referee for the thesis. Marta has been familiar with my work for some time, our research groups have interacted and collaborated ever since the joint StochVer project started in October 1998. Within this collaboration, I have greatly benefited from the discussions with Dr. Gethin Norman and Dave Parker, who are also working on symbolic representations of stochastic systems. The exchange of ideas and results with the Birmingham group has been of great value for this work.

Another person who has had a great influence on my work over the last few years is Dr.-Ing. Holger Hermanns. A former colleague at Erlangen, Holger moved to the University of Twente at the end of 1998, but still the intensity of our collaboration grew. Holger's knowledge, insight and creativity are truly admirable, and I consider myself lucky to be able to work with him.

Furthermore I would like to thank Dr. Ir. Joost-Pieter Katoen, whose lecture in the winter term 1997/98 introduced me to the area of model checking. Joost-Pieter has also been a superb coauthor on several papers which we wrote with Holger and Joachim Meyer-Kayser. Joachim is working on temporal logics and verification algorithms and has developed the tool ETMCC for the model checking of stochastic systems, and I have the pleasure of being his group leader. I would also like to thank the two other members of my group "Stochastic Modelling and Verification", Matthias Kuntz and Kai Lampka, for proofreading parts of the thesis and for their excellent cooperation. Dr. Bernhard Wentz, Detlef Kraska and Dr. Andreas Klingler from the computing centre of the Erlangen University hospital deserve thanks for the fruitful cooperation during the years 1996-2001. Together we investigated performance aspects of the Erlangen hospital communication system, which led to one of the case studies in the thesis. Last but not least I would like to thank the many students who contributed to the progress of my research with their "Studienarbeit" or "Diplomarbeit". In particular, I would like to mention Hannes Bruchner and Edgar Frank, who implemented the DNBDD and MTBDD tools which were used in this thesis.

Erlangen, May 2002,
Markus Siegle

Contents

Chapter 1

Introduction

1.1 Current and future trends in communication systems

At the beginning of the third millennium, we live in the age of information, and to be more precise, one could also say that our society has entered the age of communication. The Internet, in particular since the advent of the World Wide Web and electronic commerce, has changed people's lives and already has huge economical and sociological effects, which tendency will even increase in the future. Computers and communication services become more easily accessible and more and more affordable for a large proportion of the world's population. Industry pushes a plethora of innovations in the areas of high-speed and mobile networks, new classes of services and new application areas are being developed and marketed. Smart mobile phones and personal digital assistants, equipped with powerful CPUs and lots of memory, and featuring sophisticated applications such as hand-held Internet browsers, support the trend towards ubiquitous computing and universal access to virtually unlimited information.

Distributed computing, cooperation between physically remote processes, interoperability and information exchange between previously isolated systems are also of increasing importance in established areas such as software engineering, the development of information systems or the design of business processes. As an example, we may mention our own experiences from analysing the communication system of the hospital of the University of Erlangen-Nürnberg, where during the late 1990ies proprietary one-to-one communication has been replaced by the universal exchange of standardised messages, and where currently a distributed

component-based healthcare information architecture is being developed [220]. Improving the communication capabilities of legacy systems is impossible without partly modifying their basic concepts and architecture, and this is usually a very tedious task.

As a result of these trends, communication systems of immense complexity are being developed and will need to be developed in the future, and it will be of the greatest importance that they exhibit functionally correct behaviour and meet very strict performance and dependability requirements. In the next section, we argue that modelling and verification will thus play a more and more important role during the development process of such systems.

1.2 System design and analysis – the role of modelling and verification

The analysis of systems with respect to their performance is a crucial aspect in the design cycle of concurrent information systems. Although huge efforts are often made to analyse and tune system performance, these efforts are usually isolated from contemporary hardware and software design methodology [123, 156, 185]. This insularity of performance analysis has numerous drawbacks. Most severe, it is unclear how to incorporate performance analysis into the early stages of a design, where substantial changes are still not too costly. In these design stages, system models are nowadays developed by means of semi-formal methods such as UML or SDL. In order to overcome the insularity problem, there is a growing tendency towards the integration of performance modelling and analysis into (semi-)formal methods, such as Petri nets [1, 2], process algebras [163], or SDL [257, 256]. This integration has potential benefits for the application of both formal methods and performance analysis: Using a formal method, performance models of interest are readily available for analysis. Conversely, the availability of quantitative insight into a design clearly adds extra value to a formal design.

A typical communication system has a complex life cycle, during which a series of specifications, models, prototypes and products – in general called "artefacts" – is generated. Starting from an initial idea, from a set of (mostly informal) requirements and general design criteria (such as practical and economical considerations, compatibility issues, the preferred hardware/software technology to be used, etc.), the artefacts at subsequent stages are derived partly manually and partly automatically with the help of tools. Before the actual system exists, many kinds of analysis have to be performed on these specifications, models and prototypes, in order to ensure correct system behaviour and detect undesirable

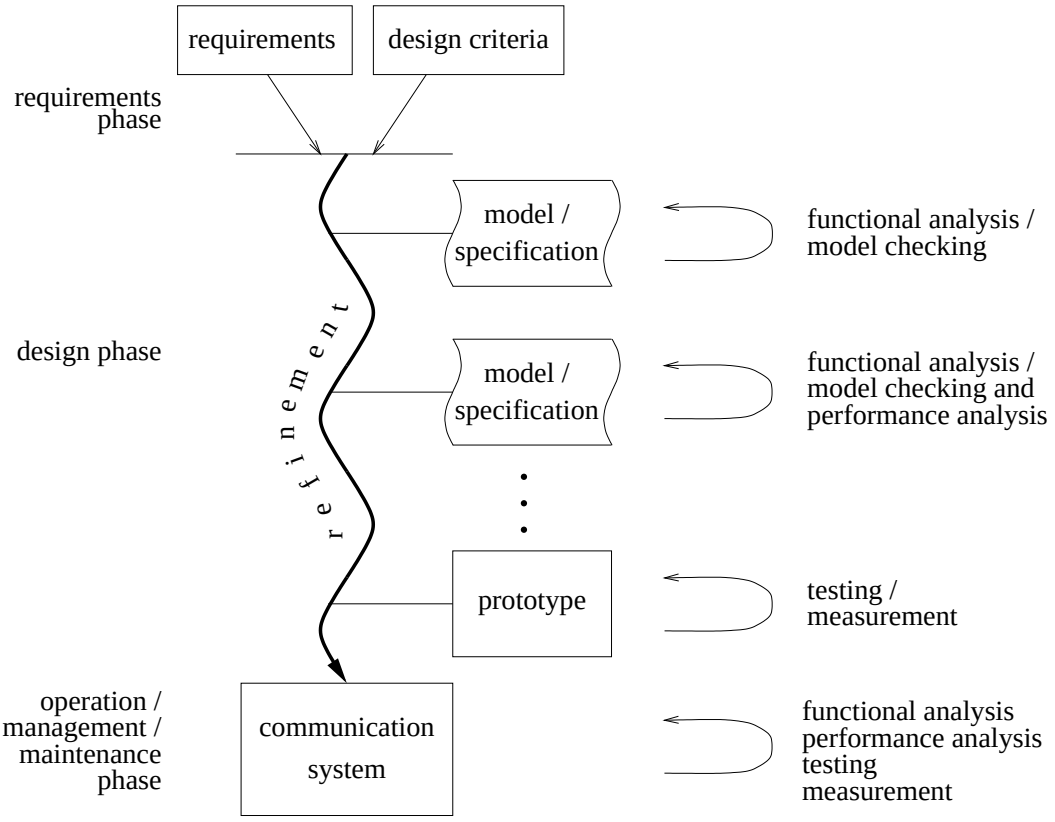


Figure 1.1: Communication system life cycle: The role of analysis during the design phase

effects. Fig. 1.1 presents a coarse overview of the life cycle of a communication system where different types of analysis are sketched at different stages. In particular, the figure shows that the design phase of a typical communication system consists of a complex refinement process. As indicated in the figure by the rollback arrows, the results of analysis may make it necessary to modify the current specification or model, or even force developers to go back to earlier stages within the life cycle. Such rollbacks may be extremely expensive because they require a lot of time and human effort. Therefore, in order to avoid rollbacks as much as possible, model-based analysis of system properties at an early stage during the system life cycle is of very high importance.

The goals of analysis may be manyfold, depending on the current stage of the system life cycle: Developers may be "only" interested in the functional behaviour of the system under development, i.e. in the question, whether the system ex-

hibits the correct functional behaviour, regardless of performance. On the other extreme, they may concentrate on the performance, wishing to know whether the system fulfils certain timing or throughput requirements. However, quite obviously, for a communication system, functional behaviour and performance cannot be considered separately since, for the system to function properly, the right activities have to be performed at the right time.

While monitoring (measurement) [124, 197, 216] can only be carried out once a functional system (or at least a prototype) has been built, model-based validation, verification and performance evaluation of certain aspects of system behaviour can be performed at an early stage, based on the available specifications and models.

In order to avoid expensive maldevelopments, integration of non-functional – i.e. temporal – behaviour analysis into the system development cycle plays an ever increasing role. Therefore, this kind of integration is currently a very active area of research, both in academia and in industry. As a practical example we mention the early integration of performance analysis into the system development cycle with SDL/MSC, involving modelling and simulation [114, 257, 256, 113, 214], testing [286] and measurement [238].

The focus of this thesis is on the study of temporal properties, based on the analysis of stochastic models of the system under consideration. Carrying out this kind of model-based performance evaluation, performance indices are derived by mathematical analysis techniques, usually dependent on a set of varying model parameters. However, from a slightly different perspective, one often needs to answer questions of the type "Does the system fulfil a performance requirement which is related to a particular functional behaviour?". For instance, one may ask the question "Is the probability that a SEND message is answered by an ACK message within 50 ms greater than 95%?". We argue that traditional techniques for the definition and calculation of performability[1] measures are not sufficiently formalised to enable a flexible and automated evaluation of such problems. This consideration leads us to the domain of model checking, where complex system requirements are specified in a formal way with the help of temporal logics, and thereafter checked by model checking algorithms. In the past (as indicated in Fig. 1.1), this area of research has dealt mostly with purely functional properties. In order to answer questions related to timing and performance behaviour, "classical" model checking techniques need to be extended in various directions, in order to take into account real-time, stochastic time and probabilistic behaviour.

[1]The term "performability" is an artificial combination of the terms "performance" and "dependability" and was coined by J.F. Meyer, see for instance [248, 249]. Note that in this thesis we employ the term "performability" in a rather broad sense, characterising general measures and behavioural properties related to performance and dependability aspects.

1.3 Limiting factors of modelling

While model-based analysis is a very valuable technique and of high importance during the life cycle of communication systems, it has of course certain limitations. It is in the nature of a model or specification that it abstracts from reality and only captures certain, carefully chosen aspects of the real system. The developer must always be aware of this fact and ensure that the level of detail of the current model is appropriate for the analysis at hand. Each model has to represent reality well enough in order to allow the modeller to derive meaningful answers from it, but on the other hand a model must not be too detailed, because an excess of detail might render the model too difficult to understand or impossible to analyse.

Apart from the aforementioned general limitations of modelling, *state space explosion* is a very serious problem in the context of state-space-based modelling (whether it be purely functional or stochastic/temporal). In many areas of system design and analysis, there is the need to generate, manipulate and analyse very large state spaces. More precisely, the models to be analysed are often represented as state-transition systems, namely labelled transition systems (LTS) or – in case of stochastic models – discrete time or continuous time Markov chains (DTMC or CTMC). These "low-level" representations are usually derived mechanically from high-level formalisms such as formal description techniques (FDT) or specification languages, queueing networks, (stochastic) automata, (stochastic) Petri nets or (stochastic) process algebra descriptions. The inherent concurrency of the high-level representation is often translated into an interleaving of all possible moves, i.e. all possible totally ordered sequences of actions are included explicitly in the low-level representation. As a consequence, the number of states tends to grow exponentially in the number of parallel components of the high-level model from which it is derived. Such large state-transition systems are often very difficult to handle in practice, due to memory limitations of the available computing equipment.

The complexity of the system to be modelled is one of the reasons that give rise to large state spaces. There are, however, a number of other reasons. As already mentioned, the use of interleaving semantics, when translating from high-level formalisms to the underlying state-transition systems, constitutes one of the general sources of state space explosion and is therefore sometimes referred to as the "interleaving trap".

The use of modular or structured models – consisting of a number of interacting components – makes it easier for the human user to specify complex systems. However, when the structured high-level model is translated into its corresponding low-level representation, the size of the state space is often exponential in

the number of components (depending on the degree of independence between the components), leading to state space explosion. Therefore, alternative representations (such as the Kronecker approach or the symbolic approach to be discussed in depth in this thesis) which avoid this exponential blow-up are of high importance when working with structured models.

As an additional factor when dealing with stochastic models, the use of approximations of non-exponential distributions by phase-type distributions can lead to an enormous further growth of the state space. Using phase-type distributions, a generally distributed duration is represented by a number of fictitious exponential phases, each of which leads to a distinct state. This phenomenon is closely related to the interleaving of events, since for two or more concurrently enabled phase-type distributions every combination of intermediate phases manifests itself by a distinct state of the overall model.

1.4 Overview of approaches to the state space explosion problem

Fig. 1.2 gives an overview of possible approaches to the state space explosion problem which we will discuss in the following sections. The figure distinguishes between the two main categories of non-state-space-based approaches and state-space-based approaches. The former are characterised by the fact that analysis is carried out on the basis of some high-level model specification without the need to explicitly generate its possibly huge underlying state space. These approaches are typically much more efficient than state-space-based analysis, but unfortunately they are restricted to special classes of models and therefore not universally applicable. Within the class of state-space-based approaches, Fig. 1.2 further distinguishes between largeness tolerance and largeness avoidance, which terms will be discussed briefly at the beginning of Sec. 1.6.

Our main focus in this thesis is on space-efficient "symbolic" encodings of large state spaces and transition systems, based on structured high-level specifications. For the encoding (i.e. the symbolic representation) we use binary decision diagrams and extensions thereof as the underlying data structure. This approach, which is summarised in Sec. 1.7.5 and elaborated on in Chaps. 4 – 8, belongs to the category of state-space-based, largeness avoidance methods. In the next sections, however, we first briefly review the other approaches contained in Fig. 1.2.

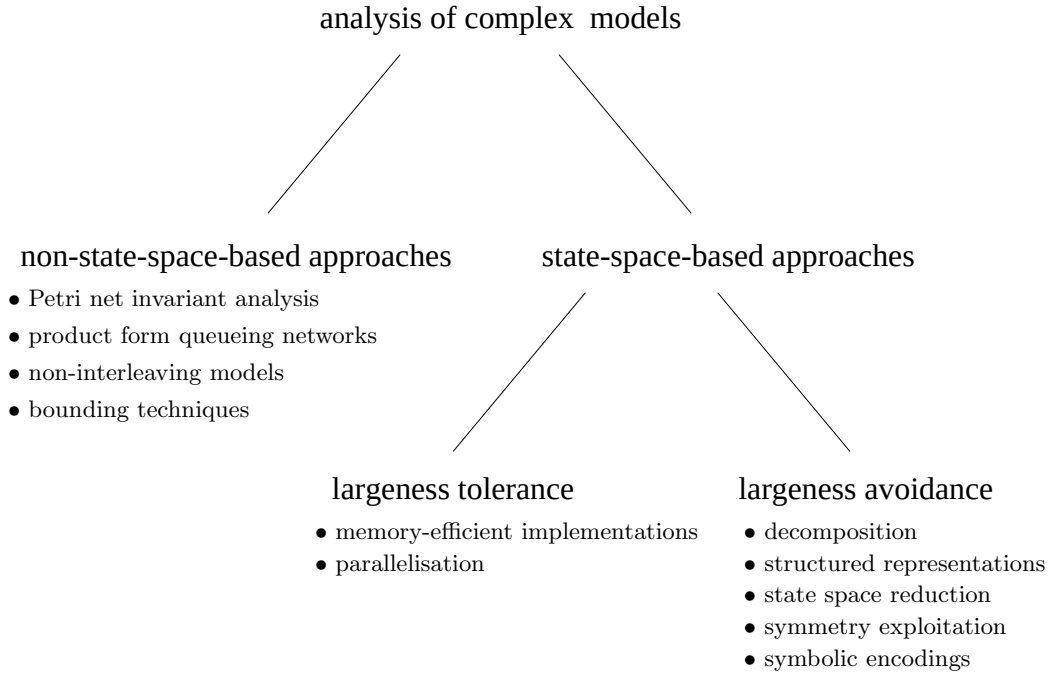


Figure 1.2: Overview of approaches to the state space explosion problem

1.5 Non-state-space-based analysis

For some special classes of models it is possible to prove functional properties or to derive performance measures, ranging from individual state probabilities to cumulative performance indices, directly from the high-level model specification, without ever generating the underlying state space. While these techniques are far more efficient than state-space-based analysis techniques, their application is restricted to specialised model classes.

1.5.1 Petri net analysis by invariants

As an example for non-state-space-based functional analysis we mention the analysis of Petri nets by invariants, which can be applied to verification, proof and analysis of behavioural properties of Petri nets. This kind of analysis works on the static Petri net structure and employs linear algebraic methods. Taking into account the information on the initial net marking it yields results concerning the reachability or non-reachability of certain markings, conditions that hold true for all reachable markings, and identifies transition firing sequences which lead back to the original marking [311, 105].

It is known that Petri net invariants can also be exploited for the purpose of performance analysis. In [74, 302], efficient algorithms for the computation of performance bounds are described, which work entirely at the structural level, without generating the reachability set (i.e. the state space).

1.5.2 Product form queueing networks

Maybe the most prominent example for non-state-space-based methods in the area of performance evaluation is the analysis of product form queueing networks (PFQN), for instance the class of BCMP queueing networks [24], where the steady-state probabilities can be derived immediately once some rather simple "traffic equations" have been solved. For closed PFQNs, performance indices such as the mean queue population or the mean residence time at a queueing station can be computed directly by the mean value analysis (MVA) algorithm [277]. However, these methods and their associated efficient computational algorithms are not applicable (in an exact mathematical sense) to general queueing networks but only to certain well-defined subclasses.

Product form solutions do not only play a role in the context of queueing networks, but have also been developed for other modelling formalisms. For instance, there are also classes of stochastic Petri net models [159, 235, 290], queueing Petri nets [25] and stochastic process algebra models with product form [150, 289]. All of this research has been stimulated by the success of product form queueing networks, and the basic idea of all these approaches is, of course, closely related to product form queueing networks.

1.5.3 Non-interleaving models

In this section, we briefly summarise a class of approaches which, generally spoken, avoid the interleaving trap. These non-interleaving approaches are often also referred to as partial order techniques or causality-based techniques.

In the interleaving approach, one abstracts from the fact that the overall model may consist of more than one interacting submodel. The overall system's behaviour is modelled by sequences of events which are totally ordered. In contrast, the partial order reflects the causal dependences between events within different submodels, but there is no need to force a particular order on those events which are not causally dependent. Therefore, following the partial order approach, "true concurrency" of events (of actions) is allowed.

As an example we consider stochastic task graphs where a node starts execution once all its predecessors are finished and each node has a given stochastic runtime distribution (for stochastic graph models see also Sec. 3.1). The special subclass of stochastic task graphs with series-parallel structure is a partial order model, where the distribution of the overall execution time can be calculated in a very efficient manner by successive computation of sum and maximum of node runtime distributions. For generally structured graphs, bounds can be obtained by modifying the graph structure in order to make it series-parallel [218, 308, 153, 151].

Another, very similar, example is the stochastic causality-based process algebra of [42] and [208], where event structures are extended in order to be able to represent stochastic process algebra models. Furthermore, techniques such as partial order reduction have been developed and successfully applied in the area of model checking in order to prevent state space explosion (see e.g. [139]).

1.5.4 Bounding techniques

As a last example from the class of non-state-space-based approaches, we now briefly sketch the idea of bounding techniques. Such techniques are described, for instance, in [236]. They are used in order to gain insight into the primary factors affecting the performance of the system under investigation. A bounding technique consists of the computation of upper and lower bounds of the performance measures as a function of the system workload. Typically, such computations can be carried out quickly by means of simple formulas, assuming extreme conditions of light or heavy loads. Applied to networks of queues, upper and lower bounds on system throughput and on the response time can be derived, and it is known that for balanced systems (where a customer's service demand is the same at every service centre, cf. [236]) tighter bounds can be obtained than for general, unbalanced systems.

1.6 State-space-based analysis: Largeness tolerance

The terms "largeness tolerance" and "largeness avoidance" (see Sec. 1.7) were coined by Trivedi et al., see for instance [318]. While the former summarises sparse storage techniques and memory-efficient — possibly parallel — numerical methods, the latter subsumes approaches that try to keep the size of the model representation as small as possible at every stage of the modelling and analysis

process. This aim may be achieved with the help of state truncation, state space reduction, hierarchical or structured model description and solution techniques, or efficient encodings.

1.6.1 Memory-efficient and parallel implementations

As an example of memory-efficient analysis, we mention the disk-based approach by Deavours et al. [103]. They employ a block Gauss-Seidel (BGS) method for the solution of Markov models on a single processor where the generator matrix is stored on disk. The solver maintains high disk throughput using a system of two cooperating processes which perform disk I/O and computation concurrently. The memory usage of the BGS solver is low, the main requirement being the space for the solution vector. By this approach, it is possible to solve systems of 10 million states and 100 million transitions on a workstation with only 128 MB RAM.

Next we consider parallel or distributed state space generation. The paper [69] describes parallel state space generation on a CM-2 SIMD machine with massive parallelism (8K processors), integrated into the stochastic Petri net tool GreatSPN [73]. The authors state that "sources for massive parallelism have actually been found in the problem, but they do not match well with the strictly SIMD type model of computation supported by the CM-2". Extensions of this are described in [70], and more related work can be found in [5, 6, 7, 78, 226]. In [157], good state encoding techniques and hashing tables are used, such that models with 55 million states can be done on a single workstation, and models with 400 million states can be done on a cluster of 16 workstations in reasonable time. In [261], an alternative approach is taken, which uses a heuristic method instead of hashing.

More recently, Knottenbelt et al. [224, 225] developed distributed disk-based techniques not only for state space generation, but for Markov chain analysis based on the Jacobi and Conjugate Gradient Square (CGS) methods. Exploiting the structure induced by breadth-first search state generation algorithms, an efficient matrix-vector multiply kernel is developed which exhibits low memory usage, low communication overhead and good load balance. Markov chains of up to 50 million states and 500 million transitions were solved on a distributed memory computer (a Fujitsu AP3000 with 16 nodes, each running at 300 MHz and equipped with 256 MB RAM). Parallel transient analysis of stochastic reward nets for both distributed and shared memory machines is described in [8].

Another approach to memory-efficient implementations is on-the-fly generation of

the state space, i.e. rather than permanently storing states and transitions these are re-generated every time they are needed. One such approach, matrix-free-iteration (MFI), is described in [222]. In this work, Markovian queueing networks (closed single class networks and extensions to (non-product-form) multi-chain networks) are considered. Gauss-Seidel iteration is used, but the concept could also be applied to other iterative schemes. The solvability of the linear system of equations depends only on the size of the iteration vector. Models with up to 70 million states and 1373 million transitions have been analysed (but one iteration took 1 hour of user time !).

1.7 State-space-based analysis: Largeness avoidance

1.7.1 Decomposition

The idea of decomposing a large state space in order to make it tractable was presented for the first time in [303] by Simon and Ando (they were interested in an application from economy). Some years later, Courtois [92] used the same approach in the context of computer performance evaluation. Instead of analysing one large system, the decomposition approach relies on analysing several small subsystems, analysing an aggregated overall system, and afterwards combining the subsystems' solutions. Thus, the decomposition/aggregation approach consists of three steps: Decomposition, aggregation and combination. In general, the decomposition approach works well if the state space can be partitioned into subsets of states, such that there is a lot of interaction between states belonging to the same subset, but little interaction between states belonging to different subsets. Systems which possess this property are called nearly completely decomposable (NCD). For the class of reversible Markov chains, the decomposition/aggregation approach yields exact results [90], and the approach may also be applied iteratively [93, 68]. Decomposition-based analysis is also considered in [77, 82, 83], where the focus is on approximate decomposition for nearly-independent GSPN structures.

Mertsiotakis et al. developed approximate decomposition-based analysis methods for Stochastic Process Algebra models [245]. Time scale decomposition (TSD) is based on the concept of nearly completely decomposable Markov chains [193, 244]. Response time approximation (RTA) relies on a structural decomposition for the special class of decision-free process algebraic models [246, 247]. Another analysis approach, based on the exploitation of the structure of a special class of process algebraic models is described in [30].

1.7.2 Structured representations

We strongly believe that the structure of the real system to be analysed should play a very important role during the construction and analysis of a model. For the human modeller, structured models are much more manageable than monolithic ones, because they allow one to concentrate on a particular part of the model at a time. During model analysis, i.e. state space construction, state space reduction, functional and temporal analysis, the structure of the model may be exploited, resulting in reduced memory requirements and opening the way for specialised analysis techniques which make even highly complex models computationally tractable.

The Kronecker approach (also called tensor approach) [271, 272, 50, 54, 57, 107, 292, 80], where an overall model is constructed from a set of interacting submodels, relies heavily on the structure of the system to be modelled. The main advantage of this approach is the fact that the generator matrix of the overall model never has to be generated explicitly, it is only described implicitly in the form of a so-called tensor descriptor, a tensor expression which involves submodel matrices of small size. The memory requirements of this approach can be kept extremely low, since it suffices to store matrices of the size of the submodels. All operations necessary for the computation of the stationary and/or transient state probabilities (mainly vector-matrix multiplication) can be performed by accessing only the submodel matrices. Specialised algorithms have been developed [59, 60], which however are notably slower than comparable algorithms which work on explicit sparse matrix representations of the overall model. Thus, the advantage of the Kronecker approach lies exclusively in the compactness of its model representation. We will discuss the Kronecker approach in more depth in Sec. 3.6.1.

1.7.3 Constructing minimal representations

A very general approach to the state space explosion problem works as follows: One defines an equivalence relation among states, which yields a partitioning of the state space, equivalent states forming a subset (class) of the partition. From this partition a reduced model is constructed where each class of states is represented by a single “macro” state. The reduced model is then analysed, which is cheaper than the analysis of the original model, and afterwards certain (if not all) results for the original model may be derived from the solution of the reduced model.

The specific equivalence relation used for constructing the partition depends on the context: For Markov chains (DTMCs or CTMCs), the well-known notion of lumpability is defined [212, 55]. More specifically, there exist the notions of ordinary lumpability, exact, strict and weak lumpability. For instance, two states of a CTMC are said to be ordinarily lumpable if they can move (directly) to the same equivalence classes with the same rates. For purely functional labelled transition systems (without rates or probabilities) we have the notions of (strong or weak) bisimilarity. Two states are bisimilar, if they can move to the same equivalence classes with the same actions. For transition systems extended by transition rates, i.e. the combination of CTMCs and LTS, we have the notions of (strong or weak) Markovian bisimulation. Two states are Markovian bisimilar, if they can move to the same equivalence classes with the same actions and with the same cumulative rates (a formal definition is provided in Section 3.7).

Algorithms for constructing a partition which corresponds to a given equivalence relation are known and their complexity has been analysed. For instance, strong Markovian bisimulation can be implemented with time complexity $\mathcal{O}(m \log n)$ where n is the number of states and m is the number of transitions. Weak Markovian bisimulation can be implemented with time complexity n^3, which is due to the fact that the transitive closure of weak transitions must be computed [162].

In practice, computing the partition of the state space can be very expensive. For that reason, one should not blindly generate the overall state space of a model and afterwards try to identify classes of equivalent states. In contrast, it is essential to use the information contained in the model structure as a basis for state space reduction. For instance, if the model consists of symmetric (or replicated) components, one can directly construct a reduced state space, because it is known a priori which states are "symmetric" to each other and thus equivalent (see Sec. 1.7.4). As another example, in the context of (stochastic) process algebras, bisimulation minimisation can be applied in a compositional fashion, which means that the state space of a component is minimised before that component is composed in parallel with other components. Following this scheme, it can be ensured that the state space of any intermediate model is always kept at a minimum, and the non-reduced state space of the overall model never needs to be constructed (or stored) explicitly.

1.7.4 Exploitation of model symmetries

Technical systems consist of components or parts, some of which may be repeated or replicated several times. For instance, a MIMD multiprocessor system contains

several identical processor/memory modules. Several stations connected to a local area network may exhibit similar behaviour and thus be considered symmetrical. In a cellular network, a number of mobile stations within reach of the same base station may all have similar statistical behaviour.

Such symmetry in the real world should also be reflected in models of these systems. Using state-space-based modelling techniques, the information about model symmetries can be exploited during the modelling process, leading to a possibly vast reduction of the state space size. The general idea is to identify "symmetric" states and combine them into a single macro state, thereafter performing analysis on the basis of the macro states.

In the modelling formalism of stochastic activity networks, an extension of stochastic Petri nets, specification of symmetric systems is supported with a special "replicate" operator [283], and during state space construction a reduced base model is directly constructed from the high-level specification. Symmetries also play a predominant role for the analysis of stochastic well-formed coloured Petri nets [76, 127] where a reduced reachability graph is constructed directly from the net description, without the need to construct the full (expanded) reachability graph first. The special role of symmetries in connection with the Kronecker approach has been studied in [291, 293], where an algorithm for directly generating the reduced state space from a structured high-level specification is described. For the Markovian framework of [293], it was shown that symmetry reduction corresponds to strict lumpability. In the the context of stochastic process algebras, it is known that model symmetries lead to transition systems with bisimilar states, which can be reduced on the basis of bisimulation equivalences. Symmetry exploitation for stochastic process algebras is described in [280, 182, 137].

We emphasise that it is essential for successful symmetry exploitation that the reduced model can be generated directly from a high-level description, without the need to generate the full-blown low-level model first, since the latter may be too large to be computationally tractable.

1.7.5 Symbolic encodings

In recent years, the problem of generating, representing and analysing large transition systems has been very successfully approached by using symbolic representations, in particular binary decision diagrams (BDD) and derivatives thereof, see e.g. [45, 63, 46, 65, 118][2]. Most of this work took place in the areas of design and

[2]BDD-based representations (also called encodings) of sets, transition relations or transition systems are commonly termed "symbolic" in the literature, as opposed to traditional, explicit

verification of digital circuits and model checking of concurrent systems, i.e. areas where state space explosion is also a severe problem. Experience showed that symbolic representations make it possible to handle much larger state spaces than traditional methods. Applying symbolic techniques, the border between manageable and unmanageable sizes of verification problems has been moved upwards by several orders of magnitude. Therefore, today, mechanised hardware verification is considered by many companies to have industrial strength [88].

BDDs are compact canonical representations of Boolean functions as directed acyclic graphs, where the representation of redundant information is completely avoided. State spaces or transition systems can be represented as BDDs by a binary encoding of state identities or state-to-state transitions. In view of the numerous success stories related to BDDs, a proper assessment of BDDs must mention that much of this success is based on heuristics concerning the efficient encoding of state spaces as Boolean functions. Without applying such heuristics, BDDs are usually no more space efficient than conventional, explicit state space representations.

Most of the literature on BDDs deals with functional behaviour only. For representing stochastic performance models, the basic BDD data structure has to be extended, such that, for instance, transition probabilities or transition rates can be included in the symbolic representation. Multi-Terminal BDDs [86] (called Algebraic Decision Diagrams in [147]), Edge-Valued BDDs [232], Binary Moment Diagrams [48] and Decision Node BDDs [294, 295, 298] are all extensions of standard BDDs capable of representing numerical information. Symbolic representation of stochastic systems has not got much consideration in the literature, but some pioneering previous work exists [147, 131, 294, 295, 177]. The latter three references document the stages of our own work on the symbolic representation and analysis of stochastic systems.

Motivated by the success of symbolic representations in other fields and by the mentioned pioneering work, we propose in this thesis an approach to the representation and manipulation of stochastic transition systems which is entirely based on symbolic techniques. We develop strategies for the space-efficient representation of large stochastic models, pointing out the importance of taking into account the model structure. In particular, we discuss BDD-based parallel composition, i.e. the parallel composition of submodels which are represented as decision diagrams. Furthermore, we discuss BDD-based algorithms for reachability analysis, hiding of functional information, vanishing state elimination, bisimulation and even numerical analysis of the stochastic process.

---

representations. This use of the term "symbolic" should not be mixed up with "symbolic" formula manipulation packages.

1.8 Organisation of this thesis

Fig. 1.3 provides an overview of the four parts of this thesis. We now briefly describe the content of each chapter and sketch the general line of thought.

In Chap. 2, the foundations of stochastic processes, transition systems and binary decision diagrams are revisited. Chap. 3 contains a survey of techniques for high-level model specification and their particular benefits and shortcomings. In view of the tractability of complex models with large state spaces, this chapter emphasises the importance of structured models and contains a detailed introduction to stochastic process algebras. In Chap. 4, we discuss the symbolic representation of transition systems with the help of binary decision diagrams, since this data structure enables extremely space-efficient representations of huge state spaces. The chapter also introduces extensions of binary decision diagrams for the representation of stochastic systems. BDD-based compositional model construction, as well as state space manipulation and reduction algorithms are presented in depth in Chap. 5. In particular, it is shown that symbolic parallel composition is the key to compact representations, and this theme is carried on in Chap. 6, which contains a study of the factors influencing the compactness of the symbolic representation. Chap. 7 describes the numerical analysis of Markov chains based on their symbolic representation, thereby rounding off our symbolic approach to modelling and analysis. In Chap. 8, after showing that the algorithms for numerical computation are currently the main bottleneck of the symbolic approach, we address the challenging problem of speeding up BDD-based analysis techniques. Chap. 9 is a short introduction to the verification of (temporal) behavioural properties by means of model checking. We motivate the importance of this topic by considerations on the class of performance and dependability measures a user may wish to specify and compute, and by the fact that symbolic techniques are well established in the area of verification. The chapter introduces the temporal logic aCSL and gives an overview of the associated model checking algorithms. In Chap. 10, we present three application case studies, after which we wrap up in Chap. 11 with conclusions and suggestions for future research.

As can be observed from Fig. 1.3 (and as already mentioned in Sec. 1.4), with Part II bearing the main weight of the thesis, our main focus is on space-efficient symbolic encodings, based on structured high-level specifications of large performability models.

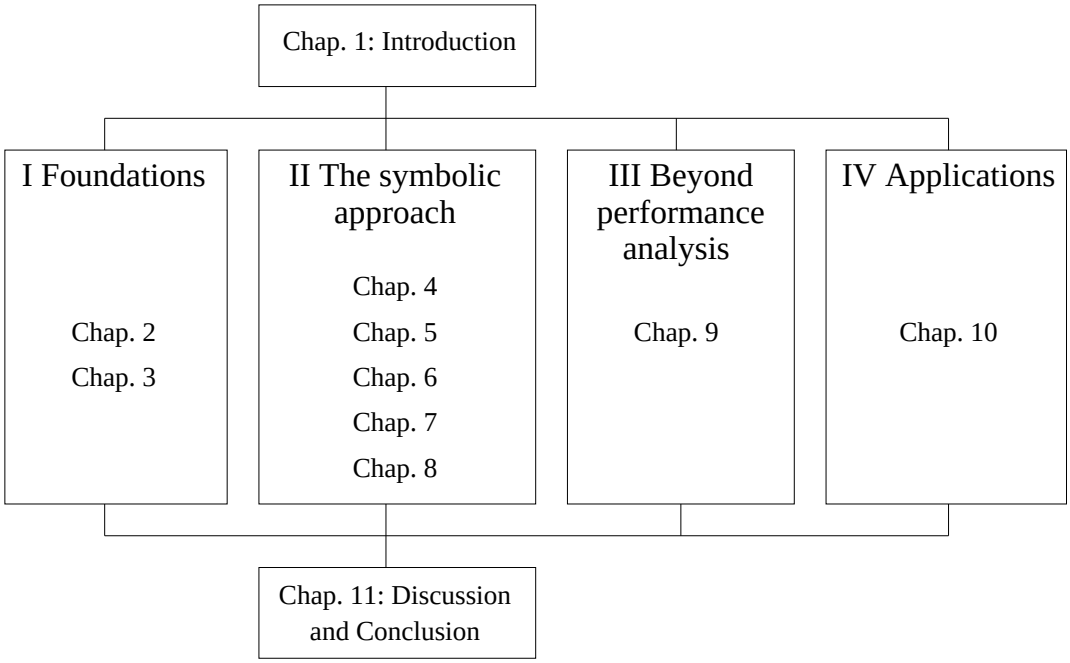


Figure 1.3: Overview of thesis

Part I

Foundations

Chapter 2

Preliminaries

In this chapter, we recall the definitions of some fundamental concepts and data structures which are needed as basic ingredients for modelling and verification.

Most "low-level" representations used for the modelling of the dynamic behaviour of sequential, parallel or distributed systems are based on the notions of "states" and "transitions" between states, together with some appropriate labelling of these states and transitions. A state characterises the system in a particular situation, often involving a rather radical abstraction from the situation of the real system, while transitions denote the change of the system's state.

Examples for such basic state-transition representations are finite state automata, labelled transition systems and Kripke structures. Among the more advanced concepts which offer structuring concepts, at the cost of a more intricate semantics, are statecharts [149, 96].

If one is not only interested in purely functional properties of a system but also wishes to express probabilistic or stochastic aspects, possibly involving quantitative timing information, then Markov chains, where transitions between states are labelled by probabilities or rates, are often an appropriate means for describing a system's behaviour. Among other formalisms including time are timed automata [11] and stochastic automata [106].

In this chapter, we focus on Markov chains and labelled transition systems, including stochastic extensions of the latter. Furthermore, we introduce binary decision diagrams which are the basis of the symbolic encodings which we shall study in depth in Chaps. 4 – 8.

2.1 Stochastic processes

A stochastic process [71, 219] is a family of random variables $\{X(t) \mid t \in T\}$ which take on values from a state space S, indexed by a parameter $t \in T$. The index may represent an arbitrary physical quantity, but is often interpreted as a time parameter. There are several classification schemes for stochastic processes: On the one hand, the state space S of the random variables may be either discrete or continuous, yielding discrete or continuous state space stochastic processes. Stochastic processes with discrete state space are also called stochastic chains. On the other hand, the (time) parameter space T may be either discrete or continuous, depending on whether the process is observed at discrete time instants or over the full real time axis. Altogether, the combination of discrete/continuous state space with discrete/continuous time parameter yields four types of stochastic processes.

Another, more intricate, classification is based on the stochastic dependence between the random variables at different time instants. In general, a complete characterisation of a stochastic process would need to specify the joint distributions

$$F_{X_1,\ldots,X_n}(x_1,\ldots,x_n;t_1,\ldots,t_n) = P[X(t_1) \leq x_1,\ldots,X(t_n) \leq x_n]$$

for all n, for all $x_1,\ldots,x_n$ and for all $t_1,\ldots,t_n$. This, of course, would be extremely tedious and is totally infeasible in practice. However, in most cases a simpler description is sufficient. For instance, discrete state stochastic processes may be characterised by two criteria:

- the probability p_{ij} that the next state will be state j, provided that the current state is state i.
- the time spent in state i (the state holding time), characterised by its distribution function $F_i(.)$.

A very general class of stochastic processes, where both the p_{ij} and the $F_i(.)$ can be arbitrary, is the class of semi-Markov processes [71, 138]. Other classes of interest are, for example, random walks and renewal processes.

Of particular interest for the area of performance and dependability modelling is the class of Markov processes, since Markov processes have proved to be a suitable means for describing many phenomena in computer and communication systems, and because their mathematical analysis is well understood. We will restrict our attention to Markov processes with discrete state space, i.e. to Markov chains.

Depending on the properties of the time parameter, one obtains either Discrete Time Markov Chains (DTMC) or Continuous Time Markov Chains (CTMC). As their distinguishing feature, Markov processes enjoy the property of *memorylessness*, which greatly simplifies their analysis, compared to general stochastic processes. Roughly speaking, memorylessness means that the behaviour of the process in the future only depends on the current state, and not on the past history of the process. One can express this formally by:

$$\begin{aligned} & P[X(t_{n+1}) = x_{n+1} \mid X(t_n) = x_n, X(t_{n-1}) = x_{n-1}, \ldots, X(t_0) = x_0] \\ = \; & P[X(t_{n+1}) = x_{n+1} \mid X(t_n) = x_n] \end{aligned}$$

for any set of time instants $t_0 < t_1 < \ldots < t_{n+1}$. As a consequence, the time already spent in a state has no influence on the time until that state will be left, from which it follows directly that for DTMCs the holding time of a state is geometrically distributed, and for CTMCs the holding time in a state is exponentially distributed[1] [314].

2.1.1 Continuous time Markov chains

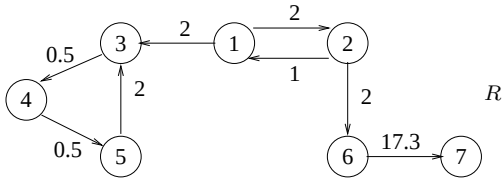
While the behaviour of DTMCs is characterised in terms of transition probabilities between states, transitions of CTMCs are determined in terms of rates, as formalised in the following definition.

Definition 2.1.1 Continuous Time Markov Chain (CTMC)
A Continuous Time Markov Chain is defined by a tuple $\mathcal{C} = (S, R)$ where S is a (finite or infinite) set of states, and $R : S \times S \to \mathbb{R}_{\geq 0}$ is the matrix of transition rates satisfying $R(s, s) = 0$ for all s. ■

Sometimes a uniquely defined initial state of the CTMC or an initial state probability distribution is included in the definition. Note that the above definition does not allow self-loops, i.e. transitions leading back to the same state, since such transitions would not have any influence on the state probabilities of the CTMCs. In a compositional context, however, where transitions are labelled by action names (as is the case for SLTSs, see Sec. 2.2), self-loops may be allowed and are a decisive factor for the behaviour of the system.

The rate matrix R characterises the transitions between the states of the CTMC. If $R(s, s') > 0$ then it is possible that a transition from state s to state s' takes

[1] The geometric (exponential) distribution is the only discrete (continuous) distribution which enjoys the memoryless property.



$$R = \begin{bmatrix} 0 & 2 & 2 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.5 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 17.3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 2.1: Example CTMC and corresponding transition rate matrix R

place. Conversely, if $R(s, s') = 0$ then no such transition is possible. If state s has only a single possible successor state s', then the probability of moving from state s to s' within t time units (for positive t) is given by $1 - e^{-R(s,s')\cdot t}$. This expression is the cumulative probability distribution function of an exponential distribution with rate $R(s, s')$.

In the case where $R(s, s') > 0$ for more than one state s', a competition between the transitions exists, also called a *race*. Let $E(s) = \sum_{s' \in S} R(s, s')$, the total rate at which any transition emanating from state s is taken ($E(s)$ is also called the exit rate of state s). This rate is the reciprocal of the mean sojourn time (also called mean holding time) in s. More precisely, $E(s)$ specifies that the probability of leaving s within t time units is $1-e^{-E(s)\cdot t}$, due to the fact that the minimum of exponential distributions (competing in a race) is again exponentially distributed, and characterised by the sum of their rates. Consequently, the probability of moving from state s to s' by a single transition, denoted $p(s, s')$, is determined by the probability that the delay of going from s to s' finishes before the delays of other outgoing edges from s; formally, $p(s, s') = R(s, s')/E(s)$ (except if s is an absorbing state, i.e. if $E(s) = 0$; in this case we define $p(s, s') = 0$).

Figure 2.1 shows an example CTMC and its rate matrix R. Note that from states 1 and 2 all other states are reachable. However, once state 3 is entered, the Markov chain will remain within the subset of states $\{3, 4, 5\}$ forever. We call such a subset (which cannot be left and whose states are all mutually reachable) a bottom strongly connected component (BSCC). Whenever state 6 is entered, the next transition will inevitably lead to state 7 which does not possess any outgoing transitions. Such a state which cannot be left is called *absorbing*. An absorbing state can also be viewed as a BSCC containing only a single state. States which do not belong to a BSCC are called *transient*. A CTMC consisting of a single BSCC is called *irreducible*.

A CTMC is called *homogeneous* if the rates are time-independent, and in the sequel we will assume homogeneity, since this assumption simplifies analysis considerably. An infinitesimal generator matrix Q is derived from R by set-

ting $Q(s,s') = R(s,s')$ for $s \neq s'$ and replacing the diagonal elements of R by $Q(s,s) = -E(s) = -\sum_{s' \neq s} R(s,s')$.

2.1.2 Transient analysis

Given a CTMC $\mathcal{C} = (S,R)$, one of the main goals is the computation of its state probabilities at a fixed time [314]. The vector $\vec{\pi}(t)$, whose length is $|S|$ and whose elements are such that $\sum_{s \in S} \pi_s(t) = 1$, denotes the probability distribution on S at time instant t. Of course, the probability distribution at time t depends on the initial state, i.e. on the distribution $\vec{\pi}(0)$ at time instant $t = 0$. The state probability distribution at time t is obtained by solving the following Kolmogorov system of differential equations

$$\frac{d\vec{\pi}(t)}{dt} = \vec{\pi}(t) \cdot Q$$

The unique solution of this system (given boundary condition $\vec{\pi}(0)$) is given in closed form by

$$\vec{\pi}(t) = \vec{\pi}(0) \cdot e^{Qt}$$

In this equation, the matrix exponential is defined by its series expansion as $e^{Q \cdot t} = \sum_{k=0}^{\infty} (Q \cdot t)^k / k!$. For the practical computation of $\vec{\pi}(t)$ one constructs the stochastic matrix $P = Q \cdot \Delta t + I$, where $1/\Delta t = q$ must be larger than the maximum exit rate of the CTMC (a common choice is $q = 1.02 \cdot \max_{s \in S}\{E(s)\}$ [322]) and I is an identity matrix of the appropriate size. This construction is called uniformisation [205, 143], and we call $1/\Delta t = q$ the uniformisation constant. Substituting $Q = P \cdot q - I \cdot q$, we obtain $e^{Q \cdot t} = e^{P \cdot q \cdot t} \cdot e^{-I \cdot q \cdot t} = e^{P \cdot q \cdot t} \cdot I \cdot e^{-q \cdot t} = e^{P \cdot q \cdot t} \cdot e^{-q \cdot t}$, which yields the following expression for $\pi(t)$:

$$\begin{aligned} \vec{\pi}(t) &= \vec{\pi}(0) \cdot e^{Q \cdot t} \\ &= \vec{\pi}(0) \cdot e^{P \cdot q \cdot t} \cdot e^{-q \cdot t} \\ &= \vec{\pi}(0) \cdot \textstyle\sum_{k=0}^{\infty} P^k \cdot \frac{(q \cdot t)^k}{k!} \cdot e^{-q \cdot t} \end{aligned}$$

In this expression, the weight factors $\frac{(q \cdot t)^k}{k!} \cdot e^{-q \cdot t}$ are known as the Poisson probabilities (the discrete probabilities of the Poisson distribution). In practice it suffices to evaluate a finite number of terms of this infinite sum, i.e. the infinite summation $\sum_{k=0}^{\infty} P^k \cdot \frac{(q \cdot t)^k}{k!} \cdot e^{-q \cdot t}$ is replaced by the finite summation $\sum_{k=L}^{R} P^k \cdot \frac{(q \cdot t)^k}{k!} \cdot e^{-q \cdot t}$, where the left and right truncation points (L and R) depend on $q \cdot t$ (the product of the uniformisation constant and the time instant) and on a pre-specified precision ϵ. For details on this and the efficient evaluation of the Poisson probabilities we refer to [126].

2.1.3 Steady-state analysis

Of particular interest is the behaviour of the Markov chain in the long run, i.e. for $t \to \infty$, given by the so-called steady-state (or stationary) probability vector $\vec{\pi} = \lim_{t\to\infty} \vec{\pi}(t)$. The steady-state probabilities exist for arbitrary homogeneous CTMCs with finite state space S, but they are known to depend on the initial behaviour of the chain if the latter is *not* irreducible. Remember that a chain is irreducible if its state graph is strongly connected, i.e. if there is a directed path of transitions with positive rates between each ordered pair of states in S.

In the case of a reducible Markov chain, the picture is quite complicated: After an infinite time, the CTMC is certainly no longer in any transient state, but will be in one of its BSCCs and remain there forever. The probability of reaching a particular BSCC can be calculated easily. For example, in the CTMC depicted in Figure 2.1, the probability of reaching BSCC $\{3, 4, 5\}$, provided that the initial state is state 1, is given by $\frac{1}{2}+\frac{1}{2}\cdot\frac{1}{3}\cdot\frac{1}{2}+(\frac{1}{2}\cdot\frac{1}{3})^2\cdot\frac{1}{2}+\ldots = \frac{1}{2}\sum_{k=0}^{\infty}(\frac{1}{6})^k = \frac{3}{5}$. Likewise it can be established that the probability of reaching BSCC $\{7\}$ is given by $\frac{2}{5}$. Within each given BSCC one can compute the steady-state distribution by solving a linear system of equations as described above. Altogether, the probability that the CTMC is in state i after an infinite time is equal to the probability of reaching the corresponding BSCC, multiplied by the steady-state probability of state i within that BSCC.

In the case of an irreducible chain, the picture is simpler. Therefore, in the sequel, we only consider finite, homogeneous, irreducible CTMCs, so we can assume the existence and uniqueness of the steady-state distribution. The steady-state probability vector $\vec{\pi} = (\pi_s)_{s\in S}$ is obtained by solving the linear system of equations

$$\vec{\pi} \cdot Q = 0$$

under the additional constraint that $\sum_{s\in S} \pi_s = 1$. This system may be solved by direct methods (such as Gaussian elimination, LU (or LDU) decomposition, inverse iteration, etc.) or by iterative methods[2] [314]. Iterative methods can be obtained by transforming the former equation into the following common fixed point equation with appropriate iteration matrix M:

$$\vec{\pi} = \vec{\pi} \cdot M$$

This equation is then used in the iteration scheme

$$\vec{\pi}^{(k+1)} = \vec{\pi}^{(k)} \cdot M$$

[2] An iterative method improves a given initial estimate in a step-by-step fashion until either convergence is achieved or some other termination criterion (such as exceeding of the maximum number of iterations) holds.

starting from an initial approximation $\vec{\pi}^{(0)}$. We now briefly recall the most basic iteration schemes.

Power method: In element-wise notation, the well-known power method can be written as follows:

$$\pi_s^{(k+1)} = \pi_s^{(k)} + \sum_{s' \in S} \pi_{s'}^{(k)} Q(s', s) \cdot \Delta t$$

which corresponds to the following matrix notation:

$$\vec{\pi}^{(k+1)} = \vec{\pi}^{(k)} \cdot (Q \cdot \Delta t + I)$$

where I is the identity matrix of appropriate size and the scaling factor Δt must be chosen such that $\Delta t < (\max_{s \in S}\{E(s)\})^{-1}$ in order to ensure that the iteration matrix $M_{power} = Q \cdot \Delta t + I$ is a stochastic matrix[3]. Note that the iteration matrix for the power method is identical (if the same Δt is chosen) with the stochastic matrix used in the uniformisation method for calculating transient solutions.

Jacobi method: The iteration scheme of Jacobi can be written in element-wise notation as follows:

$$\pi_s^{(k+1)} = \frac{-1}{Q(s,s)} \cdot \sum_{\substack{s' \in S \\ s' \neq s}} \pi_{s'}^{(k)} Q(s', s)$$

The method of Jacobi stems from a decomposition of the generator matrix in the form of $Q = R - D$, where D refers to a diagonal matrix whose entries are the row sums of the rate matrix R. The corresponding matrix formulation is

$$\vec{\pi}^{(k+1)} = \vec{\pi}^{(k)} \cdot R \cdot D^{-1}.$$

i.e. the iteration matrix M is defined as $M_{Jacobi} = R \cdot D^{-1}$.

Gauss-Seidel method: The method of Gauss-Seidel stems from a decomposition of the generator matrix in the form of $Q = D - L - U$, where L and U are the negative lower (upper) triangular portions of the rate matrix R, and D is the diagonal of the generator matrix Q (note that this is not the same D as in the method of Jacobi, but its negative). The matrix formulation of the scheme of Gauss-Seidel is

$$\begin{aligned} \vec{\pi}^{(k+1)} &= \vec{\pi}^{(k)} \cdot L \cdot (D - U)^{-1} \quad \text{(forward GS)} \\ \vec{\pi}^{(k+1)} &= \vec{\pi}^{(k)} \cdot U \cdot (D - L)^{-1} \quad \text{(backward GS)} \end{aligned}$$

[3]Note that in practice Δt should be chosen very close to $(\max_{s \in S}\{E(s)\})^{-1}$, for instance $(\max_{s \in S}\{E(s)\})^{-1} \cdot (1 - \epsilon)$ for a small value of ϵ, in order to achieve good convergence [314, p. 31 and p. 124]. In [323], the value $\epsilon = 0.01$ is recommended.

It is important to note that in practice the method of Gauss-Seidel is not usually performed by a vector-matrix multiplication but by a computation of the new probability vector $\vec{\pi}^{(k+1)}$ in an element-wise, sequential fashion, which in the case of forward GS amounts to

$$\pi_{s_i}^{(k+1)} = \frac{-1}{Q(s_i,s_i)} \cdot \left(\sum_{\substack{s_j \in S \\ j<i}} \pi_{s_j}^{(k+1)} Q(s_j,s_i) + \sum_{\substack{s_j \in S \\ j>i}} \pi_{s_j}^{(k)} Q(s_j,s_i) \right)$$

From this equation we observe that the iteration scheme of Gauss-Seidel is similar to the method of Jacobi, apart from the fact that for computing the new iterate $\pi_{s_i}^{(k+1)}$ the already updated values for $\pi_{s_j}^{(k+1)}$, $j < i$ are used *immediately*, instead of at the next iteration. If one wished to perform Gauss-Seidel using the straightforward matrix multiplication scheme, one would have to explicitly calculate the iteration matrix $M_{GS\_forward} = L \cdot (D-U)^{-1}$. This is usually not done in practice for the following reason: The inverse of the triangular matrix $D-U$ is also upper triangular. However, for the average Markov chain whose rate matrix is very sparse, the inversion step causes a lot of fill-in which destroys the efficiency of sparse matrix storage techniques.

SOR method: Another well-known iterative method for the solution of the linear system of equations $\vec{\pi} \cdot Q = 0$ is the method of successive over-relaxation (SOR), an extrapolation technique for accelerating the convergence of the method of Gauss-Seidel. Its element-wise formulation is as follows (note that the expression in parenthesis is similar to the Gauss-Seidel case):

$$\pi_{s_i}^{(k+1)} = (1-\omega) \cdot \pi_{s_i}^{(k)} + \omega \cdot \left(\frac{-1}{Q(s_i,s_i)} \cdot \left(\sum_{\substack{s_j \in S \\ j<i}} \pi_{s_j}^{(k+1)} Q(s_j,s_i) + \sum_{\substack{s_j \in S \\ j>i}} \pi_{s_j}^{(k)} Q(s_j,s_i) \right) \right)$$

The matrix formulation of the SOR scheme is

$$\begin{aligned} \vec{\pi}^{(k+1)} &= \vec{\pi}^{(k)} \cdot [(1-\omega) \cdot D + \omega \cdot L] \cdot [D - \omega \cdot U]^{-1} \quad \text{(forward SOR)} \\ \vec{\pi}^{(k+1)} &= \vec{\pi}^{(k)} \cdot [(1-\omega) \cdot D + \omega \cdot U] \cdot [D - \omega \cdot L]^{-1} \quad \text{(backward SOR)} \end{aligned}$$

The main problem of SOR consists of finding a good value for the relaxation parameter ω, which must be chosen such that $0 < \omega < 2$ in order for the iteration to converge. Since no general method is known for determining the optimal value of ω, most implementations use heuristic parameter estimation, adjusting the value of ω every few iterations, depending on the rate of convergence.

Projection methods: All iterative methods mentioned so far work with an iteration matrix which does not change from iteration to iteration. In other

words, the current approximation is always multiplied by the same, unmodified iteration matrix. Therefore these methods are also called stationary iterative methods[4].

There is, however, a whole class of iterative methods which do not work with an iteration matrix at all. These are the so-called projection methods, in particular the Krylov subspace methods, which approach the solution by a sequence of approximations taken from small dimension subspaces. Krylov methods often converge faster (i.e. within fewer iterations) than stationary methods, but they also consume more memory, because they require the storage of several vectors (of the size of the number of unknowns, i.e. the size of the state space). The most well understood Krylov methods (which enjoy optimality properties) are conjugate gradient and GMRES, but there are several other Krylov methods applicable to different scenarios [314, 23, 211]. It is beyond the scope of this thesis to elaborate on the mathematical theory of projection methods, but in Sec. 7.1.2 we will discuss some implementation considerations of the Bi-CGSTAB method.

2.2 Labelled transition systems (LTS)

Informally, a transition system consists of states and transitions between states. The transitions are labelled with symbols from a set L which may correspond, for example, to the set of actions *Act* of a process algebra[5]. A LTS can be graphically interpreted as a directed graph (with a distinguished initial node) whose edges are labelled with labels from L. Fig. 2.2 shows an example LTS.

Definition 2.2.1 Labelled Transition System (LTS)
Let S be a finite set of states. Let $s \in S$ be the initial state. Let L be a finite set of labels. Let $\dashrightarrow$ be a relation

$$\dashrightarrow \subseteq S \times L \times S$$

We call $\mathcal{T} = (S, L, \dashrightarrow, s)$ a Labelled Transition System. If $(x, l, y) \in \dashrightarrow$, we write $x \overset{l}{\dashrightarrow} y$. ■

[4]Kelley [211, p. 5] states: "Iterative methods of this form are called *stationary methods* because the transition from $\vec{\pi}^{(k)}$ to $\vec{\pi}^{(k+1)}$ does not depend on the history of the iteration".

[5]In the context of process algebras, it is often useful to have a special internal (unobservable) action denoted by the symbol τ, see Sec. 3.7.

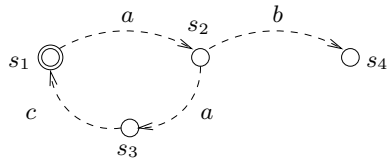


Figure 2.2: Example of a labelled transition system (LTS)

Note that in our definition the set of states S is assumed to be finite. Finiteness of the state space is a prerequisite for the symbolic encoding of states and transitions which is described below in Sec. 4.1.

2.3 Stochastic extensions of LTS

In a *stochastic* LTS each transition is associated with a stochastic delay, i.e. each transition is labelled both with an (action) label as in the LTS case, and in addition with a real number, referred to as the transition rate. Such stochastic LTSs (SLTS) appear during performance evaluation and performability analysis of distributed systems. For example, stochastic LTSs are generated during the analysis of Markovian stochastic process algebra (SPA) models, stochastic automata networks (SAN) or stochastic Petri nets (SPN). Abstracting from their functional information, SLTSs can be interpreted as Markov chains and analysed by numerical methods.

2.3.1 Stochastic LTSs

In case of *stochastic* transition systems, each transition has — in addition to the labelling with an element from the set of labels L — as a second attribute a positive real number, the *rate* of the transition, i.e. edges are labelled with tuples from $L \times \mathbb{R}$, as shown in Fig. 2.3.

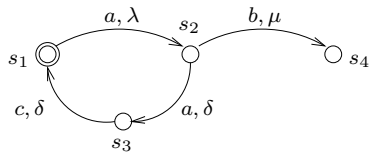


Figure 2.3: Example of a stochastic labelled transition system (SLTS)

Definition 2.3.1 Stochastic Labelled Transition System (SLTS)[6]
Let S, s and L be defined as for LTSs. Let $\longrightarrow$ be defined as follows:

$$\longrightarrow \subseteq S \times L \times \mathbb{R}^{>0} \times S$$

We call $\mathcal{T} = (S, L, \longrightarrow, s)$ a Stochastic Labelled Transition System. If $(x, a, \lambda, y) \in \longrightarrow$, we say that there is an a-transition from state x to state y with rate λ and write $x \xrightarrow{a,\lambda} y$. ■

For practical reasons and in view of the following symbolic representation, we merge multiple a-transitions between a given pair of states into a single transition. For instance, two separate transitions $x \xrightarrow{a,\lambda} y$ and $x \xrightarrow{a,\mu} y$ will be merged into $x \xrightarrow{a,\lambda+\mu} y$.

The real-valued rates determine the time T spent in a particular state x, which is a random value drawn from an exponential distribution, i.e. $Prob(T \leq t) = 1 - e^{-E(x)\cdot t}$ where $E(x)$ is the exit rate of state x, defined as $E(x) = \sum_{x \xrightarrow{a,\lambda} y} \lambda$, as in the case of CTMCs. The mean of this distribution is thus given by $1/E(x)$, the inverse of the sum of all rates of transitions leaving state x. For example, in Fig. 2.3, the mean time spent in state s_1 is $1/\lambda$, and the mean time spent in state s_2 is $1/(\mu + \delta)$.

The Continuous Time Markov Chain (CTMC) corresponding to an SLTS is obtained by abstracting from the action labels. The arcs of the CTMC are given by the union of all the transitions joining the LTS nodes (regardless of their labels), and the transition rate is the sum of the individual rates. This is justified by the properties of the exponential distribution, in particular the fact that the minimum of two exponentially distributed random variables with rates λ_1 and λ_2 is again exponentially distributed, namely with rate $\lambda_1 + \lambda_2$. Note that self-loops (transitions leading back to the same state) are allowed in an SLTS. When moving from an SLTS to a CTMC by abstracting from the action labels and summing up "parallel" rates, self-loops can be simply deleted, since they are only relevant in a compositional context, but irrelevant for the state probabilities of the CTMC (since they have no effect on the balance equations of the CTMC).

2.3.2 Extended Stochastic LTSs

In case of *extended* stochastic transition systems, there are two transition relations: One for Markovian transitions and one for immediate (action) transitions.

[6] The term "Action-labelled Markov Chain" (AMC) is sometimes used instead of "Stochastic Labelled Transition System", e.g. in [170].

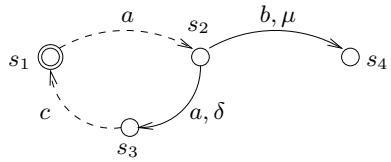


Figure 2.4: Example of an extended stochastic labelled transition system (ESLTS)

Figure 2.4 shows an example ESLTS, where Markovian transitions are drawn as solid arrows and immediate transitions as dashed arrows.

Definition 2.3.2 Extended Stochastic Labelled Transition System (ESLTS)
Let S, s and L be defined as for LTSs.
Let $\dashrightarrow$ be defined as follows:

$$\dashrightarrow \subseteq S \times L \times S$$

Let $\longrightarrow$ be defined as follows:

$$\longrightarrow \subseteq S \times L \times \mathbb{R}^{>0} \times S$$

We call $\mathcal{T} = (S, L, \dashrightarrow, \longrightarrow, s)$ an Extended Stochastic Labelled Transition System. If $(x, a, y) \in \dashrightarrow$, we say that there is an immediate a-transition from state x to state y and write $x \overset{a}{\dashrightarrow} y$. If $(x, a, \lambda, y) \in \longrightarrow$, we say that there is a Markovian a-transition from state x to state y with rate λ and write $x \overset{a,\lambda}{\longrightarrow} y$. ■

Immediate transitions are sometimes defined as Markovian transitions with the special rate value ∞ which, in the limit, yields a zero delay. However, this would imply that several immediate transitions, which are enabled at the same time, occur with the same probability. Since we do not wish to associate a particular probability with an immediate transition, but consider the choice between several immediate transitions as purely non-deterministic, we prefer to describe immediate transitions in a separate transition relation.

Immediate transitions lead to the existence of vanishing (instable) states. These are states which are left as soon as they are entered, i.e. their sojourn time is zero. Conversely, tangible (stable) states are states in which no immediate transitions are enabled. The sojourn time of a tangible state has an exponential distribution. In a compositional framework, e.g. in the context of stochastic process algebras, the notions of tangible and vanishing states may be refined in the following way: A state is called compositionally vanishing if it has at least one outgoing *internal* immediate transition, but no outgoing *visible* immediate transition. The idea is

that even an immediate transition may be delayed if it is visible, since it may be kept waiting by a synchronisation partner which is not yet ready to participate in the synchronisation. Since synchronisation on internal τ-transitions is not allowed, one can be sure that internal immediate transitions will not be delayed. For a precise definition of the notions of vanishing states and compositionally vanishing states, we refer to Def. 3.7.8 and Def. 5.2.1.

2.4 Binary Decision Diagrams (BDD)

BDDs are graph-based representations of Boolean functions which, during the recent years, have received a lot of attention. Their success is due to the fact that in many instances from different areas of application, such as hardware verification and model checking of concurrent systems, they enable a compact *symbolic* representation of Boolean functions. In particular, they are known to enable efficient encodings of very large state spaces and transition systems. Building on early work by Lee [237] and Akers [3, 4], Bryant has been the main advocate of the BDD data structure [45, 46, 47, 48].

The BDD data structure is very well suited for applications from the areas of model based verification (and — as we will see in this thesis — potentially performance analysis) for the following reason: When representing state-transition systems, the parallel composition of components can be realised directly on their BDD representations. This "symbolic parallel composition" has the potential to avoid the usually observed exponential blow-up [118]. This feature is actually one of the main strengths of the BDD approach. Parallel composition on BDDs will be discussed in detail in Chap. 5.

In the following sections, we introduce the BDD data structure and basic operations thereon. Sec. 4.1 explains how LTSs can be encoded as BDDs. Chap. 4 also discusses ways to include the rate information of (E)SLTS into this data structure. In Chap. 5 we will discuss compositional model construction on BDDs as well as BDD-based reachability analysis and BDD-based implementation of bisimulation algorithms. Chaps. 6 and 7 deal with further aspects of BDD-based modelling, namely the issues of compact symbolic representations and symbolic numerical analysis.

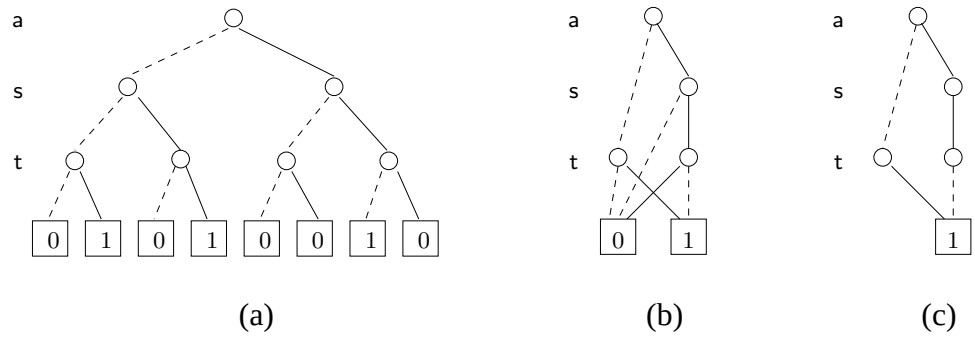


Figure 2.5: (a) Binary decision tree, (b) reduced BDD and (c) simplified graphical representation for the Boolean function $(\overline{\mathsf{a}} \wedge \mathsf{t}) \vee (\mathsf{a} \wedge \mathsf{s} \wedge \overline{\mathsf{t}})$

2.4.1 Definition

We denote by $\mathbb{B} = \{0, 1\}$ the set of Boolean values. A Binary Decision Diagram (BDD) [45, 12] is a symbolic representation of a Boolean function $f : \mathbb{B}^n \mapsto \mathbb{B}$. Its graphical interpretation is a rooted directed acyclic graph with one or two terminal vertices. Each non-terminal vertex x is associated with a Boolean variable $\mathsf{var}(x)$ and has two successor vertices, denoted by $\mathsf{then}(x)$ and $\mathsf{else}(x)$. The graph is ordered in the sense that on each path from the root to a terminal vertex, the variables are visited in the same order. A reduced BDD is essentially a collapsed binary decision tree in which isomorphic subtrees are merged and "don't care" vertices are skipped (a vertex is called "don't care" if the truth value of the corresponding variable is irrelevant for the truth value of the overall function).

As a simple example, Fig. 2.5 (a) shows the full binary decision tree for the function $(\overline{\mathsf{a}} \wedge \mathsf{t}) \vee (\mathsf{a} \wedge \mathsf{s} \wedge \overline{\mathsf{t}})$. We use the convention that all vertices drawn at one level are labelled by the same Boolean variable, as indicated at the left of the graph. The edge from vertex x to $\mathsf{then}(x)$ represents the case where $\mathsf{var}(x)$ is true; conversely, the edge from x to $\mathsf{else}(x)$ the case where $\mathsf{var}(x)$ is false. (In the graphical representation, then-edges are drawn solid, else-edges dashed. Furthermore, the direction of the edges is usually from top to bottom and therefore not shown in the graphical representation.) Part (b) of the figure shows the corresponding reduced BDD which can be obtained from the decision tree by merging isomorphic subgraphs and leaving out don't care vertices. For instance, in the diagrams shown in Fig. 2.5, if $\mathsf{a} = 0$ then s is a don't care variable. As shown in Fig. 2.5 (c), in the graphical representation of a BDD, for reasons of simplicity, the terminal vertex 0 and its adjacent edges are usually omitted. So, for a non-terminal vertex with only one outgoing edge drawn, the other outgoing edge leads to the terminal 0 vertex. In all three graphs shown in Fig. 2.5, the function value for a given

truth assignment can be determined by following the corresponding edges from the root until a terminal vertex is reached.

Next, we give a formal definition of BDD:

Definition 2.4.1 Ordered Binary Decision Diagram (BDD)
An Ordered Binary Decision Diagram (BDD for short) over $\langle Vars, \prec \rangle$ *is a rooted directed acyclic graph* $\mathsf{B} = (Vert, \mathsf{var}, \mathsf{then}, \mathsf{else})$ *defined by*

- *a finite set of vertices* $Vert = T \cup NT$*, where* T *(*NT*) is the set of terminal (non-terminal) vertices,* $|Vert| \geq 1$*,* $T \subseteq \mathbb{B}$*,*
- *a function* $\mathsf{var} : NT \mapsto Vars$*, where* $Vars = \{\mathsf{v}_1, \ldots, \mathsf{v}_n\}$ *is a set of Boolean variables with a fixed ordering relation* $\prec \subset Vars \times Vars$*,*
- *a function* $\mathsf{then} : NT \mapsto Vert$ *and a function* $\mathsf{else} : NT \mapsto Vert$*,*

with the following constraints:
$\forall x \in NT : \mathsf{then}(x) \in T \vee \mathsf{var}(\mathsf{then}(x)) \succ \mathsf{var}(x)$
$\forall x \in NT : \mathsf{else}(x) \in T \vee \mathsf{var}(\mathsf{else}(x)) \succ \mathsf{var}(x)$ ■

The function var defines the labelling of the non-terminal vertices with Boolean variables, and the functions then and else define the edges of the graph. The constraints ensure that the ordering relation among the Boolean variables is respected. For $x, y \in Vert$ we write $x \prec y$ either if $x, y \in NT$ and $\mathsf{var}(x) \prec \mathsf{var}(y)$, or if $x \in NT$ and $y \in T$. A BDD B over $\langle \{\mathsf{v}_1, \ldots, \mathsf{v}_n\}, \prec \rangle$, where $\mathsf{v}_1 \prec \ldots \prec \mathsf{v}_n$, we also call a BDD over $(\mathsf{v}_1, \ldots, \mathsf{v}_n)$.

A BDD as defined by Def. 2.4.1 is ordered, but not necessarily reduced. This motivates the need for the following definition:

Definition 2.4.2 Reducedness of a BDD
A BDD B *is called reduced iff*

1. $\forall x \in NT : \mathsf{else}(x) \neq \mathsf{then}(x)$
2. $\forall x, y \in NT : \quad \mathsf{var}(x) \neq \mathsf{var}(y)$
   $\vee\ \mathsf{else}(x) \neq \mathsf{else}(y)$
   $\vee\ \mathsf{then}(x) \neq \mathsf{then}(y)$ ■

The first condition states that redundant (don't care) vertices must be skipped (i.e. not explicitly present in the BDD), and the second condition states that

no pair of isomorphic vertices exists. Bryant [45] described an algorithm for transforming a BDD into a reduced BDD. The operations which we will describe in Sec. 2.4.2 make sure that the resulting BDD is always in reduced form, provided that the operand(s) were in reduced form. Note that unless otherwise stated, from here on we will simply use the term "BDD" instead of "reduced BDD", i.e. we will always assume that we work with BDDs which are reduced.

Each BDD vertex unambiguously defines a Boolean function. The definition is based on the so-called Shannon expansion which states that for an arbitrary Boolean function f that depends on k variables $\mathsf{v}_1, \ldots, \mathsf{v}_k$ we have

$$f(\mathsf{v}_1, \ldots, \mathsf{v}_k) \;=\; \mathsf{if}\ \mathsf{v}_1\ \mathsf{then}\ f(1, \mathsf{v}_2, \ldots, \mathsf{v}_k)\ \mathsf{else}\ f(0, \mathsf{v}_2, \ldots, \mathsf{v}_k),$$

or, in terms of Boolean operators $\wedge$ and $\vee$

$$f(\mathsf{v}_1, \ldots, \mathsf{v}_k) = (\mathsf{v}_1 \wedge f(1, \mathsf{v}_2, \ldots, \mathsf{v}_k)) \vee (\overline{\mathsf{v}_1} \wedge f(0, \mathsf{v}_2, \ldots, \mathsf{v}_k))$$

The terms $f(0, \mathsf{v}_2, \ldots, \mathsf{v}_k)$ and $f(1, \mathsf{v}_2, \ldots, \mathsf{v}_k)$ are called the cofactors of the Boolean function f with respect to the variable v_1. It is, of course, possible to expand f not only with respect to v_1, but with respect to any one of the Boolean variables $\mathsf{v}_1, \ldots, \mathsf{v}_k$.

Definition 2.4.3 Boolean function f_x represented by a BDD-vertex
The Boolean function f_x represented by a BDD-vertex $x \in Vert$ is recursively defined as follows:

- *if $x \in T$ then $f_x = x$, i.e. either 0 or 1,*
- *else (if $x \in NT$)*
  $f_x = \Big(\mathsf{var}(x) \wedge f_{\mathsf{then}(x)}\Big) \vee \Big(\overline{\mathsf{var}(x)} \wedge f_{\mathsf{else}(x)}\Big)$ ■

Most times one is interested in the case where x corresponds to the BDD root. In that case we will write f_{B} instead of f_x, where x is the root vertex of BDD B.

It is known that BDDs provide a canonical representation for Boolean functions, i.e. a given Boolean function has a unique BDD representation (assuming a fixed ordering of the Boolean variables) [45]. For this reason, some computationally hard problems (e.g. satisfiability of Boolean functions, test-for-tautology, equivalence of two Boolean functions, ...) can be solved in constant or linear time, once the BDD representations of the Boolean functions involved are known [12].

It should be noted that, given a Boolean function, the size of the resulting BDD is highly dependent on the chosen variable ordering. This issue will be discussed further in Chaps. 4, 5 and 6.

2.4.2 Operations on BDDs

We now introduce the basic operations on BDDs and provide some informal explanation of how they can be implemented. In general, algorithms for BDD construction from a Boolean expression and algorithms for BDD manipulation, such as the APPLY algorithm (see below) for performing Boolean operations on BDD arguments, all follow a recursive descent scheme according to the above Shannon expansion.

Negation: Let B be a BDD over $(\mathsf{v}_1, \ldots, \mathsf{v}_n)$, representing the Boolean function f_{B}. The BDD representing the negation of this function, denoted $\overline{\mathsf{B}}$, is obtained from B by simply swapping the terminal vertices.

The general Apply algorithm: Let B_1 and B_2 be two BDDs over the set of Boolean variables $\{\mathsf{v}_1, \ldots, \mathsf{v}_n\}$ with ordering relation $\prec$. If OP is a binary Boolean operator (e.g. conjunction $\wedge$, disjunction $\vee$, implication $\rightarrow$, ...) then $\textsc{Apply}(\mathsf{B}_1, \mathsf{B}_2, \mathrm{OP})$ returns the BDD B over $(\mathsf{v}_1, \ldots, \mathsf{v}_n)$ where $f_{\mathsf{B}} = f_{\mathsf{B}_1}$ OP f_{B_2}. The basic idea of the algorithm is as follows: $\textsc{Apply}(\mathsf{B}_1, \mathsf{B}_2, \mathrm{OP})$ calls a recursive procedure $A_{\mathrm{OP}}(r_1, r_2)$, where r_1 (r_2) is the root vertex of B_1 (B_2). In general, procedure $A_{\mathrm{OP}}(x_1, x_2)$ takes a vertex x_1 of B_1 and a vertex x_2 of B_2 as its input and returns a vertex x that is obtained according to the following rules:

- If both x_1 and x_2 are terminal vertices then $A_{\mathrm{OP}}(x_1, x_2)$ returns the terminal vertex $x = x_1$ OP x_2.
- If x_1, x_2 are non-terminal vertices and $\mathsf{var}(x_1) = \mathsf{var}(x_2) = \mathsf{v}$ then $\mathsf{var}(x) = \mathsf{v}$, $\mathsf{else}(x) = A_{\mathrm{OP}}(\mathsf{else}(x_1), \mathsf{else}(x_2))$ and $\mathsf{then}(x) = A_{\mathrm{OP}}(\mathsf{then}(x_1), \mathsf{then}(x_2))$.
- If $x_1 \prec x_2$ then $\mathsf{var}(x) = \mathsf{var}(x_1)$, $\mathsf{else}(x) = A_{\mathrm{OP}}(\mathsf{else}(x_1), x_2)$ and $\mathsf{then}(x) = A_{\mathrm{OP}}(\mathsf{then}(x_1), x_2)$.
  Conversely, if $x_2 \prec x_1$ then $\mathsf{var}(x) = \mathsf{var}(x_2)$, $\mathsf{else}(x) = A_{\mathrm{OP}}(x_1, \mathsf{else}(x_2))$ and $\mathsf{then}(x) = A_{\mathrm{OP}}(x_1, \mathsf{then}(x_2))$.

Furthermore, in order to achieve efficient implementation, procedure A_{OP} may check for the presence of special "controlling" values [45] (which depend on the instantiation of the operator OP) which can receive special treatment and thereby avoid the initiation of recursive calls. For instance, if OP is disjunction and x_1 is the terminal vertex $x_1 = 1$, then x_2 can be immediately returned as the result.

As a small example to illustrate the principle of APPLY, Fig. 2.6 shows three BDDs B_1, B_2 and B over Boolean variables $(\mathsf{a}, \mathsf{s}, \mathsf{t})$. B_1 represents the function $(\overline{\mathsf{a}} \wedge \mathsf{t}) \vee (\mathsf{a} \wedge \mathsf{s} \wedge \overline{\mathsf{t}})$, B_2 represents $(\overline{\mathsf{a}} \wedge \mathsf{s} \wedge \mathsf{t}) \vee (\mathsf{a} \wedge \overline{\mathsf{s}})$ and B represents their conjunction which is $\overline{\mathsf{a}} \wedge \mathsf{s} \wedge \mathsf{t}$. Note that some vertices appear in more than one of the three

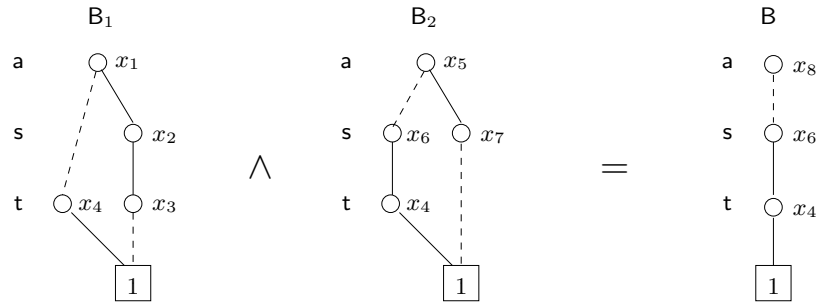


Figure 2.6: Example for APPLY where OP is instantiated by conjunction

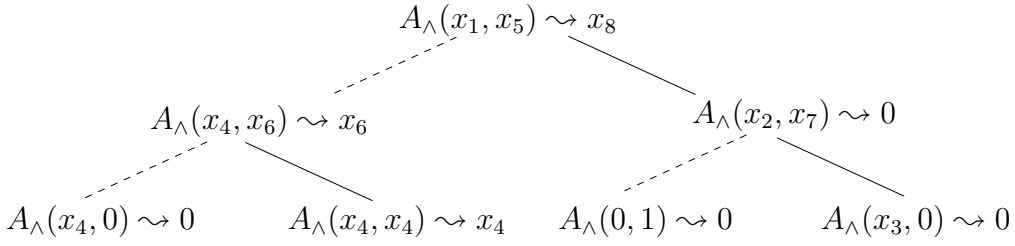


Figure 2.7: Call tree for the APPLY example from Fig. 2.6

BDDs. For instance, vertices 0 (not drawn), 1 and x_4 appear in both B_1 and B_2 since they represent the same Boolean functions in both BDDs and are therefore stored only once in memory (for this issue cf. the discussion on the unique table below). Fig. 2.7 shows the call tree associated with $\text{APPLY}(\mathsf{B}_1, \mathsf{B}_2, \wedge)$. At the top level, procedure $A_\wedge$ (an instance of A_{OP}) is called with the arguments x_1 and x_5, the root vertices of the operand BDDs. Since x_1 and x_5 are both non-terminal and $\mathsf{var}(x_1) = \mathsf{var}(x_5) = \mathsf{a}$, this causes the two recursive calls $A_\wedge(\mathsf{else}(x_1), \mathsf{else}(x_5)) = A_\wedge(x_4, x_6)$ and $A_\wedge(\mathsf{then}(x_1), \mathsf{then}(x_5)) = A_\wedge(x_2, x_7)$ which in turn cause further recursive calls. In Fig. 2.7, solid (dashed) edges denote recursive calls which will determine the then (else) successor of a node, and the notation $A_\wedge(x, y) \rightsquigarrow z$ denotes the fact that as a result of the call $A_\wedge(x, y)$ a vertex z is eventually returned. For instance, the top-level call to $A_\wedge$ eventually returns vertex x_8 with $\mathsf{var}(x_8) = \mathsf{a}$. In the call tree, one can also observe how the recursion terminates either because both operands are terminal (e.g. $A_\wedge(0, 1) \rightsquigarrow 0$) or because controlling values take effect (e.g. $A_\wedge(x_4, 0) \rightsquigarrow 0$ or $A_\wedge(x_4, x_4) \rightsquigarrow x_4$).

We now discuss some important implementation considerations [41]. In order to make sure that the BDD returned by APPLY(.) is in reduced form, the algorithm uses a "unique table" which contains all currently existing BDD vertices. A unique table entry for a non-terminal vertex x consists of the vertex identifier,

the vertex's variable labelling $\mathsf{var}(x)$ and the two references to the children vertices $\mathsf{then}(x)$ and $\mathsf{else}(x)$. (In addition, a reference counter is usually maintained for each non-terminal vertex in order to keep track of the number of references to that vertex.) A unique table entry for a terminal vertex consists of the vertex identifier and the vertex's value. As a result of procedure $A_{\mathrm{OP}}(x_1, x_2)$, a new vertex is inserted into the unique table, if an isomorphic vertex was not yet in the table. Otherwise a reference to the already existing vertex is returned.

In order to minimise the number of recursive calls to procedure A_{OP}, a second table is maintained by most BDD packages [41]. This is called the "computed table" which contains entries of the form (x, y, OP, z), where z is the identifier of the vertex which had been previously obtained when computing $A_{\mathrm{OP}}(x, y)$. Whenever A_{OP} is called, the algorithm checks whether there exists a matching entry in the computed table, and if this is the case simply returns the vertex found there. Proper use of the computed table ensures that the same computation will not be repeated during recursive descent of the decision graph.

For efficiency reasons, both the unique table and the computed table are usually implemented with the help of hashing functions, and the computed table is realised as a finite size cache [41].

The Restrict operation: Let B be a BDD depending on Boolean variables $\mathsf{v}_1, \ldots \mathsf{v}_n$. Let the Boolean vector $(b_1, \ldots, b_k) \in \mathbb{B}^k$, where $k \leq n$, be a fixed assignment for a subset of the Boolean variables $\mathsf{v}_{i_1}, \ldots, \mathsf{v}_{i_k} \in \{\mathsf{v}_1, \ldots \mathsf{v}_n\}$. The Boolean function represented by vertex $x \in Vert$ under this assignment is denoted by $f_x\Big|_{(\mathsf{v}_{i_1}=b_1,\ldots,\mathsf{v}_{i_k}=b_k)}$. For the constant Boolean value $b \in \mathbb{B}$ we define

$$\textsc{Restrict}(\mathsf{B}, \mathsf{v}_i, b) := \mathsf{B}\Big|_{\mathsf{v}_i=b}$$

which is called the restriction of B to the case $\mathsf{v}_i = b$. Note that $\textsc{Restrict}(\mathsf{B}, \mathsf{v}_i, b)$ is a BDD which depends only on Boolean variables $\mathsf{v}_1, \ldots, \mathsf{v}_{i-1}, \mathsf{v}_{i+1}, \ldots, \mathsf{v}_n$. If B represents Boolean function $f_{\mathsf{B}}(\mathsf{v}_1, \ldots, \mathsf{v}_n)$, then $\textsc{Restrict}(\mathsf{B}, \mathsf{v}_i, b)$ represents the Boolean function $f_{\mathsf{B}}(\mathsf{v}_1, \ldots, \mathsf{v}_n)\Big|_{\mathsf{v}_i=b}$, i.e. the cofactor of f_{B} with respect to Boolean variable v_i.

Restricting with respect to more than one variable at a time is defined recursively as

$$\begin{aligned} & \textsc{Restrict}(\mathsf{B}, (\mathsf{v}_{i_1}, \ldots, \mathsf{v}_{i_n}, \mathsf{v}_{i_{n+1}}), (b_1, \ldots, b_n, b_{n+1})) \\ := \; & \textsc{Restrict}(\mathsf{B}, (\mathsf{v}_{i_1}, \ldots, \mathsf{v}_{i_n}), (b_1, \ldots, b_n))\Big|_{\mathsf{v}_{i_{n+1}}=b_{n+1}} \end{aligned}$$

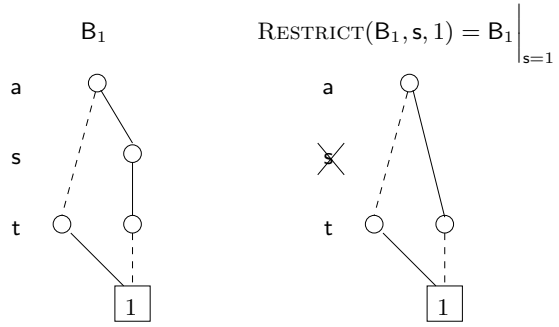


Figure 2.8: Example for RESTRICT

The BDD $\textsc{Restrict}(\mathsf{B}, \mathsf{v}_i, b)$ can be obtained from B by replacing any edge from a vertex x to a v_i-labelled vertex y by an edge from x to $\mathsf{then}(y)$ if $b = 1$ ($\mathsf{else}(y)$ if $b = 0$), and afterwards removing all v_i-labelled vertices.

In Fig. 2.8, a small example for the application of the RESTRICT operation is shown. BDD B_1, shown on the left, represents the function $(\overline{\mathsf{a}} \wedge \mathsf{t}) \vee (\mathsf{a} \wedge \mathsf{s} \wedge \overline{\mathsf{t}})$. The BDD on the right is obtained by restricting B_1 to the case $\mathsf{s} = 1$ and represents the function $(\overline{\mathsf{a}} \wedge \mathsf{t}) \vee (\mathsf{a} \wedge \overline{\mathsf{t}})$.

The general Abstract operation: Let B be a BDD depending on Boolean variables $\mathsf{v}_1, \ldots \mathsf{v}_n$. For an associative binary Boolean operator OP we define

$$\textsc{Abstract}(\mathsf{B}, \mathsf{v}_i, \textsc{op}) := \mathsf{B}\Big|_{\mathsf{v}_i=0} \textsc{op}\ \mathsf{B}\Big|_{\mathsf{v}_i=1}$$

which is called the abstraction of B with respect to Boolean variable v_i and operator OP.

As an example, consider $\textsc{Abstract}(\mathsf{B}, \mathsf{v}_i, \vee) = \mathsf{B}\Big|_{\mathsf{v}_i=0} \vee\ \mathsf{B}\Big|_{\mathsf{v}_i=1}$ which corresponds to the existential quantification on Boolean functions, since

$$\begin{aligned} \exists \mathsf{v}_i.\ f(\mathsf{v}_1, \ldots, \mathsf{v}_i, \ldots, \mathsf{v}_n) = \quad & f(\mathsf{v}_1, \ldots, \mathsf{v}_{i-1}, 0, \mathsf{v}_{i+1}, \ldots, \mathsf{v}_n) \\ \vee \quad & f(\mathsf{v}_1, \ldots, \mathsf{v}_{i-1}, 1, \mathsf{v}_{i+1}, \ldots, \mathsf{v}_n) \end{aligned}$$

Abstracting with respect to more than one variable is defined as

$$\begin{aligned} & \textsc{Abstract}(\mathsf{B}, (\mathsf{v}_{i_1}, \ldots, \mathsf{v}_{i_n}), \textsc{op}) \\ := \quad & \mathsf{B}\Big|_{(\mathsf{v}_{i_1}, \ldots, \mathsf{v}_{i_n})=(0,\ldots,0)} \textsc{op}\ \ldots\ \textsc{op}\ \mathsf{B}\Big|_{(\mathsf{v}_{i_1}, \ldots, \mathsf{v}_{i_n})=(1,\ldots,1)} \end{aligned}$$

i.e. all possible restrictions of B with respect to the Boolean variables $\mathsf{v}_{i_1}, \ldots, \mathsf{v}_{i_n}$ are combined by operator OP. It now becomes obvious that associativity of the

operator OP is required in order to ensure that the ordering of the variables and the order in which the cofactors are chosen do not influence the outcome [18].

Variable renaming: Let B be a reduced BDD over $(\mathsf{v}_1, \ldots, \mathsf{v}_n)$. Let $\mathsf{w} \notin \{\mathsf{v}_1, \ldots, \mathsf{v}_n\}$ and $i \in \{1, \ldots, n\}$ with $\mathsf{v}_{i-1} \prec \mathsf{w} \prec \mathsf{v}_{i+1}$. Then, $\mathsf{B}\{\mathsf{v}_i \leftarrow \mathsf{w}\}$ denotes the BDD over $(\mathsf{v}_1, \ldots, \mathsf{v}_{i-1}, \mathsf{w}, \mathsf{v}_{i+1}, \ldots, \mathsf{v}_n)$ that results from B if one changes the variable labelling of any v_i-labelled vertex into w. To implement this, one sets $\mathsf{var}(x) = \mathsf{w}$ for any v_i-labelled vertex x in B.

If $1 \leq i_1 < \ldots < i_m \leq n$ and $\mathsf{w}_1, \ldots, \mathsf{w}_m \notin \{\mathsf{v}_1, \ldots, \mathsf{v}_n\}$, then we write $\mathsf{B}\{\mathsf{v}_{i_1} \leftarrow \mathsf{w}_1, \ldots, \mathsf{v}_{i_m} \leftarrow \mathsf{w}_m\}$ as a shorthand for $\mathsf{B}\{\mathsf{v}_{i_1} \leftarrow \mathsf{w}_1\} \ldots \{\mathsf{v}_{i_m} \leftarrow \mathsf{w}_m\}$. For $\{\mathsf{w}_1, \ldots, \mathsf{w}_n\}$, where $\mathsf{w}_1 \prec \ldots \prec \mathsf{w}_n$, we write $\mathsf{B}\{\vec{\mathsf{v}} \leftarrow \vec{\mathsf{w}}\}$ to denote the BDD where each v variable has been renamed into the corresponding w variable.

Variable reordering: There exist algorithms for the dynamic reordering of the variables in a BDD, see e.g. [35, 36]. In general, variable reordering can lead to a decrease of the size of a BDD. However, it is known that finding the optimal ordering is an NP-complete problem [36], and therefore one has to resort to heuristics. We do not discuss variable reordering algorithms in detail, since we will not make use of them in our applications.

Chapter 3

Formalisms for "high-level" model specification

Stochastic models, in many cases Markov models, have a long history in application fields such as economics, physics, biology and medicine. For many years they have also been successfully used for the purpose of performance and dependability analysis of computer and communication systems. When constructing a performance model, the human modeller usually does not describe the behaviour of the model directly at the level of individual states and state-to-state transitions, since this would be much too tedious and error-prone. Instead, he or she employs some sort of "high-level" model specification formalism which assists him in precisely describing the behaviour he has in mind. Such a model specification formalism may provide graphical, textual or algebraic features, and typical examples are queueing networks, stochastic Petri nets, stochastic process algebras or specialised modelling languages.

In this chapter, we give a brief overview of high-level formalisms for describing stochastic performance models. We first briefly survey the main features of stochastic graph models, queueing networks, stochastic Petri nets and specialised modelling languages. We emphasise the importance of structuring concepts since these are a prerequisite for handling complex models. This argument leads our discussion to stochastic automata networks, in which context we review the Kronecker approach, and from there to the concept stochastic process algebras. In our opinion, stochastic process algebras quite ideally support the specification and analysis of complex performance models, and for that reason the section on stochastic process algebras (Sec. 3.7) takes up most of this chapter.

3.1 Stochastic graph models

In the area of project planning, where the aim is to accomplish certain sequences of activities with limited resources under critical time constraints, network-based planning methods of various form have been developed [115]. Classical project networks are represented by acyclic weighted graphs, as realised in the so-called program evaluation and review technique (PERT) and the critical path method (CPM). A more general class of projects, containing several different node types and also cycles, can be described and analysed with the graphical evaluation and review technique (GERT) [260][1]. Job shop scheduling problems [287] are among the typical application areas of such networks.

In the sequel, we focus on stochastic graph models (SGM) (also called stochastic task graphs) which are less general but amenable to efficient analysis. A SGM is a directed acyclic graph where every node is equipped with a stochastic distribution representing its execution time. Nodes represent tasks and the directed edges represent ordering relations between tasks. The standard interpretation of a SGM is that a task is causally dependent of its predecessor tasks, i.e. it starts execution when all its predecessor tasks are finished. Other interpretations, for instance that a task may start execution once at least one predecessor is finished, are also possible. SGMs are very well suited for modelling the execution of parallel programs [309, 151, 320, 241, 240] or business processes (workflows) [321]. The primary aim of analysis is to determine the mean or distribution of the overall execution time of the graph model.

For series-parallel graph models, efficient analysis, based on the series-parallel reduction of the graph to a single node is possible. For instance, the tool PEPP implements, among other analysis algorithms, a numerical series-parallel reduction algorithm [154, 152], which is based on discretised distributions and therefore works on SGMs with generally distributed task execution times.

Generally structured graph models may be analysed by exact or approximate state space analysis [308], methods which suffer from the state space explosion problem, especially if the graph model has a high degree of parallelism or if the node distributions are of phase type with many phases. From the point of view of Markovian state space analysis, only transient analysis (as opposed to steady-state analysis) is of interest with SGMs, since they have one absorbing state, namely the state where all tasks have finished execution. Therefore, determining the mean execution time of a SGM is a special case of determining the mean time to absorption (MTTA) for a general CTMC (which could have more than one absorbing state).

[1] GERT networks are also called stochastic project networks.

Bounds for the mean execution time of a generally structured SGM can be obtained by modifying the graph structure, such that it becomes series-parallel [153], and afterwards performing series-parallel reduction of the modified graph. In many instances, the bounding methods yield good approximations to the exact result at very low computational cost. So, to summarise, one can say that the analysis of SGMs is well understood and in many cases very efficient, but that SGMs are suitable only for a limited range of applications and therefore not sufficient as a universal modelling formalism.

3.2 Queueing networks

The development of queueing networks (QN) started already in the 1950ies, initiated by problems from the area of operations research [202, 203, 204]. It continued in the 1960ies with the modelling of closed networks [140], polling systems (e.g. the machine interference model [95]) and time-sharing systems (e.g. the well-known central server model [66]) and has since been a very active field of research with a plethora of applications. A QN describes customers moving between stations where they receive service after possibly waiting for a service unit to become available. The aim of analysis is typically the mean or distribution of the number of customers at a station, the customer throughput at a station, or the waiting time. The success of queueing networks stems mainly from the fact that for the class of product form networks [24] (which was already mentioned in Sec. 1.5.2) very efficient analysis algorithms, such as Buzen's algorithm [67] or mean-value analysis [277], are known, and that software tools for the specification and analysis of QN models were available at an early stage [285, 319]. Although QN have been extended in various directions, e.g. in order to model the forking and synchronisation of jobs (fork-join QNs, [188, 189, 16, 15, 215, 259]), the formalism of QNs 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.

3.3 Stochastic Petri nets

Stochastic Petri nets (SPN) were developed in the 1980ies for modelling complex dependences and synchronisation schemes which cannot easily be expressed by queueing models [258]. The modelling primitives of Petri nets (places, transitions, markings) are very basic and do not carry any application-specific semantics. For that reason, Petri nets are universally applicable and very flexible, which is shown

by the fact that they have been successfully applied to many different areas of application. In the class of Generalised SPN (GSPN) [1, 2], transitions are either timed or immediate. Timed transitions are associated with an exponentially distributed firing time, while immediate transitions fire as soon as they are enabled. During the analysis of a GSPN, the reachability graph is generated and vanishing (instable) markings, which are due to the firing of immediate transitions, are eliminated. The result is a Markov chain (CTMC) which can be analysed by numerical algorithms, yielding (steady-state or transient) state probabilities, i.e. the probabilities of the individual net markings[2].

In the basic formalism, a (G)SPN model is monolithic, i.e. it consists of a single net which models the whole system to be studied. Therefore, SPN models of complex systems tend to become very large and confusing. Moreover, such complex SPNs suffer from the state space explosion problem which can make state space generation and analysis prohibitively expensive. In Sec. 1.7.4 we had already mentioned the formalism of stochastic well-formed coloured Petri nets [76, 127] which enables a reduction of the state space, based on marking symmetries. In the 1990ies, some work has been published which is concerned with building SPNs in a structured way, basically by synchronising subnets via common transitions [51, 53, 52, 61, 107, 108], and exploiting the structure during analysis. These approaches are all related to the Kronecker approach described below in Sec. 3.6.1. A different approach to the structuring of SPNs, namely through the sharing of places between subnets, lead to Stochastic Activity Networks [283, 94]. In summary, one can safely say that such structuring techniques for stochastic Petri nets are mandatory for successfully combating the state space explosion problem.

3.4 Tool-specific model specification languages

Some software tools for performance modelling implement their own specialised model description languages. We mention the tools USENUM [288], MARCA [313], MOSEL [32, 33] and DNAmaca [223] which support the specification and analysis of Markov models. Such languages provide constructs for specifying states (usually described as tuples of discrete state variables) and state-to-state transitions (usually described by enabling conditions, transition rate functions and state variable changing functions). In addition to the actual model specification, these languages also contain constructs for specifying the measures to be calculated, the kind of experiment to be carried out, the solution method to be used and other control parameters.

[2] Another line of research is concerned with non-Markovian Petri nets, i.e. SPNs where some transitions may have generally distributed firing times [136, 239, 135].

It is, of course, difficult for tool-specific languages to achieve wide-spread propagation and acceptance. A further problem of the languages mentioned above is the fact that the models thus specified are monolithic, i.e. consisting of a single component, and state space explosion is a serious problem. As a first step towards compositional model specification, in [233] the DNAmaca input language and state space generation component were extended, such that it is possible to specify and analyse structured models consisting of several interacting components.

3.5 The need for structured models

In their standard form, queueing models, stochastic Petri nets and the tool-specific modelling languages mentioned above do not offer the possibility of composing an overall model from components which can be specified in isolation. Some proposals for modular extensions of these formalisms have been made, but no general concept exists. Modular composition of submodels, however, is a highly desirable feature when modelling complex systems, since it enables human users to focus on manageable parts from which a whole system can be constructed. As a specific example, suppose one wished to model a communication system where a sender communicates with a receiver over some communication medium. The model should reflect this structure, i.e. it should consist of three interacting submodels, one for the sender, one for the receiver and one for the medium, and the user should be able to specify these three submodels more or less independently of each other and then simply specify the way in which the submodels interact. As another, more general example, we mention the very fundamental concept of separating aspects related to the machine from aspects related to the load, a concept that had been developed already in [218, 184, 186]. Following this approach, a so-called system model is constructed from two interacting submodels, namely a machine model and a load model[3].

As mentioned already in Chap. 1, and as we shall see in the sequel, modularity and structuredness do not only help to make the construction of complex models easier and more convenient, but can also be exploited for efficient representation, compositional state-space reduction and efficient analysis. Prominent examples for such structure-based efficient modelling techniques are the Kronecker approach (discussed briefly in Sec. 3.6.1) and the symbolic approach (which is the main focus of this thesis).

[3]Similar ideas are applied in stochastic rendezvous networks [324] and layered queueing networks [281].

Stochastic automata networks (SAN) and stochastic process algebras (SPA) are formalisms which provide operators for constructing large models from small components, i.e. they explicitly support the construction of structured models. We shall discuss these formalisms in the following sections. We also point out the structured modelling framework described in [293], which allows the user to construct models consisting of interacting submodels, and where symmetries in the model structure are exploited in order to construct a reduced state space.

3.6 Stochastic automata networks

Stochastic Automata Networks (SAN) [4] were developed in the 1980ies and 1990ies [271, 273, 272, 312]. A SAN consists of several stochastic automata, basically CTMCs whose transitions are labelled with event names, which run in parallel and may perform certain synchronising events together. Thus, the SAN formalism is truly structured, since it allows the user to specify an overall model as a collection of interacting submodels. The major achievement of SANs was the formulation of the infinitesimal generator matrix of the Markov chain underlying the overall model as a so called tensor descriptor, as described below in Sec. 3.6.1. The potentially very large generator matrix of the overall model therefore never needs to be explicitly generated and stored. The SAN modelling approach is supported by the tool PEPS [274] and by the toolbox described in [62].

3.6.1 The Kronecker approach

The Kronecker approach was originally developed for the SAN framework [271], which is a rather low-level modelling formalism since its submodels are basically labelled CTMCs. However, the Kronecker approach has since been adapted to queueing networks [54], stochastic Petri nets [51, 107, 61], stochastic process algebras [56] and the structured modelling framework of [293].

The Kronecker approach realises an "implicit", space-efficient representation of the transition matrix of a structured Markov model, which also carries over to the generator matrix and to the iteration matrices for some of the common stationary iterative methods. Suppose we have two *independent* CTMCs $\mathcal{C}_1$ and $\mathcal{C}_2$ which

[4] The acronym "SAN" is also used for Stochastic Activity Networks (cf. Sec. 1.7.4 and Sec. 3.3), an extension of GSPNs, which has been developed by Sanders et al. [283, 94] and also offers structuring concepts and symmetry exploitation. In this thesis, unless otherwise stated, we will use "SAN" for Stochastic Automata Network.

are given by their transition rate matrices R_1 and R_2 (of size d_1 and d_2). Let us consider the combined stochastic process $\mathcal{C}$ whose state space is the Cartesian product of the state spaces of $\mathcal{C}_1$ and $\mathcal{C}_2$. Process $\mathcal{C}$ possesses the transition rate matrix R which is given by the Kronecker sum of R_1 and R_2:

$$R = R_1 \oplus R_2 \;\; (= R_1 \otimes I_{d_2} + I_{d_1} \otimes R_2)$$

where $\otimes$ denotes Kronecker product, $\oplus$ denotes Kronecker sum and I_d denotes an identity matrix of size d [101]. If, however, $\mathcal{C}_1$ and $\mathcal{C}_2$ are *not* independent, but perform certain transitions synchronously, the expression for the overall transition rate matrix changes to

$$R = R_{1,i} \oplus R_{2,i} + \sum_{a \in Sync} \lambda_a \cdot R_{1,a} \otimes R_{2,a}$$

where $R_{1,i}$ and $R_{2,i}$ contain those transitions which $\mathcal{C}_1$ and $\mathcal{C}_2$ perform independently of each other, and $R_{1,a}$ and $R_{2,a}$ contain those transitions which are caused by an event a from the set of synchronising events $Sync$. Here it is assumed that the resulting rate of the synchronising event a is given by λ_a, i.e. it is a predetermined rate, and matrices $R_{1,a}$ and $R_{2,a}$ are indicator matrices which contain only zeroes and ones. (It is also possible that $R_{1,a}$ and $R_{2,a}$ contain rates, in which case in the above subexpression $\lambda_a \cdot R_{1,a} \otimes R_{2,a}$ has to be replaced by $R_{1,a} \otimes R_{2,a}$. This would mean that the resulting rate of a synchronising event is equal to the product of the rates of the participating processes.) For the general case, where the overall model consists of K submodels, the expression for the overall transition rate matrix is given by

$$R = \bigoplus_{k=1}^{K} R_{k,i} + \sum_{a \in Sync} \lambda_a \cdot \bigotimes_{k=1}^{K} R_{k,a}$$

The strength of the Kronecker approach lies in its memory-efficiency and in the fact that for performing numerical analysis, the overall transition matrix never needs to be constructed explicitly. Rather, iterative numerical schemes which rely on matrix-vector multiplication as their basic operation, can be performed directly on the tensor descriptor of the iteration matrix (which can be derived from the tensor descriptor of the transition rate matrix). Plateau [271] used the power method, and Buchholz [50] describes Kronecker-based power, Jacobi, modified Gauss-Seidel, JOR (extrapolated Jacobi method) and modified SOR methods. Efficient algorithms for the multiplication of a vector with a Kronecker descriptor are analysed in depth in [59, 60], where, however, the authors state that "...all Kronecker-based algorithms are less computationally efficient than a conventional multiplication where [the matrix] R is stored in sparse format ..." and "This suggests that, in practice, the real advantage of Kronecker-based methods lies exclusively in their large memory savings". Efficient numerical solution based on Kronecker representation is also the focus of [312] and [120].

When working with the Kronecker approach, the set of states reachable from the initial state may be a strict subset of the Cartesian product of the involved submodel state spaces. This is known as the "potential versus actual state space" problem. If the actual state space is not known before numerical analysis starts, a probability vector of the size of the potential state space must be allocated, which can waste a considerable amount of memory space and even make the whole analysis impracticable. For that reason, Kronecker-based reachability analysis has been developed [213, 81] which may be performed as a preprocessing step before numerical analysis, at the cost of extra execution time.

3.7 Stochastic process algebras

Stochastic process algebras (SPA) [141, 192, 28] were developed in the 1990ies as a formal approach to performance evaluation. They are based on classical process algebras such as CSP [196], CCS [252] and LOTOS [37] and make it possible to carry out the modelling of distributed systems in a compositional fashion.

According to [26], a process algebra is a formal description technique for complex computer systems, especially those with communicating, concurrently executing components. A process algebra can be seen as a formal language for specifying the behaviour of processes in a structured way. This is achieved by defining the possible sequences of actions which a process may perform, and by specifying the interaction between processes. In addition to the formal language, a calculus that allows one to establish the equivalence between processes is an integral part of a process algebra.

In a *stochastic* process algebra, actions are associated with quantifiable stochastic delays, such that it is possible to model the passage of time as needed, for example, for the purpose of performance evaluation. The most popular stochastic distribution used for SPAs is the exponential distribution, and therefore the mathematical analysis of the resulting *Markovian* SPAs does not pose any specific problems. In some instances, timeless (immediate) actions are added [181, 278, 279], which raises some semantical questions but does not pose problems with respect to numerical analysis. In addition to the Markovian case, SPAs with general distributions have been developed, see e.g. [187, 98, 99]. One interesting branch of current SPA research combines performance analysis with model checking [161, 170, 172], a topic which we will discuss in more detail in Chap. 9.

3.7.1 Syntax and semantics

The basis of a process algebra is a formal language, which allows one to specify the behaviour of processes and their interaction. The underlying semantic model is usually a labelled transition system, whose transitions are labelled by actions, and which is generated with the help of a structural operational semantics (SOS). In general, process algebras offer the following features:

1. Complex models are built by composing small-size components in a hierarchical fashion.

2. An abstraction operator can be used in order to make a component's internal behaviour invisible to the environment. This feature may also be exploited for the reduction of the state space through "weak" bisimulation relations.

3. A calculus, i.e. a set of axiomatic rules, establishes the equivalence (w.r.t. a given bisimulation equivalence) between two specifications. The equivalence is the basis for state space reduction (also called state space aggregation).

These three features, which are also present in *stochastic* process algebras, are often summarised by the term "constructivity".

The LTS generated from a stochastic process algebra specification carries additional labels concerning the stochastic timing behaviour of the process. In the case of Markovian SPAs, those labels are in the form of transition rates. From such an SLTS the underlying CTMC can be derived by abstracting from all action names, and analysed by standard numerical methods.

On the level of the underlying labelled transition system, the equivalence between two specifications can be established through the concept of *bisimulation* relations [267]. Examples of such relations are Milner's strong and weak bisimilarity [252], strong equivalence [192], (strong and weak) Markovian bisimilarity [167] and extended Markovian bisimilarity [28]. These relations have been shown to be congruences (with some limitations), which fact can be used in order to minimise the state space of an SPA model in a compositional fashion. Thus, the structure of a system consisting of several interacting components can be exploited during Markov chain generation. Any component can be replaced by an equivalent but smaller one before it is used in a composition with other components, which ensures that the state space is kept minimal at any intermediate level of model construction. This strategy, known as *compositional aggregation* helps to

circumvent the state space explosion problem, and has been applied successfully in many applications, see for instance [174].

We now introduce an example stochastic process algebra language that will be considered in the sequel, and define its operational semantics.

Definition 3.7.1 Stochastic process algebra language $\mathcal{L}$
Let Act be the set of valid action names and Pro the set of process names. We distinguish the action $\tau \in Act$ as an internal, invisible activity. Let $P, P_i \in \mathcal{L}$, $a \in Act$, $S \subseteq Act \setminus \{\tau\}$, and $X \in Pro$. The set $\mathcal{L}$ of valid expressions is defined by the following language elements:

stop	inaction		
$a\ ;\ P$	action prefix	$(a, \lambda)\ ;\ P$	Markovian prefix
$P_1\ []\ P_2$	choice	$P_1\ \|[S]\|\ P_2$	parallel composition
hide a **in** P	hiding	X	process instantiation

A set of process definitions (of the form $X := P$) constitutes a process environment. ■

The operational semantic rules shown in Fig. 3.1 (formulated in the style of [275]) define a transition system which contains action transitions (also called immediate transitions), $\overset{a}{\dashrightarrow}$, and Markovian transitions, $\xrightarrow{a,\lambda}$. The semantic model is a multi-transition system, i.e. a transition system where the number of instances of a Markovian transition is recognised. This multi-transition system is defined as the tuple $(\mathcal{L}, Act, \dashrightarrow, \Longrightarrow, P^0)$, where $\mathcal{L}$ is the set of derivable process terms, Act is the set of actions, $P^0 \in \mathcal{L}$ is the initial process, $\dashrightarrow$ is the ordinary transition relation for action transitions and $\Longrightarrow = \{\!|(P, a, \lambda, Q) \mid P, Q \in \mathcal{L},\ a \in Act,\ \lambda \in \mathbb{R}^{>0}|\!\}$ is the multi-relation for Markovian transitions ($\{\!|$ and $|\!\}$ denote multi-set brackets). The multiplicity of a certain Markovian transition is defined as the number of its distinct derivations according to the semantic rules in Fig. 3.1. For more details see for instance [141, 192]. It is possible to flatten the multi-relation to an ordinary transition relation as follows: Transitions with multiplicity greater than one can be amalgamated into a single transition whose rate is the sum of the individual rates, and such a cumulation preserves the behaviour with respect to Markovian bisimulation. Therefore, from now on, we can safely assume that multiple transitions are already cumulated and that the semantic model is an ESLTS (with two ordinary transition relations, one for action transitions and one for Markovian transitions).

Note that the semantic rule for synchronisation of Markovian transitions is parametric in a function ϕ determining the rate of synchronisation, in response to the

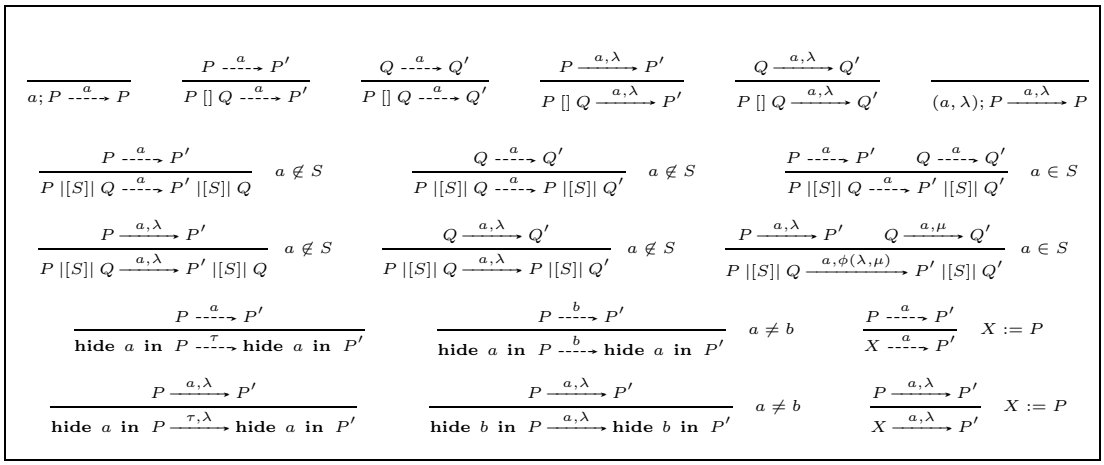


Figure 3.1: Semantic rules for the language $\mathcal{L}$

fact that different synchronisation policies (minimum, maximum, product, ...) are possible. In the process algebra TIPP [178, 179, 180], ϕ is instantiated by multiplication, since strong bisimilarity is a congruence with respect to parallel composition and abstraction, provided that ϕ is distributive over summation of real values, see [162, 167]. Note, also, that the apparent rate construction of PEPA [192] requires a function $\phi(P, Q, \lambda, \mu)$ instead of $\phi(\lambda, \mu)$.

3.7.2 Bisimulation equivalences

In this and the subsequent sections we discuss the important concept of bisimulation. The discussion follows [183], further details may be found in [162].

Bisimulation relations are important since they allow one to compare the behaviour of different processes. Two processes (two expressions of a process algebra language, two states of a transition system) exhibit the same behaviour, if they are bisimilar. For the practical work with transition systems, bisimulation equivalences make it possible to reduce the state space of a process in the following way: Each class of (potentially many) equivalent states is replaced by a single macro state, thereby reducing the size of the state space. In the case where a bisimulation relation is not only an equivalence, but also a congruence, processes may be replaced by equivalent but smaller ones before they are composed with other processes, thereby preserving the behaviour of the overall process.

In the sequel, we consider the language $\mathcal{L}$ and two distinct sub-languages thereof, for which we define strong and weak bisimulation relations. The first sub-

language, $\mathcal{L}_1$, arises by disallowing Markovian prefix. This sub-language is an ordinary, non-stochastic process algebra, a subset of Basic LOTOS [37], where only action transitions appear in the underlying LTS. On this language, strong and weak bisimilarity coincide with Milner's strong and weak bisimilarity [252]. The complementary sub-language, $\mathcal{L}_2$, is obtained by disallowing action prefix. The resulting language is a Markovian process algebra (called MTIPP [179] if ϕ is instantiated with multiplication), for which (strong) Markovian bisimilarity can be defined (the notion of weak bisimilarity has no parallel in this purely Markovian context). Note that Markovian bisimilarity agrees with Hillston's strong equivalence [192]. The semantics of $\mathcal{L}_2$ contains only Markovian transitions, so the semantics of an $\mathcal{L}_2$-specification is a stochastic LTS (SLTS). The complete language, where both prefixes coexist, involves both types of transitions, so the semantics of an $\mathcal{L}$-specification in the general case is an extended SLTS (ESLTS). For this general case we define strong and weak Markovian bisimilarity.

The non-stochastic case $\mathcal{L}_1$: We now recall the notions of strong and weak bisimilarity for the language $\mathcal{L}_1$.

Definition 3.7.2 Strong bisimulation on LTS
An equivalence relation $\mathcal{B}$ on the set of states of an LTS is a strong bisimulation, if $(P,Q) \in \mathcal{B}$ implies that

$$\text{if } P \xrightarrow{a} P', \text{ then } Q \xrightarrow{a} Q', \text{ for some } Q' \text{ with } (P',Q') \in \mathcal{B}$$

Two expressions P and Q are strongly bisimilar if they are contained in a strong bisimulation. ■

Weak bisimilarity is obtained from strong bisimilarity by basically replacing $\xrightarrow{a}$ with $\stackrel{a}{\Longrightarrow}$. Here, $\stackrel{a}{\Longrightarrow}$ denotes an observable a transition that is preceded and followed by an arbitrary number (possibly zero) of invisible $\xrightarrow{\tau}$ activities, i.e. $\stackrel{a}{\Longrightarrow} := \xrightarrow{\tau^*}\xrightarrow{a}\xrightarrow{\tau^*}$. In the case where a is internal ($a = \tau$), $\stackrel{a}{\Longrightarrow}$ abbreviates $\xrightarrow{\tau^*}$.

Definition 3.7.3 Weak bisimulation on LTS
An equivalence relation $\mathcal{B}$ on the set of states of an LTS is a weak bisimulation, if $(P,Q) \in \mathcal{B}$ implies that

$$\text{if } P \stackrel{a}{\Longrightarrow} P', \text{ then } Q \stackrel{a}{\Longrightarrow} Q', \text{ for some } Q' \text{ with } (P',Q') \in \mathcal{B}$$

Two expressions P and Q are weakly bisimilar if they are contained in a weak bisimulation. ■

The purely Markovian case $\mathcal{L}_2$: The equivalence relation on which we focus now is known as Markovian bisimulation [179]. Informally, two states are

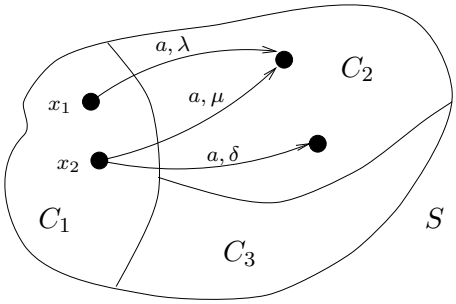


Figure 3.2: Partitioning of the state space S of an SLTS

Markovian bisimilar (members of the same equivalence class) if from both states the same equivalence classes can be reached in one step by the same actions and with the same "cumulative rate" (defined below). There is a strong connection between Markovian bisimulation and classical Markov chain *lumpability* [212, 55]. Informally, Markovian bisimulation is a refinement of lumpability, by distinguishing between different action names. Fig. 3.2 illustrates how the state space S of an SLTS is partitioned into three disjoint subsets, $C_1 \ldots C_3$, also called classes.

Definition 3.7.4 Cumulative rate γ
The cumulative rate from a state P by action a to a set of states C is denoted by the function $\gamma : \mathcal{L} \times Act \times 2^{\mathcal{L}} \mapsto \mathbb{R}$ which is defined as

$$\gamma(P, a, C) = \sum_{\lambda \in E(P,a,C)} \lambda$$

where $E(P, a, C) := \{\!|\ \lambda \mid P \xrightarrow{a,\lambda} P' \wedge P' \in C\ |\!\}$. In this expression, $\{\!|$ and $|\!\}$ denote multi-set brackets. ■

For example, in Fig. 3.2, $\gamma(x_1, a, C_2) = \lambda$ and $\gamma(x_2, a, C_2) = \mu + \delta$.

We can now give the formal definition of Markovian bisimulation on SLTS (defined in a style similar to that of probabilistic bisimulation in [234]).

Definition 3.7.5 (Strong) Markovian bisimulation on SLTS
An equivalence relation $\mathcal{B}$ on the set of states of an SLTS is a (strong) Markovian bisimulation, if $(P, Q) \in \mathcal{B}$ implies that for all equivalence classes C of $\mathcal{B}$ and all actions a it holds that

$$\gamma(P, a, C) = \gamma(Q, a, C)$$

Two expressions P and Q are strong Markovian bisimilar if they are contained in a strong Markovian bisimulation. ■

We do not define a weak bisimulation equivalence for the case $\mathcal{L}_2$, since in this setting all internal transitions $\xrightarrow{\tau,\lambda}$ are associated with a strictly positive (exponentially distributed) delay, which cannot be ignored, and which cannot be "merged" with another such delay. For instance, the delay associated with a sequence $\xrightarrow{\tau,\lambda_1}\xrightarrow{\tau,\lambda_2}$ has a different (non-exponential!) distribution than the delay of a single transition $\xrightarrow{\tau,\lambda}$, for whatever choice of λ_1, λ_2 and λ.

The general case $\mathcal{L}$: We now recall the notions of strong and weak Markovian bisimilarity for ESLTSs, again using the function γ, the cumulative rate.

Definition 3.7.6 Strong Markovian bisimulation for ESLTS
An equivalence relation $\mathcal{B}$ on the set of states of an ESLTS is a strong Markovian bisimulation, if $(P,Q) \in \mathcal{B}$ implies that

(i) if $P \overset{a}{\dashrightarrow} P'$, then $Q \overset{a}{\dashrightarrow} Q'$, for some Q' with $(P',Q') \in \mathcal{B}$,

(ii) for all equivalence classes C of $\mathcal{B}$ and all actions a it holds that
$$\gamma(P,a,C) = \gamma(Q,a,C).$$
Two expressions P and Q are strong Markovian bisimilar if they are contained in a strong Markovian bisimulation. ■

As in the non-stochastic case, weak bisimilarity is obtained from strong bisimilarity by replacing $\overset{a}{\dashrightarrow}$ with $\overset{a}{\Longrightarrow}$, which denotes an observable a transition that is preceded and followed by an arbitrary number (possibly zero) of invisible $\overset{\tau}{\dashrightarrow}$ activities. As discussed in [167], the extension from strong to weak Markovian bisimilarity has to take into account the interplay of Markovian and immediate transitions. Priority of *internal* immediate transitions leads to the following definition [162].

Definition 3.7.7 Weak Markovian bisimulation for ESLTS
An equivalence relation $\mathcal{B}$ on the set of states of an ESLTS is a weak Markovian bisimulation, if $(P,Q) \in \mathcal{B}$ implies that

(i) if $P \overset{a}{\Longrightarrow} P'$, then $Q \overset{a}{\Longrightarrow} Q'$, for some Q' with $(P',Q') \in \mathcal{B}$,

(ii) if $P \overset{\tau}{\Longrightarrow} P' \not\overset{\tau}{\dashrightarrow}$ then there exists Q' such that $Q \overset{\tau}{\Longrightarrow} Q' \not\overset{\tau}{\dashrightarrow}$, and for all equivalence classes C of $\mathcal{B}$ and all actions a
$$\gamma(P',a,C) = \gamma(Q',a,C).$$
Two expressions P and Q are weak Markovian bisimilar if they are contained in a weak Markovian bisimulation. ■

In this definition, $P \not\xrightarrow{\tau}$ denotes that P does not possess an outgoing internal immediate transition. We call such a state *tangible*, as opposed to *vanishing* states which may internally and immediately evolve to another behaviour (denoted $P \xrightarrow{\tau}$). Formally:[5]

Definition 3.7.8 Vanishing and tangible states
A state P of an ESLTS is called vanishing if it possesses an outgoing internal immediate transition (written $P \xrightarrow{\tau}$). Otherwise it is called tangible. ■

It can be shown that strong Markovian bisimilarity is a congruence with respect to the language operators, provided that ϕ is distributive over summation of real values. The same result holds for weak Markovian bisimilarity except for congruence with respect to choice, see [162].

3.7.3 Bisimulation in non-stochastic process algebras

Practical applicability of compositional aggregation relies on efficient algorithms for computing minimised components. In this section, we introduce the general idea of iterative partition refinement, working with the non-stochastic language $\mathcal{L}_1$. Algorithms for the the purely Markovian case (language $\mathcal{L}_2$) and for the general case (language $\mathcal{L}$) will be discussed in the next two sections (Sec. 3.7.3 – Sec. 3.7.5 have been adapted from [183]). BDD-based implementations of these algorithms will be described in Chap. 5.

To illustrate the key ideas, we use as an example a queueing system, consisting of an arrival process and a finite queue, which we specify using the language $\mathcal{L}_1$. The arrival process is modelled as an infinite sequence of incoming arrivals (*arrive*), each followed by an enqueue action (*enq*).

$$Arrival := arrive;\ enq;\ Arrival$$

The behaviour of the queue is described by a family of processes, one for each value of the current queue population.

$$\begin{aligned} Queue_0 &:= enq;\ Queue_1 \\ Queue_i &:= enq;\ Queue_{i+1} \ [] \ deq;\ Queue_{i-1} \qquad 1 \leq i < max \\ Queue_{max} &:= deq;\ Queue_{max-1} \end{aligned}$$

[5] This characterisation of tangible and vanishing states is slightly less restrictive than the one which will be given in the definition of compositionally vanishing states (see Def. 5.2.1), since the latter is devised for a compositional context.

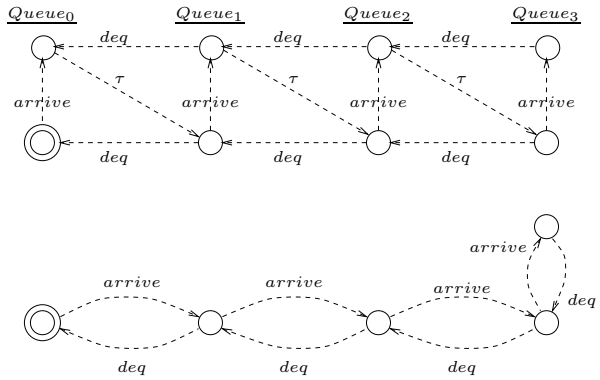


Figure 3.3: LTS of the queueing system example, before and after applying weak bisimilarity

These separate processes are combined by parallel composition in order to describe the whole queueing system. Hiding is used to internalise action *enq*, since it is not needed for further synchronisation.

$$System := \textbf{hide } enq \textbf{ in } \Big(Arrival \;\; |[enq]| \;\; Queue_0 \Big)$$

Fig. 3.3 (top) shows the LTS associated with the *System* specified above for the case that the maximum queue population is $max = 3$. The LTS has 8 states, the initial state being emphasised by a double circle. Fig 3.3 (bottom) shows an equivalent representation, minimised with respect to weak bisimilarity. The reduced LTS is obtained from the original one by replacing every class of weakly bisimilar states by a single (macro) state.

The bisimulation algorithms to be discussed in the sequel are variants of the well-known partition refinement algorithm [266, 121, 207]. (TIPPtool [217, 164, 176, 165], for instance, contains implementations of such algorithms.) The basic idea is as follows: For a given finite state space, partitions of equivalent states are computed by the method of iterative refinement, until a fixed point is reached. This means that starting from an initial partition of the state space (which consists of a single class containing all states), classes are refined until the resulting partition corresponds to a bisimulation equivalence. The result thus obtained is the largest existing bisimulation, in a sense the "best" bisimulation, since it has a minimal number of equivalence classes. We only explain the basic concepts of the algorithms, which can be optimised in several ways [121, 266, 57, 162, 183].

For the refinement of a partition, the notion of a "splitter" is very important. A splitter is a pair (a, C_{spl}), consisting of an action a and a class C_{spl}. During re-

finement, a class C is split with respect to a splitter, which means that subclasses C^+ and C^- are computed, such that subclass C^+ contains all those states from C which can perform an a-transition leading to class C_{spl}, and C^- contains all remaining states.

In the following, an algorithm for strong bisimulation is sketched, which uses a dynamic set of splitters, denoted by $Splitters$. We mention that, by a clever treatment of splitters, it is possible to obtain a time complexity $\mathcal{O}(m \log n)$, where n is the number of states and m is the number of transitions [121].

1. **Initialisation**
   $Partition := \{S\}$
   /* the initial partition consists of a single class which contains all states */
   $Splitters := Act \times Partition$
   /* all pairs of actions and classes have to be considered as splitters */

2. **Main loop**

   while $(Splitters \neq \emptyset)$
   choose splitter $(a, C_{spl}) \in Splitters$
   forall $C \in Partition$
   $split(C, a, C_{spl}, Partition, Splitters)$
   /* all classes (including C_{spl} itself) are split */
   $Splitters := Splitters - (a, C_{spl})$
   /* the processed splitter is removed from the set of splitters */

It remains to specify the procedure *split*. Its task is to split a class C, using (a, C_{spl}) as a splitter. If splitting actually takes place, the input class C is split into subclasses C^+ and C^-.

procedure $split(C, a, C_{spl}, Partition, Splitters)$
$C^+ := \{P \mid P \in C \ \wedge \ \exists\, Q : \ (P \xrightarrow{a} Q \ \wedge \ Q \in C_{spl})\}$
/* the subclass C^+ is computed */
if $(C^+ \neq C \ \wedge \ C^+ \neq \emptyset)$
/* only continue if class C actually needs to be split */
$C^- := C - C^+$
/* C^- is the complement of C^+ with respect to C */
$Partition := Partition \cup \{C^+, C^-\} - \{C\}$
$Splitters := Splitters \cup (Act \times \{C^+, C^-\}) - Act \times \{C\}$
/* the partition and the set of splitters are updated */

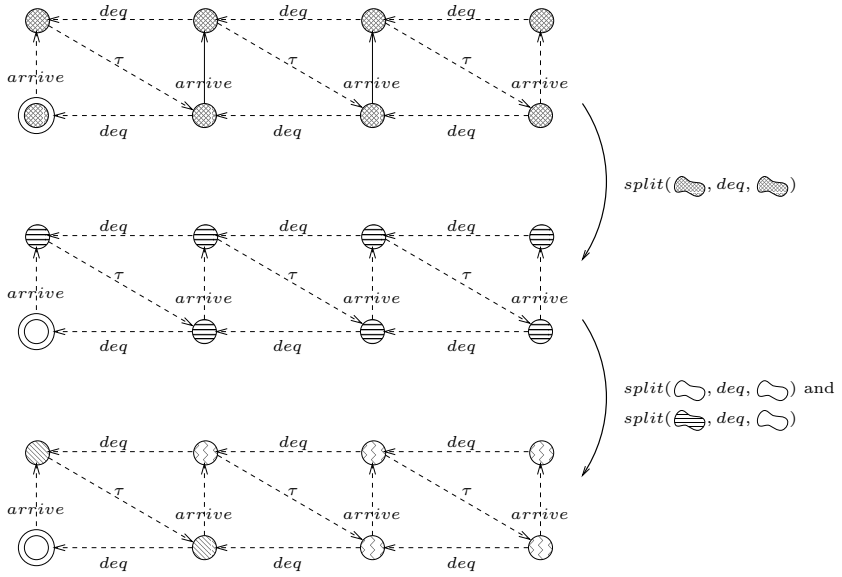


Figure 3.4: Initialisation, first and second refinement step of the algorithm

We illustrate the algorithm by means of the above queueing example. However, we will compute weak instead of strong bisimilarity. The only change required for this purpose is to replace the transition relation $\xrightarrow{a}$ (used in procedure *split*) by the weak relation $\stackrel{a}{\Longrightarrow}$. This, of course, requires the computation of $\stackrel{a}{\Longrightarrow}$ during the initialisation phase. It is known that this computation dominates the complexity of partition refinement, basically because the reflexive and transitive closure $\xrightarrow{\tau^*}$ of internal moves has to be computed in order to build the weak transition relation. The usual way of computing a transitive closure has cubic complexity. (Some slight improvements are known for this task, see for instance [91]. In any case, this is the computationally expensive part.)

The LTS is depicted in Fig. 3.4 (top) where we use a particular shading of states in order to visualise the current partition. At the beginning all states are assumed to be equivalent, and hence, all states are shaded with the same pattern. We use to refer to the set of states shaded like . So, $Partition := \{$$\}$, and *Splitters* is initialised accordingly.

After computing the weak transition relation $\Longrightarrow$, we start partition refinement by choosing a splitter, say $(deq,$ $)$ and computing $split($, $deq,$ $)$. The initial state has no possibility to perform a $\stackrel{deq}{\Longrightarrow}$ transition in contrast to all other states. Therefore $^+ =$ and $^- =$. As a consequence, *Partition* becomes $\{$, $\}$ and new splitters are added to *Splitters* while the currently processed one, $(deq,$ $)$, is removed. This completes the first iteration and leads to the

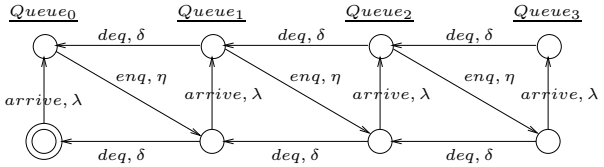


Figure 3.5: Semantic model of the Markovian queueing system, isomorphic to a CTMC

situation depicted in Fig. 3.4 (middle).

By choosing a different splitter, say $(deq, \bigcirc)$, we start the next iteration. Since *Partition* now contains two elements, we compute both $split(\bigcirc, deq, \bigcirc)$ and $split(\bigcirc, deq, \bigcirc)$. $\bigcirc$ cannot be split any further, while splitting of $\bigcirc$ returns $\bigcirc^+ = \bigcirc$ and $\bigcirc^- = \bigcirc$. Updating *Partition* to $\{\bigcirc, \bigcirc, \bigcirc\}$ and adding new splitters leads to the situation depicted in Fig. 3.4 (bottom). Subsequent iterations of the algorithm will divide $\bigcirc$ further, leading to five partitions in total. The algorithm terminates once the set *Splitters* is empty.

3.7.4 Bisimulation in Markovian process algebras

In this section, we consider the MTIPP-style language $\mathcal{L}_2$ where all actions are associated with an exponential delay. The semantic model of a process from the language $\mathcal{L}_2$ is an SLTS, which only contains transitions of the form $\xrightarrow{a,\lambda}$.

We return to the example of a queueing system. The arrival process is now modelled as follows, employing the Markovian action prefix:

$$Arrival := (arrive, \lambda);\ (enq, 1);\ Arrival$$

Action *arrive* occurs with rate λ, whereas for action *enq* we specified the (passive) rate 1, the neutral element of multiplication. The queue process determines the actual rate of *enq*, occurring as a result of synchronisation via *enq*.

$$\begin{array}{rcll} Queue_0 & := & (enq, \eta);\ Queue_1 & \\ Queue_i & := & (enq, \eta);\ Queue_{i+1} \ [] \ (deq, \delta);\ Queue_{i-1} & 1 \leq i < max \\ Queue_{max} & := & (deq, \delta);\ Queue_{max-1} & \end{array}$$

Fig. 3.5 depicts the SLTS obtained from the parallel composition of processes *Arrival* and $Queue_0$ synchronised over action *enq*.

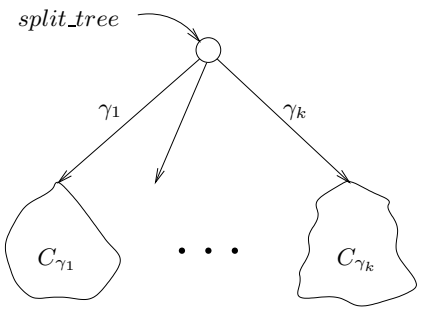


Figure 3.6: *split_tree* used by procedure *split′*

For the refinement of a partition in the Markovian case, again the notion of a "splitter", defined by a pair (a, C_{spl}), is very important. When, during refinement, a class C_i is split with respect to a splitter, now possibly more than two (!) subclasses $C_{\gamma_1}, C_{\gamma_2}, \ldots, C_{\gamma_k}$ are computed ($k \geq 1$).

The basic bisimulation algorithm is the same as in Sec. 3.7.3, only the procedure *split* needs to be replaced by a new procedure *split′*, whose task it is to split a class C_i, using the combination (a, C_{spl}) as a splitter. Procedure *split′* works on a data structure *split_tree* shown in Fig. 3.6. It essentially sorts states according to their γ-values. Input class C is split according to the cumulative rate of a-labelled transitions from a state $P \in C_i$ to class C_{spl}, such that the cumulative rate $\gamma(P, a, C_{spl}) = \gamma_j$ is the same for all the states P belonging to the same subclass C_{γ_j}. Each branch of the *split_tree* corresponds to one particular subclass.

procedure $split'(C, a, C_{spl}, Partition, Splitters)$
  forall $P \in C$
    $\gamma := \gamma(P, a, C_{spl})$
    /* the cumulative rate from state P to C_{spl} is computed */
    $insert(split\_tree, P, \gamma)$
    /* state P is inserted into the *split_tree* */
  /* now, *split_tree* contains k leaves $C_{\gamma_1}, \ldots, C_{\gamma_k}$ */
  if $(k > 1)$
    /* only continue if C has been split into $k > 1$ subclasses */
    $Partition := Partition \cup \{C_{\gamma_1}, C_{\gamma_2}, \ldots, C_{\gamma_k}\} - \{C\}$
    $Splitters := Splitters \cup (Act \times \{C_{\gamma_1}, C_{\gamma_2}, \ldots, C_{\gamma_k}\}) - Act \times \{C\}$
    /* the partition and the set of splitters are updated */

In the **forall** loop of procedure *split′*, the cumulative rate γ is computed for every state P in class C, and state P is inserted (by procedure *insert*) into the *split_tree*

such that states with the same cumulative rate fall into the same branch. The *split_tree* finally has k leaves, i.e. k different values of γ have appeared. If splitting has taken place (i.e. if $k > 1$), the partition must be refined and the set of splitters has to be updated.

It can be shown that such an algorithm for computing Markovian bisimilarity on a given SLTS can be implemented in time complexity $\mathcal{O}(m \log n)$ and in space complexity $\mathcal{O}(m + n)$, where n is the number of states and m is the number of transitions. The proof is given in [162].

Note that, since Markovian bisimulation corresponds to Markov chain lumpability, the algorithm can be used to efficiently compute lumpable partitions of an SPA description as well as a CTMC in isolation. Note further that the SLTS from Fig. 3.5 does not contain any pair of states which are Markovian bisimilar, i.e. each of its 8 states forms its own equivalence class.

3.7.5 Bisimulation with Markovian and immediate transitions

We now consider the complete language $\mathcal{L}$ with both immediate and Markovian actions. The semantic model of such a specification is an ESLTS with two types of transitions: Markovian transitions $\xrightarrow{a,\lambda}$ and action transitions $\overset{a}{\dashrightarrow}$.

Again, we return to the queueing system example. In the arrival process, action *arrive* now has an exponential delay, whereas action *enq* is immediate.

$$Arrival := (arrive, \lambda); enq; Arrival$$

The specification of the Queue is modified such that action *enq* is immediate and action *deq* has exponential delay.

$$\begin{array}{rcll} Queue_0 & := & enq; Queue_1 & \\ Queue_i & := & enq; Queue_{i+1} \,[]\, (deq, \delta); Queue_{i-1} & 1 \leq i < max \\ Queue_{max} & := & (deq, \delta); Queue_{max-1} & \end{array}$$

The overall queueing system is again given by the composition of *Arrival* and $Queue_0$, where *enq* is hidden after synchronisation. Fig. 3.7 (top) depicts the resulting ESLTS, whose equivalence classes are indicated at the bottom of the figure, in order to illustrate the effect of weak Markovian bisimilarity.

In the context of the complete language $\mathcal{L}$, the notion of weak Markovian bisim-

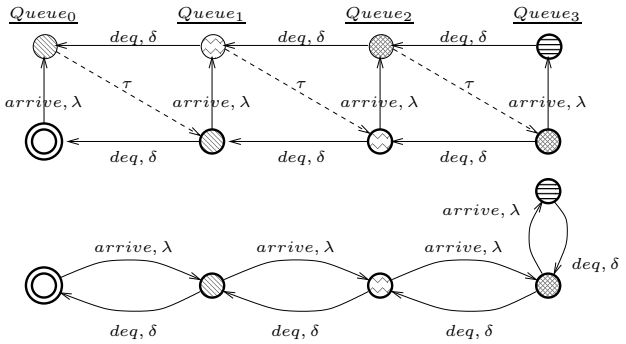


Figure 3.7: ESLTS of the queueing system example, before and after applying weak Markovian bisimilarity

ilarity allows one to derive a CTMC from a given specification. Immediate transitions do not have a counterpart on the level of the CTMC but weak Markovian bisimilarity justifies to eliminate *internal* immediate transitions such that a SLTS, i.e. an (action labelled) CTMC is obtained. We emphasise that elimination of immediate transitions requires abstraction of immediate actions before applying weak Markovian bisimilarity. In other words: Visible immediate actions cannot be eliminated. Therefore, in order to construct a CTMC, it is mandatory to hide all immediate actions before applying weak Markovian bisimulation. Furthermore, a *unique* CTMC exists only if non-determinism is absent after applying weak Markovian bisimilarity.

We now discuss an algorithm to compute weak Markovian bisimilarity which is based on the ones given in the two previous sections, but proceeds in a different way. In general, the algorithm needs to distinguish between *convergent* and *divergent* states. A divergent state is a vanishing state which has no possibility to internally and immediately evolve to a tangible state. All other states are called convergent. The algorithm for the general case is quite involved, see [174, 162, 183], therefore we restrict the present discussion to the divergence-free case. We use the notation $P \searrow\!\!\!\!\!^{\tau}\ P'$ to indicate that P may internally and immediately evolve to a tangible state P', i.e. to a state P' where no further internal immediate transition is possible. The following algorithm computes weak Markovian bisimilarity for divergence-free ESLTS.

1. **Initialisation** as before in Sec. 3.7.3.
   In addition, weak transitions $\Longrightarrow$ are computed from $\dashrightarrow$.

2. **Main loop**

while $(Splitters \neq \emptyset)$
choose splitter (a, C_{spl})
forall $C \in Partition$ $split(C, a, C_{spl}, Partition, Splitters)$
/* all classes are split with respect to weak transitions */
forall $C \in Partition$ $split''(C, a, C_{spl}, Partition, Splitters)$
/* all classes are split with respect to Markovian transitions */
$Splitters := Splitters - (a, C_{spl})$
/* the processed splitter is removed from the set of splitters */

The main loop calls two different procedures, *split* and *split''* requiring further explanation. The first, *split*, refines with respect to clause *(i)* of Def. 3.7.7, which is achieved using procedure *split* from Sec. 3.7.3, but applied on weak transitions (as in the example of Sec. 3.7.3). The second procedure, *split''*, is more complicated. It refines with respect to the clause *(ii)* of Def. 3.7.7. The details are as follows:

procedure $split''(C, a, C_{spl}, Partition, Splitters)$
forall $P \in C \ \wedge \ P \not\xrightarrow{\tau}$
/* P is a tangible state */
$\gamma := \gamma(P, a, C_{spl})$
/* the cumulative rate to C_{spl} is computed */
$insert(split\_tree, P, \gamma)$
/* state P is inserted into the *split_tree* */
/* now, *split_tree* contains k leaves $C_{\gamma_1}, \ldots, C_{\gamma_k}$ */
forall $P \in C \ \wedge \ P \xrightarrow{\tau}$
/* P is a vanishing state */
if $(\exists\, \gamma_j : \ P \searrow^{\tau} Q \ \Rightarrow \ Q \in C_{\gamma_j})$
/* P can internally and immediately reach tangible states of class C_{γ_j} only */
$insert(split\_tree, P, \gamma_j)$
$Partition := Partition \cup \{C_{\gamma_1}, C_{\gamma_2}, \ldots, C_{\gamma_k}\} - \{C\}$
$Splitters := Splitters \cup (Act \times \{C_{\gamma_1}, C_{\gamma_2}, \ldots, C_{\gamma_k}\}) - Act \times \{C\}$
/* the partition and the set of splitters are updated */
if $(C \neq \bigcup_1^k C_{\gamma_j})$
/* some vanishing states have not been covered yet */
$Partition := Partition \cup \{C - \bigcup_1^k C_{\gamma_j}\}$
$Splitters := Splitters \cup (Act \times \{C - \bigcup_1^k C_{\gamma_j}\})$
/* all remaining vanishing states form a new class, since they can internally
and immediately evolve to tangible states belonging to different classes */

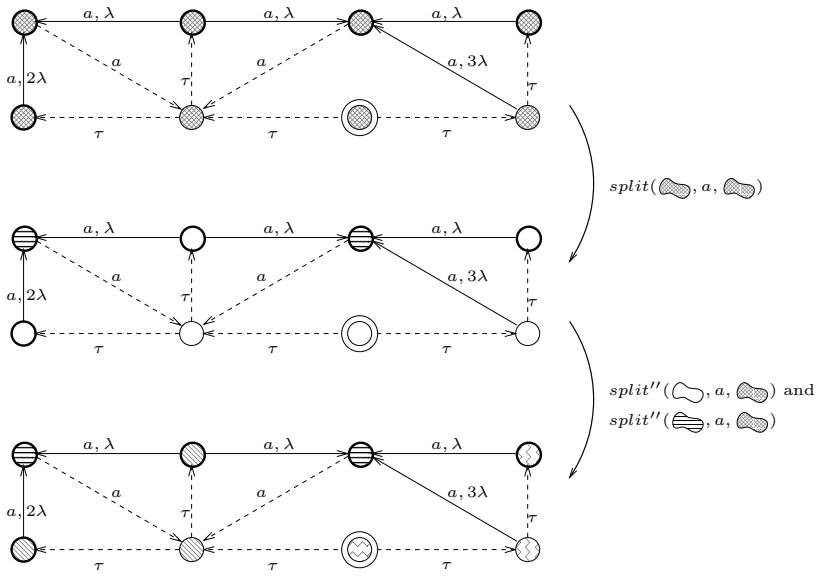


Figure 3.8: Initialisation and first refinement step of the algorithm

Fig. 3.7 (bottom) shows the result obtained when this algorithm is applied to the ESLTS in Fig. 3.7 (top) (tangible states are highlighted by bold circles in the figure). In order to illustrate the algorithm on a nontrivial example, where all the distinctions between different types of vanishing states occur, we consider a different example, depicted in Fig. 3.8 (top).

The initialisation proceeds as usual: After computing the weak transition relation $\Longrightarrow$, the algorithm chooses a splitter, say $(a, \text{[bean]})$ and computes $split(\text{[bean]}, a, \text{[bean]})$. Two states may perform a $\overset{a}{\Longrightarrow}$ transition in contrast to all other states. Therefore $\text{[bean]}^+ = \text{[bean]}$ and $\text{[bean]}^- = \text{[bean]}$. *Partition* and *Splitters* are updated accordingly, leading to the situation depicted in Fig. 3.8 (middle).

The subsequent invocation of $split''(\text{[bean]}, a, \text{[bean]})$ is most interesting (as opposed to $split''(\text{[bean]}, a, \text{[bean]})$). First, the three tangible markings (indicated by bold circles) in class [bean] are inserted into *split_tree*, according to their cumulative rates of moving into (former) class [bean]. This leads to a tree with two leaves, $C_{2\lambda}$ and C_λ, containing two, respectively one state. Now the three remaining vanishing states are considered: The rightmost vanishing states can internally and immediately evolve only to tangible states of class C_λ (note that according to Def. 3.7.3 the transition $\xrightarrow{a,3\lambda}$ is irrelevant since it originates in a vanishing state). For the same reason, the left vanishing state is inserted into class $C_{2\lambda}$. Only the initial state is not covered yet, since it has an internal, non-deterministic choice of behaving as a member of either of the classes. Hence, this state forms a new

class, . In total, $split''(\ldots, a, \ldots)$ has split into , (representing $C_{2\lambda}$ and C_λ), and , leading to the situation depicted in Fig. 3.8 (bottom) (the *Partition* and *Splitters* are updated accordingly). This situation incidentally coincides with the classes of weak Markovian bisimilarity, because subsequent refinement steps do not reveal any distinction in one of these four classes. The algorithm terminates once the set *Splitters* is emptied.

This algorithm computes weak Markovian bisimilarity on a given ESLTS. Its implementation, based on [198, 144, 38, 20] requires $\mathcal{O}(n^3)$ time and $\mathcal{O}(n^2)$ space, where n is the number of states. The proof is given in [162] for the divergence-free as well as the general case. As in the non-stochastic case, the time complexity of weak bisimulation is due to the fact that a transitive closure operation is needed to compute weak transitions in either case.

For the moment, this concludes our discussion of efficient algorithms for computing bisimulation equivalences. Their BDD-based implementation will be discussed in Chap. 5. As an important result, we have seen that the computational complexity of computing bisimulation equivalences does *not* increase when moving from a non-stochastic to a stochastic setting. For Markovian bisimilarity this fact is also mentioned (in similar settings) in [198] and in [28]. We also mention that, in principle, minimisation with respect to a given bisimulation relation can be carried out directly on a process algebraic specification by purely syntactic transformations, using a complete axiomatisation (as given for weak Markovian bisimulation in [175]).

Part II

The symbolic approach

Chapter 4

Symbolic representation of transition systems

In this chapter, we describe how labelled transition systems (LTS) and their stochastic extensions, that is SLTSs and ESLTSs, can be represented with the help of decision diagrams. We first consider the symbolic representation of LTSs by binary decision diagrams (BDD), the data structure which was introduced already in Sec. 2.4. Then we introduce Decision Node BDDs and Multi Terminal BDDs as extensions of BDDs which are capable of representing the numerical information within an SLTS or ESLTS. The chapter concludes with some considerations concerning the size of decision diagrams and the complexity of operations thereon. While this chapter focuses on the techniques to represent transition systems symbolically, the following chapters will demonstrate how manipulations and analyses of various sort can be carried out efficiently on the basis of symbolic representations.

4.1 Representing LTSs with the help of BDDs

We first define how elements from finite sets (e.g. actions, states) are encoded as Boolean vectors.

Definition 4.1.1 Encoding function $\mathcal{E}(.)$
Let $S = \{s_1, \ldots, s_m\}$ be a finite set. An encoding function is an injective mapping $\mathcal{E} : S \mapsto \mathbb{B}^n$ where $n \geq \lceil \log_2 m \rceil$. ■

For $s \in S$, $\mathcal{E}(s) = (b_1, \ldots, b_n)$ is a Boolean vector of length n. For convenience, we introduce the following notation which we shall need later (in Chap. 5):

Definition 4.1.2 Minterm function $\mathcal{M}(.)$
Given n distinct Boolean variables $\mathsf{a}_1, \ldots, \mathsf{a}_n$ and a Boolean vector $(b_1, \ldots, b_n)$ of length n, we denote by $\mathcal{M}(\mathsf{a}_1, \ldots, \mathsf{a}_n; b_1, \ldots, b_n)$ the minterm consisting of the conjunction of n literals (a literal is either a Boolean variable or its negation), i.e. $\mathcal{M}(\mathsf{a}_1, \ldots, \mathsf{a}_n; b_1, \ldots, b_n) = \mathsf{a}_1^* \wedge \ldots \wedge \mathsf{a}_n^*$ where $\mathsf{a}_i^* = \mathsf{a}_i$ if $b_i = 1$ and $\mathsf{a}_i^* = \overline{\mathsf{a}_i}$ if $b_i = 0$. ■

As an example, we have $\mathcal{M}(\mathsf{a}_1, \mathsf{a}_2, \mathsf{a}_3; 0, 1, 1) = \overline{\mathsf{a}_1} \wedge \mathsf{a}_2 \wedge \mathsf{a}_3$.

We now discuss how a given LTS can be represented symbolically by a BDD. The idea is to encode states and transition labels by Boolean vectors. Each transition $s \xrightarrow{a} t$ of the LTS corresponds to a Boolean vector of length $n_a + 2n_s$, whose positions correspond to Boolean variables $\mathsf{a}_1 \ldots \mathsf{a}_{n_a}, \mathsf{s}_1 \ldots \mathsf{s}_{n_s}, \mathsf{t}_1 \ldots \mathsf{t}_{n_s}$. This vector encodes the action label, the source and the target state of the transition. (We assume that the number of distinct actions to be encoded is less than $2^{n_a}+1$, so that n_a bits are suitable to encode them, and similarly for the number of states). The next definition states in which way an LTS is represented by a BDD.

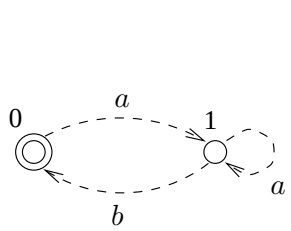
Definition 4.1.3 Symbolic Representation of an LTS by a BDD
Let $\mathcal{T} = (S, L, \dashrightarrow, s)$ be a LTS. Let $\mathsf{B} = (Vert, \mathsf{var}, \mathsf{then}, \mathsf{else})$ be a BDD and $Vars = \{\mathsf{a}_1, \ldots, \mathsf{a}_{n_a}, \mathsf{s}_1, \ldots, \mathsf{s}_{n_s}, \mathsf{t}_1, \ldots, \mathsf{t}_{n_s}\}$, such that $n_a \geq \lceil \log_2 |L| \rceil$ and $n_s \geq \lceil \log_2 |S| \rceil$. We say that B is the symbolic representation of $\mathcal{T}$ iff

$$x \xrightarrow{a} y \;\Leftrightarrow\; f_{\mathsf{B}}\Big|_{(\mathsf{a}_1,\ldots,\mathsf{a}_{n_a})=\mathcal{E}(a),(\mathsf{s}_1,\ldots,\mathsf{s}_{n_s})=\mathcal{E}(x),(\mathsf{t}_1,\ldots,\mathsf{t}_{n_s})=\mathcal{E}(y)} = 1$$

In this case, we write $\mathsf{B} \triangleright \mathcal{T}$ for short. ■

Note that by this definition, the BDD B, which is the symbolic representation of LTS $\mathcal{T}$, does not encode the information about the initial state s of the LTS. This is not a problem, since the identity of the initial state can easily be stored in memory as a separate data item.

As a first example, Fig. 4.1 shows a simple LTS, the way transitions are encoded and the corresponding BDD B. Since there are only two distinct action labels (a and b), one Boolean variable (a) suffices to encode that label, and we set $\mathcal{E}(a) = 0$ and $\mathcal{E}(b) = 1$. The number of state variables n_s is also equal to one,



	$(\mathsf{a},\mathsf{s},\mathsf{t})$
$0 \xrightarrow{a} 1$	$(0,0,1)$
$1 \xrightarrow{a} 1$	$(0,1,1)$
$1 \xrightarrow{b} 0$	$(1,1,0)$

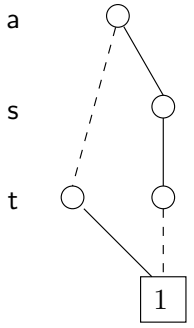
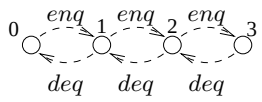


Figure 4.1: Simple LTS, transition encoding and corresponding BDD



	$(\mathsf{a}_1,\mathsf{a}_2,\mathsf{s}_1,\mathsf{t}_1,\mathsf{s}_2,\mathsf{t}_2)$
$0 \xrightarrow{enq} 1$	$(0,1,0,0,0,1)$
$1 \xrightarrow{enq} 2$	$(0,1,0,1,1,0)$
$2 \xrightarrow{enq} 3$	$(0,1,1,1,0,1)$
$1 \xrightarrow{deq} 0$	$(1,0,0,0,1,0)$
$2 \xrightarrow{deq} 1$	$(1,0,1,0,0,1)$
$3 \xrightarrow{deq} 2$	$(1,0,1,1,1,0)$

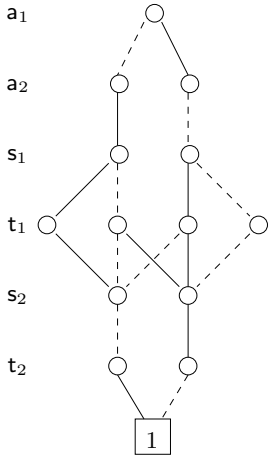


Figure 4.2: Queue LTS, transition encoding and corresponding BDD

since there are only two states. Note that this is the same BDD as in Fig. 2.5 which represents the Boolean function $f_{\mathsf{B}}(\mathsf{a},\mathsf{s},\mathsf{t}) = (\overline{\mathsf{a}} \wedge \mathsf{t}) \vee (\mathsf{a} \wedge \mathsf{s} \wedge \overline{\mathsf{t}})$.

As a second example, Fig. 4.2 shows the LTS corresponding to a simple queue process, the way transitions are encoded and the resulting BDD. Here we deliberately choose to encode the action labels *enq* and *deq* with the help of two Boolean variables[1] a_1 and a_2, i.e. we choose $n_a = 2$. In particular, we set $\mathcal{E}(enq) = (0,1)$ and $\mathcal{E}(deq) = (1,0)$. The LTS has four states, therefore two bits are needed to represent the state, i.e. we can set $n_s = 2$. Note that in this BDD we use a special "interleaved" ordering of the Boolean variables encoding the source and target state, which leads us to the following discussion.

[1]Since there are only two distinct actions in the LTS, one bit would actually be enough to encode the action. However, the encoding 0 is often reserved for the special internal action τ, and in any case it is not mandatory to use the smallest possible number of bits.

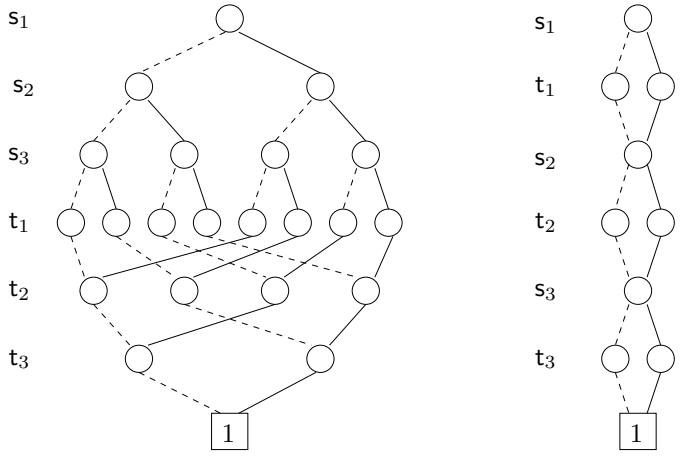


Figure 4.3: The BDD Stab for $n = 3$ pairs of s and t variables. Non-interleaved (left) and interleaved variable ordering (right).

As already mentioned, the size of a BDD, i.e. its number of vertices, is highly dependent on the chosen variable ordering. As a prominent example, consider the function

$$f_{\mathsf{Stab}}(\mathsf{s}_1, \ldots, \mathsf{s}_n, \mathsf{t}_1, \ldots, \mathsf{t}_n) = \bigwedge_{i=1}^{n} (\mathsf{s}_i \leftrightarrow \mathsf{t}_i)$$

Under the naïve non-interleaved ordering $s_1 \prec \ldots \prec s_n \prec t_1 \prec \ldots \prec t_n$ the number of vertices for BDD Stab would be $3 \cdot 2^n - 1$, i.e. exponential in the number of state variables. On the other hand, it is possible to represent this function by a very compact BDD, whose number of vertices is only $3 \cdot n + 2$, i.e. linear in the number of state variables, provided that the following "interleaved" ordering of Boolean variables is employed: $s_1 \prec t_1 \prec \ldots \prec s_n \prec t_n$. Figure 4.3 shows the BDD Stab for a process with $n = 3$ state variables, using the non-interleaved and the interleaved orderings.

In the context of transition systems, experience has shown that the following variable ordering yields small BDD sizes [118]:

$$\mathsf{a}_1 \prec \ldots \prec \mathsf{a}_{n_a} \prec \mathsf{s}_1 \prec \mathsf{t}_1 \prec \ldots \prec \mathsf{s}_{n_s} \prec \mathsf{t}_{n_s}$$

i.e. the variables encoding the action come first, followed by the variables for source and target state interleaved. As we shall see, this ordering is also advantageous in view of the parallel composition operator discussed below (see Sec. 5.1).

4.2 BDD extensions for representing real-valued functions

Clearly, pure BDDs are not capable of representing the numerical information within a *stochastic* LTS, where transitions between states are labelled not only by an action label, but also by a transition rate, i.e. by a non-negative real number. In order to represent a SLTS with a fixed finite number of different values of the transition rates, one could encode these values as Boolean vectors, as has been done in [221]. However, this approach is impractical in most cases, since computations on the numerical values have to be performed, whereby new values are generated whose encoding is undefined.

In the literature, several modifications and augmentations of the BDD data structure have been proposed for representing functions of the type $f : \mathbb{B}^n \mapsto \mathbb{D}$, where $\mathbb{D}$ denotes an arbitrary set (we are particularly interested in the case where $\mathbb{D}$ is a finite subset of $\mathbb{R}$) [284]. Most prominent among these are multi-terminal BDDs (MTBDD) [86, 17], edge-valued BDDs (EVBDD) [232] and Binary Moment Diagrams (BMD) [47]. Another BDD extension with the same capability are decision-node BDDs (DNBDD) [298]. These data structures can capture not only functional, but also numerical information, a feature which is needed for representing the information about the transition rates of SLTSs. In this thesis, we concentrate on the DNBDD and MTBDD data structures, since the former is a new approach developed by ourselves, and the latter offers very good support for linear algebra operations which we shall employ for numerical analysis based on symbolic representations (cf. Chap. 7).

For the sake of completeness, we mention some other data structures which are related to BDDs and also have the capability of representing numerical information. Ciardo and Miner developed matrix diagrams [79, 80, 254, 255, 253] for the space-efficient representation of structured Markov models. This graph-based data structure is based on multi-valued decision diagrams that were originally used for hardware verification [310, 206]. Bozga and Maler developed probabilistic decision graphs [40] as a concise representation formalism for probability vectors and probabilistic transition system. They employ this data structure for the simulation of large probabilistic systems by successive calculation of next-state probabilities, which amounts to symbolic vector-matrix multiplication.

4.3 Decision Node BDDs (DNBDD)

In this section, the data structure DNBDD, developed by the author at the University of Erlangen [294, 296, 295, 297, 298], is introduced. DNBDDs are tailored for the symbolic representation of SLTSs and in many instances allow a very compact representation.

When moving from an LTS to an SLTS encoding, with the MTBDD, BMD and EVBDD approaches there is less sharing of subgraphs compared to the original BDD, which means that the compactness of the representation is diminished. Therefore, when developing the DNBDD data structure, it was our aim to use the unmodified BDD (which represents the functional information of the LTS) and decorate it with the additional rate information in a fully orthogonal fashion. DNBDDs are the only type of decision diagram where the basic graph structure remains unmodified when moving from an LTS to an SLTS encoding. The addition of the rate information does not alter the basic structure of the decision diagram. Rather, additional information is superimposed on it.

In the sequel, we define the DNBDD data structure and explain how it is employed for the symbolic representation of SLTSs. In Sec. 5.1.3, we will discuss a DNBDD-based procedure for the parallel composition of submodels which can be used for efficiently building complex models from small components, without inducing the problem of state space explosion. Also, the MTBDD-based minimisation algorithm for stochastic LTSs which we describe in Sec. 5.3.2, based on the concept of Markovian bisimulation, can be implemented on the DNBDD data structure, as has been realised in the tool DNBDDTOOL [43]. In this thesis, we do not describe DNBDD-based numerical analysis, since efficient algorithms and tool support for linear algebra operations on this data structure have not yet been developed. Therefore, for numerical analysis we switch to MTBDDs, where such support is already available.

4.3.1 Definition of DNBDDs

As we have seen, BDDs represent functions of the type $\mathbb{B}^n \mapsto \mathbb{B}$. As we shall see in Section 4.4, MTBDDs can represent functions of the type $\mathbb{B}^n \mapsto \mathbb{R}$ (note that the image of function f is of course a finite subset of $\mathbb{R}$). The following consideration led to the development of DNBDDs: When extending BDDs in order to represent SLTS, they must associate a real value (a rate) with exactly those Boolean vectors which are valid encodings of a transition, i.e. they must actually be capable of representing functions of the type $f : \mathbb{B}^n \mapsto \{(0,0)\} \cup$

($\{1\} \times \mathbb{R}$). The first component of the image determines whether a certain Boolean vector is a valid transition encoding. This information can be captured by the terminal vertex of a conventional BDD. The second component of the image is only meaningful if the first component is equal to 1, and represents the transition rate. For those Boolean vectors $(b_1, \ldots, b_n)$ which are mapped to a tuple of the form $(1, \lambda)$, where $\lambda \in \mathbb{R}$, the question is where to store the value λ, without changing the basic BDD-structure.

Based on our original definition of a BDD (cf. Def. 2.4.1), we next provide the definition of a path through a BDD:

Definition 4.3.1 Path
A path through a BDD (over Boolean variables $\mathsf{v}_1, \ldots, \mathsf{v}_n$*) is a vector of vertices* $(x_1, \ldots, x_k)$*,* $1 \leq k \leq n+1$*, where* $x_i \in Vert$*,* x_1 *is the BDD root vertex,* $x_k \in T$ *(*x_k *is a terminal vertex) and* $\forall i < k : \ (x_{i+1} = \mathsf{else}(x_i)) \vee (x_{i+1} = \mathsf{then}(x_i))$*. A path is called a true-path iff* $x_k = 1$*, otherwise it is called a false-path. We denote the set of all (true-) paths through a BDD by Paths (True-Paths). For a given Boolean assignment* $(b_1, \ldots, b_n) \in \mathbb{B}^n$*, the function* $\mathsf{path}(b_1, \ldots, b_n) = (x_1, \ldots, x_k)$ *returns the corresponding path through the BDD. We define the length of a path by* $length(x_1, \ldots, x_k) = k$*.* ■

We use the following notation: Boolean vectors are written as $(b_1, \ldots, b_n)$, paths (i.e. vectors of BDD vertices) are written as $(x_1, \ldots, x_k)$, and rate lists (see below) are written as $[\lambda_1, \ldots, \lambda_{2^{n+1-k}}]$.

If a given path $(x_1, \ldots, x_k)$ has length $k = n+1$, which is the maximal possible length for a path, that path contains a vertex for every Boolean variable, formally $\forall\ 1 \leq i \leq n : \mathsf{var}(x_i) = \mathsf{v}_i$. This means that the path corresponds to exactly one Boolean assignment $(b_1, \ldots, b_n)$. In this case, we say that the path does not contain any don't cares. If a path is of length $k < n+1$, it contains $n+1-k = d$ don't cares. Such a path corresponds to 2^d different Boolean assignments (because for every don't care two Boolean values are possible). Every Boolean assignment is mapped onto exactly one path. Several Boolean assignments (always a power of 2) may be mapped onto the same path, in which case the path has one or more don't cares. Therefore we assign to each true-path a list of real numbers (rates), also called a rate list, whose length (a power of 2) is determined by the number of don't cares of the path. Formally, we introduce the function $\mathsf{rates}(x_1, \ldots, x_k) = [\lambda_1, \ldots, \lambda_{2^{n+1-k}}]$. Thus, the correspondence of Boolean assignments to rates is one to one, uniquely defined by the lexical ordering of the Boolean assignments. We illustrate this concept in Fig. 4.4.

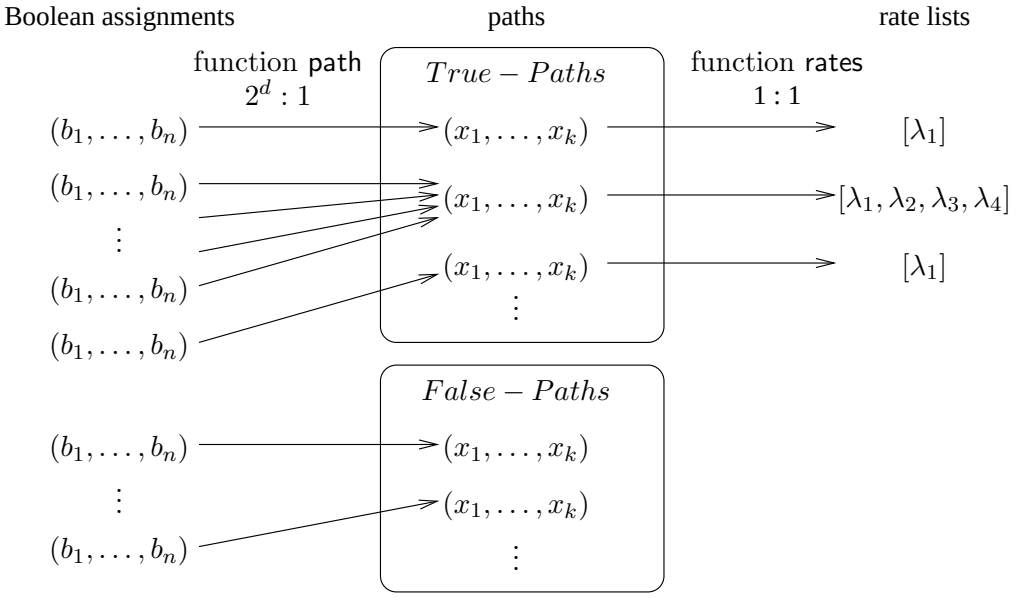


Figure 4.4: Correspondence between Boolean assignments, paths and rate lists

So far, we decided that every true-path is mapped onto a real-valued list whose length is given by the number of Boolean assignments corresponding to the path. Next we must find a practical method for attaching that information to the BDD. Thinking about the way a path is characterised, we find that a subset of the BDD vertices, the so-called *Decision Nodes* play a key role in this consideration.

Definition 4.3.2 Decision Node
A non-terminal BDD-vertex $x \in NT$ *is called decision node iff* $\mathsf{else}(x) \neq 0 \wedge \mathsf{then}(x) \neq 0$*, i.e. iff the terminal vertex* 1 *can be reached via both outgoing edges of vertex* x*. The set of decision nodes is denoted* DN*.* ■

Let $(x_1, \ldots, x_k) \in$ *True-Paths*. Let x_j be the "last" decision node on that path, i.e. $x_j \in DN \wedge \forall\ j < l \leq k : x_l \notin DN$. We then attach the rate list $\mathsf{rates}(x_1, \ldots, x_k) = [\lambda_1, \ldots, \lambda_{2^{n+1-k}}]$ to the edge (x_j, x_{j+1}). (In the special (trivial) case where the path $(x_1, \ldots, x_k)$ does not contain any decision node, the rate list is attached to the edge (x_1, x_2).) We can now give the definition for the DNBDD data structure:

a	s	t	$f(\mathsf{a},\mathsf{s},\mathsf{t})$
0	0	1	$(1,\lambda)$
0	1	1	$(1,\mu)$
1	0	1	$(1,\alpha)$
1	1	0	$(1,\beta)$
else			$(0,0)$

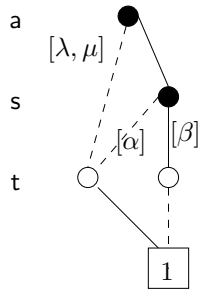


Figure 4.5: A function f and the corresponding DNBDD

Definition 4.3.3 Decision Node BDD (DNBDD)
A Decision Node BDD (DNBDD) is a BDD extended by a function

$$\mathsf{rates} : True\text{–}Paths \mapsto (\mathbb{R})^+$$

(where $(\mathbb{R})^+$ is the set of finite lists over $\mathbb{R}$), such that for any true-path p, $\mathsf{rates}(p) \in \mathbb{R}^{2^d}$ if d is the number of don't cares on path p. The list $\mathsf{rates}(p)$ is attached to the outgoing edge of the last decision node on path p, i.e. the decision node nearest to the terminal vertex 1. (In the special case where path p does not include any decision node, the list $\mathsf{rates}(p)$ is attached to the first edge of the path.) ■

Note that in this definition, the length of the list $\mathsf{rates}(p)$ is not a global constant but depends on the number of don't cares d (i.e. on the length k) of the individual true-path p.

The concept of DNBDDs is illustrated in Fig. 4.5 (in the figure, decision nodes are drawn solid black). In this example, there are four Boolean vectors mapped to a tuple $(1, x)$, where x is the numeric (rate) value (as shown in the left part of the figure). The first two assignments are mapped onto the same path, a path which has a don't care in the Boolean variable s. Therefore, the corresponding rate list $[\lambda, \mu]$ has length two.

The fact that DNBDDs are a *canonical* representation of functions of the type $f : \mathbb{B}^n \mapsto \{(0,0)\} \cup (\{1\} \times \mathbb{R})$ follows from the following considerations: The decision graph of a DNBDD is a standard BDD; it is uniquely determined since BDDs are canonical for functions of the type $f : \mathbb{B}^n \mapsto \mathbb{B}$. The additional association of true-paths with rate lists, the length and contents of the rate lists, and their attachment to certain edges of the graph are also well-defined and uniquely determined.

The practical realisation of the DNBDD concept introduced so far induces the

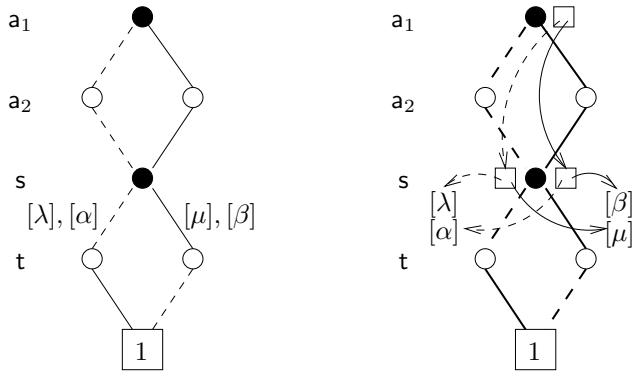
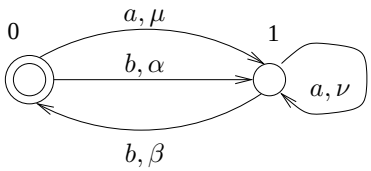


Figure 4.6: Two true-paths sharing their last decision node, DNBDD with rate tree

following problem: There are situations, where several true-paths share their last decision node. This is the case if and only if there exists a decision node which can be reached from the root by more than one path. As an example, see Fig. 4.6 (left), where a decision node has two incoming edges. In such a case, several rate lists will be attached to the same edge. From the point of view of canonicity this would not be a problem, since the one-to-one correspondence between true-paths and rate lists is preserved based on lexicographical ordering. However, in the algorithms for manipulating the data structure this would result in a confusion, since during recursive descent it would not be clear any more which rate list corresponds to which path. In order to overcome this problem, we introduce a pointer structure, the so-called *rate tree*, as illustrated in Fig. 4.6 (right). The rate tree is an unbalanced binary tree which makes it possible to access rate lists during recursive descent through the BDD [296, 298]. The terminal vertices of the rate tree contain the rate lists. Its internal vertices are associated with the decision nodes of the BDD as indicated in the figure (right). The rate tree is built and maintained as a separate data structure from the BDD. The standard algorithms for BDDs (e.g. Apply) can be adapted and enhanced for DNBDDs such that the rate tree is manipulated simultaneously with the BDD, i.e. the rate tree is traversed (and possibly modified) in a recursive fashion by an appropriate extension of the procedures which manipulate the BDD data structure.

In our current prototypical implementation of DNBDDs, the rate tree is implemented in a rather straight-forward manner as an actual binary tree. This implementation has the drawback that it requires the explicit storage of one rate for each encoded transition which is basically against the grain of BDDs and may cause considerable overhead, especially in those cases where the same rate appears multiple times in the encoded SLTS. In order to avoid such redundancies, an efficient data structure to represent rate trees might itself be a decision diagram. However, this issue is currently still under investigation.



	(a, s, t)	*rate*
$0 \xrightarrow{a,\mu} 1$	$(0,0,1)$	μ
$1 \xrightarrow{a,\nu} 1$	$(0,1,1)$	ν
$0 \xrightarrow{b,\alpha} 1$	$(1,0,1)$	α
$1 \xrightarrow{b,\beta} 0$	$(1,1,0)$	β

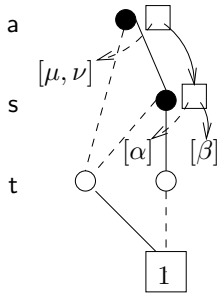


Figure 4.7: Simple SLTS, mapping of Boolean assignments to rates and corresponding DNBDD

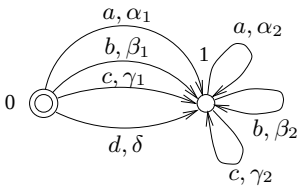
4.3.2 Representing SLTSs with the help of DNBDDs

The concept of representing an SLTS by a DNBDDs is illustrated in Fig. 4.7 (note that it is the same DNBDD as in Fig. 4.5). Since the SLTS has four transitions, there are four Boolean vectors mapped to a tuple of the form $(1, \lambda)$.

In order to formally define how an SLTS is represented by a DNBDD, in addition to the Boolean function f_x represented by a BDD vertex x (which can remain unchanged as defined before for BDDs), we now define a function *Num*, which, given a DNBDD and a Boolean assignment, computes the numerical result.

Definition 4.3.4 Numeric result value *Num*
Let r be the root vertex of DNBDD D *over Boolean variables* $\mathsf{v}_1, \ldots, \mathsf{v}_n$*, and let* $(b_1, \ldots, b_n)$ *be a fixed assignment to these variables. If* $f_r\Big|_{\mathsf{v}_1=b_1,\ldots,\mathsf{v}_n=b_n} = 0$ *then the function* $Num_r\Big|_{\mathsf{v}_1=b_1,\ldots,\mathsf{v}_n=b_n}$ *is undefined. Else let* $(x_1, \ldots, x_k) = \mathsf{path}(b_1, \ldots, b_n)$ *and* $\mathsf{rates}(x_1, \ldots, x_k) = [\lambda_1, \ldots, \lambda_{2^{n+1-k}}]$*. Then* $Num_r\Big|_{\mathsf{v}_1=b_1,\ldots,\mathsf{v}_n=b_n} = \lambda_i$*, where i is determined unambiguously by those positions of* $(b_1, \ldots, b_n)$ *which correspond to don't care variables.* ■

In other words, each of the 2^{n+1-k} Boolean assignments sharing path $(x_1, \ldots, x_k)$ corresponds to exactly one element of the rate list $[\lambda_1, \ldots, \lambda_{2^{n+1-k}}]$, and this correspondence is according to the lexicographical ordering of the Boolean assignments. We are now able to define how to represent a SLTS by a DNBDD:



	$(\mathsf{a}_1, \mathsf{a}_2, \mathsf{s}, \mathsf{t})$	*rate*
$\xrightarrow{a,\alpha_1}$	$(0,0,0,1)$	α_1
$\xrightarrow{b,\beta_1}$	$(0,1,0,1)$	β_1
$\xrightarrow{a,\alpha_2}$	$(0,0,1,1)$	α_2
$\xrightarrow{b,\beta_2}$	$(0,1,1,1)$	β_2
$\xrightarrow{c,\gamma_1}$	$(1,0,0,1)$	γ_1
$\xrightarrow{c,\gamma_2}$	$(1,0,1,1)$	γ_2
$\xrightarrow{d,\delta}$	$(1,1,0,1)$	δ

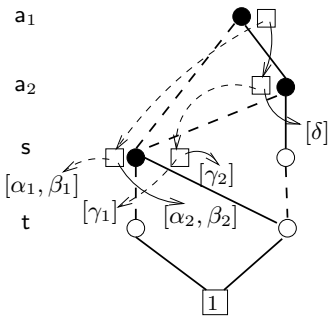


Figure 4.8: SLTS, mapping of Boolean assignments to rates and corresponding DNBDD

Definition 4.3.5 Symbolic Representation of a SLTS by a DNBDD
Let $\mathcal{T} = (S, L, \longrightarrow, s_1)$ *be a SLTS. Let* $\mathsf{D} = (Vert, \mathsf{var}, \mathsf{then}, \mathsf{else}, \mathsf{rates})$ *be a DNBDD with* $Vars = \{\mathsf{a}_1, \ldots, \mathsf{a}_{n_a}, \mathsf{s}_1, \ldots, \mathsf{s}_{n_s}, \mathsf{t}_1, \ldots, \mathsf{t}_{n_s}\}$*, such that* $n_a \geq \lceil \log_2 |L| \rceil$ *and* $n_s \geq \lceil \log_2 |S| \rceil$*. Let* r *be the root vertex of* D*. We say that* D *is a symbolic representation of* $\mathcal{T}$ *iff*

$$x \xrightarrow{a,\lambda} y \quad \Leftrightarrow \quad f_r\Big|_{(\mathsf{a}_1,\ldots,\mathsf{a}_{n_a})=\mathcal{E}(a),(\mathsf{s}_1,\ldots,\mathsf{s}_{n_s})=\mathcal{E}(x),(\mathsf{t}_1,\ldots,\mathsf{t}_{n_s})=\mathcal{E}(y)} = 1$$
$$\wedge \quad Num_r\Big|_{(\mathsf{a}_1,\ldots,\mathsf{a}_{n_a})=\mathcal{E}(a),(\mathsf{s}_1,\ldots,\mathsf{s}_{n_s})=\mathcal{E}(x),(\mathsf{t}_1,\ldots,\mathsf{t}_{n_s})=\mathcal{E}(y)} = \lambda$$

In this case, we write $\mathsf{D} \triangleright \mathcal{T}$ *for short.* ■

Note that again, the initial state of the SLTS is not encoded in the symbolic representation. Fig. 4.8 shows another example SLTS and its DNBDD representation. There are four different actions which are encoded by two Boolean variables (a_1 and a_2). In this example, the BDD contains five true-paths, two of which have a don't care in the Boolean variable a_2. Therefore, two rate lists have length two. The other three true-paths do not contain any don't cares, therefore the remaining three rate lists have length one.

4.4 Multi Terminal BDDs (MTBDD)

Multi Terminal BDDs (MTBDDs) are a graph-based representation of functions from a multidimensional Boolean domain to an arbitrary finite range $\mathbb{D}$, i.e. functions of the type $f : \mathbb{B}^n \to \mathbb{D}$. For instance, $\mathbb{D}$ can be a finite subset of the real numbers, or the set $\mathbb{B}$. (In the latter case the MTBDD reduces to a BDD,

representing a Boolean function.[2]) Thus, MTBDDs can be seen as an extension of BDDs such that they can represent functions with an arbitrary finite range.

MTBDDs [132] have been developed by Clarke et al. in the early 1990ies [86, 89, 131]. Among others, Hachtel et al. [17, 147], who use the term "algebraic decision diagrams (ADD)" for the same data structure, have made important contributions to this field.

The facts that MTBDDs are very well suited for matrix representation [86, 131], and that MTBDD-based algorithms for the manipulation and analysis of matrices are known, make this data structure very attractive to applications in the area of model-based performance evaluation. In particular, MTBDDs are capable of representing the matrices associated with discrete and continuous time Markov chains.

Similarly to BDDs, the main idea behind the MTBDD representation of $\mathbb{D}$-valued functions is the use of rooted directed acyclic graphs as a more compact representation of the binary decision tree which results from the Shannon expansion

$$f(\mathsf{v}_1, \ldots, \mathsf{v}_2) \;=\; \mathsf{if}\ \mathsf{v}_1\ \mathsf{then}\ f(1, \mathsf{v}_2, \ldots, \mathsf{v}_n)\ \mathsf{else}\ f(0, \mathsf{v}_2, \ldots, \mathsf{v}_n),$$

or, in terms of arithmetics, if the operations $+$ and $\cdot$ are defined on $\mathbb{D}$ and denote ordinary addition and multiplication,

$$f(\mathsf{v}_1, \ldots, \mathsf{v}_2) \;=\; \mathsf{v}_1 \cdot f(1, \mathsf{v}_2, \ldots, \mathsf{v}_n) \;+\; (1 - \mathsf{v}_1) \cdot f(0, \mathsf{v}_2, \ldots, \mathsf{v}_n).$$

Note that in this expression, either v_1 or $(1 - \mathsf{v}_1)$ is zero, while the other is one.

Definition 4.4.1 Multi Terminal Binary Decision Diagram (MTBDD)
An (ordered) Multi Terminal Binary Decision Diagram over $\langle Vars, \prec, \mathbb{D}\rangle$ *is a rooted directed acyclic graph* $\mathsf{M} = (Vert, \mathsf{var}, \mathsf{then}, \mathsf{else}, \mathsf{value})$ *defined by*

- *a finite set of vertices* $Vert = T \cup NT$*, where* T *(*NT*) is the set of terminal (non-terminal) vertices,* $|Vert| \geq 1$*,*
- *a function* $\mathsf{var} : NT \mapsto Vars$*, where* $Vars = \{\mathsf{v}_1, \ldots, \mathsf{v}_n\}$ *is a set of Boolean variables with a fixed ordering relation* $\prec \subset Vars \times Vars$*,*
- *a function* $\mathsf{then} : NT \mapsto Vert$ *and a function* $\mathsf{else} : NT \mapsto Vert$*,*
- *a function* $\mathsf{value} : T \mapsto \mathbb{D}$*,*

with the following constraints:
$\forall x \in NT : \mathsf{then}(x) \in T \vee \mathsf{var}(\mathsf{then}(x)) \succ \mathsf{var}(x)$
$\forall x \in NT : \mathsf{else}(x) \in T \vee \mathsf{var}(\mathsf{else}(x)) \succ \mathsf{var}(x)$ ■

[2]We also use the expression "0-1-MTBDD" for an MTBDD whose terminal vertices are from the set $\mathbb{B}$, i.e. for an MTBDD which is actually a BDD.

s_1	t_1	s_2	t_2	value
0	1	0	0	3
0	1	0	1	3
0	1	1	1	1
1	0	0	0	4
1	0	1	0	2
1	0	1	1	2
else				0

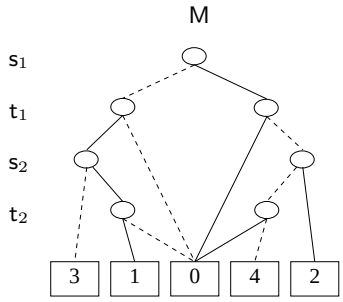


Figure 4.9: A function $f_{\mathsf{M}} : \mathbb{B}^4 \mapsto \{0, \ldots, 4\}$ and its representation by an MTBDD M

A BDD is an MTBDD with $\mathsf{value}(x) \in \mathbb{B}$ for all terminal vertices x. Each MTBDD M over $(\mathsf{v}_1, \ldots, \mathsf{v}_n)$ represents a function $f_{\mathsf{M}} : \mathbb{B}^n \mapsto \mathbb{D}$ and has two *cofactors* $\mathsf{M}^{\mathsf{then}}$ and $\mathsf{M}^{\mathsf{else}}$, resulting from a top-level Shannon expansion, i.e. $\mathsf{M}^{\mathsf{then}}$ ($\mathsf{M}^{\mathsf{else}}$) represents $f_{\mathsf{M}}(1, \mathsf{v}_2, \ldots, \mathsf{v}_n)$ ($f_{\mathsf{M}}(0, \mathsf{v}_2, \ldots, \mathsf{v}_n)$), respectively. Note that an MTBDD over $\langle Vars, \prec \rangle$ is also an MTBDD over $\langle Vars', \prec' \rangle$ for any superset $Vars'$ of $Vars$ and ordering $\prec'$ on $Vars'$ such that $\mathsf{v}_1 \prec \mathsf{v}_2$ iff $\mathsf{v}_1 \prec' \mathsf{v}_2$ for all v_1, $\mathsf{v}_2 \in Vars$. An MTBDD over $\langle \{\mathsf{v}_1, \ldots, \mathsf{v}_n\}, \prec \rangle$, where $\mathsf{v}_1 \prec \ldots \prec \mathsf{v}_n$, we also call an MTBDD over $(\mathsf{v}_1, \ldots, \mathsf{v}_n)$.

Fig. 4.9 shows a simple MTBDD M over $(\mathsf{s}_1, \mathsf{t}_1, \mathsf{s}_2, \mathsf{t}_2)$ together with the function f_{M} it represents. In the graphical representation, vertices are grouped into four levels, and all vertices on the same level are assumed to be labelled with the variable denoted on the left. Furthermore, we again adopt the convention that edges from vertex x to $\mathsf{then}(x)$ are drawn solid, while edges to $\mathsf{else}(x)$ are drawn dashed.

Definition 4.4.2 Reducedness of an MTBDD
A MTBDD M *is called reduced iff*

1. $\forall x \in NT : \mathsf{else}(x) \neq \mathsf{then}(x)$
2. $\forall x, y \in NT : \quad \mathsf{var}(x) \neq \mathsf{var}(y)$
   $\vee\ \mathsf{else}(x) \neq \mathsf{else}(y)$
   $\vee\ \mathsf{then}(x) \neq \mathsf{then}(y)$
3. $\forall x, y \in T : x \neq y \Rightarrow \mathsf{value}(x) \neq \mathsf{value}(y)$ ■

The first two conditions are identical with the ones given in Def. 2.4.2 for BDDs, and the third condition states that each terminal vertex x has a distinct label $\mathsf{value}(x)$. The recursive procedure proposed by Bryant [45] for reducing BDDs

can be applied to MTBDDs as well. From now on, unless otherwise stated, we will assume that we work with MTBDDs which are reduced.

Similarly to the Boolean function represented by a BDD vertex (cf. Def. 2.4.3) we now define the $\mathbb{D}$-valued function represented by an MTBDD vertex.

Definition 4.4.3 $\mathbb{D}$-valued function f_x represented by an MTBDD vertex
The $\mathbb{D}$-valued function f_x represented by a MTBDD vertex $x \in Vert$ is recursively defined as follows:

- *if $x \in T$ then $f_x = \mathsf{value}(x)$, i.e. an element from $\mathbb{D}$,*
- *else (if $x \in NT$)*
  $f_x = (1 - \mathsf{var}(x)) \cdot f_{\mathsf{else}(x)} + \mathsf{var}(x) \cdot f_{\mathsf{then}(x)}$ ■

Again, as in the BDD case, most times one is interested in the case where the vertex x corresponds to the MTBDD root. In that case we will write f_{M} instead of f_x, where x is the root vertex of MTBDD M.

For a fixed ordering of Boolean variables, reduced MTBDDs form a *canonical* representation of $\mathbb{D}$-valued functions, i.e. if M, M' are two reduced MTBDDs over the same ordered set $Vars$ such that $f_{\mathsf{M}} = f_{\mathsf{M}'}$, then M and M' are isomorphic.

MTBDD M of Fig. 4.9 satisfies Def. 4.4.2, it is *reduced*. Note that the valuations of some variable levels are irrelevant on certain paths through the MTBDD. For instance, for function f_{M} to return the values 3 or 2, the truth value of variable t_2 is irrelevant. Hence the vertices on these paths are skipped, a consequence of the first clause of Def. 4.4.2. Thus, variable t_2 is a don't-care variable for the respective paths.

4.4.1 Operations on MTBDDs

In this section, we describe how standard logical and arithmetic operations can be realised on MTBDDs. Let M, M_1, M_2 be reduced MTBDDs over $(\mathsf{v}_1, \ldots, \mathsf{v}_n)$. In what follows, we write $x_1 \prec x_2$ if either x_2 is a terminal vertex while x_1 is non-terminal or both x_1, x_2 are non-terminal vertices and $\mathsf{var}(x_1) \prec \mathsf{var}(x_2)$. From here on, unless otherwise stated, we assume that $\mathbb{D} = \mathbb{R}$.

Variable renaming, restriction (Restrict) and abstraction (Abstract) work similarly as on BDDs. The operator op used in abstraction can be any associative binary operator defined on $\mathbb{R}$ (such as addition $+$, or multiplication $\cdot$).

Converting an MTBDD to a BDD: The function MTBDDtoBDD converts an MTBDD $\mathsf{M}(\mathsf{v}_1, \ldots, \mathsf{v}_n)$ to a BDD $\mathsf{B}(\mathsf{v}_1, \ldots, \mathsf{v}_n)$ by abstracting from the numerical information. The resulting BDD is obtained by replacing the value of every nonzero terminal vertex of M by the value 1 and afterwards reducing the decision diagram, for instance by applying Bryant's reduction algorithm.

Maximum terminal value: Let $\{x_1, \ldots, x_k\}$ be the terminal vertices of M. Maxval(M) is the MTBDD consisting of a single terminal vertex labelled with $\max_{1 \leq i \leq k}\{\ |\mathsf{value}(x_i)|\ \}$, the maximum absolute value of the function f_{M}. This requires a simple traversal of the terminal vertices of M.

Combining two MTBDDs via binary arithmetic operators: If op is a binary operator (e.g. addition $+$, or multiplication $\cdot$) then Apply($\mathsf{M}_1, \mathsf{M}_2,$ op) returns the MTBDD M over $(\mathsf{v}_1, \ldots, \mathsf{v}_n)$ where $f_{\mathsf{M}} = f_{\mathsf{M}_1}$ op f_{M_2}. The algorithm behind this operator is exactly as in the BDD case, see Section 2.4.2, i.e. Apply($\mathsf{M}_1, \mathsf{M}_2,$ op) calls a recursive procedure $A_{\mathrm{OP}}(x_1, x_2)$ which works according to the following rules:

- If x_1 and x_2 are terminal vertices then $A_{\mathrm{OP}}(x_1, x_2)$ returns just the single terminal vertex x labelled by $\mathsf{value}(x_1)$ op $\mathsf{value}(x_2)$.
- If x_1, x_2 are non-terminal vertices and $\mathsf{var}(x_1) = \mathsf{var}(x_2) = \mathsf{v}$ then $\mathsf{var}(x) = \mathsf{v}$, $\mathsf{else}(x) = A_{\mathrm{OP}}(\mathsf{else}(x_1), \mathsf{else}(x_2))$ and $\mathsf{then}(x) = A_{\mathrm{OP}}(\mathsf{then}(x_1), \mathsf{then}(x_2))$.
- If $x_1 \prec x_2$ then $\mathsf{var}(x) = \mathsf{var}(x_1)$, $\mathsf{else}(x) = A_{\mathrm{OP}}(\mathsf{else}(x_1), x_2)$ and $\mathsf{then}(x) = A_{\mathrm{OP}}(\mathsf{then}(x_1), x_2)$.
  Conversely, if $x_2 \prec x_1$ then $\mathsf{var}(x) = \mathsf{var}(x_2)$, $\mathsf{else}(x) = A_{\mathrm{OP}}(x_1, \mathsf{else}(x_2))$ and $\mathsf{then}(x) = A_{\mathrm{OP}}(x_1, \mathsf{then}(x_2))$.

As in the BDD case, the algorithm may check for the presence of special "controlling" values which can receive special treatment and thereby avoid the initiation of recursive calls. For instance, if op is multiplication and x_1 is a terminal vertex with $\mathsf{value}(x_1) = 1$, then x_2 can be immediately returned as the result.

As in the BDD case, the Apply algorithm uses a "unique table" which contains all currently existing MTBDD vertices. A unique table entry for a terminal vertex now consists of the vertex identifier and the vertex's value labelling $\mathsf{value}(x)$. Proper use of the unique table again ensures that the MTBDD resulting from an Apply-operation is in reduced form. The use of the "computed table" is also similar to the BDD case.

Scalar multiplication: If T just consists of a terminal vertex x labelled with $\mathsf{value}(x)$, then Smult(M, T) returns the unique reduced MTBDD over $(\mathsf{v}_1, \ldots, \mathsf{v}_n)$ representing the function $\mathsf{value}(x) \cdot f_{\mathsf{M}}$. Even though this operation could be easily

realised using APPLY, it may be more efficient to implement as an update of the terminal vertices y of M (provided the particular implementation at hand enables direct access to the terminal vertices), changing the value of each terminal vertex y into $\mathsf{value}(x) \cdot \mathsf{value}(y)$.

Matrix multiplication: We first describe how (real-valued) matrices are represented by MTBDDs. For simplicity we assume square matrices whose size is a power of 2, i.e. 2^n. Rectangular matrices of general sizes can be represented with the same basic scheme by padding them with an appropriate number of columns and rows of zeroes. By construction, these additional entries do not contribute to the size of the MTBDD representing the matrix. A $2^n \times 2^n$ matrix M can be seen as a function from $\{0, \ldots, 2^n - 1\} \times \{0, \ldots, 2^n - 1\}$ to $\mathbb{R}$. If the row position r is encoded by Boolean variables r_i and the column position c by Boolean variables c_i (where in both cases $i = 1, \ldots, n$) then the MTBDD M over $\langle Vars, \prec \rangle$ where $f_{\mathsf{M}}(\mathsf{r}_1, \ldots, \mathsf{r}_n, \mathsf{c}_1, \ldots, \mathsf{c}_n) = M(r, c)$ is a canonical representation of matrix M

Concerning the variable ordering, it turns out that an interleaving of the Boolean variables encoding row and column position, i.e. the ordering $\mathsf{r}_1 \prec \mathsf{c}_1 \prec \ldots \prec \mathsf{r}_n \prec \mathsf{c}_n$, is usually the best choice for the following reasons.

- The cofactors of the MTBDD correspond to block submatrices of the matrix. For instance, $\mathsf{M}^{\mathsf{else}}$ corresponds to the upper half of matrix M, $\mathsf{M}^{\mathsf{then}^{\mathsf{else}}}$ (to be read as $(\mathsf{M}^{\mathsf{then}})^{\mathsf{else}}$) corresponds to the lower left quadrant of matrix M, etc..
- The identity matrix, corresponding to the function $f_{\mathsf{Id}} = \prod_{k=1}^{n} (\mathsf{r}_k \equiv \mathsf{c}_k)$, which is nothing else but the matrix equivalent of the function f_{Stab} introduced in Sec. 4.1, can be represented in a number of vertices which is logarithmic in the size of the matrix. More precisely, the number of vertices needed to represent an identity matrix of size $2^n \times 2^n$ is $3n + 2$. In contrast, using the straightforward ordering $\mathsf{r}_1 \prec \ldots \prec \mathsf{r}_n \prec \mathsf{c}_1 \prec \ldots \prec \mathsf{c}_n$ the size of the MTBDD would be $3 \cdot 2^n - 1$ vertices. The MTBDDs representing function f_{Id} under the two variable orderings are similar to the BDDs shown in Fig. 4.3 (for the case $n = 3$). Since identity matrices play an important role during the parallel composition of transition systems (see Section 5.1), their compact representation is an essential feature of MTBDDs.

As an example, consider the simple CTMC $\mathcal{C} = (\{0, 1, 2, 3\}, R)$ with transition rate matrix R as shown on the left of Fig. 4.10. Since the size of this matrix is $2^n = 4$, we need $n = 2$ bits for addressing its rows and 2 bits for addressing its columns. We use Boolean variables $\mathsf{r}_1, \mathsf{r}_2$, $(\mathsf{c}_1, \mathsf{c}_2)$ for encoding the row (column) position.

Based on this encoding scheme for matrices, we now discuss a (naïve) realisation of

$$R = \begin{pmatrix} - & 4 & - & - \\ 3 & - & 3 & - \\ - & 2 & - & 2 \\ - & - & 1 & - \end{pmatrix}$$

transition	r_1	c_1	r_2	c_2	value
$0 \xrightarrow{4} 1$	0	0	0	1	4
$1 \xrightarrow{3} 0$	0	0	1	0	3
$1 \xrightarrow{3} 2$	0	1	1	0	3
$2 \xrightarrow{2} 1$	1	0	0	1	2
$2 \xrightarrow{2} 3$	1	1	0	1	2
$3 \xrightarrow{1} 2$	1	1	1	0	1
		else			0

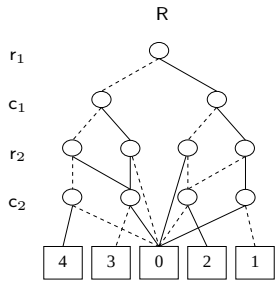


Figure 4.10: Rate matrix R, transition encoding and MTBDD R for a simple CTMC

matrix multiplication. Consider two square matrices M_1 and M_2, represented as MTBDDs M_1 and M_2 over variables $(\mathsf{r}_1, \mathsf{c}_1 \ldots, \mathsf{r}_n, \mathsf{c}_n)$ and $(\mathsf{c}_1, \mathsf{c}'_1 \ldots, \mathsf{c}_n, \mathsf{c}'_n)$. Let us for the moment assume that M_1 and M_2 are not fully reduced, since this assumption makes the algorithm much more straight-forward. In particular, we assume that don't care variables are not skipped but explicitly present. $\textsc{Mmult}(\mathsf{M}_1, \mathsf{M}_2)$ produces an MTBDD M over $(\mathsf{r}_1, \mathsf{c}'_1 \ldots, \mathsf{r}_n, \mathsf{c}'_n)$, representing the matrix product $M = M_1 \cdot M_2$. This MTBDD is generated by recursive descent. The four quadrants of M corresponding to the cofactors $\mathsf{M}^{\mathsf{else}\,\mathsf{else}}$, $\mathsf{M}^{\mathsf{else}\,\mathsf{then}}$, $\mathsf{M}^{\mathsf{then}\,\mathsf{else}}$, and $\mathsf{M}^{\mathsf{then}\,\mathsf{then}}$ are computed on the basis of the cofactors of M_1 and M_2. For instance:

$$\mathsf{M}^{\mathsf{else}\,\mathsf{else}} = \textsc{Apply}\left(\textsc{Mmult}(\mathsf{M}_1^{\mathsf{else}\,\mathsf{else}}, \mathsf{M}_2^{\mathsf{else}\,\mathsf{else}}), \textsc{Mmult}(\mathsf{M}_1^{\mathsf{else}\,\mathsf{then}}, \mathsf{M}_2^{\mathsf{then}\,\mathsf{else}}), +\right)$$

is the MTBDD reformulation of the fact that the upper left quadrant of $M_1 \cdot M_2$ equals the sum of (1) the product of the upper left quadrants of M_1 and M_2, and (2) the product of the upper right quadrant of M_1 and the lower left quadrant of M_2. The products $\textsc{Mmult}(\mathsf{M}_1^{\mathsf{else}\,\mathsf{else}}, \mathsf{M}_2^{\mathsf{else}\,\mathsf{else}})$ and $\textsc{Mmult}(\mathsf{M}_1^{\mathsf{else}\,\mathsf{then}}, \mathsf{M}_2^{\mathsf{then}\,\mathsf{else}})$ are recursively computed in the same way. The recursion terminates when the operands of Mmult are terminal vertices x_1 and x_2, in which case a terminal vertex labelled by $\mathsf{value}(x_1) \cdot \mathsf{value}(x_2)$ is returned.

Vector-matrix (and matrix-vector) multiplication VMmult (MVmult) can be performed by the same basic strategy. If M_1 is as above, and P over variables $(\mathsf{r}_1, \ldots, \mathsf{r}_n)$ represents a row vector $\vec{p}$, then $\textsc{VMmult}(\mathsf{P}, \mathsf{M}_1)$ computes an MTBDD Q over variables $(\mathsf{c}_1, \ldots, \mathsf{c}_n)$ representing $\vec{q} = \vec{p} \cdot M_1$ by means of the cofactors of its arguments. The left half of the vector $\vec{q}$ is, for instance, obtained from

$$\mathsf{Q}^{\mathsf{else}} = \textsc{Apply}\left(\textsc{VMmult}(\mathsf{P}^{\mathsf{else}}, \mathsf{M}_1^{\mathsf{else}\,\mathsf{else}}), \textsc{VMmult}(\mathsf{P}^{\mathsf{then}}, \mathsf{M}_1^{\mathsf{then}\,\mathsf{else}}), +\right)$$

Note that Q does not depend on the same variables as P, since the common variables, $\vec{\mathsf{r}}$, are abstracted from during multiplication. Nevertheless, Q and P both represent row vectors, as is obvious from standard linear algebra.

$$M_1 = \left[\begin{array}{cc|cc} 2 & 2 & 3 & 3 \\ 2 & 2 & 3 & 3 \\ \hline 4 & 4 & 5 & 5 \\ 4 & 4 & 5 & 5 \end{array}\right] \quad M_2 = \left[\begin{array}{cc|cc} 6 & 6 & 7 & 7 \\ 6 & 6 & 7 & 7 \\ \hline 8 & 8 & 9 & 9 \\ 8 & 8 & 9 & 9 \end{array}\right] \quad M = M_1 \cdot M_2 = \left[\begin{array}{cc|cc} 72 & 72 & 82 & 82 \\ 72 & 72 & 82 & 82 \\ \hline 128 & 128 & 146 & 146 \\ 128 & 128 & 146 & 146 \end{array}\right]$$

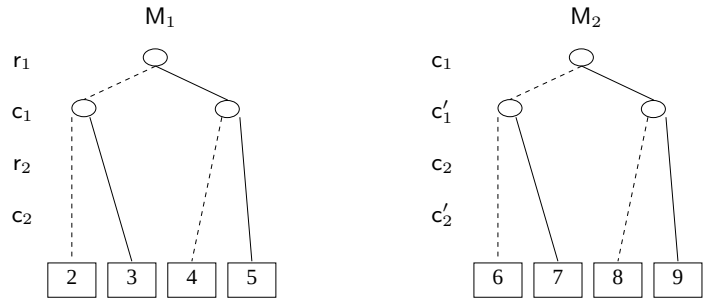


Figure 4.11: Example where the naïve matrix multiplication algorithm produces wrong results

This naïve approach to matrix multiplication is not sufficient (it does not work correctly) if M_1 and M_2 are reduced MTBDDs where variable levels may be skipped (as is the case, for example, when representing regularly structured matrices with repeated submatrices). As a very simple example for such a situation consider the multiplication of the two matrices shown in Fig. 4.11. Their MTBDD representations are also shown in the figure. Due to the fact that the same entry is repeated within each 2×2 block, variables r_2 and c_2 (c_2 and c'_2) are skipped in the first (second) MTBDD, i.e. these are don't care variables. If the above recursive scheme was applied without modification, all entries of the resulting matrix would be too small by a factor of 2, i.e. a scaling factor of 2 would be missing in all the entries of the result matrix. For instance, $\mathsf{M}^{\mathsf{else}^{\mathsf{else}}}$ would be computed as $2 \cdot 6 + 3 \cdot 8 = 36$ instead of $2 \cdot 6 + 2 \cdot 6 + 3 \cdot 8 + 3 \cdot 8 = 72$. Note that the smaller size matrices $M'_1 = \left[\begin{array}{cc} 2 & 3 \\ 4 & 5 \end{array}\right]$ and $M'_2 = \left[\begin{array}{cc} 6 & 7 \\ 8 & 9 \end{array}\right]$ are represented by the same MTBDDs as M_1 and M_2 (without depending on the above don't care variables) and their multiplication by the described basic scheme would of course yield a correct result.

The literature on MTBDDs describes matrix multiplication algorithms which overcome this shortcoming of the given simple scheme. Their implementations work on reduced MTBDDs and return MTBDDs that are reduced (and hence canonical) by construction. The general idea is to pass additional integer parameters to function MMULT (MVMULT, VMMULT), basically to take care of variable levels that are skipped (don't care variables) .

An early algorithm of Clarke et al. [89] (commonly referred to as the CMU method) uses the operation APPLY and an "ABSTRACT" operation to symbolically calculate the matrix product in a rather straight-forward fashion. It separates multiplication and summation and therefore requires two recursive descents: One through the operand MTBDDs, and one through an MTBDD which represents an intermediate result (the latter MTBDD may become very large).

Another algorithm of Clarke et al. [86] (resp. Fujita et al. [131]) (called the Berkeley method) is based on a rigorous splitting on all Boolean variables, even if one (or both) of the operands does not depend on certain variables. It uses an integer variable to keep track of the set of Boolean variables that is used at the current level of recursion.

The algorithm of Bahar et al. [18] (called the Boulder method) aims at minimising the number of microoperations needed and is probably the most sophisticated one published up to now. It keeps track of missing variables in the two operands by computing a scaling factor (a power of 2) which depends on the current level of recursion and the current 'top-level" variable. This algorithm highly relies on the computed table for bookkeeping of functions whose MTBDDs are already computed, thereby avoiding the recomputation of the same arithmetic or logic operation with the same arguments (even if the scaling factor had previously been different). When the terminal case is encountered or when the result is found in the computed table, the result value is multiplied by the appropriate scaling factor before it is returned.

Inversion of triangular matrices: Inversion of triangular matrices, denoted INVTRI, can be performed on their MTBDD-based representation by a recursive algorithm similar to matrix multiplication, as described in [86]. Let U be an upper diagonal matrix, represented by an MTBDD U. Again, for simplicity, let us assume that U is not fully reduced. INVTRI(U) produces an MTBDD V representing the inverse $V = U^{-1}$. This MTBDD is also created by recursive descent. From the following decomposition of the matrices into quadrants

$$\begin{pmatrix} U_{00} & U_{01} \\ 0 & U_{11} \end{pmatrix} \cdot \begin{pmatrix} V_{00} & V_{01} \\ V_{10} & V_{11} \end{pmatrix} = \begin{pmatrix} I & 0 \\ 0 & I \end{pmatrix}$$

it follows that

$$\begin{aligned} U_{00}V_{00} + U_{01}V_{10} &= I \\ U_{00}V_{01} + U_{01}V_{11} &= 0 \\ U_{11}V_{10} &= 0 \\ U_{11}V_{11} &= I \end{aligned}$$

From these four equations one can easily derive that

$$\begin{array}{lcl} V_{10} & = & 0 \\ V_{11} & = & U_{11}^{-1} \\ V_{00} & = & U_{00}^{-1} \\ V_{01} & = & -V_{00}U_{01}V_{11} \end{array}$$

which translates into the following MTBDD operations

$$\begin{array}{lcl} \mathsf{V}^{\mathsf{then}\,\mathsf{else}} & = & 0 \\ \mathsf{V}^{\mathsf{then}\,\mathsf{then}} & = & \textsc{Invtri}(\mathsf{U}^{\mathsf{then}\,\mathsf{then}}) \\ \mathsf{V}^{\mathsf{else}\,\mathsf{else}} & = & \textsc{Invtri}(\mathsf{U}^{\mathsf{else}\,\mathsf{else}}) \\ \mathsf{V}^{\mathsf{else}\,\mathsf{then}} & = & \textsc{Smult}(\textsc{Mmult}(\mathsf{V}^{\mathsf{else}\,\mathsf{else}}, \textsc{Mmult}(\mathsf{U}^{\mathsf{else}\,\mathsf{then}}, \mathsf{V}^{\mathsf{then}\,\mathsf{then}})), -1) \end{array}$$

In the algorithm, the smaller size inverses $\textsc{Invtri}(\mathsf{U}^{\mathsf{then}\,\mathsf{then}})$ and $\textsc{Invtri}(\mathsf{U}^{\mathsf{else}\,\mathsf{else}})$ are recursively computed in the same way. The recursion terminates when the operand is a terminal vertex x, in which case $\mathsf{value}(x)^{-1}$ is returned.

Matrix diagonal: Let M be an MTBDD over $(\mathsf{v}_1, \ldots, \mathsf{v}_n)$. For Boolean variables $\mathsf{v}'_1, \ldots, \mathsf{v}'_n$, $\textsc{Diag}(\mathsf{M}(\vec{\mathsf{v}}), \vec{\mathsf{v}}')$ is the MTBDD over $(\mathsf{v}_1, \mathsf{v}'_1, \ldots, \mathsf{v}_n, \mathsf{v}'_n)$ representing f_{M} if $\mathsf{v}_i = \mathsf{v}'_i$ for $1 \leq i \leq n$ and 0 otherwise. So, it turns a vector into a diagonal matrix of the same size. For non-reduced MTBDDs, where don't care variables are not skipped, the algorithm for Diag takes a vertex x of M and introduces new vertices x_1 and x_2 with $\mathsf{var}(x_1) = \mathsf{var}(x_2) = \mathsf{var}(x)'$, $\mathsf{else}(x_1) = \mathsf{else}(x)$, $\mathsf{then}(x_2) = \mathsf{then}(x)$, and $\mathsf{then}(x_1) = \mathsf{else}(x_2) = 0$. Afterwards it sets $\mathsf{else}(x) = x_1$ and $\mathsf{then}(x) = x_2$, and (recursively) proceeds by taking the vertices $\mathsf{else}(x_1)$ and $\mathsf{then}(x_2)$. For reduced MTBDDs the same algorithm may be used, but the intermediate result thus obtained needs to be multiplied with the MTBDD Id representing the identity matrix, in order to obtain the correct final result.

Inversion of a diagonal matrix: A diagonal matrix is inverted by individually inverting its elements (which, of course, must be all non-zero). For an MTBDD M representing a diagonal matrix, $\textsc{InvDiag}(\mathsf{M})$ returns an MTBDD representing the inverse of the matrix. The MTBDD operation InvDiag is implemented by means of a single update of each terminal vertex x of the argument MTBDD, replacing each non-zero $\mathsf{value}(x)$ by $\mathsf{value}(x)^{-1}$.

4.4.2 Representing SLTSs with the help of MTBDDs

In this section, we discuss how SLTS and ESLTS can be encoded as MTBDDs.

The encoding of non-stochastic transition systems has been already described in Sec. 4.1, and the encoding of matrices has already been described in Sec. 4.4.1. These techniques are now combined.

We use the encoding function $\mathcal{E}(.)$ to encode states and action labels. Each transition $s \xrightarrow{a,\lambda} t$ of the SLTS is mapped to a Boolean vector of length $n_a + 2n_s$, whose positions correspond to Boolean variables $\mathsf{a}_1, \ldots, \mathsf{a}_{n_a}, \mathsf{s}_1, \ldots, \mathsf{s}_{n_s}, \mathsf{t}_1, \ldots, \mathsf{t}_{n_s}$. This vector encodes the action label, the source and the target state of the transition, but not the transition rate. The transition rates are stored in the terminal vertices of the MTBDD. The following definition states in which way an SLTS is represented by an MTBDD.

Definition 4.4.4 Symbolic Representation of an SLTS by an MTBDD
Let $\mathcal{T} = (S, L, \longrightarrow, s)$ *be a SLTS. Let* $\mathsf{M} = (Vert, \mathsf{var}, \mathsf{then}, \mathsf{else}, \mathsf{value})$ *be an MTBDD with* $Vars = \{\mathsf{a}_1, \ldots, \mathsf{a}_{n_a}, \mathsf{s}_1, \ldots, \mathsf{s}_{n_s}, \mathsf{t}_1, \ldots, \mathsf{t}_{n_s}\}$*, such that* $n_a \geq \lceil \log_2 |L| \rceil$ *and* $n_s \geq \lceil \log_2 |S| \rceil$*. We say that* M *is the symbolic representation of* $\mathcal{T}$ *iff*

$$s \xrightarrow{a,\lambda} t \;\Leftrightarrow\; f_{\mathsf{M}}\Big|_{(\mathsf{a}_1,\ldots,\mathsf{a}_{n_a})=\mathcal{E}(a),(\mathsf{s}_1,\ldots,\mathsf{s}_{n_s})=\mathcal{E}(s),(\mathsf{t}_1,\ldots,\mathsf{t}_{n_s})=\mathcal{E}(t)} = \lambda$$

In this case, we write $\mathsf{M} \triangleright \mathcal{T}$ *for short.* ■

Note that again, the initial state of SLTS $\mathcal{T}$ is not encoded by the MTBDD. Fig. 4.12 depicts two example SLTSs represented by MTBDDs. The example at the top of the figure is the same as in the section on DNBDDs (cf. Fig 4.8). It shows clearly the advantages of DNBDDs, namely their feature of optimal subgraph sharing in the case where there are many different rates. Obviously, MTBDDs do not allow a compact encoding if all transitions of the SLTS carry different rate labels, since this causes a large number of terminal vertices and allows only little sharing of subgraphs. The example at the bottom of Fig. 4.12 shows the *Queue*-process from Fig. 4.2 enhanced with rate information. MTBDDs are obviously well-suited for this example, since it has a regular structure and only two different rates are involved, leading to two terminal vertices.

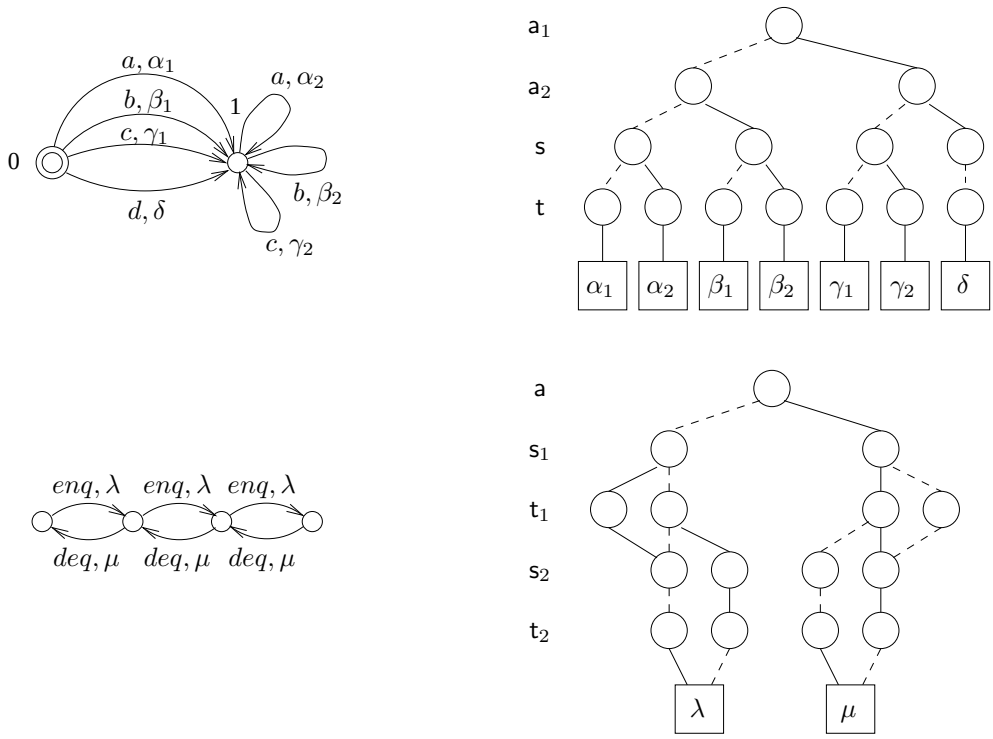


Figure 4.12: SLTSs and corresponding MTBDDs

4.4.3 Representing ESLTSs with the help of MTBDDs and BDDs

In this section, we discuss how ESLTSs can be represented symbolically. Remember that an ESLTS contains two transition relations: Immediate (action) transitions, denoted by $\overset{a}{\dashrightarrow}$, and Markovian transitions, denoted by $\xrightarrow{a,\lambda}$. Basically, one has two options when representing an ESLTS $\mathcal{T}$.

1. The first option is to employ two separate data structures, i.e. a BDD B which encodes all immediate transitions, and an MTBDD M which encodes all Markovian transitions. In this case we write $(\mathsf{B}, \mathsf{M}) \triangleright \mathcal{T}$, i.e. the ESLTS is represented by a pair, consisting of a BDD and an MTBDD.

   In certain situations, one may wish to associate immediate transitions with numerical values different from 1, for instance in order to associate immediate transitions with weights or probabilities. This feature may be needed when resolving non-determinism between several concurrently enabled internal immediate transitions, cf. Sec. 5.2.2. To support this feature, an MTBDD M^I can be employed instead of BDD B for representing imme-

diate transitions, while Markovian transitions are still represented by an MTBDD as before, but now called M^M. In this case we write $(\mathsf{M}^I, \mathsf{M}^M) \triangleright \mathcal{T}$.

2. The second option is to work with a single MTBDD M which comprises Markovian as well as immediate transitions. If this option is chosen, it must be ensured that the two types of transitions can be properly distinguished. The potential problem is the following: The terminal vertex with value 1 would represent both a logical one and the numerical rate value 1, i.e. a Markovian transition with rate 1 would be indistinguishable from an immediate transition. One solution in order to avoid such confusion is to store the rates of Markovian transitions as negative real values[3]. Using this option, immediate transitions can also be easily associated with values different from 1, for instance to associate immediate transitions with probabilities.

Figure 4.13 contains a very simple ESLTS example and its encoding following the two options just mentioned. While the second option allows additional sharing of the vertices of the decision diagram, the first option is conceptually simpler, since there is no "mixing" of immediate and Markovian transitions within the same decision diagram. For this reason, in the implementation of the tool IM-CAT [128, 129], we followed the first approach[4].

For the sake of completeness, we now provide the formal definitions for both alternatives. The first alternative leads to the following definition:

Definition 4.4.5 Symbolic Representation of an ESLTS by a BDD and an MTBDD
Let $\mathcal{T} = (S, L, \dashrightarrow, \longrightarrow, s)$ *be an ESLTS. Let* $\mathsf{B} = (Vert, \mathsf{var}, \mathsf{then}, \mathsf{else})$ *be a BDD. Let* $\mathsf{M} = (Vert, \mathsf{var}, \mathsf{then}, \mathsf{else}, \mathsf{value})$ *be an MTBDD. Both* B *and* M *are decision diagrams over* $\langle Vars, \prec \rangle = \langle \{\mathsf{a}_1, \ldots, \mathsf{a}_{n_a}, \mathsf{s}_1, \ldots, \mathsf{s}_{n_s}, \mathsf{t}_1, \ldots, \mathsf{t}_{n_s}\}, \prec \rangle$*, where* n_a *and* n_s *must be chosen such that* $n_a \geq \lceil \log_2 |L| \rceil$ *and* $n_s \geq \lceil \log_2 |S| \rceil$*. We say that the tuple* (B, M) *is the symbolic representation of* $\mathcal{T}$ *iff*

1. $s \overset{a}{\dashrightarrow} t \;\Leftrightarrow\; f_{\mathsf{B}}\Big|_{(\mathsf{a}_1,\ldots,\mathsf{a}_{n_a})=\mathcal{E}(a),(\mathsf{s}_1,\ldots,\mathsf{s}_{n_s})=\mathcal{E}(s),(\mathsf{t}_1,\ldots,\mathsf{t}_{n_s})=\mathcal{E}(t)} = 1$
2. $s \overset{a,\lambda}{\longrightarrow} t \;\Leftrightarrow\; f_{\mathsf{M}}\Big|_{(\mathsf{a}_1,\ldots,\mathsf{a}_{n_a})=\mathcal{E}(a),(\mathsf{s}_1,\ldots,\mathsf{s}_{n_s})=\mathcal{E}(s),(\mathsf{t}_1,\ldots,\mathsf{t}_{n_s})=\mathcal{E}(t)} = \lambda$

In this case, we write $(\mathsf{B}, \mathsf{M}) \triangleright \mathcal{T}$ *for short.* ■

[3] Another solution could be to use a special value, e.g. ∞ or MAXFLOAT, in the terminal vertex representing the immediate transitions. We specifically do *not* choose this option, since it would implicitly assign equal probabilities to simultaneously enabled internal immediate transitions, while the intended semantics is that of a non-deterministic choice.

[4] IM-CAT does indeed use an MTBDD (and not a BDD) for immediate transitions because it can associate a probability with each immediate transition.

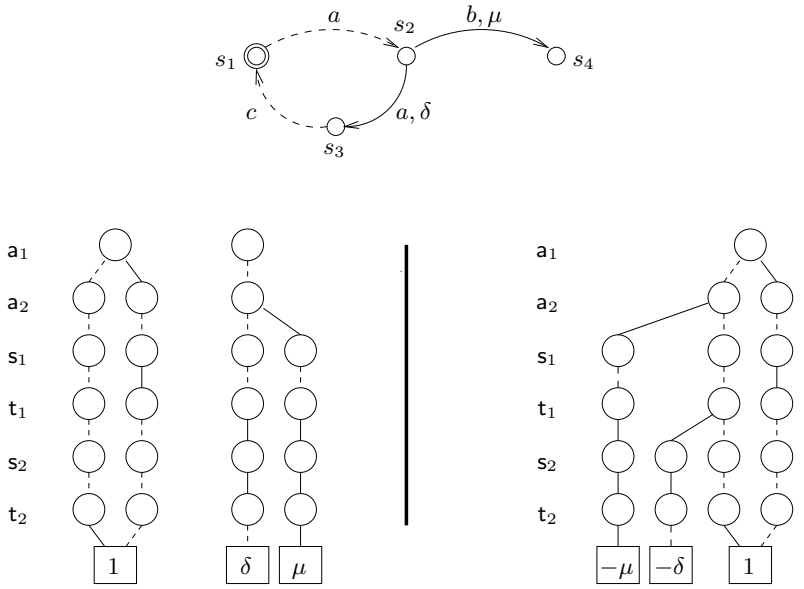


Figure 4.13: Encoding of an example ESLTS. Left: separation of immediate and Markovian transitions. Right: encoding by a single MTBDD.

The second alternative leads to the following definition:

Definition 4.4.6 Symbolic Representation of an ESLTS by a single MTBDD
Let $\mathcal{T} = (S, L, \dashrightarrow, \longrightarrow, s)$ *be an ESLTS. Let* $\mathsf{M} = (Vert, \mathsf{var}, \mathsf{then}, \mathsf{else}, \mathsf{value})$ *be an MTBDD with* $Vars = \{\mathsf{a}_1, \ldots, \mathsf{a}_{n_a}, \mathsf{s}_1, \ldots, \mathsf{s}_{n_s}, \mathsf{t}_1, \ldots, \mathsf{t}_{n_s}\}$*, such that* $n_a \geq \lceil \log_2 |L| \rceil$ *and* $n_s \geq \lceil \log_2 |S| \rceil$*. We say that* M *is the symbolic representation of* $\mathcal{T}$ *iff*

1. $s \overset{a}{\dashrightarrow} t \;\Leftrightarrow\; f_{\mathsf{M}}\Big|_{(\mathsf{a}_1,\ldots,\mathsf{a}_{n_a})=\mathcal{E}(a),(\mathsf{s}_1,\ldots,\mathsf{s}_{n_s})=\mathcal{E}(s),(\mathsf{t}_1,\ldots,\mathsf{t}_{n_s})=\mathcal{E}(t)} = 1$
2. $s \xrightarrow{a,\lambda} t \;\Leftrightarrow\; f_{\mathsf{M}}\Big|_{(\mathsf{a}_1,\ldots,\mathsf{a}_{n_a})=\mathcal{E}(a),(\mathsf{s}_1,\ldots,\mathsf{s}_{n_s})=\mathcal{E}(s),(\mathsf{t}_1,\ldots,\mathsf{t}_{n_s})=\mathcal{E}(t)} = -\lambda$

In this case, we write $\mathsf{M} \triangleright \mathcal{T}$ *for short.* ■

4.5 Complexity considerations

When working with decision diagrams, it is very difficult to predict the size of the resulting data structures or the computation time needed to generate or manipulate them. As we shall see for instance in Chap. 6, some large problem instance may be encoded as a surprisingly small BDD, while another, almost identical problem leads to a much larger BDD. Similar observations can be made concerning BDD construction times and computation times: In Chap. 10, for instance, we will see that the construction times for the MTBDDs representing iteration matrices vary considerably, even if the resulting MTBDDs are of about the same size. Furthermore, for two matrices represented by MTBDDs of almost the same BDD size, matrix-vector multiplication with one may take much longer than with the other.

We also stress the important point that the size of the decision diagram resulting from some computation is only part of the picture. One should always be aware of the fact that an intermediate BDD that occurs in the course of the computation may be larger by orders of magnitude, which can be a crucial factor for both memory and time.

When encoding a transition system, the size of the resulting BDD, DNBDD or MTBDD depends on many different factors, such as the size and the structure of the transition system, the chosen encoding of states and actions and the ordering of the Boolean variables. In the best case, the decision diagram has constant size (it may even consist of a single vertex), as we observed for the MTBDD-based encoding of regularly structured matrices in Sec. 4.4.1. In the worst case, the size of the decision diagram will be exponential in n, where n is the number of Boolean variables, as we observed for the encoding of the identity matrix with non-interleaved variable ordering. As we shall see in Chap. 5, the size of a decision diagram becomes predictable if it is built in a structured way from components whose size is already known. We shall prove that the size of the decision diagram resulting from the composition of two components is linear in the sizes of the components.

The cost of BDD operations is of course dependent of the size of the BDD. Bryant stated the following important complexity result for BDDs [45]: The worst case time complexity of $\textsc{Apply}(\mathsf{B}_1, \mathsf{B}_2, \textsc{op})$ is $O(|\mathsf{B}_1| \cdot |\mathsf{B}_2|)$, where $|\mathsf{B}_i|$ is the number of vertices of BDD B_i. Bryant argues that no Apply algorithm can have better worst case time complexity, since there exist cases where the resulting BDD actually has $O(|\mathsf{B}_1| \cdot |\mathsf{B}_2|)$ vertices. It can be shown that this worst case time complexity carries over to DNBDDs and MTBDDs.

We shall see in Chaps. 8 and 10 that MTBDD-based matrix multiplication and vector-matrix multiplication is a serious bottleneck for symbolic numerical analysis. Basically, MTBDD-based matrix multiplication algorithms have the same time complexity as their conventional sparse-matrix counterparts which is cubic[5] in the size of the matrix. However, we are not aware of any complexity analysis of these symbolic algorithms that is based on the number of vertices of the MTBDDs by which the matrices involved in the multiplication are represented.

[5] A slight improvement over cubic complexity is possible by Strassen's algorithm [315].

Chapter 5

Working with symbolic representations

In this chapter, we describe how complex models can be built from small components in a stepwise fashion, based on the symbolic model representations introduced in Chap. 4. Since the composition of submodels may lead to unreachable states, we also discuss symbolic reachability analysis. Furthermore, we address special issues concerning the combination of Markovian and immediate transitions: Apart from parallel composition of ESLTSs, we discuss symbolic techniques for the hiding of actions and for the elimination of compositionally vanishing states in such transition systems. In addition to the composition of submodels and hiding, we address symbolic techniques for state space minimisation which are based on bisimulation equivalences between states.

5.1 Compositional state space construction

In this section, we describe how BDDs are constructed from a given LTS and how the parallel composition of components can be carried out at the level of their BDD-based representation. We observe that this composition can be implemented on BDDs in such a way that the size of the data structure only grows linearly in the number of parallel components. This compares very favourably to the exponential growth caused by the usual interleaving of causally independent transitions (as resulting, for instance, from the operational semantics of process algebras). This feature is actually so strong that one can safely state that symbolic representations are only beneficial if they are used in the context of parallel

composition of components (or in a context where similar information about the inherent system structure is exploited). Although not explicitly stated, this observation can be deduced, for instance, from the findings of Enders et al. [118], who considered the parallel composition of BDDs generated from CCS terms and showed that the symbolic representation is proportional to the sum of the sizes of its components, provided that the components are loosely coupled and provided that the interleaved variable ordering is used.

5.1.1 BDD construction

We first introduce the following notation for vectors of Boolean variables: Unless otherwise stated, we will write $\vec{\mathsf{a}} = (\mathsf{a}_1, \ldots, \mathsf{a}_{n_a})$, $\vec{\mathsf{s}} = (\mathsf{s}_1, \ldots, \mathsf{s}_{n_s})$ and $\vec{\mathsf{t}} = (\mathsf{t}_1, \ldots, \mathsf{t}_{n_s})$.

The algorithm for constructing the BDD representation from a given LTS is straight-forward and works as follows: Transitions from the LTS are processed one by one, each transition being encoded in a simple BDD which is subsequently combined by a Boolean "or" operation with the BDD representing all the previously processed transitions. The algorithm can be sketched like this:

(1) $\mathsf{B} := 0$
(2) **for** each transition $x \xrightarrow{a} y$ of the LTS **do**
(3) $\quad Newtrans := \mathcal{M}(\vec{\mathsf{a}}; \mathcal{E}(a)) \wedge \mathcal{M}(\vec{\mathsf{s}}; \mathcal{E}(x)) \wedge \mathcal{M}(\vec{\mathsf{t}}; \mathcal{E}(y))$
(4) $\quad \mathsf{B} := \mathsf{B} \vee Newtrans$
(5) **od**

On line (1), the BDD to be constructed, B, is initialised as 0. i.e. it does not represent any transition. On line (3), one transition of the SLTS is encoded in BDD *Newtrans* (which consists of a single path from the root to the terminal vertex 1, encoding action label a and source and target states x and y). Remember that the Boolean function $\mathcal{M}$ yields the minterm corresponding to a given Boolean vector. On line (4), the "or" between the previous result and the new transition is computed, i.e. the new transition is added to the previous result.

The construction of a DNBDD or MTBDD from an SLTS follows the same basic algorithm. When generating a DNBDD from a given SLTS, transitions are processed one by one. Each transition is first translated into a DNBDD which is then combined by an or-operation with the previously obtained intermediate result (the or-operation now also takes care of manipulating the rate tree). When

working with MTBDDs, addition is used instead of disjunction on line (4) of the algorithm when combining the encoding of a new transition with the previous result.

5.1.2 Parallel composition on BDDs

Consider the parallel composition of two LTSs $\mathcal{T}_1$ and $\mathcal{T}_2$ where actions from the set $S \subseteq Act$ shall take place in a synchronised way. Using process algebraic notation, we can express this as $\mathcal{T} = \mathcal{T}_1 \,|[S]|\, \mathcal{T}_2$, where $\mathcal{T}$ is the resulting LTS generated from the two components $\mathcal{T}_1$ and $\mathcal{T}_2$. Now assume that the BDDs which correspond to LTSs $\mathcal{T}_1$ and $\mathcal{T}_2$ have already been generated and are denoted B_1 and B_2. The set of actions S can also be encoded[1] in the standard way as a BDD, say S. The BDD B which corresponds to the resulting process $\mathcal{T}$ can then be obtained from $\mathsf{B}_1, \mathsf{B}_2$ and S according to the following theorem:

Theorem 5.1.1 BDD-based parallel composition of LTSs
Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two LTSs, and for $i \in \{1,2\}$, let B_i over $(\mathsf{a}_1, \ldots, \mathsf{a}_{n_a}, \mathsf{s}_1^{(i)}, \mathsf{t}_1^{(i)}, \ldots, \mathsf{s}_{n_i}^{(i)}, \mathsf{t}_{n_i}^{(i)})$ be two BDDs, such that $\mathsf{B}_i \triangleright \mathcal{T}_i$. Let $S \subseteq Act$ be a set of action labels encoded by BDD S over $(\mathsf{a}_1, \ldots, \mathsf{a}_{n_a})$. Let BDD B be constructed as follows

$$\begin{aligned} \mathsf{B} \;=\; & (\mathsf{B}_1 \wedge \mathsf{S}) \wedge (\mathsf{B}_2 \wedge \mathsf{S}) \\ \vee\; & (\mathsf{B}_1 \wedge \overline{\mathsf{S}} \wedge \mathsf{Stab}_2) \\ \vee\; & (\mathsf{B}_2 \wedge \overline{\mathsf{S}} \wedge \mathsf{Stab}_1) \end{aligned}$$

Then $\mathsf{B} \triangleright \mathcal{T}$, where $\mathcal{T} = \mathcal{T}_1 \,|[S]|\, \mathcal{T}_2$. ∎

The meaning of Stab_2 (Stab_1) is a BDD which expresses stability of the non-moving partner of the parallel composition, i.e. the fact that the source state of process $\mathcal{T}_2$ ($\mathcal{T}_1$) equals its target state. Such a BDD Stab_i actually represents the function $f_{\mathsf{Stab}_i} = \bigwedge_{j=1}^{n_i} (\mathsf{s}_j^{(i)} \leftrightarrow \mathsf{t}_j^{(i)})$, where n_i is the number of state variables of LTS $\mathcal{T}_i$ and variables $\mathsf{s}_j^{(i)}$ ($\mathsf{t}_j^{(i)}$) encode the source (target) state of a transition. Remember that this BDD is actually very compact; its number of BDD vertices is only $3 \cdot n_i + 2$, i.e. linear in the number of state variables, provided that the interleaved ordering of Boolean variables is employed.

Note that we have fixed the variable ordering within BDDs B_1 and B_2 (such that Boolean variables encoding source and target states are interleaved), although

[1] Although we do not define this formally, it goes without saying that a particular action label a has to be encoded by Boolean variables $\mathsf{a}_1, \ldots, \mathsf{a}_{n_a}$ in the same way in both B_1 and B_2.

the theorem holds for any variable ordering. However, unless otherwise stated, when performing BDD-based parallel composition, we will always assume the following overall variable ordering:

Definition 5.1.1 Standard interleaved variable ordering
Let Boolean variables $\mathsf{a}_1, \ldots, \mathsf{a}_{n_a}$ *be used to encode elements from a set of actions Act. Let Boolean variables* $\mathsf{s}_1^{(i)}, \ldots, \mathsf{s}_{n_i}^{(i)}$ *(*$\mathsf{t}_1^{(i)}, \ldots, \mathsf{t}_{n_i}^{(i)}$*) be used to encode the source (target) state of LTS* $\mathcal{T}_i$*, where* $i = 1, 2$*. The standard interleaved variable ordering is defined by*

$$\mathsf{a}_1 \prec \ldots \prec \mathsf{a}_{n_a} \prec \mathsf{s}_1^{(1)} \prec \mathsf{t}_1^{(1)} \prec \ldots \prec \mathsf{s}_{n_1}^{(1)} \prec \mathsf{t}_{n_1}^{(1)} \prec \mathsf{s}_1^{(2)} \prec \mathsf{t}_1^{(2)} \prec \ldots \prec \mathsf{s}_{n_2}^{(2)} \prec \mathsf{t}_{n_2}^{(2)}$$

This overall ordering has the advantage that it is compatible with parallel composition, i.e. plugging in two BDDs with the standard interleaved ordering will yield a BDD which also obeys the standard interleaved ordering.

Instead of a formal proof, we now explain the expression for B in Thm. 5.1.1. The term on the first line is for the synchronising actions (actions from the set S) in which both $\mathcal{T}_1$ and $\mathcal{T}_2$ participate. The conjunction $\mathsf{B}_1 \wedge \mathsf{S}$ selects that part of LTS $\mathcal{T}_1$ which corresponds to actions from the set S, and similarly for $\mathsf{B}_2 \wedge \mathsf{S}$. By then taking the conjunction of these two terms one obtains the encoding of those transitions where both partners simultaneously make a move with an action from S. The term on the second (third) line is for those actions which $\mathcal{T}_1$ ($\mathcal{T}_2$) performs independently of $\mathcal{T}_2$ ($\mathcal{T}_1$) — these actions are all from the complement of S, encoded by $\overline{\mathsf{S}}$ — and the conjunction with Stab_2 (Stab_1) ensures that $\mathcal{T}_2$ ($\mathcal{T}_1$) remains stable, i.e. does not change its state.

We now illustrate parallel composition by means of a simple queueing example, consisting of the parallel composition of an arrival process *Arrival* and a queue process *Queue*. The arrival process has only 2 states, while the queue process, which describes the behaviour of an (initially empty) queue with 3 buffer places, has 4 states. The two transition systems together with their BDD-representations are shown in Fig 5.1. Fig. 5.2 shows the intermediate and final BDDs when performing BDD-based parallel composition of processes *Arrival* and *Queue*. In the second (third) BDD one can observe the parts which express stability of process *Queue* (*Arrival*). Even in this small example we observe the general tendency that the size of the resulting BDD (25 vertices, including the terminal vertex 0 which is not shown) is in the order of the sum of the sizes of the two partner BDDs (8 vertices for *Arrival* and 15 vertices for *Queue*). Thus, using BDD-based parallel composition, the typically observed exponential growth of the memory requirement can be avoided.

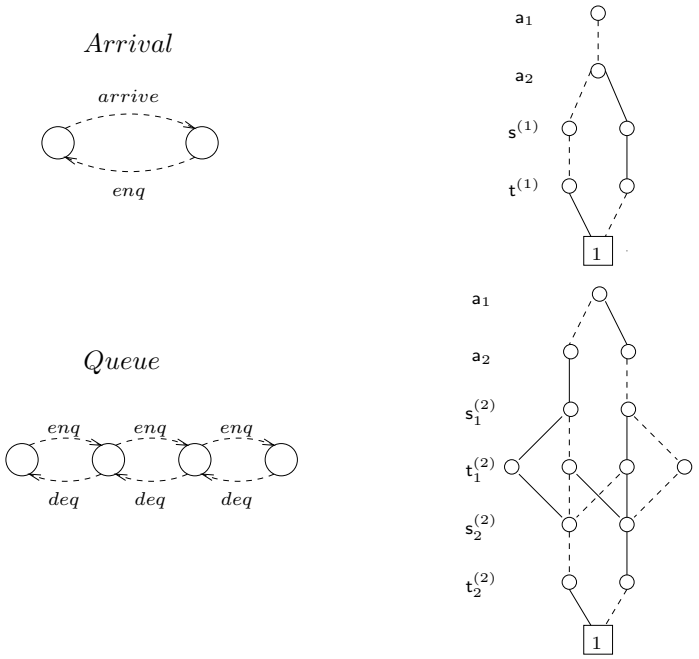


Figure 5.1: LTS and BDD for processes *Arrival* and *Queue*

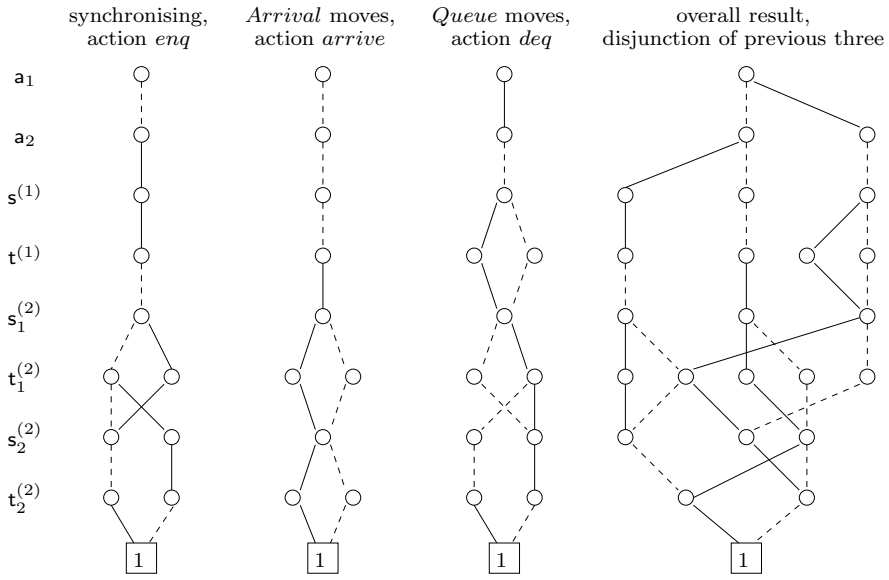


Figure 5.2: Intermediate and final BDD results for parallel composition of *Arrival* and *Queue*, synchronised over action *enq*

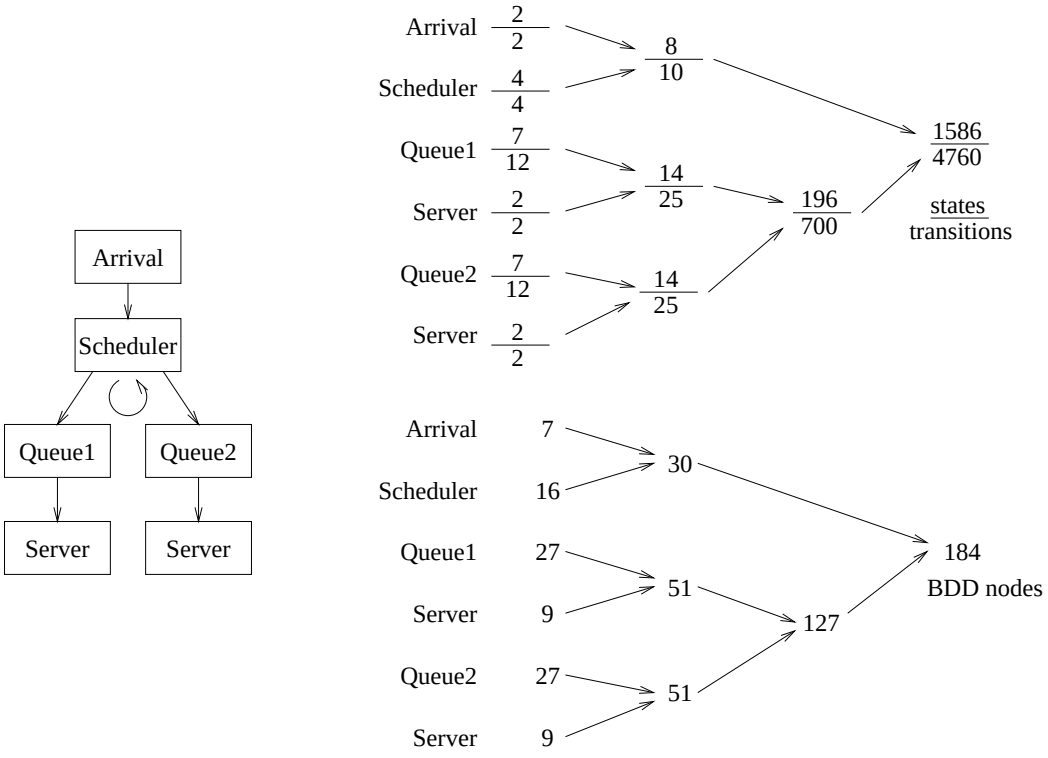


Figure 5.3: Example process model (left), corresponding number of states and transitions (top right) and corresponding number of BDD vertices (bottom right)

Extending this example, Fig. 5.3 (left) shows the block diagram of a queueing model, consisting now of an arrival process, a scheduler (which assigns incoming jobs to queues in a simple alternating fashion) and two queue-server pairs. The model was specified as the parallel composition of six sequential processes, using the (stochastic) process algebra TIPP. The numbers of states and transitions for the six component processes, as well as for all intermediate processes which one obtains by performing parallel composition in a step by step manner, are given in the top right portion of the figure. For this model, the operational semantics of the process algebra generates an overall (S)LTS with 1568 reachable states and 4760 transitions. In general, when performing parallel composition of SLTSs, the size of the resulting SLTS is exponential in the number of parallel components. Note that in this example all states of the product state space are reachable, which is often not the case.

The bottom right portion of Fig. 5.3 illustrates the use of BDD-based parallel composition. The LTSs for the six sequential processes were generated by the process algebra semantics and then translated into their corresponding BDDs. The BDD for the overall model was then generated by applying the BDD-based

parallel composition algorithm in a stepwise manner. The figure gives the number of BDD vertices at every composition step. This example demonstrates well the main advantage of symbolic parallel composition, namely the fact that the BDD size is roughly linear in the number of parallel components. In this example, the BDD for the overall model has only 184 BDD vertices. (This, by the way, is the same number of vertices which the DNBDD for the stochastic system would have if it were constructed in a similar compositional way). Note that the number of vertices for *Arrival* and $Queue_i$ is not exactly the same in Fig. 5.1 and in Fig. 5.3 since a) the action labels are encoded differently, and b) $Queue_i$ now has capacity 6 instead of 3.

We now focus our attention on the size of the BDD resulting from parallel composition of two LTSs. First recall that parallel composition of two transition systems $\mathcal{T}_1$ and $\mathcal{T}_2$ with state sets S_1 and S_2 yields an overall transition system $\mathcal{T}$ with up to $|S_1| \cdot |S_2|$ states, i.e. in the worst case the state space grows multiplicatively. The fact that the BDD representation only grows linearly can be established as follows on the base of theoretical reasoning, which will be confirmed by the application case studies presented in Chap. 10.

Theorem 5.1.2 Size of the BDD resulting from parallel composition
Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two LTSs represented by BDDs B_1 and B_2, i.e. $\mathsf{B}_i \triangleright \mathcal{T}_i$ ($i = 1, 2$), using the standard interleaved variable ordering. For $S \subseteq Act$, let B be the BDD representing the parallel composition $\mathcal{T}_1|[S]|\mathcal{T}_2$ (constructed according to Thm. 5.1.1), written $\mathsf{B} \triangleright \mathcal{T}_1|[S]|\mathcal{T}_2$. Then the number of vertices of BDD B is bounded by $|Act| \cdot (|\mathsf{B}_1| + |\mathsf{B}_2| + |\mathsf{Stab}_1| + |\mathsf{Stab}_2|)$. ■

Proof: We now sketch a proof. The chosen (standard interleaved) variable ordering for BDD B_i is $\mathsf{a}_1 \prec \ldots \prec \mathsf{a}_{n_a} \prec \mathsf{s}_1^{(i)} \prec \mathsf{t}_1^{(i)} \prec \ldots \prec \mathsf{s}_{n_i}^{(i)} \prec \mathsf{t}_{n_i}^{(i)}$, i.e. the variables encoding the action name are at the top, followed by an interleaving of the variables for source and target state. For the BDD B resulting from parallel composition the variable ordering is $\mathsf{a}_1 \prec \ldots \prec \mathsf{a}_{n_a} \prec \mathsf{s}_1^{(1)} \prec \mathsf{t}_1^{(1)} \prec \ldots \prec \mathsf{s}_{n_1}^{(1)} \prec \mathsf{t}_{n_1}^{(1)} \prec \mathsf{s}_1^{(2)} \prec \mathsf{t}_1^{(2)} \prec \ldots \prec \mathsf{s}_{n_2}^{(2)} \prec \mathsf{t}_{n_2}^{(2)}$. The proof considers three cases:

1. We consider first the case of parallel composition with maximal synchronisation, i.e. the case $S = Act$, which means synchronisation on all actions. Let $|\mathsf{B}_i|$ be the number of vertices of B_i, and for an action $a \in Act$, let A_a be the BDD encoding that action. $\mathsf{B}_{i,a} = \mathsf{B}_i \wedge \mathsf{A}_a$ is the "restriction" of B_i to action a, i.e. the subgraph of B_i which corresponds to action a (as highlighted in Fig. 5.4, top). To obtain the subgraph of the resulting BDD B which corresponds to action a one has to build $\mathsf{B}_a = \mathsf{B}_{1,a} \wedge \mathsf{B}_{2,a} = (\mathsf{B}_1 \wedge \mathsf{A}_a) \wedge (\mathsf{B}_2 \wedge \mathsf{A}_a)$

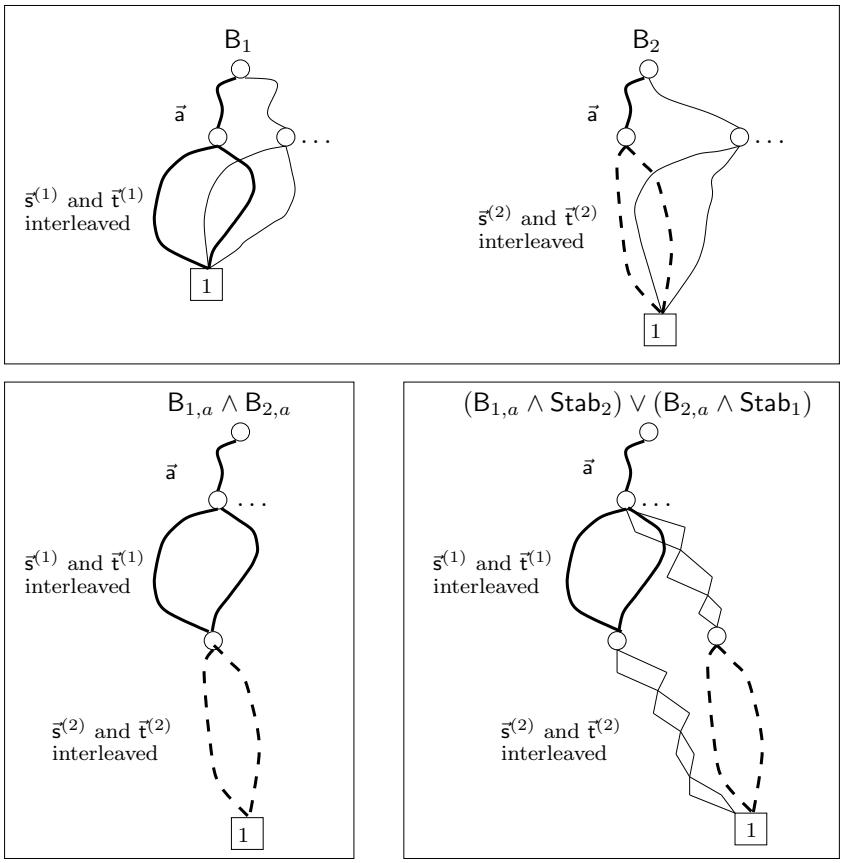


Figure 5.4: Sketch of the BDD shapes involved in parallel composition

whose number of vertices can be bounded as (cf. Fig. 5.4, bottom left).

$$\begin{array}{rcl} |\mathsf{B}_a| & \leq & |\mathsf{B}_{1,a}| + |\mathsf{B}_{2,a}| \\ & \leq & |\mathsf{B}_1| + |\mathsf{B}_2| \end{array}$$

Summing up over all actions we obtain $|\mathsf{B}| \leq |Act| \cdot (|\mathsf{B}_1| + |\mathsf{B}_2|)$. Note that $|Act|$ is usually a small value and that this is a rather coarse worst case bound which assumes that there is no sharing of the subgraphs which correspond to different action labels.

2. In the case where action a is non-synchronising the picture is as follows: $\mathsf{B}_a = (\mathsf{B}_{1,a} \wedge \mathsf{Stab}_2) \vee (\mathsf{Stab}_1 \wedge \mathsf{B}_{2,a}) = ((\mathsf{B}_1 \wedge \mathsf{A}_a) \wedge \mathsf{Stab}_2) \vee (\mathsf{Stab}_1 \wedge (\mathsf{B}_2 \wedge \mathsf{A}_a))$ whose number of vertices can be bounded by (cf. Fig. 5.4, bottom right, where the zig-zag structures symbolise the parts corresponding to Stab_i).

$$\begin{array}{rcl} |\mathsf{B}_a| & \leq & |\mathsf{B}_{1,a}| + |\mathsf{Stab}_2| + |\mathsf{B}_{2,a}| + |\mathsf{Stab}_1| \\ & \leq & |\mathsf{B}_1| + |\mathsf{Stab}_2| + |\mathsf{B}_2| + |\mathsf{Stab}_1| \end{array}$$

In the case of pure interleaving, i.e. the case $S = \emptyset$, which means that all actions are non-synchronising, the overall size can thus be bounded by $|\mathsf{B}| \leq |Act| \cdot (|\mathsf{B}_1| + |\mathsf{Stab}_2| + |\mathsf{B}_2| + |\mathsf{Stab}_1|)$. Remember that Stab_i is represented in a compact manner with only $|\mathsf{Stab}_i| = 3n_i + 2 = 3\lceil \log_2 |S_i| \rceil + 2$ vertices, i.e. $|\mathsf{Stab}_i|$ is usually much smaller than $|\mathsf{B}_i|$.

3. In the general case where there is only partial synchronisation, i.e. the case $S \subset Act$, which means that there is synchronisation on a subset of the actions and interleaving of the remaining actions, the two extremal results from above can be combined, resulting in the overall bound

$$|\mathsf{B}| \leq |Act| \cdot (|\mathsf{B}_1| + |\mathsf{Stab}_2| + |\mathsf{B}_2| + |\mathsf{Stab}_1|)$$

This concludes the proof. ■

5.1.3 Parallel composition on DNBDDs

In this section, we describe how parallel composition can be performed symbolically in a stochastic setting with the help of DNBDDs. The basic procedure is as for the BDD case. Let (similarly as before) D_i be the DNBDD representing SLTS $\mathcal{T}_i$ $(i = 1, 2)$, and let S and Stab_i be BDDs defined as before. The DNBDD D which corresponds to the combined SLTS $\mathcal{T} = \mathcal{T}_1 | [S] | \mathcal{T}_2$ is obtained by evaluating the following expression:

$$\begin{array}{rcl} \mathsf{D} & = & (\mathsf{D}_1 \wedge \mathsf{S}) \wedge (\mathsf{D}_2 \wedge \mathsf{S}) \\ & \vee & (\mathsf{D}_1 \wedge \overline{\mathsf{S}} \wedge \mathsf{Stab}_2) \\ & \vee & (\mathsf{D}_2 \wedge \overline{\mathsf{S}} \wedge \mathsf{Stab}_1) \end{array}$$

Note that this expression is structurally similar to the one given in Theorem 5.1.1. Note further that in this expression the $\wedge$ and $\vee$ operations are used both to combine a DNBDD with a BDD (as for instance in $\mathsf{D}_1 \wedge \mathsf{S}$), and to combine two DNBDDs (as for instance in $(\mathsf{D}_1 \wedge \mathsf{S}) \wedge (\mathsf{D}_2 \wedge \mathsf{S})$). So, $\wedge$ and $\vee$ are now not just Boolean operators but in addition must be capable of manipulating the rate trees if one or both of the operands are DNBDDs.

An important question is about the result rate of synchronising actions. Suppose transition $x_1 \xrightarrow{a,\lambda} y_1$ in SLTS $\mathcal{T}_1$ synchronises with transition $x_2 \xrightarrow{a,\mu} y_2$ in SLTS $\mathcal{T}_2$. This yields a transition $(x_1, x_2) \xrightarrow{a,\phi(\lambda,\mu)} (y_1, y_2)$ in the combined SLTS $\mathcal{T}$. Depending on the application, different expressions for the result rate $\phi(\lambda, \mu)$ may apply, cf. Sec. 3.7.1. Typical examples are the maximum, minimum, sum or product of the two partner rates (when using PEPA's concept of apparent rate

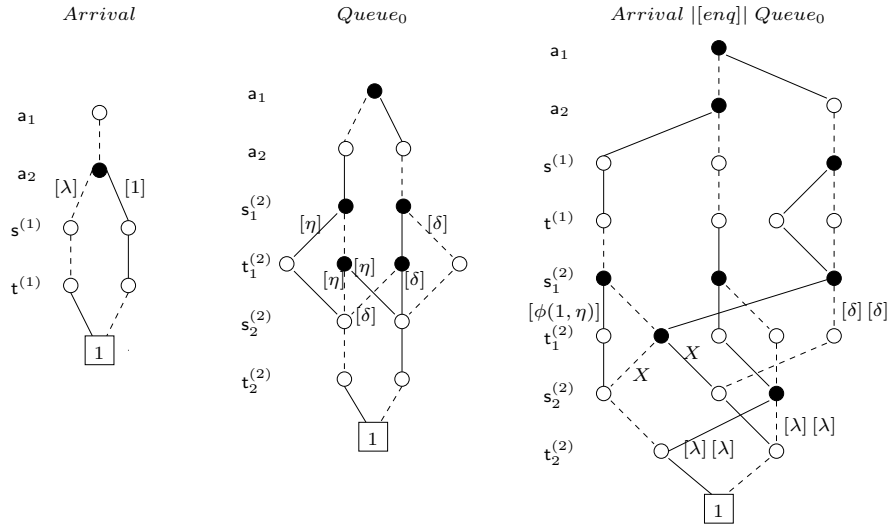


Figure 5.5: DNBDDs for the queueing example (shorthand notation: $X = [\phi(1,\eta)][\delta][\delta]$)

[192], the definition and calculation of the function ϕ becomes more involved). Using DNBDDs, the result rate is calculated from the two partner rates during the $\wedge$ operation at the centre of the first line of the above equation. This $\wedge$ operation is flexible enough to realise any of the above alternatives (maximum, minimum, ...), i.e. DNBDDs are able to cover any of those cases.

Concerning the size of the DNBDD D resulting from the parallel composition of two SLTSs, its number of vertices can be bounded in a similar way as stated in Thm. 5.1.2 for the BDD case. We do not give a bound on the size of the rate tree of D, since such a bound depends on the actual implementation of the rate tree, which issue, as mentioned in Sec. 4.3.1, is still under investigation.

To illustrate parallel composition on DNBDDs, we return to the simple queueing example. Fig. 5.5 shows the DNBDDs associated with processes *Arrival*, *Queue* and *Arrival* |[*enq*]| *Queue* (in the figure, decision nodes are drawn black). In order to keep the figure clear, the rate trees are omitted, only the rate lists are shown beside their corresponding BDD edges. On the left, rates λ and 1 are attached to the outgoing edges of the (single) decision node of the BDD. In the middle, six individual rates are attached to the appropriate edges. On the right hand side, up to three rate lists, each consisting of a single rate, are attached to BDD edges. For instance, the rate lists $[\delta][\delta]$ specify the rates of the two transitions encoded as Boolean vectors $(1,0,1,1,0,0,1,0)$ and $(1,0,0,0,0,0,1,0)$ whose paths share the last decision node.

5.1.4 Parallel composition on MTBDDs

In this section, we consider the parallel composition of SLTSs represented by MTBDDs as described in Section 4.4.2.

Theorem 5.1.3 MTBDD-based parallel composition of SLTS
Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two SLTSs, and for $i \in \{1,2\}$, let M_i over $(\mathsf{a}_1, \ldots, \mathsf{a}_{n_a}, \mathsf{s}_1^{(i)}, \mathsf{t}_1^{(i)}, \ldots, \mathsf{s}_{n_i}^{(i)}, \mathsf{t}_{n_i}^{(i)})$ be two MTBDDs, such that $\mathsf{M}_i \triangleright \mathcal{T}_i$. Let $S \subseteq Act$ be a set of action labels encoded by BDD S over $(\mathsf{a}_1, \ldots, \mathsf{a}_{n_a})$. Let MTBDD M be constructed as follows

$$\begin{array}{rcl} \mathsf{M} & = & (\mathsf{M}_1 \cdot \mathsf{S}) \cdot (\mathsf{M}_2 \cdot \mathsf{S}) \\ & + & \mathsf{M}_1 \cdot (1 - \mathsf{S}) \cdot \mathsf{Id}_2 \\ & + & \mathsf{M}_2 \cdot (1 - \mathsf{S}) \cdot \mathsf{Id}_1 \end{array}$$

Then $\mathsf{M} \triangleright \mathcal{T}$, where $\mathcal{T} = \mathcal{T}_1 |[S]| \mathcal{T}_2$. ∎

One immediately recognises the similarity with the expressions given for the BDD- and DNBDD-cases in the two previous sections. Since we are now working with MTBDDs, disjunction $\vee$ and conjunction $\wedge$ (resp. the extensions of these operations to the DNBDD data structure) are replaced by addition $+$ and multiplication $\cdot$. The set of synchronising actions is encoded by MTBDD S, which is actually a BDD, since its terminal vertices are 0 and 1. In order to obtain its complement, i.e. the set of non-synchronising actions, instead of using Boolean negation $\overline{\mathsf{S}}$ as before, we now use the expression $1 - \mathsf{S}$, which turns a terminal 0 into a 1 and vice versa. Instead of BDD Stab_i we now use the MTBDD Id_i, which represents an identity matrix of appropriate size. Note, however, that Stab_i and Id_i are actually identical.

Note that for the synchronising transitions, calculated by the first line in the above expression, the resulting rate $\phi(\lambda, \mu)$ is now given by the product $\lambda \cdot \mu$, which is in accordance with the stochastic process algebra TIPP and ensures important congruence properties [179, 162, 167]. Should one wish to employ a different function $\phi(\lambda, \mu)$, for instance the maximum function, one would simply have to replace the first line of the above expression by $\textsc{Max}(\mathsf{M}_1 \cdot \mathsf{S}, \mathsf{M}_2 \cdot \mathsf{S})$, where Max is the maximum function on MTBDDs which can be realised with the help of a particular instance of the standard Apply algorithm.

We now consider the size of the MTBDD resulting from the parallel composition of two SLTSs and derive a similar bound as we did before for the BDD case.

Theorem 5.1.4 Size of a MTBDD resulting from parallel composition
Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two SLTSs represented by MTBDDs M_1 and M_2, i.e. $\mathsf{M}_i \triangleright \mathcal{T}_i$ ($i = 1, 2$), using the standard interleaved variable ordering. For $S \subseteq Act$, let M be the MTBDD representing the parallel composition $\mathcal{T}_1|[S]|\mathcal{T}_2$ (constructed according to Thm. 5.1.3), written $\mathsf{M} \triangleright \mathcal{T}_1|[S]|\mathcal{T}_2$. Then the number of vertices of MTBDD M is bounded by $|Act| \cdot \eta \cdot (|\mathsf{M}_1| + |\mathsf{M}_2| + |\mathsf{Id}_1| + |\mathsf{Id}_2|)$ where η depends on the number of distinct rate values that are associated with a particular action. ■

Proof: We analyse the size of the resulting MTBDD in a similar fashion as we did for BDDs in Section 5.1.2. There are, however, some important modifications to the bounds, since the number of terminal vertices of the partner MTBDDs has to be taken into account. We know that SLTS $\mathcal{T}_i$ is represented by MTBDD M_i, i.e. $\mathsf{M}_i \triangleright \mathcal{T}_i$, and that, as above, the standard interleaved variable ordering is used. The proof again considers three cases:

1. Again, we consider first the case of parallel composition with maximal synchronisation. Let $|\mathsf{M}_i|$ be the number of vertices of M_i, and for an action $a \in Act$, let A_a be the BDD encoding that action. $\mathsf{M}_{i,a} = \mathsf{M}_i \cdot \mathsf{A}_a$ is the "restriction" of M_i to action a. To obtain the subgraph of the resulting MTBDD M which corresponds to action a one has to build $\mathsf{M}_a = \mathsf{M}_{1,a} \cdot \mathsf{M}_{2,a} = (\mathsf{M}_1 \cdot \mathsf{A}_a) \cdot (\mathsf{M}_2 \cdot \mathsf{A}_a)$ whose number of vertices can be bounded as

$$\begin{aligned} |\mathsf{M}_a| &\leq |\mathsf{M}_{1,a}| + \eta_{1,a} \cdot |\mathsf{M}_{2,a}| \\ &\leq |\mathsf{M}_1| + \eta_{1,a} \cdot |\mathsf{M}_2| \\ &\leq \eta_{1,a} \cdot (|\mathsf{M}_1| + |\mathsf{M}_2|) \end{aligned}$$

   In the latter equation, $\eta_{1,a}$ denotes the number of terminal vertices of $\mathsf{M}_{1,a}$ which is usually a small value. Summing up over all actions we obtain $|\mathsf{M}| \leq |Act| \cdot \eta_1 \cdot (|\mathsf{M}_1| + |\mathsf{M}_2|)$, where $\eta_1 = \max_{a \in Act}\{\eta_{1,a}\}$. As for the BDD case, $|Act|$ is also usually a small value.

2. In the case where action a is non-synchronising the picture is as follows: $\mathsf{M}_a = \mathsf{M}_{1,a} \cdot \mathsf{Id}_2 + \mathsf{Id}_1 \cdot \mathsf{M}_{2,a} = (\mathsf{M}_1 \cdot \mathsf{A}_a) \cdot \mathsf{Id}_2 + \mathsf{Id}_1 \cdot (\mathsf{M}_2 \cdot \mathsf{A}_a)$ whose number of vertices can be bounded by

$$\begin{aligned} |\mathsf{M}_a| &\leq |\mathsf{M}_{1,a}| + \eta_{1,a} \cdot |\mathsf{Id}_2| + |\mathsf{M}_{2,a}| + |\mathsf{Id}_1| \\ &\leq |\mathsf{M}_1| + \eta_{1,a} \cdot |\mathsf{Id}_2| + |\mathsf{M}_2| + |\mathsf{Id}_1| \end{aligned}$$

   In the case of pure interleaving, the overall size can thus be bounded by $|\mathsf{M}| \leq |Act| \cdot (|\mathsf{M}_1| + \eta_1 \cdot |\mathsf{Id}_2| + |\mathsf{M}_2| + |\mathsf{Id}_1|)$. Remember that Id_i is represented in a compact manner with only $|\mathsf{Id}_i| = 3n_i + 2 = 3\lceil \log |S_i| \rceil + 2$ vertices, i.e. $|\mathsf{Id}_i|$ is usually much smaller than $|\mathsf{M}_i|$.

3. For the general, mixed case, where there are both synchronising and non-synchronising transitions, we can combine the bounds for the two extremal cases and obtain the overall bound: $|\mathsf{M}| \leq |Act| \cdot \eta_1 \cdot (|\mathsf{M}_1| + |\mathsf{Id}_2| + |\mathsf{M}_2| + |\mathsf{Id}_1|)$.

This concludes the proof. ∎

5.1.5 Reachability analysis

The BDD B resulting from the parallel composition of two partners B_1 and B_2 describes all transitions which are possible in the product space of the two partner processes. Given a pair of initial states for LTSs $\mathcal{T}_1$ and $\mathcal{T}_2$, however, only part of this product space may be reachable due to synchronisation constraints. Therefore, B potentially includes transitions emanating from unreachable states.

For example, consider the two transition systems $\mathcal{T}_1$ and $\mathcal{T}_2$ shown in Figure 5.6 (top). Their parallel composition, starting from the initial state $(1, 1)$ and synchronising over action set $S = \{a\}$, yields the LTS $\mathcal{T}'$ shown at the bottom left of the figure. On the other hand, if state $(1, 2)$ were the initial state, parallel composition would yield the LTS $\mathcal{T}''$, as shown at the bottom right. Since BDD-based parallel composition does not take into account the information about the initial state, the LTS $\mathcal{T}$ resulting from BDD-based parallel composition is the "union" of the two previous results. (If we were building $\mathcal{T}$ from $\mathcal{T}'$ and $\mathcal{T}''$, we could obtain its state space and its transition relation by $S_{\mathcal{T}} = S_{\mathcal{T}'} \cup S_{\mathcal{T}''}$ and $\dashrightarrow_{\mathcal{T}} = \dashrightarrow_{\mathcal{T}'} \cup \dashrightarrow_{\mathcal{T}''}$, whereas the initial state of $\mathcal{T}$ would be ambiguous. However, since $\mathcal{T}$ is generated directly from $\mathcal{T}_1$ and $\mathcal{T}_2$, the initial state is of course given by $(1, 1)$ and the portion of $\mathcal{T}$ which corresponds to $\mathcal{T}''$ is simply unreachable.)

This situation is the same, whether we use BDDs for the parallel composition of LTSs, or whether we use DNBDDs or MTBDDs for the parallel composition of SLTSs. The reason is that the basic approach to building the symbolic representation of the combined process with these data structures works according to the following scheme: For a synchronising action a, each a-transition $x_1 \overset{a}{\dashrightarrow} y_1$ of the first partner is combined with each a-transition $x_2 \overset{a}{\dashrightarrow} y_2$ of the second partner, regardless whether the state (x_1, x_2) is reachable or not. For non-synchronising action b, each b-transition $x_1 \overset{b}{\dashrightarrow} y_1$ of the first partner is supposed to be enabled in any state (x_1, x_2), i.e. $(x_1, x_2) \overset{b}{\dashrightarrow} (y_1, x_2)$, regardless whether the state (x_1, x_2) is reachable or not.

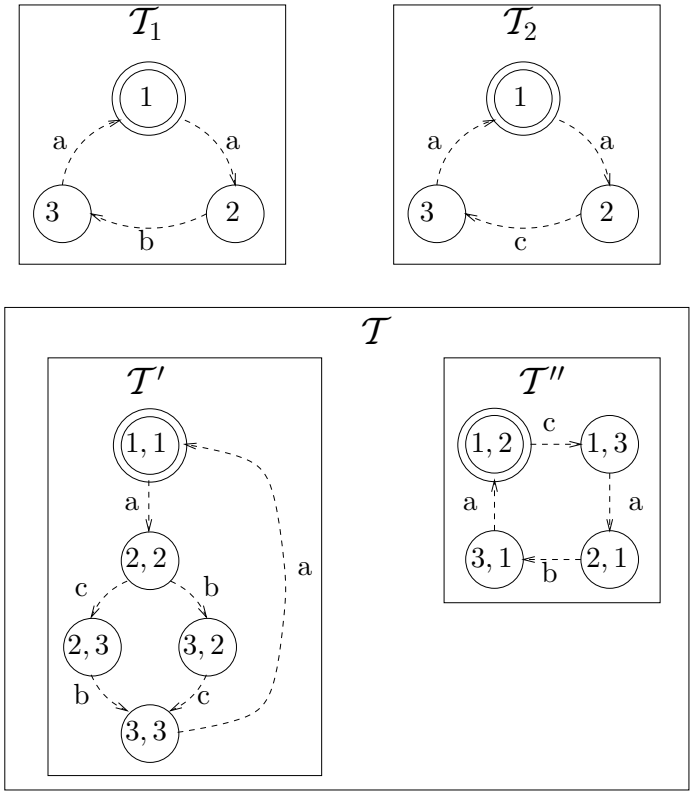


Figure 5.6: Parallel composition of two LTSs

In this situation, reachability analysis is an important tool for reducing the size of the underlying (S)LTS. Reachability analysis can be performed directly on the BDD representation of the resulting process, as has been described, for instance, by [39]. The following general reachability algorithm computes a BDD which represents all states which are reachable from a given initial state. At every step of the algorithm, new states reachable by a single transition from states previously found are added to this BDD. This is repeated until a fixed point is reached.

(1) $Reach(\vec{\mathsf{t}}) := \mathcal{M}(\vec{\mathsf{t}}; \mathcal{E}(s_1))$
(2) $Unex(\vec{\mathsf{s}}) := \mathcal{M}(\vec{\mathsf{s}}; \mathcal{E}(s_1))$
(3) **do while** $(Unex(\vec{\mathsf{s}}) \neq 0)$
(4) $\quad New(\vec{\mathsf{t}}) := \textsc{Abstract}\left((Trans(\vec{\mathsf{s}}, \vec{\mathsf{t}}) \wedge Unex(\vec{\mathsf{s}})), \vec{\mathsf{s}}, \vee\right) \wedge \overline{Reach(\vec{\mathsf{t}})}$
(5) $\quad Reach(\vec{\mathsf{t}}) := Reach(\vec{\mathsf{t}}) \vee New(\vec{\mathsf{t}})$
(6) $\quad Unex(\vec{\mathsf{s}}) := New(\vec{\mathsf{t}})\{\vec{\mathsf{t}} \leftarrow \vec{\mathsf{s}}\}$
(7) **od**
(8) **return** $Reach(\vec{\mathsf{t}})$

The algorithm assumes that the transition relation is encoded in a BDD $Trans$ which depends on the two vectors of Boolean variables $\vec{\mathsf{s}}$ and $\vec{\mathsf{t}}$. ($Trans$ can be easily obtained from B by abstracting from the action variables.) In lines (1) and (2), the BDDs $Reach$ (encoding the set of reachable states) and $Unex$ (encoding the set of yet unexplored states), are initialised such that they both contain the encoding of the starting state s_1. The main loop of the algorithm starts in line (3) and ends when the set of unexplored states is empty (when the BDD $Unex$ is simply the 0-vertex). The core operation within every iteration is performed in line (4), where the increment New to the set of reachable states is computed: The conjunction $Trans \wedge Unex$ restricts the transition relation to source states from the set $Unex$. The ABSTRACT-operation abstracts from the vector of s-variables, i.e. from all variables encoding the source state of a transition. Therefore the result $\textsc{Abstract}\,((Trans \wedge Unex), \vec{\mathsf{s}}, \vee)$ encodes all states which are reachable from the set $Unex$. The conjunction with $\overline{Reach}$ is needed in order to restrict the set New to states not previously in $Reach$. In line (5), the set of reachable states is incremented, and in line (6), the set of unexplored states is set to New. Note how renaming of Boolean variables is used in line (6), since New depends on the vector of Boolean variables $\vec{\mathsf{t}}$ while $Unex$ has to depend on vector $\vec{\mathsf{s}}$, in order to fit properly into the conjunction in line (4).

After the BDD $Reach$ encoding the set of reachable states has been determined, the BDD (DNBDD, MTBDD) B representing the overall (S)LTS can be restricted to those transitions which originate in reachable states. This can be easily realised by a single conjunction: $\mathsf{B}(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}})_{reach} := \mathsf{B}(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}}) \wedge Reach(\vec{\mathsf{s}})$, where $Reach$ must now be a BDD depending on the vector of Boolean variables $\vec{\mathsf{s}}$, i.e. on the variables $\mathsf{s}_1, \ldots, \mathsf{s}_{n_s}$ encoding the source state.

We now mention a very interesting issue concerning the sizes of the BDDs before and after reachability analysis: Contrary to what one might expect, it can often be observed that the BDD B_{reach}, representing the reachable part of the combined (S)LTS, is larger than the original BDD B encoding all transitions within the product space. In a way, this is against the intuition, since one would expect that the BDD becomes bigger if it encodes more transitions, but as we shall see, BDD sizes are often counter-intuitive. In the example from Figure 5.6, the number of BDD vertices for $\mathcal{T}$ is 36, while the number of BDD vertices for $\mathcal{T}'$ is 38, although the former encodes 10 and the latter only 6 transitions[2]. The reason is that the restriction to the reachable part often destroys the regularity of the BDD-based representation. Therefore it may often be better to keep the unreachable states and transitions.

[2]Actually, $\mathcal{T}$ encodes 12 transitions, since there are extra phantom states which stem from the fact that 2 bits are used to encode the state of either partner, but the systems have only 3 states.

Another effect that can often be observed, and which may also seem surprising, is the following: Suppose a BDD B has been generated as the parallel composition of two partners B_1 and B_2 which depend on n_1 resp. n_2 state variables (thus, B depends on $n_1 + n_2$ state variables). Suppose further that symbolic reachability analysis has been performed on B, resulting in a BDD B_{reach} (which also depends on the same $n_1 + n_2$ state variables). In the case where the number of reachable states is substantially smaller than the number of states in the product space of the two partner state spaces, one could rename and re-encode the states, such that less than $n_1 + n_2$ variables would suffice in order to characterise a state. However, although the re-encoded BDD then has fewer variable levels, its size, i.e. its number of vertices, is often larger than before the re-encoding. Again, the reason for this "strange" behaviour is loss of regularity.

5.2 Issues related to Markovian and immediate transitions

In this section, we discuss issues related to ESLTSs, i.e. transition systems which have both Markovian and immediate transitions, which are represented symbolically by MTBDDs as discussed in Sec. 4.4.3. For reasons of brevity, we restrict this discussion to the case of MTBDDs, although in principle DNBDDs could be employed as well as the underlying data structure.

5.2.1 Parallel composition

We now consider symbolic parallel composition for the ESLTS case. Note that in accordance with the semantics of stochastic process algebras such as TIPP and PEPA, synchronisation between two a-transitions is only possible if they are either both immediate transitions or if they are both Markovian transitions, i.e. synchronisation between an immediate and a Markovian transition is not allowed (cf. the SPA semantics given in Sec 3.7.1). As a result, the concepts for parallel composition developed in Sections 5.1.2 through 5.1.4 can be applied without modification to the ESLTS case.

To describe this formally, suppose that the immediate transitions of ESLTS $\mathcal{T}_i$ are represented by BDD M_i^I and that the Markovian transitions of ESLTS $\mathcal{T}_i$ are represented by MTBDD M_i^M, where $i = 1, 2$ (i.e. we follow the second option described in Sec. 4.4.3, where Markovian and immediate transitions are represented

by two separate MTBDDs). Let S encode the set of synchronising actions. Then

$$\begin{array}{rcl} \mathsf{M}^I & = & (\mathsf{M}_1^I \wedge \mathsf{S}) \wedge (\mathsf{M}_2^I \wedge \mathsf{S}) \\ & \vee & (\mathsf{M}_1^I \wedge \overline{\mathsf{S}} \wedge \mathsf{Stab}_2) \\ & \vee & (\mathsf{M}_2^I \wedge \overline{\mathsf{S}} \wedge \mathsf{Stab}_1) \end{array}$$

represents the immediate transitions of the combined process $\mathcal{T} = \mathcal{T}_1 | [S] | \mathcal{T}_2$, and

$$\begin{array}{rcl} \mathsf{M}^M & = & (\mathsf{M}_1^M \cdot \mathsf{S}) \cdot (\mathsf{M}_2^M \cdot \mathsf{S}) \\ & + & \mathsf{M}_1^M \cdot (1 - \mathsf{S}) \cdot \mathsf{Id}_2 \\ & + & \mathsf{M}_2^M \cdot (1 - \mathsf{S}) \cdot \mathsf{Id}_1 \end{array}$$

represents the Markovian transitions of the combined process $\mathcal{T}$. Therefore, altogether we have $(\mathsf{M}^I, \mathsf{M}^M) \triangleright \mathcal{T}$.

5.2.2 Symbolic hiding and elimination of compositionally vanishing states

Process algebras provide the concept of hiding, i.e. making visible actions invisible. Hiding a (visible) action b in a transition system $\mathcal{T}$ causes all b-transitions within $\mathcal{T}$ to be relabelled by the special internal action τ. The action τ is invisible from the environment; therefore synchronisation on τ-transitions is not possible.

Hiding is important for several reasons:

- In the context of LTSs, internal transitions may be eliminated through the concept of weak bisimulation. It is possible to reduce the size of an LTS by finding a weakly bisimilar one with fewer states.
- In the context of ESLTSs, internal immediate transitions may be eliminated through the concept of weak Markovian bisimulation, which again can be employed in order to reduce the state space. Furthermore, internalising immediate transitions and eliminating them (thereby eliminating the vanishing states, see below), is mandatory for transforming the ESLTS into a CTMC which can then be analysed by numerical methods.

Within a purely Markovian framework, making certain transitions internal has the sole effect that the system may no longer synchronise via these. A reduction of the state space is not possible in this case, since τ-transitions still have a strictly positive (to be precise: exponential) delay from which one cannot abstract without modifying the underlying stochastic process.

We now describe symbolic hiding formally for the case of an LTS $\mathcal{T}$ represented by a BDD M. Although this is the simplest case, it is general enough, since for SLTS or ESLTS represented by DNBDDs or MTBDDs symbolic hiding works in a similar fashion. So, let $\mathsf{M} \triangleright \mathcal{T}$, let $\mathsf{a}_1, \ldots, \mathsf{a}_{n_a}$ be the Boolean variables encoding the action label, and let b be an action label. The hiding of action b can then be achieved at the level of the symbolic representation by the following operation

$$\begin{aligned} \mathsf{M}^{\textbf{hide } b \textbf{ in } \mathcal{T}} \quad = \quad & \left(\mathsf{M}\Big|_{\vec{\mathsf{a}}=\mathcal{E}(b)} \wedge \ \mathcal{M}(\vec{\mathsf{a}}; \mathcal{E}(\tau)) \right) \\ \vee \quad & \left(\mathsf{M} \wedge \neg \mathcal{M}(\vec{\mathsf{a}}; \mathcal{E}(b)) \right) \end{aligned}$$

In this equation, the cofactor $\mathsf{M}\Big|_{\vec{\mathsf{a}}=\mathcal{E}(b)}$ represents all transitions which correspond to b-actions (note that such a cofactor does not depend on Boolean variables $\mathsf{a}_1, \ldots, \mathsf{a}_{n_a}$). These transitions are relabelled by τ through the conjunction with the term $\mathcal{M}(\vec{\mathsf{a}}; \mathcal{E}(\tau))$. The part of the BDD M not corresponding to action b, which is given by the conjunction $\mathsf{M} \wedge \neg \mathcal{M}(\vec{\mathsf{a}}; \mathcal{E}(b))$, remains unmodified.

We now consider the case of ESLTS. In particular, we describe how vanishing states, which are a result of internal immediate transitions, can be eliminated, thereby reducing the size of the state space[3]. We remark that the concept of vanishing states and their elimination has been studied extensively in the context of Generalised Stochastic Petri Nets (GSPN) [1, 75]. However, in a compositional framework as considered in this thesis, the definition and treatment of vanishing states is somewhat more complicated than in the monolithic GSPN case, as explained in the sequel. We now refine the concept of a vanishing state as defined in Def. 3.7.8, leading to the notion of a compositionally vanishing state [299].

Definition 5.2.1 Compositionally vanishing state
A state s of an ESLTS is called compositionally vanishing if there is at least one internal immediate transition emanating from s (written $s \overset{\tau}{\dashrightarrow} s'$), but no visible immediate transition emanating from s (written $s \not\overset{a}{\dashrightarrow} s''$, where $a \neq \tau$). ■

A vanishing state is left as soon as it is entered via one of its (possibly several) $\overset{\tau}{\dashrightarrow}$ -transitions. Note that a necessary condition for a state to be compositionally vanishing is that it may not be left via visible immediate transitions. At a first glance, this condition may seem to be unjustified, its meaning only becomes clear in a compositional scenario: This condition is introduced in order to delay the elimination of states with both emanating $\overset{\tau}{\dashrightarrow}$ -transitions and $\overset{a}{\dashrightarrow}$ -transitions, thereby preserving the possibility to perform a visible immediate

[3]Vanishing (tangible) states are sometimes also called instable (stable) states.

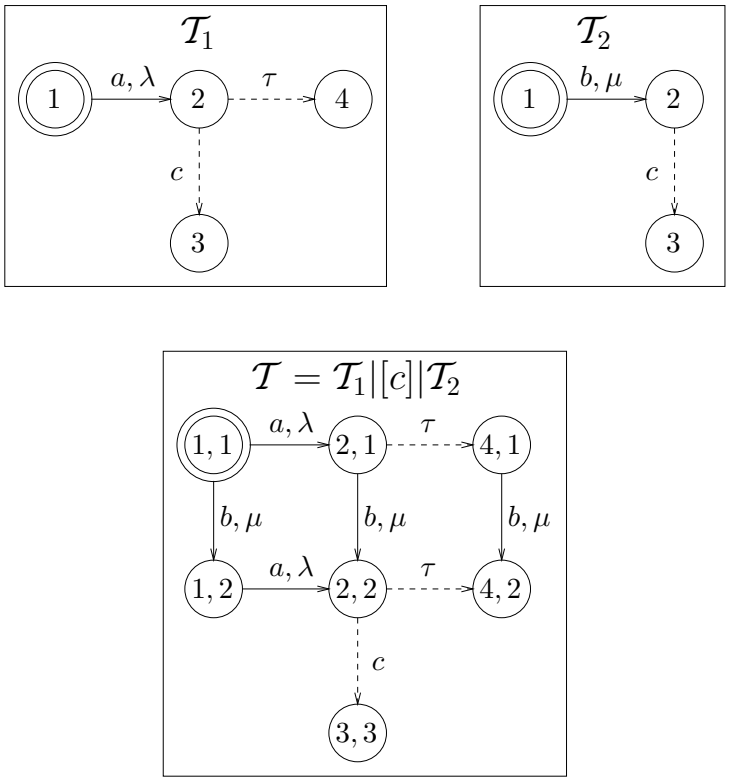


Figure 5.7: Role of visible immediate transitions during parallel composition

transition together with a partner process. An example for such a situation is shown in Fig. 5.7 which shows two ESLTSs, $\mathcal{T}_1$ and $\mathcal{T}_2$, which are composed in parallel, synchronising on action c. The resulting ESLTS, $\mathcal{T}$, is shown at the bottom of the figure. State 2 in ESLTS $\mathcal{T}_1$ must not be eliminated before parallel composition takes place (and therefore state 2 must not be considered a compositionally vanishing state), since its elimination would disable any c-transition in the combined transition system $\mathcal{T}$. In the resulting ESLTS, state $(2, 1)$ is a compositionally vanishing state which can be eliminated, whereas state $(2, 2)$ is not. However, if action c is hidden in ESLTS $\mathcal{T}$ (since further synchronisation on c is not required), state $(2, 2)$ becomes compositionally vanishing and can be also eliminated (its elimination, however, requires a proper treatment of non-determinism as explained below).

Note also that there may be one or several Markovian transitions emanating from a vanishing state, but they are never taken, since internal immediate transitions have priority over them (they always win the race). As an example, in Fig. 5.7 the transition $(2, 1) \xrightarrow{b,\mu} (2, 2)$ will never be taken, since the competing internal

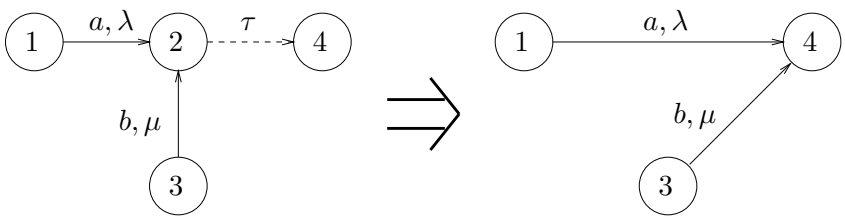


Figure 5.8: Simple redirection of transitions

immediate transition $(2,1) \xrightarrow{\tau} (4,1)$ will always take place first. Therefore transition $(2,1) \xrightarrow{b,\mu} (2,2)$ can safely be deleted without changing the behaviour of the ESLTS.

With Def. 3.7.8 and Def. 5.2.1 we have defined the important notions of vanishing states and compositionally vanishing states. In order to complete the picture, we also classify the remaining states: A state s is called compositionally tangible if there is no immediate transition (i.e. neither visible nor internal) emanating from s. In the remaining case (where there is at least one visible immediate transition, but no internal immediate transition emanating from s) the state is called inconclusive.

Since compositionally vanishing states are left as soon as they are entered (via an internal immediate transition, which cannot be prevented from the environment), the idea is to eliminate them, in order to reduce the state space. The basic strategy of elimination is to redirect transitions leading to a compositionally vanishing state to its successor state, as shown in Fig. 5.8. In the case where a compositionally vanishing state has more than one outgoing internal immediate transitions, it is not specified which of them will be taken. This is an instance of non-determinism. In order to resolve such non-determinism, one may assign probabilities or weights to internal immediate transitions, as exemplified by the two transitions emanating from state 2 in the ESLTS shown in Fig. 5.9 (left), where the probabilities p and $q = 1 - p$ have been assigned to those two immediate transitions in order to resolve the non-deterministic choice between them. Transitions leading to the compositionally vanishing state can then be redirected to its successor states, taking into account the probabilities, as shown in Fig. 5.9 (right). Note in the figure how the probabilities of the two internal immediate transitions are 'inherited' by the redirected transitions.

The following is a sketch of a general algorithm for eliminating the compositionally vanishing states of an ESLTS:

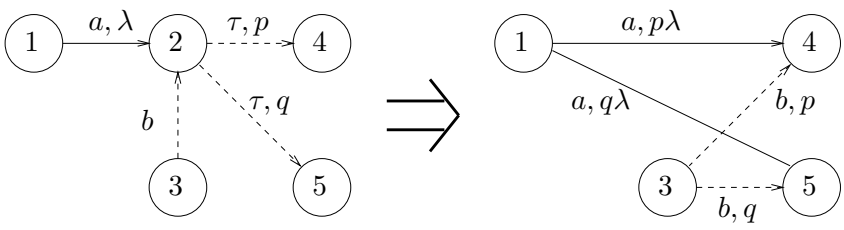


Figure 5.9: Resolving non-determinism by probabilities

1. Identify the compositionally vanishing states. For those compositionally vanishing states which have more than one emanating $\stackrel{\tau}{\dashrightarrow}$ -transition, assign probabilities to those transitions[4].

2. Delete Markovian transitions emanating from those states which have at least one outgoing $\stackrel{\tau}{\dashrightarrow}$ -transition[5] (since these Markovian transitions would never be taken).

3. Step 2 may have rendered some states unreachable. Determine the unreachable states and delete them and all transitions (regardless of their type) emanating from them. This is an optional step.

4. While there are still compositionally vanishing states, select one of them (let it be called state y) and do the following: Redirect transitions leading to y (regardless of their type) to the successor states of y, thereby taking into account the probabilities. (More precisely, if $x \xrightarrow{a,\lambda} y$ and $y \stackrel{\tau,p}{\dashrightarrow} z$ then modify the former transition as $x \xrightarrow{a,p\lambda} z$; if $x \stackrel{a,q}{\dashrightarrow} y$ and $y \stackrel{\tau,p}{\dashrightarrow} z$ then modify the former transition as $x \stackrel{a,pq}{\dashrightarrow} z$.) Afterwards, delete y and all transitions emanating from it. This step may lead to the existence of immediate loops of the kind $x \stackrel{\tau,pq}{\dashrightarrow} x$. Such loops can be eliminated by deleting them and multiplying all other $\stackrel{\tau}{\dashrightarrow}$ -transitions emanating from x with the factor $1/(1-pq)$.

We emphasise that this algorithm is suitable for a symbolic realisation, i.e. it can be implemented, for example, with the help of MTBDDs and the operations thereon (as realised in the tool IM-CAT). To begin with, consider step 1 of the algorithm. Suppose that M^I is the (MT)BDD which encodes all immediate

[4]These probabilities may be specified explicitly by the modeller. A software tool may also provide the default option to assign equal probabilities, as is indeed the case with the tool IM-CAT [128, 129].

[5]Note that this condition does not only apply to compositionally vanishing states, but also to vanishing states.

transitions of an ESLTS. The following symbolic computation determines a BDD S^{τ} encoding those states which have emanating $\stackrel{\tau}{\dashrightarrow}$ -transitions,

$$\mathsf{S}^{\tau}(\vec{\mathsf{s}}) = \textsc{Abstract}\left((\mathsf{M}^{I} \wedge \mathcal{M}(\vec{\mathsf{a}}; \mathcal{E}(\tau))), (\vec{\mathsf{a}}, \vec{\mathsf{t}}), \vee\right)$$

The conjunction $\mathsf{M}^{I} \wedge \mathcal{M}(\vec{\mathsf{a}}; \mathcal{E}(\tau))$ selects the τ-transitions, and the Abstract-operation abstracts from the Boolean variables encoding the action type and the target state of a transition, which yields a (MT)BDD that encodes only the source state of immediate τ-transitions. Similarly, states which have emanating immediate transitions labelled with actions other than τ are determined by

$$\mathsf{S}^{O}(\vec{\mathsf{s}}) = \textsc{Abstract}\left((\mathsf{M}^{I} \wedge \neg\mathcal{M}(\vec{\mathsf{a}}; \mathcal{E}(\tau)), (\vec{\mathsf{a}}, \vec{\mathsf{t}}), \vee\right)$$

The BDD S^{V} encoding the set of compositionally vanishing states can then be computed as

$$\mathsf{S}^{V}(\vec{\mathsf{s}}) = \mathsf{S}^{\tau}(\vec{\mathsf{s}}) \wedge \overline{\mathsf{S}^{O}(\vec{\mathsf{s}})}$$

The computation in step 2 of the algorithm is realised by

$$\mathsf{M}^{M_{relevant}}(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}}) = \mathsf{M}^{M}(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}}) \wedge \overline{\mathsf{S}^{\tau}(\vec{\mathsf{s}})}$$

i.e. only those Markovian transitions are selected, which do not originate in states that also have emanating $\stackrel{\tau}{\dashrightarrow}$ -transitions. Symbolic reachability analysis, needed for step 3, has already been discussed in Section 5.1.5. The symbolic realisation of step 4 is the most complicated. We only sketch how it can be done: One first computes

$$\mathsf{M}^{I,\tau}(\vec{\mathsf{t}}, \vec{\mathsf{u}}) = \left(\textsc{Restrict}(\mathsf{M}^{I}(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}}) \wedge \mathsf{S}^{V}(\vec{\mathsf{s}}), \vec{\mathsf{a}}, \mathcal{E}(\tau))\right) \{\vec{\mathsf{u}} \leftarrow \vec{\mathsf{t}}, \vec{\mathsf{t}} \leftarrow \vec{\mathsf{s}}\}$$

which is the MTBDD representing the internal immediate transitions emanating from compositionally vanishing states. The renaming of Boolean variables $\{\vec{\mathsf{u}} \leftarrow \vec{\mathsf{t}}, \vec{\mathsf{t}} \leftarrow \vec{\mathsf{s}}\}$ has the effect that the source state is encoded by Boolean variables t_i and the target state by u_i. Redirecting Markovian transitions leading to compositionally vanishing states is then achieved simply by

$$\mathsf{M}^{M_{redir}}(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{u}}) = \textsc{Abstract}(\mathsf{M}^{M_{relevant}}(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}}) \wedge \mathsf{M}^{I,\tau}(\vec{\mathsf{t}}, \vec{\mathsf{u}}), \vec{\mathsf{t}}, +)$$

The redirection of immediate transitions leading to compositionally vanishing states is done in a similar way, namely by computing

$$\mathsf{M}^{I_{redir}}(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{u}}) = \textsc{Abstract}(\mathsf{M}^{I}(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}}) \wedge \mathsf{M}^{I,\tau}(\vec{\mathsf{t}}, \vec{\mathsf{u}}), \vec{\mathsf{t}}, +)$$

It should be emphasised, that the hiding of actions and the symbolic elimination of compositionally vanishing states can be employed as part of a compositional model construction procedure. Starting from small size submodels, these can be

composed in parallel, where synchronisation takes place on some actions. After each composition step, the actions which will not be needed for further synchronisation can be hidden. The compositionally vanishing states which may have resulted from this hiding can then be eliminated as discussed, before the current submodel is composed further with other submodels.

At this point, we have to mention an interesting phenomenon concerning the size of the symbolic data structures. Although the transition system which results from the elimination of compositionally vanishing states has fewer states (and fewer transitions) than the original transition system, its symbolic representation is typically less compact than the original one, i.e. typically we have $|\mathsf{M}^{I_{redir}}(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}})| > |\mathsf{M}^{I}(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}})|$ and $|\mathsf{M}^{M_{redir}}(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}})| > |\mathsf{M}^{M}(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}})|$. The increase of the size of the symbolic representation is generally due to some loss of regularity. We shall take up this issue again in Sec. 5.3.5, and also in the context of the application case studies in Chap. 10.

Concerning the point in time when non-determinism is resolved, we wish to point out the following: The implementation in the tool IM-CAT offers the possibility to detect and resolve non-determinism at any point of time. Therefore the decision is left to the modeller, whether to resolve non-determinism as soon as possible (i.e. before further parallel composition), or to postpone the resolution (and the necessary assignment of probabilities to the non-deterministic internal immediate transitions) until the current system is composed further. Contrary to this, the SPA tool TIPPTOOL does not offer such a flexibility. It always builds the complete model (thereby obeying a maximal progress assumption which says that Markovian transitions emanating from states with emanating $\stackrel{\tau}{\dashrightarrow}$ -transitions can be disregarded [162]). When constructing a CTMC, which is needed for performance evaluation, TIPPTOOL automatically hides all immediate transitions and in all cases of non-deterministic choice assigns equal probabilities.

5.3 Symbolic bisimulation and state space minimisation

As we have seen in Sec. 3.7, state space reduction based on bisimulation equivalences is an important concept when working with stochastic process algebra models. The basic idea is to reduce the state space by representing states which are equivalent (in the sense of a given bisimulation relation) by a single macro state. In particular, if the bisimulation is a congruence, reduction may be applied in a compositional manner by reducing intermediate models after every

construction step, before further composition is carried out.

In this section, we discuss how bisimulation algorithms can be realised symbolically, using BDDs and their extensions as the underlying data structures. We describe bisimulation algorithms which are entirely based on BDD operations, both for non-stochastic and for stochastic scenarios.

Before we actually discuss symbolic bisimulation algorithms, we mention some related work (all of which concerns bisimulation in non-stochastic systems). A symbolic minimisation algorithm for networks of parallel processes has been described in [39]. The algorithm, which is implemented in a prototype tool, is capable of calculating strong and weak bisimulation relations and works according to the principle of iterative refinement. In [65], symbolic model checking for a powerful version of the mu-calculus is described. Both strong and weak equivalence are expressed in this calculus, and thus an efficient decision procedure for these equivalences is provided. Likewise, [118] also considers symbolic bisimilarity checking, but this paper focuses on BDD construction from elementary finite transition systems by applying CCS operations.

5.3.1 Symbolic non-stochastic bisimulation

We now describe how the basic non-stochastic bisimulation algorithm introduced in Sec. 3.7.3 can be realised using BDD-based data structures. For convenience, let the transition relation be represented not by a single BDD $\mathsf{B}(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}})$, but by a set of BDDs $\mathsf{B}_a(\vec{\mathsf{s}}, \vec{\mathsf{t}})$, one for each action label a. These BDDs can be easily obtained by restricting $\mathsf{B}(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}})$ as follows:

$$\mathsf{B}_a(\vec{\mathsf{s}}, \vec{\mathsf{t}}) = \textsc{Restrict}(\mathsf{B}(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}}), \vec{\mathsf{a}}, \mathcal{E}(a))$$

The current partition $\{C_1, C_2, \ldots\}$ is stored as a set of BDDs $\{\mathsf{C}_1(\vec{\mathsf{s}}), \mathsf{C}_2(\vec{\mathsf{s}}), \ldots\}$, one for each equivalence class. The dynamic set of splitters, *Splitters*, whose elements are pairs (a, C) consisting of an action a and a class C, can be realised by a pointer structure as shown in Fig. 5.10, i.e. for each action a there is a linked list of pointers to the roots of those BDDs representing the relevant classes.

In procedure *split*, when splitting of class C takes place, the subclasses C^+ and C^- are also represented by BDDs, namely $\mathsf{C}^+(\vec{\mathsf{s}})$ and $\mathsf{C}^-(\vec{\mathsf{s}})$. The core operation, i.e. the computation of the subclass C^+ in procedure *split* is carried out with the help of conjunction and existential quantification

$$\mathsf{C}^+(\vec{\mathsf{s}}) := \mathsf{C}(\vec{\mathsf{s}}) \ \wedge \ \exists\, \vec{\mathsf{t}} : \ \big(\mathsf{B}_a(\vec{\mathsf{s}}, \vec{\mathsf{t}}) \wedge \mathsf{C}_{spl}(\vec{\mathsf{t}})\big)$$

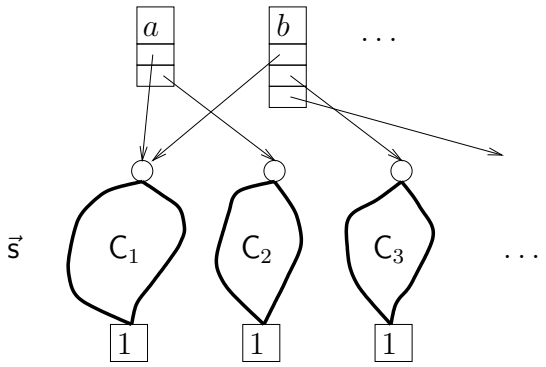


Figure 5.10: Pointer structure which realises the dynamic set of splitters

(note that existential quantification is also performed as a BDD operation[6]). The subsequent check whether class C actually needs to be split is decided based on the Boolean expression

$$\mathsf{C}^+(\vec{s}) \neq \mathsf{C}(\vec{s}) \ \wedge \ \mathsf{C}^+(\vec{s}) \neq 0$$

which can be checked in constant time since the equivalence check on BDDs only takes constant time. The computation of $\mathsf{C}^-(\vec{s})$ is achieved by

$$\mathsf{C}^-(\vec{s}) := \mathsf{C}(\vec{s}) \wedge \overline{\mathsf{C}^+(\vec{s})}$$

Having pointed out these details, we now give a symbolic version of procedure *split*:

procedure $split(\mathsf{C}, a, \mathsf{C}_{spl}, Partition, Splitters)$
  $\mathsf{C}^+(\vec{s}) := \mathsf{C}(\vec{s}) \wedge \exists\, \vec{t}: \ \big(\mathsf{B}_a(\vec{s},\vec{t}) \wedge \mathsf{C}_{spl}(\vec{t})\big)$
  /* the subclass $\mathsf{C}^+(\vec{s})$ is computed */
  if $(\mathsf{C}^+(\vec{s}) \neq \mathsf{C}(\vec{s}) \ \wedge \ \mathsf{C}^+(\vec{s}) \neq 0)$
    /* only continue if class $\mathsf{C}(\vec{s})$ actually needs to be split */
    $\mathsf{C}^-(\vec{s}) := \mathsf{C}(\vec{s}) \wedge \overline{\mathsf{C}^+(\vec{s})}$
    /* $\mathsf{C}^-(\vec{s})$ is the complement of $\mathsf{C}^+(\vec{s})$ with respect to $\mathsf{C}(\vec{s})$ */
    $Partition := Partition \cup \{\mathsf{C}^+(\vec{s}), \mathsf{C}^-(\vec{s})\} - \{\mathsf{C}(\vec{s})\}$
    $Splitters := Splitters \cup (Act \times \{\mathsf{C}^+(\vec{s}), \mathsf{C}^-(\vec{s})\}) - Act \times \{\mathsf{C}(\vec{s})\}$
    /* the partition and the set of splitters are updated */

For weak bisimulation, the algorithm is similar, but the weak transition relation must be calculated first (which, as already mentioned in Sec. 3.7.3, is the

[6]Existential quantification of Boolean functions is defined in terms of disjunction of cofactors (i.e. as a special instance of abstraction) as $\exists \mathsf{v}_i: \ f(\ldots,\mathsf{v}_i,\ldots) := f(\ldots,0,\ldots) \vee f(\ldots,1,\ldots)$.

computationally most expensive part). We now sketch how this can be done symbolically.

The first step is to calculate the transitive closure $\mathsf{B}_\tau^*(\vec{s},\vec{t})$ of τ-transitions, i.e. the transitive closure of internal transitions. Naïvely this can be done by computing a series of relations

$$\begin{array}{lcl} \mathsf{B}_\tau^{(0)}(\vec{s},\vec{t}) & := & \mathsf{B}_\tau(\vec{s},\vec{t}) \ \vee \ (\vec{s}=\vec{t}) \\ \mathsf{B}_\tau^{(i+1)}(\vec{s},\vec{t}) & := & \mathsf{B}_\tau^{(i)}(\vec{s},\vec{t}) \ \vee \ \exists \vec{u} : \Big(\mathsf{B}_\tau^{(i)}(\vec{s},\vec{u}) \wedge \mathsf{B}_\tau(\vec{u},\vec{t})\Big) \end{array}$$

until convergence, i.e. until $\mathsf{B}_\tau^{(i)}(\vec{s},\vec{t}) = \mathsf{B}_\tau^{(i+1)}(\vec{s},\vec{t})$, in which case we have $\mathsf{B}_\tau^{(i+1)}(\vec{s},\vec{t}) = \mathsf{B}_\tau^*(\vec{s},\vec{t})$. In order to reach the fixed point in fewer steps, the method of iterative squaring [317] may be employed, which means that $\mathsf{B}_\tau^{(i+1)}(\vec{s},\vec{t})$ is computed by $\mathsf{B}_\tau^{(i)}(\vec{s},\vec{t}) \ \vee \ \exists \vec{u} : \Big(\mathsf{B}_\tau^{(i)}(\vec{s},\vec{u}) \wedge \mathsf{B}_\tau^{(i)}(\vec{u},\vec{t})\Big)$ instead of the above expression (note the subtle difference). However, as already observed in [65], iterative squaring may not be beneficial in practice if the BDDs needed to represent the intermediate relations become too large.

The second step consists of the actual calculation of the BDDs $\mathsf{B}_{a,weak}(\vec{s},\vec{t})$ for all action labels a, encoding weak transitions $\stackrel{a}{\Longrightarrow}$ (defined by $\stackrel{\tau}{\Longrightarrow}\stackrel{a}{\longrightarrow}\stackrel{\tau}{\Longrightarrow}$). This can be achieved by conjunction and existential quantification:

$$\mathsf{B}_{a,weak}(\vec{s},\vec{t}) := \exists \vec{u} : \exists \vec{v} : \big(\mathsf{B}_\tau^*(\vec{s},\vec{u}) \wedge \mathsf{B}_a(\vec{u},\vec{v}) \wedge \mathsf{B}_\tau^*(\vec{v},\vec{t})\big)$$

Note that this requires renaming of Boolean variables within BDDs B_τ^* (twice!) and B_a. This renaming does not cause any problems if the overall variable ordering, including the additional variables $\vec{u}$ and $\vec{v}$, is as follows:

$$\mathsf{s}_1 \prec \mathsf{u}_1 \prec \mathsf{v}_1 \prec \mathsf{t}_1 \prec \ldots \prec \mathsf{s}_{n_s} \prec \mathsf{u}_{n_s} \prec \mathsf{v}_{n_s} \prec \mathsf{t}_{n_s}$$

5.3.2 Symbolic Markovian bisimulation

We now discuss aspects of a symbolic algorithm which computes Markovian bisimulation on SLTSs. Concerning the use of symbolic data structures, we basically proceed as in Sec 5.3.1, i.e. the current partition is stored as a set of BDDs $\{\mathsf{C}_1(\vec{s}),\mathsf{C}_2(\vec{s}),\ldots\}$, and the realisation of the dynamic set of splitters is by a pointer structure similar to the one shown in Fig. 5.10.

As we have seen in Sec. 4.3.2 and Sec. 4.4.2, the transition relation of the SLTS — now including information about the transition rates — can be represented by a DNBDD $\mathsf{D}(\vec{a},\vec{s},\vec{t})$ or by an MTBDD $\mathsf{M}(\vec{a},\vec{s},\vec{t})$. Since the bisimulation algorithms

for these two data structures proceed in a similar way, we will restrict our further discussion to the MTBDD case only. As in the previous section, the decision diagram encoding the transition relation is again broken up into several parts, such that there is one MTBDD per action label, denoted by $\mathsf{M}_a(\vec{\mathsf{s}}, \vec{\mathsf{t}})$.

The basic algorithm for Markovian bisimulation is as in Sec. 3.7.4, but some essential modifications are necessary within procedure *split'*. An important issue within procedure *split'* is the calculation of the cumulative rate $\gamma(P, a, C_{spl})$ of a-transitions from an individual state P (being a member of the class C to be split) to class C_{spl}. Using non-symbolic data structures, these calculations must be carried out in a state by state manner. In contrast to that, using MTBDDs, the calculation can be performed at the same time for all states of a given class, involving only a few basic MTBDD operations. This is done as follows: In a first step, an MTBDD, encoding all a-transitions which originate in C and lead to C_{spl}, is calculated by selecting the appropriate transitions from $\mathsf{M}_a(\vec{\mathsf{s}}, \vec{\mathsf{t}})$:[7]

$$\mathsf{M}_{C \xrightarrow{a} C_{spl}}(\vec{\mathsf{s}}, \vec{\mathsf{t}}) := \mathsf{C}(\vec{\mathsf{s}}) \wedge \mathsf{M}_a(\vec{\mathsf{s}}, \vec{\mathsf{t}}) \wedge \mathsf{C}_{spl}(\vec{\mathsf{t}})$$

The cumulative rates are now computed for all states $P \in C$ by a single abstraction operation:

$$\mathsf{M}_{cum\_rates}(\vec{\mathsf{s}}) := \textsc{Abstract}(\mathsf{M}_{C \xrightarrow{a} C_{spl}}(\vec{\mathsf{s}}, \vec{\mathsf{t}}), \vec{\mathsf{t}}, +)$$

The resulting MTBDD $\mathsf{M}_{cum\_rates}(\vec{\mathsf{s}})$, a sketch of which is shown in Fig. 5.11 (top), encodes a real-valued function, which maps the encoding $\mathcal{E}(P)$ of a state P to its cumulative rate. Formally this can be written as

$$f_{\mathsf{M}_{cum\_rates}}\Big|_{\vec{\mathsf{s}}=\mathcal{E}(P)} = \gamma(P, a, C_{spl})$$

The information contained in $\mathsf{M}_{cum\_rates}(\vec{\mathsf{s}})$ is almost the same as the information that was contained in the split-tree in the non-symbolic version of procedure *split'* (one can view $\mathsf{M}_{cum\_rates}(\vec{\mathsf{s}})$ as an upside-down version of the *split_tree*). Therefore, the split-tree as such is not needed in the symbolic algorithm. Instead, subclasses are extracted directly from $\mathsf{M}_{cum\_rates}(\vec{\mathsf{s}})$ by a procedure called *extract_subclasses*(), which generates from $\mathsf{M}_{cum\_rates}(\vec{\mathsf{s}})$ a finite number of BDDs, denoted by $\mathsf{C}_{\gamma_1}(\vec{\mathsf{s}}), \ldots, \mathsf{C}_{\gamma_k}(\vec{\mathsf{s}})$, one for each nonzero terminal vertex of the MTBDD. The effect of procedure *extract_subclasses*() is depicted in the lower part of Fig. 5.11.

The symbolic version of procedure *split'* can now be formulated as follows:

[7] Here and in some other instances we use the conjunction operator $\wedge$ to combine a BDD with an MTBDD. Strictly speaking, the BDD argument should be considered as an MTBDD and the multiplication operator $\cdot$ should be used.

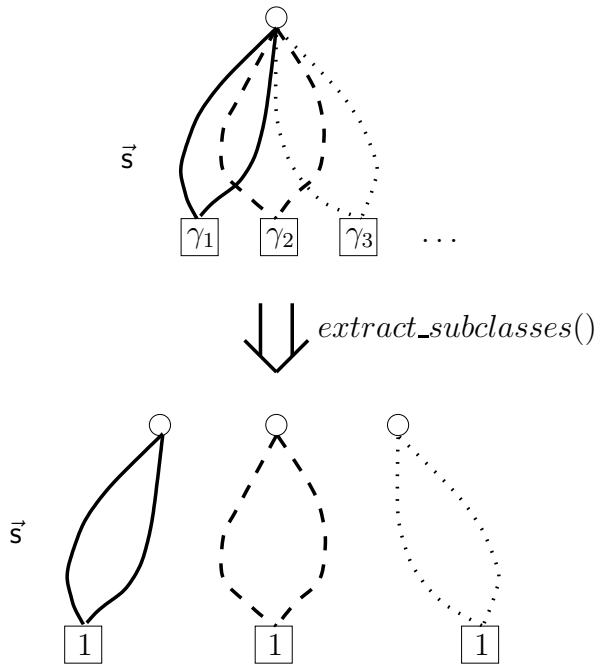


Figure 5.11: Sketch of MTBDD $\mathsf{M}_{cum\_rates}(\vec{s})$ (top), and effect of procedure *extract_subclasses*()

procedure $split'(\mathsf{C}(\vec{s}), a, \mathsf{C}_{spl}(\vec{t}), Partition, Splitters)$
$\mathsf{M}_{C \xrightarrow{a} C_{spl}}(\vec{s},\vec{t}) := \mathsf{C}(\vec{s}) \wedge \mathsf{M}_a(\vec{s},\vec{t}) \wedge \mathsf{C}_{spl}(\vec{t})$
/* all a-transitions from $\mathsf{C}(\vec{s})$ to $\mathsf{C}_{spl}(\vec{t})$ are selected */
$\mathsf{M}_{cum\_rates}(\vec{s}) :=$ ABSTRACT$(\mathsf{M}_{C \xrightarrow{a} C_{spl}}(\vec{s},\vec{t}), \vec{t}, +)$
/* all cumulative rates are computed by a single abstraction */
extract_subclasses$(\mathsf{M}_{cum\_rates}(\vec{s}))$
/* k BDDs, corresponding to the subclasses, are extracted from $\mathsf{M}_{cum\_rates}(\vec{s})$ */
if $(k > 1)$
/* only continue if $\mathsf{C}(\vec{s})$ has been split into $k > 1$ subclasses */
$Partition := Partition \cup \{\mathsf{C}_{\gamma_1}(\vec{s}), \ldots, \mathsf{C}_{\gamma_k}(\vec{s})\} - \{\mathsf{C}(\vec{s})\}$
$Splitters := Splitters \cup (Act \times \{\mathsf{C}_{\gamma_1}(\vec{s}), \ldots, \mathsf{C}_{\gamma_k}(\vec{s})\}) - Act \times \{\mathsf{C}(\vec{s})\}$
/* the partition and the splitter set are updated */

We mention at this point that our prototype tool DNBDDTOOL [43], a tool for SLTS construction and analysis based on DNBDDs, contains an implementation of Markovian bisimulation whose basic strategy, however, follows more closely the non-symbolic algorithm given in Sec. 3.7.4. Our MTBDD-based tool IMCAT does currently not include bisimulation algorithms, since so far its focus has been more on compositional model construction, handling of ESLTSs, and numerical analysis.

5.3.3 Symbolic weak Markovian bisimulation

In this section we consider a symbolic algorithm that realises weak Markovian bisimulation for ESLTSs. Remember that an ESLTS, comprising both immediate transitions $\xrightarrow{a}$ and Markovian transitions $\xrightarrow{a,\lambda}$, can be represented by means of two separate data structures: An (MT)BDD M^I to encode immediate transitions and an MTBDD M^M to encode Markovian transitions[8]. (Of course, one could also work with DNBDDs instead of MTBDDs.) For simplicity, as in Sec. 3.7.5, we assume that the ESLTS is divergence-free.

As a preprocessing step, the BDDs encoding the transitive closure of immediate τ-transitions, encoded by $\mathsf{B}^*_\tau(\vec{\mathsf{s}},\vec{\mathsf{t}})$, and the weak transition relation $\stackrel{a}{\Longrightarrow}$, encoded by a set of BDDs $\mathsf{B}_{a,weak}(\vec{\mathsf{s}},\vec{\mathsf{t}})$, must be calculated. This can be done symbolically as described in Sec. 5.3.1.

The basic bisimulation algorithm is the same as in Sec. 3.7.5, where in the main loop the following is done for each splitter: Firstly, all classes are split by procedure *split* with respect to weak transitions. Since we already discussed a symbolic version of *split* in Sec. 5.3.1, we do not have to describe this step again. Secondly, all classes are split with respect to Markovian transitions by means of procedure *split″*. So it remains to discuss a symbolic version of procedure *split″*:

Since procedure *split″* distinguishes between vanishing[9] and tangible states, two BDDs encoding these sets have to be calculated:

$$\begin{aligned}\mathsf{Van}(\vec{\mathsf{s}}) &:= \exists\vec{\mathsf{t}} : \mathsf{M}^I_\tau(\vec{\mathsf{s}},\vec{\mathsf{t}})\\ \mathsf{Tan}(\vec{\mathsf{s}}) &:= \overline{\mathsf{Van}(\vec{\mathsf{s}})}\end{aligned}$$

Since BDDs $\mathsf{Van}(\vec{\mathsf{s}})$ and $\mathsf{Tan}(\vec{\mathsf{s}})$ are the same in every invocation of procedure *split″*, it is best to carry out these calculations during the initialisation of the algorithm.

At the beginning of procedure *split″*, tangible states of class C are classified according to their cumulative rates in a similar way as in procedure *split′*. The result of this step is an MTBDD $\mathsf{M}^{Tan}_{cum\_rates}(\vec{\mathsf{s}})$, whose terminal vertices carry the values $\gamma_1,\ldots,\gamma_k$. The tricky part is then to associate vanishing states with

[8]Here, contrary to Sec. 5.2.2, we are not interested in resolving non-determinism by probabilities, but follow the concept of weak Markovian bisimulation as in Sec. 3.7.5. Therefore, M^I is a BDD and not an MTBDD.

[9]In this section we are concerned with bisimulation and not with the elimination of instable states as in Sec. 5.2.2. Therefore we return to the original notions of vanishing and tangible states as defined in Def. 3.7.8, as opposed to the notion of compositionally vanishing states as defined in Def. 5.2.1.

cumulative rates. Each vanishing state must either be associated with one of the cumulative rates γ_j which were found while processing the tangible states, or — if the cumulative rate is ambiguous — must be associated with the undefined cumulative rate $\perp$. In order to support this decision, we define a binary operator ∇ on the set of reals. For $r_1, r_2 \in I\!R$, let

$$r_1 \nabla r_2 := \begin{cases} r_1 & \text{if } r_1 = r_2 \\ \perp & \text{otherwise} \end{cases}$$

i.e. if both values agree then ∇ returns that value, otherwise undefined $\perp$ is returned. This ∇-operator is used within an ABSTRACT-operation which essentially classifies vanishing states according to the cumulative rate of those tangible states which they can reach via internal immediate transitions. The result of this abstraction is an MTBDD $\mathsf{M}^{Van}_{cum\_rates}(\vec{\mathsf{s}})$ whose terminal values are from the set $\{\gamma_1, \ldots, \gamma_{k+1}\}$, where $\gamma_{k+1} = \perp$.

Once the cumulative rates have been determined both for the tangible and the vanishing states of class C, the information contained in $\mathsf{M}^{Tan}_{cum\_rates}(\vec{\mathsf{s}})$ and $\mathsf{M}^{Van}_{cum\_rates}(\vec{\mathsf{s}})$ is merged into a single MTBDD $\mathsf{M}_{cum\_rates}(\vec{\mathsf{s}})$, and (as in procedure *split′*) procedure *extract_subclasses*() is called in order to extract the BDDs encoding the subclasses. The rest of procedure *split″* is as in procedure *split′*. The complete symbolic version of procedure *split″* is as follows:

procedure $split''(\mathsf{C}(\vec{\mathsf{s}}), a, \mathsf{C}_{spl}(\vec{\mathsf{t}}), Partition, Splitters)$

/* first process tangible states: */
$\mathsf{M}^{Tan}_{C \xrightarrow{a} C_{spl}}(\vec{\mathsf{s}}, \vec{\mathsf{t}}) := \mathsf{C}(\vec{\mathsf{s}}) \wedge \mathsf{Tan}(\vec{\mathsf{s}}) \wedge \mathsf{M}^{M}_{a}(\vec{\mathsf{s}}, \vec{\mathsf{t}}) \wedge \mathsf{C}_{spl}(\vec{\mathsf{t}})$
/* all a-transitions from tangible states within $\mathsf{C}(\vec{\mathsf{s}})$ to $\mathsf{C}_{spl}(\vec{\mathsf{t}})$ are selected */
$\mathsf{M}^{Tan}_{cum\_rates}(\vec{\mathsf{s}}) :=$ ABSTRACT$(\mathsf{M}^{Tan}_{C \xrightarrow{a} C_{spl}}(\vec{\mathsf{s}}, \vec{\mathsf{t}}), \vec{\mathsf{t}}, +)$
/* $k \geq 1$ distinct cumulative rates for the tangible states are computed */

/* next process vanishing states: */
$\mathsf{B}_{Van \to Tan}(\vec{\mathsf{s}}, \vec{\mathsf{t}}) := \mathsf{C}(\vec{\mathsf{s}}) \wedge \mathsf{Van}(\vec{\mathsf{s}}) \wedge \mathsf{B}^{*}_{\tau}(\vec{\mathsf{s}}, \vec{\mathsf{t}}) \wedge \mathsf{Tan}(\vec{\mathsf{t}})$
/* weak transitions from vanishing to tangible states are computed */
$\mathsf{M}_{Van \to Tan}(\vec{\mathsf{s}}, \vec{\mathsf{t}}) := \mathsf{B}_{Van \to Tan}(\vec{\mathsf{s}}, \vec{\mathsf{t}}) \cdot \mathsf{M}^{Tan}_{cum\_rates}(\vec{\mathsf{t}})$
/* cumulative rate information is added to BDD $\mathsf{B}_{Van \to Tan}(\vec{\mathsf{s}}, \vec{\mathsf{t}})$ */
$\mathsf{M}^{Van}_{cum\_rates}(\vec{\mathsf{s}}) :=$ ABSTRACT$(\mathsf{M}_{Van \to Tan}(\vec{\mathsf{s}}, \vec{\mathsf{t}}), \vec{\mathsf{t}}, \nabla)$
/* vanishing states are classified with respect to "reachable" cumulative rates */

/* then combine both results in one MTBDD: */
$\mathsf{M}_{cum\_rates}(\vec{\mathsf{s}}) := \mathsf{M}^{Tan}_{cum\_rates}(\vec{\mathsf{s}}) + \mathsf{M}^{Van}_{cum\_rates}(\vec{\mathsf{s}})$

/* from hereon proceed as in procedure *split'*: */
$extract\_subclasses(\mathsf{M}_{cum\_rates}(\vec{\mathsf{s}}))$
/* $l = k$ (or $l = k + 1$) BDDs, corresponding to the subclasses, are extracted */
if $(l > 1)$
/* only continue if $\mathsf{C}(\vec{\mathsf{s}})$ has been split into $l > 1$ subclasses */
$Partition := Partition \cup \{\mathsf{C}_{\gamma_1}(\vec{\mathsf{s}}), \ldots, \mathsf{C}_{\gamma_l}(\vec{\mathsf{s}})\} - \{\mathsf{C}(\vec{\mathsf{s}})\}$
$Splitters := Splitters \cup (Act \times \{\mathsf{C}_{\gamma_1}(\vec{\mathsf{s}}), \ldots, \mathsf{C}_{\gamma_l}(\vec{\mathsf{s}})\}) - Act \times \{\mathsf{C}(\vec{\mathsf{s}})\}$
/* the partition and the splitter set are updated */

5.3.4 Constructing the minimised transition system

In the last three sections, we have described symbolic bisimulation algorithms for non-stochastic, Markovian and mixed transition systems. As a result of these algorithms, it is known which states are equivalent with respect to a certain bisimulation relation. The algorithms return this information in the form of m BDDs, denoted by $\mathsf{C}_1(\vec{\mathsf{s}}), \ldots, \mathsf{C}_m(\vec{\mathsf{s}})$, each of which encodes the states belonging to one of the m equivalence classes.

As already mentioned, one aim of bisimulation is to construct a minimised transition system where every class of equivalent states is represented by a single macro state. The state space of the minimised transition system is often dramatically smaller than the original one. In this section, we describe how the minimised transition system can be obtained, working on symbolic data structures.

Strong bisimulation: We consider first strong bisimulation for a given LTS $\mathcal{T}$. Let $\mathsf{B}(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}})$ be a BDD (using the standard interleaved variable ordering), such that $\mathsf{B} \triangleright \mathcal{T}$. Let $\mathsf{C}_1(\vec{\mathsf{s}}), \ldots, \mathsf{C}_m(\vec{\mathsf{s}})$ be m BDDs (calculated as the result of a symbolic bisimulation algorithm like the one described in Sec. 5.3.1), encoding the equivalence classes $C_1, \ldots, C_m$.

As a first step, from each equivalence class C_j, where $1 \leq j \leq m$, one state has to be chosen to be the representative state[10]. This choice could be made randomly or deterministically (following some heuristic strategy). For example, the lexicographically smallest (or largest) state within each equivalence class could be chosen. While all possible choices are functionally equivalent, it is clear that the choice will affect the size of the resulting BDD (in a way that is rather hard

[10]Note that we choose one of the states from equivalence class C_j to stand as representative for that class. We do not recommend to generate a new, "artificial" representative state per class, which could possibly be encoded by less than n_s bits, since our experiments with symbolic Markovian bisimulation in the tool DNBDDTOOL suggested that such a reencoding of macro states does not usually decrease the (DN)BDD size.

to predict in general). However, we assume that the choice has been made in some way or other and that $\{r_1, \ldots, r_m\}$ are the m representative states. We then construct

$$\mathsf{Rep}(\vec{\mathsf{s}}) := \mathcal{M}(\vec{\mathsf{s}}; \mathcal{E}(r_1)) \vee \ldots \vee \mathcal{M}(\vec{\mathsf{s}}; \mathcal{E}(r_m))$$

which is a BDD encoding the set of representative states.

As a second step, only the transitions emanating from representative states are selected from the overall transition relation (which is encoded by $\mathsf{B}(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}})$). Note that all other transitions may simply be dropped, since they only describe bisimilar behaviour. This selection can be realised by a single conjunction:

$$\mathsf{B}'(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}}) := \mathsf{B}(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}}) \wedge \mathsf{Rep}(\vec{\mathsf{s}})$$

The following third step is the most complicated and the most expensive one: BDD $\mathsf{B}'(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}})$ encodes transitions which emanate from representative states, but which may lead to any state of the original state space. Therefore, within $\mathsf{B}'(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}})$ the target states have to be replaced by the corresponding representative states. This is done as follows:

$$\mathsf{B}^{red}(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}}) := \bigvee_{j=1}^{m} \left((\exists \vec{\mathsf{t}} : (\mathsf{B}'(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}}) \wedge \mathsf{C}_j(\vec{\mathsf{t}}))) \wedge \mathcal{M}(\vec{\mathsf{t}}; \mathcal{E}(r_j)) \right)$$

The inner conjunction selects transitions leading to class C_j, and the existential quantification then abstracts from the original target state (i.e. altogether deletes the information on the target state). The outer conjunction sets the target state to be the representative state of class C_j (i.e. state r_j, encoded by the minterm $\mathcal{M}(\vec{\mathsf{t}}; \mathcal{E}(r_j))$. Note that this third step requires $m - 1$ disjunctions at the outer level, where m can be very large in practice.

Weak bisimulation: For non-stochastic *weak* bisimulation, the construction of the minimised transition system works in a similar way as for strong bisimulation, but the weak transition relation (encoded by BDD $\mathsf{B}_{weak}(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}})$) must be employed instead of the original transition relation (encoded by BDD $\mathsf{B}(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}})$).

Markovian bisimulation: In the case of Markovian bisimulation on an SLTS, whose transition relation is represented by MTBDD $\mathsf{M}(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}})$, the basic procedure for constructing the minimised transition system is also similar. First, an MTBDD representing transitions emanating from representative states is calculated as

$$\mathsf{M}'(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}}) := \mathsf{M}(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}}) \wedge \mathsf{Rep}(\vec{\mathsf{s}})$$

and then the target states are replaced by the corresponding representative states as follows:

$$\mathsf{M}^{red}(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}}) := \sum_{j=1}^{m} \left(\textsc{Abstract}(\mathsf{M}'(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}}) \wedge \mathsf{C}_j(\vec{\mathsf{t}}), \vec{\mathsf{t}}, +) \wedge \mathcal{M}(\vec{\mathsf{t}}; \mathcal{E}(r_j)) \right)$$

i.e. abstraction is now used instead of existential quantification, and at the outer level summation is used instead of disjunction. Note how this abstraction operation, in addition to deleting the information on the target state, includes the calculation of the cumulative rate.

Weak Markovian bisimulation: When constructing the minimised transition system for an ESLTS (from the known equivalence classes of weak Markovian bisimulation), one can follow the same basic strategy as in the previous cases, but some fine issues must be considered. The first issue concerns the choice of the representative states: For those equivalence classes which contain at least one tangible state, one of the tangible states should be chosen as the representative state (in order to make sure that the Markovian transitions are not lost when selecting the transitions emanating from the representative state). For classes which do not contain any tangible state, any state may be chosen as the representative state. The BDD encoding the weak transition relation of the minimised transition system, $\mathsf{B}^{red}_{weak}(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}})$, is obtained by proceeding as described above for weak bisimulation in the non-stochastic case. The MTBDD encoding the Markovian transition relation of the minimised transition system, $\mathsf{M}^{red}(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}})$, is obtained by proceeding as described above for Markovian bisimulation in the SLTS case.

5.3.5 The role of symbolic state space reduction

The basic complexity of the described bisimulation algorithms remains the same when moving from an explicit to a symbolic representation of the transition relation. In the case of procedure *split′*, where the cumulative rates have to be calculated for all states of a class, we have seen that a single MTBDD abstraction operation suffices to carry out this calculation for all states simultaneously. In many cases, this will be substantially faster than the non-symbolic algorithm, where the calculation of the cumulative rate is carried out in a state-by-state fashion. A similar argument applies to procedure *split″*, both concerning the calculation of the cumulative rates for the tangible states, and concerning the classification of the vanishing states.

Together with BDD-based compositional state space construction, as discussed in Sec. 5.1, BDD-based state space reduction algorithms realise the concept of compositional reduction in a totally symbolic manner. The advantages for performance analysis seem to be obvious, since the transition system of a complex system can be built from small components by applying the BDD-based parallel composition operator step by step, where after every parallel composition step the intermediate model can be minimised without leaving the BDD world. This procedure guarantees that the size of the state space is kept at a minimum at any

time. Joined with BDD-based algorithms for numerical analysis (cf. Chap. 7), this yields a totally BDD-based framework for the construction and analysis of stochastic models.

Note, however, that unfortunately a reduction of the underlying state space does not usually coincide with a reduction of the BDD sizes. In practice, the opposite is often the case, i.e. one typically observes that BDD sizes grow when the state space is reduced. At a first glance this seems to be totally counter-intuitive. However, remember that a similar phenomenon had been observed when we discussed reachability analysis in Sec. 5.1.5, where it was found that the symbolic representation of a transitions system with unreachable transitions is often smaller than the representation of the reachable part. As a further example for such a counter-intuitive behaviour, we saw that symbolic elimination of compositionally vanishing states (discussed in Sec. 5.2.2), which also reduces the state space of the underlying ESLTS, often increases the BDD size. Generally speaking, we argue that in all these cases the increase of the BDD size is due to the same basic reasons, which may be stated as follows: Symbolic representations work best if they are constructed in a compositional fashion, such that the underlying transition system has a lot of regularity (and possibly redundancy). Deleting certain transitions, redirecting certain transitions, or deleting certain states (all states of an equivalence class, except the representative state), are all operations which to a certain extent destroy the regularity of the transition system and therefore diminish the efficiency (i.e. the compactness) of the symbolic representation.

Therefore, we have to ask the question whether symbolic bisimulation is really useful, and whether symbolic reachability analysis and symbolic elimination of compositionally vanishing states are useful. It is obvious that one cannot blindly recommend to replace a transition system by its minimised version when using BDDs as the underlying data structure. However, we argue that it is certainly advantageous to have efficient symbolic bisimulation algorithms at one's disposal, which can provide users with valuable information about the equivalence of states, even if in many cases one will prefer to work with the symbolic representation of the non-minimised transitions system instead of the minimised one.

Chapter 6

Compact encodings

In this chapter, we will discuss issues related to the size of a decision diagram. This discussion will confirm the important role of symbolic parallel composition and emphasise the implications of our findings concerning the size of BDDs resulting from parallel composition, as stated in Thm. 5.1.2 and Thm. 5.1.4.

The size of BDDs has been the object of study in several published works: In [46] and [243], lower and upper bounds for the size of BDDs representing digital circuits are given. In these works, the structure of the circuits, which consist of "blocks" with limited interaction, plays a key role for determining the bounds. As already mentioned in Chap. 5, [118] developed bounds for the size of BDDs constructed from CCS terms. Here, again, the structure of the underlying model, i.e. the way the CCS process is composed of subprocesses, determines the BDD size. Our own findings, which have been partly summarised as "rules of thumb" in [177], also confirm this point: The structure of the system to be modelled — and the way in which that structure is exploited during the modelling process — is the main factor which determines the size of a decision diagram.

Many of the statements and observations made in the following sections are equally valid for BDDs, DNBDDs, MTBDDs and other types of decision diagrams. Therefore, we will use the term "BDD" in a broader sense, referring to all of these types.

6.1 Factors influencing the size of a BDD

We already mentioned in Sec. 2.4 that the size of a BDD representing a Boolean function is highly dependent on the chosen ordering of the Boolean variables. It is known that the problem of finding the optimal variable ordering is NP complete [316, 35, 36, 109, 110]. Therefore, determining the optimal ordering is only feasible for very small problems in practice.

When representing transition systems or matrices with the help of decision diagrams, the modeller or developer of a software tool has many degrees of freedom. Several crucial decisions have to be made, each of which may have a strong influence on the size of the decision diagram:

1. The encoding of the state identifiers by Boolean vectors (whose length is $n_s \geq \lceil \log_2 |S| \rceil$, where S is the state space) has to be defined.

2. In the case of action-labelled transition systems, the encoding of the action labels by Boolean vectors (of length $n_a \geq \lceil \log_2 |Act| \rceil$, where Act is the set of actions) has to be defined.

3. It is necessary to define a total ordering on the Boolean variables involved.

For the first and the second issue, there is no commonly accepted heuristics for finding good mappings from states and actions to Boolean vectors, such that the resulting BDD is small. In most cases, states are numbered from 1 to $|S|$ (or from 0 to $|S| - 1$) and the state number is simply encoded as a bit vector, while action labels are simply encoded in the order in which they are encountered in the model. However, the first and second issue often do not have a strong influence on the BDD size, the third issue is usually more important.

Concerning the third issue, the most commonly accepted heuristics has already been introduced in Def. 5.1.1, namely the use of the standard interleaved variable ordering, which is given by

$$\mathsf{a}_1 \prec \ldots \prec \mathsf{a}_{n_a} \prec \mathsf{s}_1 \prec \mathsf{t}_1 \prec \ldots \prec \mathsf{s}_{n_s} \prec \mathsf{t}_{n_s}$$

where $\mathsf{s}_1, \ldots, \mathsf{s}_{n_s}$ ($\mathsf{t}_1, \ldots, \mathsf{t}_{n_s}$) encode the source (target) state of the transition system, and (if applicable) $\mathsf{a}_1, \ldots, \mathsf{a}_{n_a}$ encode the action. The benefits of this rule can be visualised by an inspection of the (by now well-known) function $f_{\mathsf{I}} = f_{\mathsf{Stab}} = \prod_{i=1}^{n} (\mathsf{t}_i \cdot \mathsf{s}_i + (1 - \mathsf{t}_i)(1 - \mathsf{s}_i))$ that can be interpreted as the identity matrix of size 2^n. Using the variable ordering $(\mathsf{s}_1, \mathsf{t}_1, \ldots, \mathsf{s}_n, \mathsf{t}_n)$ leads to an

MTBDD with $3 \cdot n + 2$ vertices, whereas the naïve ordering $(\mathsf{s}_1, \ldots, \mathsf{s}_n, \mathsf{t}_1, \ldots, \mathsf{t}_n)$ blows up exponentially in n, it requires $3 \cdot 2^n - 1$ vertices, cf. Fig. 4.3.

In a matrix setting, this phenomenon implies that everything that "happens" on the main diagonal, or between states whose encodings are "close" to each other, enables a lot of subgraph sharing and therefore helps to keep the size of the decision diagram small. Furthermore, the interleaved ordering fits in perfectly well with the structure of high-level formalisms and parallel composition, as we had already observed in Sec. 5.1.2.

In the following sections, we exemplify that high-level structure can be turned into space-efficient MTBDD encodings.

6.2 Observations concerning the BDD size

6.2.1 Effect of structure and reducedness of the state space

In this section, we consider a simple failure-repair model taken from [314, p. 135]. The model describes two classes of subsystems, each class consisting of two identical components that are subject to failures and repairs. A component in class $i \in \{1, 2\}$ fails with rate λ_i and is subsequently repaired with rate μ_i. Taking into account that equally behaving components can be lumped [212], we obtain a CTMC with state space $S = \{0, \ldots, 8\}$ and the following transition rate matrix R:

$$R = \begin{pmatrix} - & 2\lambda_2 & - & 2\lambda_1 & - & - & - & - & - \\ \mu_2 & - & \lambda_2 & - & 2\lambda_1 & - & - & - & - \\ - & 2\mu_2 & - & - & - & 2\lambda_1 & - & - & - \\ \mu_1 & - & - & - & 2\lambda_2 & - & \lambda_1 & - & - \\ - & \mu_1 & - & \mu_2 & - & \lambda_2 & - & \lambda_1 & - \\ - & - & \mu_1 & - & 2\mu_2 & - & - & - & \lambda_1 \\ - & - & - & 2\mu_1 & - & - & - & 2\lambda_2 & - \\ - & - & - & - & 2\mu_1 & - & \mu_2 & - & \lambda_2 \\ - & - & - & - & - & 2\mu_1 & - & 2\mu_2 & - \end{pmatrix}$$

The nine states can be encoded in the straight-forward way, mapping them onto bit vectors of length 4, such that $0 \mapsto 0000, \ldots, 8 \mapsto 1000$. Using interleaved variable ordering, the MTBDD R based on this encoding, which is shown in Fig. 6.1, has 66 vertices in total.

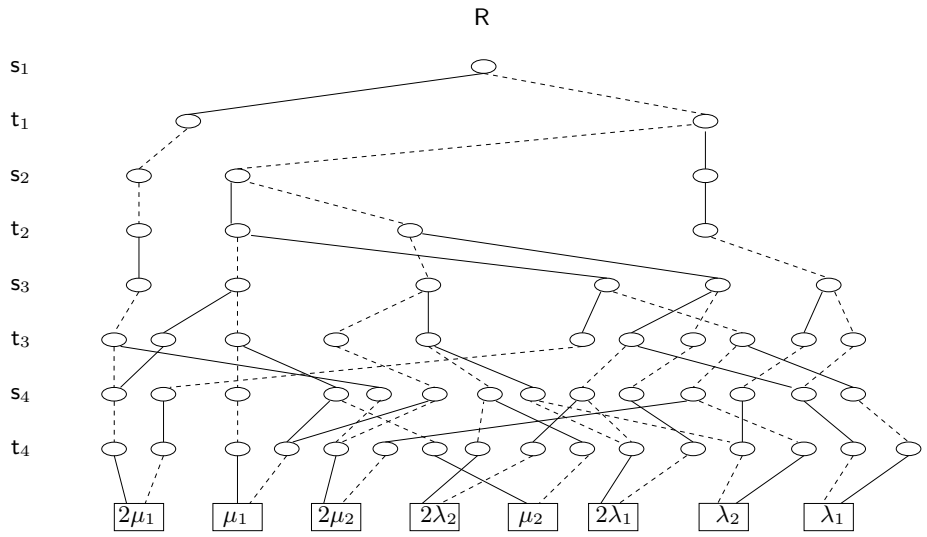


Figure 6.1: MTBDD R for the unstructured encoding of the failure-repair model

It turns out, however, that this straight-forward encoding of states is not the best choice. It is better to exploit the information that the above system consists of two subsystem classes, and that each class has two components that may fail independently. We may view the states as tuples (w_1, w_2), where each of the elements of this tuple ranges from 0 to 2 and indicates how many components of each class are currently operational. Note that this is reflected by the fact that matrix R can be written as $R = R_1 \oplus R_2$ (Kronecker sum of two matrices of size 3), where R_i is given by

$$R_i = \begin{pmatrix} - & 2\lambda_i & - \\ \mu_i & - & \lambda_i \\ - & 2\mu_i & - \end{pmatrix}$$

We need two bits to encode each of the elements of the state tuple, and choose to encode these elements in the straight-forward way. For instance, state $(1, 2)$ (corresponding to state 3) is encoded as a bit vector 0110. The resulting MTBDD is depicted in Fig. 6.2. It has fewer vertices than the one in Fig. 6.1, namely 59 vertices in total (instead of the previous 66 vertices).

We emphasise that in general such a structured encoding may be beneficial even if the sum of the number of bits needed to encode the individual elements of a state tuple were greater than the number of bits needed to encode the unstructured state identifiers[1].

[1]In the present example, the number of bits is 2+2 for encoding the state tuple $(w_1, w_2) \in \{0, 1, 2\} \times \{0, 1, 2\}$ and 4 for encoding the states $\{0, \ldots, 9\}$, so the effect of increasing the depth of the MTBDD does not show.

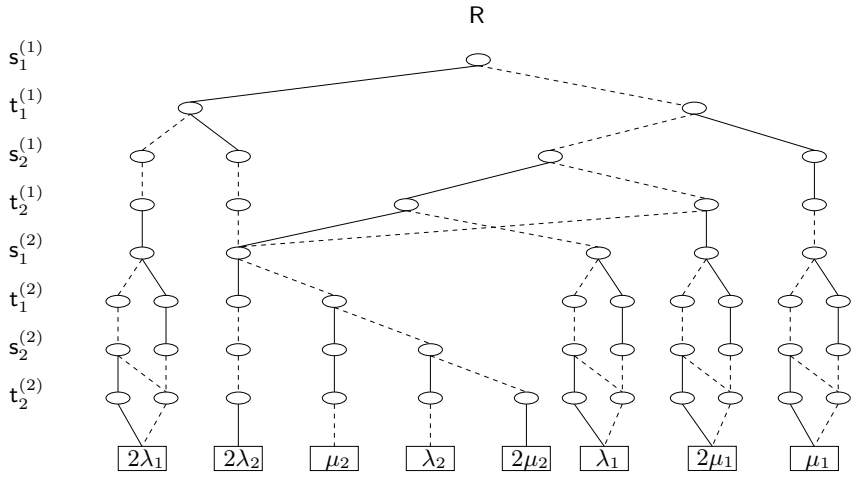


Figure 6.2: MTBDD R for the structured encoding of the failure-repair model

The next observation is particularly interesting: We observe that established techniques for compressing the state space, such as lumping (or its refinement, bisimulation reduction), can be counterproductive in a symbolic setting, since structure gets lost. To illustrate this rule, we consider the above example without applying lumpability beforehand, i.e. we model four independent subsystems, each of which is either operational or failed. As a consequence, we now deal with a CTMC with $2^4 = 16$ states, which is nearly twice the number of states as before. The states may now be viewed as tuples (w_1, w_2, w_3, w_4), where each of the elements is either 0 or 1. The value 1 is used to represent an operational component, and 0 for a failed component. The rate matrix R' of this CTMC contains more non-zero entries than the lumped variant, but only four distinct values, taken from the set $\{\lambda_1, \lambda_2, \mu_1, \mu_2\}$, appear in this matrix. Again, R' can be written as a Kronecker expression, namely the Kronecker sum $R' = R_1 \oplus R_1 \oplus R_2 \oplus R_2$, where R_i is now defined as

$$R_i = \begin{pmatrix} - & \lambda_i \\ \mu_i & - \end{pmatrix}$$

As a consequence, we obtain an MTBDD representation which is more compact than the previous ones. It is depicted in Fig. 6.3, and has only 39 vertices, although the underlying state space is much bigger. The encoding used for this example is simple. Each of the four state variables represents the status of one of the components. The first two represent the components of class 1, the third and fourth represent the components of class 2.

One may argue that a compression of the state space, even if it does increase the memory requirements, results in a reduction of the solution effort in time,

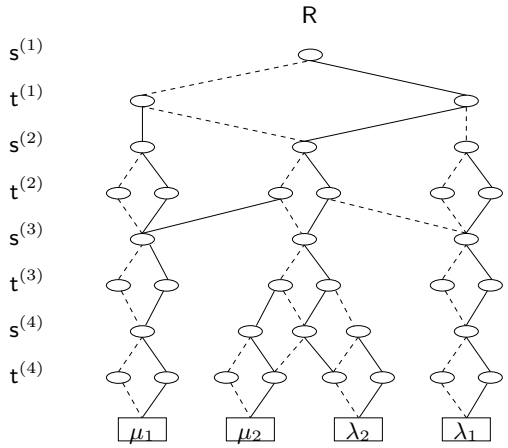


Figure 6.3: MTBDD R for the non-lumped failure-repair model

since less computations have to be performed. However, if we consider strict lumpability (as in the above failure-repair example), and provided that the start vector for the iterative numerical method is chosen appropriately, lumpable states will be involved in exactly the same arithmetic operations during the course of the numerical computation. Assuming that by means of an efficient use of the computed table virtually every operation has its result remembered for later reuse, the solution effort is not substantially increased by keeping lumpable states distinct. Thus, quite surprisingly, working with reduced state spaces can be counterproductive, as it destroys the structure of the MTBDD without saving solution time.

6.2.2 Modelling formalisms with parallel composition operator

Stochastic process algebras possess a parallel composition operator with whose help complex models can be built from small components. Parallel composition is also implicitly featured by networks of stochastic automata (SAN), where the individual automata interact by synchronising over common events. Furthermore, composition of submodels is also a growing topic of research in the area of stochastic Petri nets, where subnets may be superposed by synchronised transitions or shared places.

Models which are built with the help of a parallel composition operator are an excellent source for structure, and therefore well-suited to be used together with BDDs. In Chap. 5 we have shown that symbolic parallel composition causes only

linear growth of the BDD-size. We saw that this result carries over from the non-stochastic to the stochastic setting, a fact that had already been observed (in different contexts) in [295, 183, 177, 102].

Since symbolic parallel composition thus alleviates the explosion of the state space, it is wise to invest into an optimal encoding of the lowest level component state spaces. One may optimise the component encodings either by means of the exact algorithm[2] or by means of an adaption of Rudell's sifting algorithm [282] or other heuristic methods for BDDs, e.g. [130, 27]. We strongly recommend to stick to these encodings from there on, i.e. never to change the encodings which result from the parallel composition, since from the previous section we know that the structure gained by applying composition operators should not be sacrificed, even though it might be tempting to shorten the bit vector encoding the states.

It is important to mention that if one composes an overall model from several submodels, the ordering of the sets of Boolean variables which encode the states of the individual submodels deserves special attention, since this ordering can have a strong effect on the size of the resulting BDD. In a sense, this ordering reflects the encoding of the states of the overall model (which states are actually tuples of submodel states). As a general rule, it is our experience that the sets of state variables of submodels which synchronise with each other should be placed closely together in the overall ordering in order to obtain a compact representation.

In the context of parallel composition, where synchronisation constraints may imply that considerable parts of the composed state space are actually unreachable (cf. Sec. 5.1.5), we recommend as a general rule *not* to explicitly construct the BDD corresponding to the reachable part (even though it might be tempting to restrict the BDD to its reachable parts and even to introduce new state encodings over a smaller dimension Boolean space). Our experience has shown that it is usually much more space efficient to keep the unrestricted BDD and leave the state encoding of the composed MTBDD unchanged, and to construct an additional BDD that encodes a reachability predicate (which has been computed via BDD-based reachability analysis).

At this point, a brief comparison of the BDD-based approach and the Kronecker approach is in order. As mentioned in Sec. 3.6.1, the main strength of the Kronecker approach is also its memory-efficiency, which is achieved by avoiding the explicit enumeration of all possible interleavings of actions in the participating submodels. When composing submodels, the Kronecker approach features the same additive growth characteristics as the BDD-based approach (as opposed to

[2]Since the lowest level components are usually quite small, so are their BDD representations, and therefore NP-completeness of the minimisation problem is not a problem in practice.

the the usually observed multiplicative growth which is due to the interleaving of independent moves). However, with the Kronecker approach, the matrix representing the overall model is never explicitly constructed, which has the effect that rates need to be computed "on the fly" whenever they are needed, for instance during numerical analysis. In contrast, with the BDD-based approach, every transition between two states of the composed model is *explicitly* encoded by a path leading to a non-terminal vertex, where in the case of MTBDDs the rate is stored[3]. Both, the Kronecker approach and the BDD-based approach are faced with the problem that a large portion of the product space of the components (of the "potential state space") may be unreachable. In the context of the Kronecker approach, it has been shown that this problem can be dealt with by performing Kronecker-based reachability analysis as a preprocessing step and taking the reachability information into account during numerical analysis. In the context of the BDD-based approach, the situation is more subtle: Under certain conditions, the unreachable states may not hurt at all, i.e. they may not require extra memory and may not slow down the computation. These conditions are: All unreachable states must receive initial probability zero and the numerical method must be such that this zero probability is preserved during the course of the computation. In that case, since MTBDDs require only a single terminal vertex for storing all identical (zero-)values, the actual size of the matrix and the vector do not matter.

6.2.3 Compact encodings for networks of queues

In this section, we study a particular class of queueing networks and show the effect which structurings of the state space can have on the size of the MTBDD representation.

We start with a simple M/M/1 queue with finite capacity c. For simplicity, we assume that $c = 2^k - 1$ for some natural k, since this implies that the state space size is a power of two. Enumerating the states in the usual way from 0 to $2^k - 1$, we obtain that the rate matrix R is non-zero at $R(i, i+1) = \delta$, $R(i+1, i) = \xi$, for $0 \leq i \leq 2^k - 2$, where δ is the arrival rate, and ξ is the departure rate. For $k = 2$ we obtain the rate matrix R as

[3]In consequence, with MTBDDs, the look-up of a matrix entry can be done in (worst-case) logarithmic time in the size of the matrix, without involving arithmetic operations.

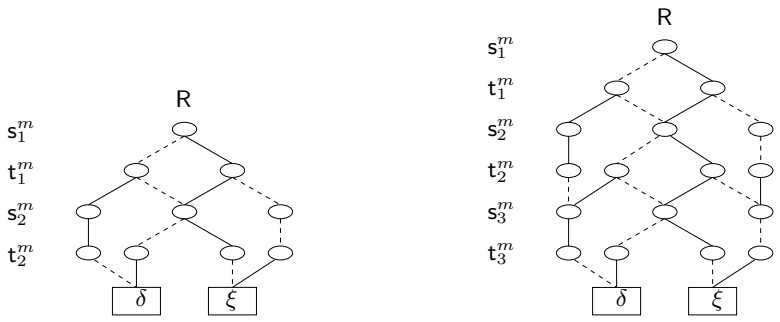


Figure 6.4: MTBDD R of the M/M/1 example with queue capacity 3 and 7

$$R = \begin{pmatrix} - & \delta & - & - \\ \xi & - & \delta & - \\ - & \xi & - & \delta \\ - & - & \xi & - \end{pmatrix}$$

For this queue, the MTBDD R over $(s_1^m, t_1^m, s_2^m, t_2^m)$ is shown in Fig. 6.4, together with the MTBDD for the case $k = 3$, i.e. for an M/M/1 queue with capacity 7 (the superscript m is used to distinguish the Boolean variables from other variables that will be introduced later). The encoding of states as bit vectors is done in the "natural" way, where 0 is encoded as a vector of all zeroes, and $2^k - 1$ is encoded as a vector of all ones (both of length k). The crucial observation from Fig. 6.4 is that doubling the state space size (and hence essentially the queue capacity) does *not* double the memory requirements of the MTBDD needed to symbolically represent the rate matrix. In contrast, the MTBDD increases only linearly, by a constant of 7 non-terminal vertices. This is true in general, the M/M/1 with queue capacity $2^k - 1$ requires $7 \cdot k - 1$ vertices to be represented as an MTBDD. This striking feature is of course due to the perfect regularity of the M/M/1 queue, but as far as we know, it is not present in any other method to store Markov chains explicitly.

This observation confirms our general statement, namely that structure exploitation is the key to obtaining compact BDD representations. In general, one may say that if repetitive sub-blocks of a matrix are encoded "close" to each other, there may be exponential savings in memory space.

Since the M/M/1 system is the simplest of all queueing systems, its efficient encoding is not yet a convincing argument. We will now show that the same effect, an increase of the memory requirements which is only logarithmic in the size of the queue lengths, can be obtained for more complex queueing models

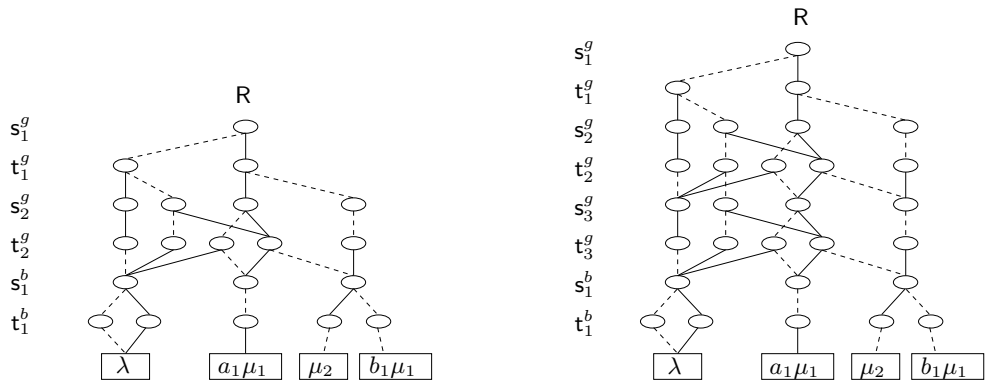


Figure 6.5: MTBDD R of the M/Cox$_2$/1 example with queue capacity 3 and 7

as well. To illustrate this fact, we extend the above result to the broad class of single queues with phase-type arrival and service time distributions. Consider, for instance, an M/Cox$_p$/1 queue with finite capacity $c = 2^k - 1$ and $p = 2^l$ phases[4]. Assuming $k = 2$ and $l = 1$, i.e. a queue capacity of $c = 3$ and a Coxian distribution with two phases, we end up with an example discussed in [314, p. 237] (see there for the special treatment of the "double" empty state). The rate matrix for this queue is given by

$$R = \left(\begin{array}{c|c|c|c} - & A & - & - \\ \hline B & C & A & - \\ \hline - & B & C & A \\ \hline - & - & B & C \end{array}\right)$$

$$\text{where } A = \begin{pmatrix} \lambda & - \\ - & \lambda \end{pmatrix},\ B = \begin{pmatrix} b_1\mu_1 & - \\ \mu_2 & - \end{pmatrix} \text{ and } C = \begin{pmatrix} - & a_1\mu_1 \\ - & - \end{pmatrix}.$$

Obviously, the matrix possesses a block tridiagonal structure, and again, for larger values of the queue capacity c, or the number of phases p the matrix is simply extended in a regular fashion. In terms of k and l, the matrix R has size 2^{k+l}, and hence it is quite natural to encode the global (diagonal) structure with $2k$ Boolean variables $(\mathsf{s}_1^g, \mathsf{t}_1^g, \ldots, \mathsf{s}_k^g, \mathsf{t}_k^g)$, and to represent the block matrices with $2l$ variables $(\mathsf{s}_1^b, \mathsf{t}_1^b, \ldots, \mathsf{s}_l^b, \mathsf{t}_l^b)$. For our example, we get the MTBDD shown in Fig. 6.5 (left). Note how the bottom variable levels $(\mathsf{s}_1^b, \mathsf{t}_1^b)$ directly encode the block matrices A, B and C. In Fig. 6.5 (right) we have also depicted the representation of the M/Cox$_2$/1 queue with capacity $c = 7$, in order to illustrate the logarithmic growth of the MTBDD. It turns out that the MTBDD corresponding to capacity

[4]If the number of phases is not a power of 2, the same encoding applies, but some dummy rows and columns in the block matrices are filled with zero entries. The MTBDD representation, however, has the same characteristics.

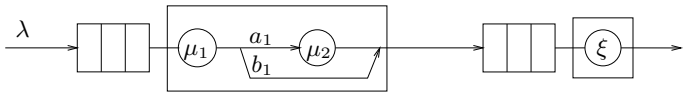


Figure 6.6: Simple tandem queueing network

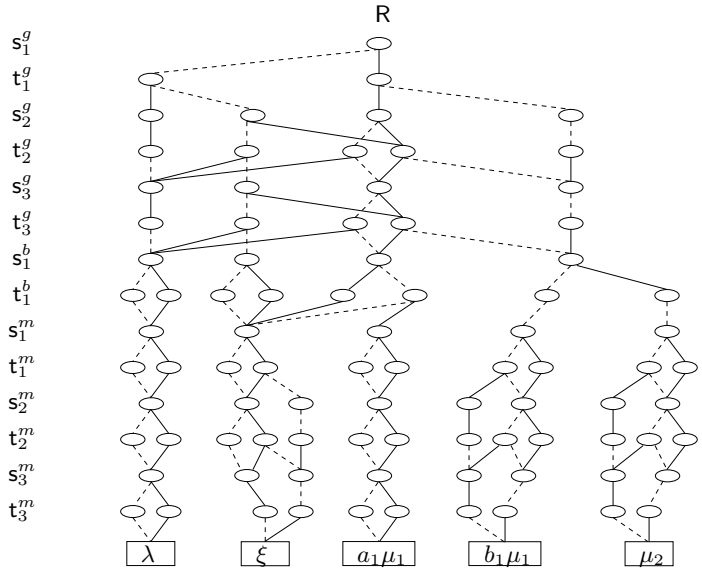


Figure 6.7: MTBDD for the simple tandem queueing network

$(2^k - 1)$ has $9 \cdot k + 7$ vertices, and a similar bound, linear in l, can be derived for an exponential increase of the number of phases of the Coxian distribution.

The same basic principle applies to any queue with block-structured rate or generator matrix, and we will now show that it is not restricted to single queues. Let us, for instance, combine the two previous queues in a tandem network as shown in Fig. 6.6. In this network, customers which have completed their service in the upstream (Coxian) queue are routed to the downstream (Markovian) queue. In the case where the downstream queue is fully occupied, the upstream queue is blocked. This means that the second phase of the Coxian server is blocked and that the "bypass" of the second phase (which is taken with probability b_1) is also blocked. In other words, while blocked, the Coxian server may only be active in its first phase with rate $a_1 \cdot \mu_1$.

We can, of course, consider this tandem queueing system as a system consisting of two components (the two queueing stations) which are composed in parallel. Their synchronisation is such that the departure event of the upstream station is synchronised with the arrival event of the downstream station. From this

consideration, it follows directly that the MTBDD encoding the rate matrix of the tandem queueing system can be constructed from the MTBDDs encoding the rate matrices of the two components, following MTBDD-based parallel composition as discussed in Sec. 5.1.4. From Thm. 5.1.4 it further follows that the logarithmic growth which we observed for both the Markovian and the Coxian queue carries over to the tandem queueing system. This is indeed the case: The resulting MTBDD for the case where both queues have a finite capacity of $c = 7$ is shown in Fig. 6.7. The state space for this model contains $16 \cdot 8 = 128$ states, and the MTBDD has 93 vertices. It is easy to verify that, in general, if each of the two queueing stations has a capacity of $c = 2^k - 1$, the state space size is 2^{2k+1} while the MTBDD representation only requires $30 \cdot k + 3$ vertices. So the tandem queueing system is another example where, for exponentially growing state space, the growth of the MTBDD is linear.

In order to give an impression of the memory requirements of such an MTBDD, assume that the size of an MTBDD vertex is 16 byte[5]. Using just 1 MB of memory, MTBDDs with up to 65536 vertices could be stored. For the tandem queueing network example, we have seen that $30 \cdot k + 3$ vertices are needed to represent a state space of size 2^{2k+1}. For the 1 MB limit, the maximum value for k is thus 2184, which corresponds to $2^{4369} = 10^{1315}$ states, generated by queue lengths of capacity 10^{657} for each of the two queueing stations. Mind, however, as already stressed in Sec. 4.5, that one has to keep in mind that the given sizes are for the resulting MTBDD, and that intermediate MTBDDs may be larger than the final ones.

There obviously exists a large class of queueing networks which have MTBDD representations that are logarithmic in the size of the state space. These networks consist of queues with finite capacity, phase-type service time distributions, and blocking[6]. The interarrival times of customers arriving from the environment may have phase-type distribution. The example tandem queueing network has a particularly compact MTBDD representation, since its state space size is a power of 2 and its rate matrix has a perfectly regular structure. But even for networks of queues with arbitrary capacity, non-trivial routing between stations and possibly cyclic structures, one can find very space-efficient symbolic encodings.

[5]The size of a vertex within the decision diagram package CUDD [307] is indeed 16 byte. A vertex in CUDD comprises two half words (containing the variable index and the reference counter) and three pointers (two pointing to the children and one pointing to the next vertex in the hash-collision list).

[6]Here, the term "blocking" is defined in the following sense: The handover of a customer from one station to another may only take place if the capacity of the receiving station is not yet exhausted. Otherwise, the activity leading to the handover event is blocked, i.e. disabled. For an in-depth study of queueing networks with blocking see, for instance, [270].

Chapter 7

Numerical analysis based on symbolic representations

The previous chapters have dealt with the symbolic representation of transition systems, where the emphasis of our discussion was on heuristics for achieving compact representations. We have also already covered some symbolic analysis techniques, such as reachability analysis and symbolic bisimulation reduction. The present chapter covers symbolic numerical analysis, which is an important cornerstone within a fully symbolic modelling and analysis methodology. In particular, we will describe how the numerical analysis of a CTMC can be carried out completely on the basis of MTBDD operations[1]. Note that the MTBDD representation of the CTMC (i.e. of its rate matrix R) underlying an SLTS $\mathcal{T}$ can be obtained easily from the MTBDD representation of $\mathcal{T}$ (denoted M), by simply abstracting from the action variables of M. In case of an ESLTS, vanishing states must be eliminated before numerical analysis can start, which can be achieved with the help of the technique described in Sec. 5.2.2.

7.1 Steady-state analysis based on MTBDDs

The idea of carrying out the numerical analysis of Markov chains based on a symbolic representation, i.e. by using decision diagrams as the principal data structure, is not new. The work of Hachtel et al. [146, 145, 147] contains an inves-

[1] We restrict this discussion of numerical analysis to MTBDDs, since efficient algorithms for DNBDD-based linear algebra operations — though feasible in principle — are currently not available.

tigation of the benefits when representing discrete time Markov chains (DTMCs), derived from digital circuits, by MTBDDs. It describes how steady-state probabilities of DTMCs of up to 10^{27} states can be computed by applying iterative numerical solution methods, namely the power method and a variant of the method of Jacobi.

Direct solution methods, as opposed to iterative methods, can — in principle — be realised on symbolic data structures. For instance, recursive formulation of LU-decomposition is possible and can be implemented in a straight-forward way with the help of MTBDDs [264]. However, it turns out that direct methods are not well-suited for large Markov chains in general, and for the MTBDD framework in particular, for the following reasons: First of all, direct methods are not suitable for large state spaces since they cause a lot of fill-in of the coefficient matrix (which is usually very sparse at the beginning). The storing of large, non-sparse matrices may cause the system to run out of memory. Using MTBDDs, the fill-in would blow up the MTBDD during the elimination process. In addition to this general memory problem, when using BDD-based data structures the following problem occurs: Each step of the direct method modifies the structure of the coefficient matrix and thus the MTBDD structure, and hence causes serious overhead to keep the representation canonical [18]. For these reasons, in the sequel we will describe iterative methods for the analysis of large Markov chains, based on their symbolic representation.

7.1.1 Stationary iterative methods

In this section, we discuss the MTBDD-based realisation of some iterative methods for computing steady-state probabilities of CTMCs. The algorithms use the operations on MTBDDs which were described in Sec. 4.4.1. MTBDDs are particularly well suited for matrix operations, but not so well suited for elementwise operations. Therefore, the central operation on which we build our symbolic implementation is vector-matrix multiplication, and we will discuss the power, Jacobi and Gauss-Seidel methods in the framework of a general matrix powering algorithm in which the new iterate is calculated according to the scheme $\vec{\pi}^{(k+1)} := \vec{\pi}^{(k)} \cdot M$. In this general algorithm only the iteration matrix M depends on the particular method. We will elaborate on the problems which such a procedure causes for Gauss-Seidel-type iteration schemes.

The general matrix powering algorithm: We assume for the moment that the MTBDD $\mathsf{M}(\vec{\mathsf{s}}, \vec{\mathsf{t}})$, representing the iteration matrix M, has already been constructed. $\mathsf{M}(\vec{\mathsf{s}}, \vec{\mathsf{t}})$ is an MTBDD over $(\mathsf{s}_1, \mathsf{t}_1, \ldots, \mathsf{s}_{n_s}, \mathsf{t}_{n_s})$. The algorithm uses two vectors $\vec{\pi}$ and $\vec{\pi}'$, represented by MTBDDs $\mathsf{P}(\vec{\mathsf{s}})$ and $\mathsf{P}'(\vec{\mathsf{t}})$, which contain the state

```
algorithm  IterativeSolve (M(s⃗,t⃗), n_s, diff_max)
(1)        Initialise(P(s⃗))
(2)        repeat
(3)            P'(t⃗)  :=  VMmult(P(s⃗), M(s⃗,t⃗))
(4)            P'(s⃗)  :=  P'(t⃗){t⃗ ← s⃗}
(5)            T      :=  Maxval(Apply(P(s⃗), P'(s⃗), −))
(6)            P(s⃗)  :=  P'(s⃗)
(7)        until value(T) < diff_max
(8)        return P(s⃗)
```

Figure 7.1: Symbolic matrix powering algorithm

probabilities before and after each iteration step. Note that we do not assume that the number of states of the underlying CTMC is a power of 2. Therefore, some assignments to the Boolean vectors $\vec{s}$ and $\vec{t}$ may correspond to non-existent "phantom" states, a circumstance which must be kept in mind during initialisation and normalisation of the probability vectors. Another related issue is the structure of the CTMC: If the CTMC, represented by $\mathsf{M}(\vec{s},\vec{t})$, is irreducible (i.e. if it has only a single BSCC), then the long-run probability distribution is independent of the initial state (or the initial probability distribution). If, on the other hand, the CTMC contains more than one BSCCs, then the initial state matters, as explained in Sec. 2.1.3.

The symbolic matrix powering algorithm is given in Fig. 7.1. The algorithm has three parameters: The MTBDD $\mathsf{M}(\vec{s},\vec{t})$ representing the iteration matrix, the length of the state encoding n_s, and the maximally tolerated elementwise difference between successive iterates, $\mathit{diff}_{\max}$. In line (1), procedure Initialise is called in order to set $\mathsf{P}(\vec{s})$ to the initial estimate. Lines (2) - (7) constitute the main loop of the algorithm. In line (3), the multiplication of the current estimate with the iteration matrix M is performed. The result vector of this multiplication, P', depends on Boolean variables $\vec{t}$, since the multiplication abstracts from variables $\vec{s}$ which are common to P and M. The variable renaming in line (4) makes P' dependent on $\vec{s}$ instead of $\vec{t}$, as required for the Apply operation in line (5) (note that the ordering of variables respects the precondition required by the renaming operation, cf. Sec. 2.4.2). In line (5), the maximal absolute elementwise difference between the old and the new iterate is calculated, in order to be used in line (7) as a termination criterion[2]. In line (6), the old iterate is overwritten

[2]Mind that, in general, the fact that two successive iterates are reasonably close is no guarantee that the correct solution has been approximated. For a discussion on termination

by the new one. Line (7) checks whether the difference lies within the tolerated bound $diff_{\max}$, in which case the algorithm terminates, returning the probability vector $\vec{\pi}$, represented as the MTBDD P. Of course, other termination criteria, such as the exceeding of a pre-specified maximum number of iterations, should be added in practice.

In procedure INITIALISE, there are several possibilities of how to choose the initial estimate:

- One possibility is to make all 2^{n_s} states (including phantom states and unreachable states) equiprobable. In this case the MTBDD $\mathsf{P}(\vec{\mathsf{s}})$ representing the probability vector initially consists of only a single terminal vertex whose value is $1/2^{n_s}$. With this initialisation, if phantom states exist or if the CTMC contains more than one BSCC, restriction of the solution vector to the set of relevant states and normalisation have to be performed after convergence has been reached. A minor advantage of this initialisation policy is that the steady state solutions within all BSCCs of the Markov chain are actually computed simultaneously.
- (a) One may make all reachable states equiprobable and assign probability zero to the unreachable and phantom states, or (b) if a unique initial state is known it can be assigned initial probability one (and all other states, be they phantom or not, reachable or unreachable, receive initial probability zero). These two initialisations avoid unnecessary calculations related to the unreachable states and have the advantage that during iteration, phantom states or unreachable states always keep their initial probability zero.
- If an estimate to the solution is already known, for instance from a previous solution of a related problem, this can serve as the initial probability vector. Such a choice, if available, may drastically reduce the number of iterations until convergence.

Depending on the particular choice of the iteration matrix M, depending on the initial probability vector, and depending on the existence of phantom or unreachable states, there may be a need to normalise the result vector returned by the above matrix powering algorithm, such that the sum of the entries corresponding to relevant states is equal to one. Normalisation can easily be carried out with the help of MTBDD operations as follows: First, all entries of the result vector which correspond to unreachable states are set to zero by a single multiplication

$$\mathsf{P}(\vec{\mathsf{s}}) := \textsc{Apply}(\mathsf{P}(\vec{\mathsf{s}}), \mathit{Reach}(\vec{\mathsf{s}}), \cdot)$$

criteria for iterative methods see [314, p. 156].

Then the sum of the relevant entries is calculated as

$$Sum := \textsc{Abstract}(\mathsf{P}(\vec{\mathsf{s}}), \vec{\mathsf{s}}, +)$$

and finally, the vector is scaled as follows:

$$\mathsf{P}(\vec{\mathsf{s}}) := \textsc{Smult}(\mathsf{P}(\vec{\mathsf{s}}), 1/Sum)$$

It remains to describe how the MTBDD-representation of the iteration matrix M is constructed for the different iteration schemes. We assume that the transition relation of an SLTS is stored as an MTBDD $\mathsf{M}(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}})$. As a first step, it is necessary to abstract from the action names in order to obtain the rate matrix R of the underlying CTMC. This can be achieved by the assignment $\mathsf{R}(\vec{\mathsf{s}}, \vec{\mathsf{t}}) := \textsc{Abstract}(\mathsf{M}(\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}}), \vec{\mathsf{a}}, +)$.

Power method: The iteration matrix for the power method is given by $M_{power} = Q \cdot \Delta t + I$. As a first step towards building the symbolic iteration matrix M_{power}, the symbolic infinitesimal generator matrix Q must be derived from the rate matrix R. We first compute the MTBDD D containing the row sums of R (which will be the negative diagonal elements of Q) as

$$\mathsf{D}(\vec{\mathsf{s}}) := \textsc{Abstract}(\mathsf{R}(\vec{\mathsf{s}}, \vec{\mathsf{t}}), \vec{\mathsf{t}}, +)$$

From MTBDDs $\mathsf{R}(\vec{\mathsf{s}}, \vec{\mathsf{t}})$ and $\mathsf{D}(\vec{\mathsf{s}})$ one can then construct the MTBDD Q, using the operator Diag in order to turn a vector into a matrix (see Sec. 4.4.1):

$$\mathsf{Q}(\vec{\mathsf{s}}, \vec{\mathsf{t}}) := \textsc{Apply}\left(\mathsf{R}(\vec{\mathsf{s}}, \vec{\mathsf{t}}), \textsc{Diag}(\mathsf{D}(\vec{\mathsf{s}}), \vec{\mathsf{t}}), -\right)$$

Next, a suitable value for Δt must be chosen. It can be obtained, for instance, by taking $\Delta t = \textsc{Maxval}(\mathsf{D}(\vec{\mathsf{s}}))^{-1} \cdot 0.99$. Let T be the MTBDD consisting of only a single terminal vertex labelled with Δt. The actual scaling of the generator matrix, i.e. calculating the product $Q \cdot \Delta t$ amounts to a simple scalar multiplication. Afterwards, as the last step, the identity matrix is added. Summarising these steps, the MTBDD M_{power} representing the iteration matrix is obtained by

$$\mathsf{M}_{power}(\vec{\mathsf{s}}, \vec{\mathsf{t}}) := \textsc{Apply}(\textsc{Smult}(\mathsf{Q}(\vec{\mathsf{s}}, \vec{\mathsf{t}}), \mathsf{T}), \mathsf{Id}(\vec{\mathsf{s}}, \vec{\mathsf{t}}), +)$$

where $\mathsf{Id}(\vec{\mathsf{s}}, \vec{\mathsf{t}})$ is a 0-1-MTBDD representing the identity matrix of size 2^{n_s}.

For the CTMC example from Fig. 4.10, the MTBDD Q is shown on the left of Fig. 7.2. In this example, the biggest row sum of matrix R is $f_{\mathsf{D}}(0,1) = 6$. Therefore, for simplicity, we choose $\Delta t = 1/7$. The resulting symbolic iteration matrix M_{power} is shown in Fig. 7.2 on the right[3].

[3] A reduction of the number of terminal vertices when moving from Q to M_{power}, as observed in this example, is not typical.

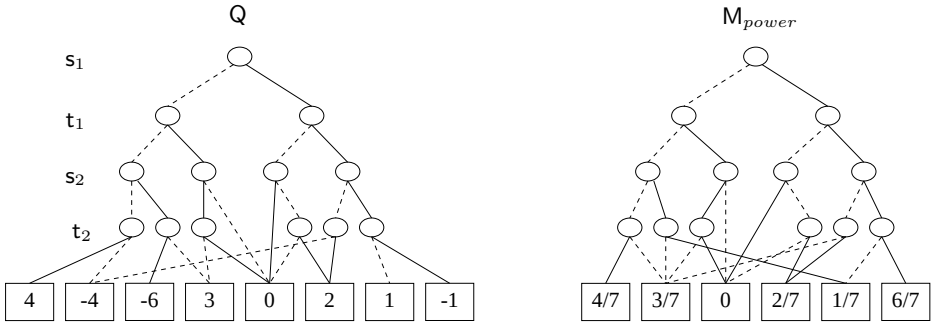


Figure 7.2: MTBDD representation of matrices Q and M_{power} for the CTMC of Fig. 4.10

Jacobi method: Using D to refer to the diagonal matrix of the row sums of R, the iteration matrix for the Jacobi methods is $M_{Jacobi} = R \cdot D^{-1}$. Let MTBDD R represent the rate matrix R, and let D be the MTBDD encoding the row sums of R (as above). The inversion of D can be achieved by performing operation InvDiag (which realises elementwise inversion) on MTBDD $\textsc{Diag}(\mathsf{D}(\vec{s}), \vec{t})$ [4]. Overall, the symbolic representation of the iteration matrix M_{Jacobi}, is calculated as follows:

$$\mathsf{M}_{Jacobi}(\vec{s}, \vec{t}) := \textsc{Mmult}(\mathsf{R}(\vec{s}, \vec{t}), \textsc{InvDiag}(\textsc{Diag}(\mathsf{D}(\vec{s}), \vec{t})))$$

Note that for (time) efficiency reasons, an MTBDD-based algorithm for matrix multiplication can be devised for the special case where the right argument is a diagonal matrix. This multiplication simply amounts to column-wise scaling of the matrix R.[5]

Gauss-Seidel and Successive Over-relaxation: The iteration scheme of Gauss-Seidel is similar to the method of Jacobi, but for computing the new iterate $\pi_{s_i}^{(k+1)}$ the already updated values for $\pi_{s_j}^{(k+1)}$, $j < i$ are used. The specific strength of the symbolic computation is the simultaneous computation of common parts of a matrix-vector product. Therefore, at first sight, the strategy of Gauss-Seidel does not seem to be well-suited for symbolic implementation. But in principle it is of course possible to perform Gauss-Seidel by straight-forward matrix multiplication. One can explicitly calculate the iteration matrix $L \cdot (D - U)^{-1}$ where L and U are the negative lower (upper) triangular portions of R and D is now the *negative* diagonal matrix of the row sums of R. A recursive MTBDD-based algorithm Invtri for inverting triangular matrices has been sketched in Sec. 4.4.1. In

[4] Of course, the terminal vertices of MTBDD D could also be inverted before D is turned into a diagonal matrix.

[5] A matrix column of R is addressed with the help of restriction $\mathsf{R}\big|_{\mathsf{t}_1=b_1 \wedge \ldots \wedge \mathsf{t}_{n_s}=b_{n_s}}$ of the t-labelled vertices, where the vector $(b_1, \ldots, b_{n_s})$ encodes the column index.

general, the inversion of the triangular matrix causes a lot of fill-in, and therefore this approach is counterproductive in the setting of sparse matrices. However, in the MTBDD setting, the fill-in can be tolerated as long as many of the newly computed entries of $(D-U)^{-1}$ are identical, because this induces a sharing of subgraphs in the resulting MTBDD. For specific cases, such as a simple M/M/1 queue with finite capacity c, this is the case indeed: Its matrix $D-U$ possesses $2 \cdot c+1$ non-zeroes. The inverse has $(c+1) \cdot (c+2)/2$ non-zeroes (fill-in). However, the number of *distinct* non-zeroes is only $2 \cdot c$. Therefore, in specific cases, the straight-forward vector-matrix realisation of Gauss-Seidel might turn out to be rather efficient in a symbolic setting.

Having said this, the symbolic iteration matrix for the Gauss-Seidel scheme can be calculated as follows:

$$\mathsf{M}_{Gauss-Seidel}(\vec{\mathsf{s}},\vec{\mathsf{t}}) := \textsc{Mmult}(\mathsf{L}(\vec{\mathsf{s}},\vec{\mathsf{t}}), \textsc{Invtri}(\textsc{Apply}(\textsc{Diag}(\mathsf{D}(\vec{\mathsf{s}}),\vec{\mathsf{t}}), \mathsf{U}(\vec{\mathsf{s}},\vec{\mathsf{t}}), -)))$$

where D is obtained as

$$\mathsf{D}(\vec{\mathsf{s}}) := \textsc{Smult}(\textsc{Abstract}(\mathsf{R}(\vec{\mathsf{s}},\vec{\mathsf{t}}),\vec{\mathsf{t}},+),-1)$$

and L and U are obtained from R by setting all elements on or above (below) the diagonal to zero. In the tool Im-Cat, this latter step is realised by a special procedure which works its way recursively through the MTBDD.

A symbolic version of the successive over-relaxation method (SOR) raises essentially the same issues as with Gauss-Seidel: Performing SOR by a simple matrix-vector multiplication scheme (which is not what is usually done) involves the inversion of a triangular matrix which incurs the above mentioned fill-in.

7.1.2 Projection methods

In this section, we discuss the Bi-CGSTAB (Bi-Conjugate Gradient Stabilised) method as an example projection method. Bi-CGSTAB [104] is an algorithm from the class of Krylov subspace methods for the solution of non-symmetric linear systems of equations (which is the type of system to which our problem $\vec{\pi} \cdot Q = 0$ belongs). Bi-CGSTAB features attractive convergence speed and stable convergence behaviour. It is a suitable candidate for MTBDD-based realisation because it uses vector-matrix multiplication as its central operation[6].

[6]GMRES, for instance is not suitable for MTBDD-based implementation, because it uses too many scalar (column-wise) accesses.

algorithm BI-CGSTAB $(\vec{\pi}, Q)$
INITIALISATION
(1) $\vec{r} := \vec{r_0} := -\vec{\pi} \cdot Q$
(2) $\rho := \alpha := \omega := 1$
(3) $\vec{v} := \vec{p} := 0$
ITERATION
(4) **repeat**
(5) $\rho_{old} := \rho$
(6) $\rho := \vec{r_0}^T \cdot \vec{r}$
(7) $\beta := \frac{\rho}{\rho_{old}} \cdot \frac{\alpha}{\omega}$
(8) $\vec{p} := \vec{r} + \beta \cdot (\vec{p} - \omega \cdot \vec{v})$
(9) $\vec{v} := \vec{p} \cdot Q$
(10) $\alpha := \rho / (\vec{r_0}^T \cdot \vec{v})$
(11) $\vec{s} := \vec{r} - \alpha \cdot \vec{v}$
(12) $\vec{t} := \vec{s} \cdot Q$
(13) $\omega := \vec{t}^T \cdot \vec{s} / (\vec{t}^T \cdot \vec{t})$
(14) $\vec{\pi} := \vec{\pi} + \alpha \cdot \vec{p} + \omega \cdot \vec{s}$
(15) $\vec{r} := \vec{s} - \omega \cdot \vec{t}$
(16) **until** convergence of $\vec{\pi}$ (or breakdown)
(17) **return** $\vec{\pi}$

Figure 7.3: The Bi-CGSTAB algorithm for solving the system $\vec{\pi} \cdot Q = 0$

Fig. 7.3 shows the basic Bi-CGSTAB algorithm (adapted to the problem of solving the system $\vec{\pi} \cdot Q = 0$). The algorithm has two arguments: An initial approximation to the solution ($\vec{\pi}$) and the generator matrix (Q). Note that for practical implementation the algorithm in Fig. 7.3 must be augmented by additional details (such as normalisation of the vector $\vec{\pi}$ and restarting of the algorithm after a certain number of iterations, a proper convergence check and checking for breakdown). During one iteration, Bi-CGSTAB performs two vector-matrix multiplications with the generator matrix Q (lines (9) and (12) of the algorithm) and four inner products of vectors of the size of the state space (lines (6), (10) and twice in (13)). The storage requirements are quite substantial: Bi-CGSTAB needs to store seven vectors (namely $\vec{\pi}$, $\vec{r}$, $\vec{r_0}$, $\vec{p}$, $\vec{v}$, $\vec{s}$ and $\vec{t}$).

The tool IM-CAT contains a prototypical MTBDD-based realisation of the Bi-CGSTAB algorithm, which is an adaptation from a sparse matrix implementation by Buchholz [58]. The vector-matrix multiplications (and actually also the inner products) are implemented by a general MTBDD-based matrix multiplication algorithm.

In our experiments, this implementation of Bi-CGSTAB was not notably faster than for example the method of Jacobi. It is expected, though, that Krylov subspace methods such as Bi-CGSTAB outperform stationary methods if used with appropriate preconditioners. The question whether preconditioning (which improves the spectral properties of the coefficient matrix) can be beneficially employed in a symbolic scenario has not been studied yet. Explicit calculation of the product of the generator matrix Q and a preconditioning matrix M is problematic, though, since the non-zero structure of $M \cdot Q$ (or $Q \cdot M$) is different from the non-zero structure of Q (sparsity is lost) and therefore the compactness of the MTBDD representation is likely to be destroyed.

7.2 Transient analysis based on MTBDDs

In Sec. 2.1.2, we briefly described the uniformisation method for determining the probability distribution of a CTMC at a fixed time instant t. The vector of time-dependent state probabilities, denoted by $\vec{\pi}(t)$, can be calculated as follows:

$$\vec{\pi}(t) = \vec{\pi}(0) \cdot \sum_{k=L}^{R} P^k \cdot \frac{(q \cdot t)^k}{k!} \cdot e^{-q \cdot t}$$

(where, as described in Sec. 2.1.2, an infinite sum has been replaced by a finite sum, dropping those terms which are sufficiently small).

Fig. 7.4 shows an algorithm which is based on the above formula. The algorithm has the following parameters: The initial probability distribution $\vec{\pi}(0)$, the time instant t, the stochastic matrix P, and the left (L) and right (R) truncation points for the summation. Within the algorithm, the vector $\vec{h}$ is used for storing the products $\vec{\pi}(0) \cdot P^k$ (for $k = 0, 1, 2, \ldots$), and the precomputation part is needed to calculate $\vec{\pi}(0) \cdot P^{L-1}$. Note that the stochastic matrix $P = Q \cdot \Delta t + I$ is identical to the iteration matrix M_{power} for the power method (the same uniformisation constant $q = 1/\Delta t$ can be used), so its MTBDD representation can be constructed as described in Sec. 7.1.1. Note further that the algorithm in Fig. 7.4 assumes that L and R, as well as the Poisson probabilities, denoted by $Prob_P(k) = \frac{(q \cdot t)^k}{k!} \cdot e^{-q \cdot t}$ (which are needed in line (8)), have already been calculated. These are all scalar calculations which do not involve any MTBDDs, therefore we do not give further details here. We emphasise that the algorithm shown in Fig. 7.4 is rather rudimentary. In practical implementations, additional details such as steady-state detection[7], need to be added.

[7]Steady state is reached if $\vec{h}$ remains (approximately) the same in line (4) or line (7) of the algorithm.

algorithm UNIFORMISATION $(\vec{\pi}(0), t, P, L, R)$
INITIALISATION
(1) $\vec{\pi}(t) := 0$
(2) $\vec{h} := \vec{\pi}(0)$
PRECOMPUTATION
(3) **for** $k = 1, \ldots, L-1$
(4) $\quad \vec{h} := \vec{h} \cdot P$
ITERATION
(5) **for** $k = L, \ldots, R$
(6) $\quad$ **if** $k > 0$
(7) $\quad\quad \vec{h} := \vec{h} \cdot P$
(8) $\quad \vec{\pi}(t) := \vec{\pi}(t) + \vec{h} \cdot Prob_P(k)$
(9) **return** $\vec{\pi}(t)$

Figure 7.4: The uniformisation algorithm for calculating the time-dependent probability distribution $\vec{\pi}(t)$

From this description of the uniformisation method, one may observe that the only operations necessary are vector-matrix multiplication, multiplication of a vector with a scalar, and the summation of vectors. These operations are realised on MTBDDs with the help of VMMULT, SMULT and APPLY, as introduced in Sec. 4.4.1. Thus, the uniformisation method can be implemented without difficulty by using MTBDDs as its underlying data structure. One such implementation is described in [209].

7.3 Discussion of symbolic numerical analysis

In this chapter, we have pointed out that well-known methods for the numerical analysis of Markov chains can readily be implemented on the MTBDD data structure. We mention at this point, that apart from the calculation of steady-state or transient state probabilities of CTMCs, some optimisation problems can also be solved on the basis of MTBDDs (for instance in the context of concurrent probabilistic systems, where the problem is to find minimal or maximal probabilities associated with sets of paths, assuming that the system's behaviour is governed by a scheduler of a certain type [230]). In Chap. 5 we have already shown that the MTBDD representations of the rate matrix of a CTMC can be extremely memory-efficient, provided that it is constructed in a compositional

fashion. This memory-efficiency carries over to the generator matrix and (for some stationary iteration schemes) also to the iteration matrix, see the examples considered in Chap. 8 and Chap. 10.

In the previous sections, we always assumed that the vectors used within the numerical methods (probability vectors, residual vector, ...) are also represented by MTBDDs. It is clear that the MTBDD representation of a probability vector, for instance, may in the worst case become a full binary tree, namely in the case where all its 2^{n_s} entries are distinct, i.e. when no sharing of leaves or subgraphs is possible. For large n_s, the memory requirements for the vectors may thus become prohibitive.

The most serious problem of MTBDD-based numerical analysis is its CPU time consumption. It has been observed that MTBDD-based matrix multiplication and vector-matrix multiplication are about two orders of magnitude slower than sophisticated sparse-matrix implementations [18, 128, 102]. Basically, this has to do with the nature of the symbolic algorithms, which involve a possibly huge number of recursive calls. In principle, the number of recursive calls can be kept at a minimum with the help of the computed table, but this is only beneficial if the same computation (up to a scaling factor) is repeated several times and if the cache size is sufficiently large, which is not the case, for instance, if one of the operands is a large vector whose entries are all distinct. Therefore, even if a large matrix can be represented in a very compact way, its multiplication with a vector may require a lot of time. The fact that the involved MTBDDs are kept in canonical form at every step of the computation is another reason for the slow speed, since the maintenance of the unique table is costly [250].

From the above discussion, it is obvious that more research is needed in order to speed up (MT)BDD algorithms. Some approaches towards this aim are sketched in Chap. 8.

Chapter 8

Speeding up BDDs

We have shown that symbolic representations of stochastic labelled transition systems can be extremely space-efficient, in particular if the structure of the underlying system is taken into account during construction of the decision diagram. But unfortunately some of the algorithms for analysis, especially those for numerical analysis, may be slow. In this chapter, we present the results of some performance analysis studies, and discuss possible strategies for speeding up BDD-based algorithms.

8.1 Performance analysis of BDD algorithms

8.1.1 Measurement of iteration times

We start this section with some illustrative examples, studying the effect which structuredness of stochastic models (more precisely: SLTSs) and their MTBDD representations has on the time needed for numerical analysis. We compare the following three medium size models of approximately the same size:

- The model of a multiprocessor mainframe, taken from the literature [190, 166, 97, 170], and to be discussed in more detail in Sec. 10.3. This model is scalable (by adjusting the size of the multiprocessor's job queues), but here we only consider a fixed model size of 2,640 states, which has 12,295 transitions. On purpose, we did not consider the inherent structure of

the model[1], but simply encoded the monolithic transition system (which resulted from breadth-first search state space generation[2] and is very irregular) as an MTBDD. Note further that this model does not have any symmetrical states.

- The failure-repair model introduced in Sec. 6.2.1 with different numbers of components: 5 components of each type result in 1,024 states, 6 components of the first and 5 components of the second type result in 2,048, and 6 components of each type result in 4,096 states (note that a model with n components has $n \cdot |S|$ transitions, where $|S|$ is the size of the state space). We chose the non-lumped versions of these models, since this had shown to yield the most compact MTBDD representations. The models were generated in a compositional fashion by applying MTBDD-based parallel composition of the n components. As a result, the models are very regular and possess symmetric (lumpable) states.

- The M/M/1 - K queue introduced in Sec. 6.2.3 with state space size ranging from 1,024 to 4,096 (note that an M/M/1 - K queue with n states has $2n-2$ transitions). These models are also very regular and have very compact MTBDD representations, but they do not possess symmetric (lumpable) states.

Neither of these three models has unreachable states. However, we still performed reachability analysis, which in all cases took only a fraction of a second[3].

Tab. 8.1 shows some experimental results for these three models which were obtained with our tool Im-Cat [128, 129]. The table shows the number of states and transitions, the number of MTBDD vertices for the rate and the iteration matrix, as well as the time needed to construct the iteration matrix from the rate matrix (the power iteration scheme was used). The last two columns give the number of iterations until convergence and the mean computation time per iteration step[4].

We first have a closer look at the construction times for the iteration matrix: For the two scalable models, the construction time is proportional to the model size, which is as expected. For the mainframe model, although its MTBDD representation is about two orders of magnitude larger than those of the other models,

[1]The mainframe model was originally specified as a Stochastic Process Algebra model whose structure could of course be exploited during MTBDD construction, see Sec. 10.3.

[2]State space generation was carried out by TIPPtool [165].

[3]Note that the M/M/1 - K example constitutes the worst case for the reachability algorithm (cf. Sec. 5.1.5), since only a single new state is found at each step.

[4]All measured times in this thesis were recorded on a SUN 5/10 workstation, equipped with 1GB of main memory and running at 300 MHz.

model	states	transitions	rate matrix vertices	iteration matrix vertices	iteration matrix constr. time [sec]	iterations	mean time per iter. [sec]
mainframe	2,640	12,295	12,896	16,241	1.63	11,050	0.344
failure-	1,024	10,240	98	406	0.37	90	0.045
repair	2,048	22,528	107	476	0.84	100	0.095
	4,096	49,152	119	590	1.73	100	0.166
M/M/1 - K	1,024	2,046	69	107	0.33	6,010	0.034
queue	2,048	4,094	76	118	0.74	11,350	0.066
	4,096	8,190	83	129	1.75	21,820	0.128

Table 8.1: Comparing performance characteristics for different models (results obtained with the tool IM-CAT)

the construction time is very moderate. From this one can conclude that the operations involved in building the power iteration matrix (1 ABSTRACT operation, 1 DIAG operation, 2 APPLY operations, 1 SMULT operation and 1 generation of the identity matrix Id) do not constitute a major performance bottleneck of MTBDDs.

When we look at the last column, the picture is different. The mean time per iteration for the mainframe model is decisively longer than for the other two models. Even compared to the largest of the three failure-repair models (which has 55% more states and whose power iteration matrix has $\frac{49142+4096}{12295+2640} = 3.56$ times as many non-zeroes), the mainframe model needs more than twice the time per iteration. Since vector-matrix multiplication is the major operation during one iteration step, we may conclude that this is the critical operation and that the MTBDD sizes of the vector and matrix have a strong influence on the speed of that operation.

Note that Tab. 8.1 only gives the *mean* time per iteration. It is important to mention at this point that the time per iteration is not constant, but may vary considerably from iteration to iteration. This is quite obvious, since the MTBDD size of the probability vector may change considerably during iteration[5] and since the number of successful lookups of the computed table may vary. While we did

[5]For instance, in the M/M/1 - K case, suppose that all entries of the probability vector are initialised with the same value $1/|S|$, so the MTBDD representation is initially a single vertex. Since in the general case all steady-state probabilities are different, the final MTBDD representation of the probability vector is a full binary tree. However, in practise many of the probabilities are extremely small, such that they fall below the smallest representable number and become zero.

indeed observe large variations in the time per iteration, we did not pursue this phenomenon further, since it depends strongly on the particular case at hand and is hard to generalise.

8.1.2 Profiling MTBDD and BDD applications

In this section we present some results which were obtained from the profiling[6] of two applications which are both based on the decision diagram package CUDD [307].

The first application is the MTBDD-based tool Im-Cat whose behaviour was already studied in the previous section. We consider the same three example models as in the previous section, i.e. the mainframe model, the failure repair model and the M/M/1 queueing model. Both, for the failure-repair model and the M/M/1 model, the profile was obtained for the 2048 state instance.

The profiling results for these models are shown in Tables 8.2, 8.3 and 8.4. In each table, the twelve most time-consuming functions are listed. For each function, the relative and absolute CPU time, as well as the number of calls, are listed. The last column is introduced in order to categorise functions as follows: "apply" means that the function directly realises part of an Apply operation, "unique" means that the function performs unique table manipulation, "cache" means that the function is associated with the computed table, and "GC" means that the function is part of garbage collection. The functions "internal_mcount" and "_mcount" are responsible for collecting the measurement data. Therefore, the first column contains corrected time percentages, which are obtained by normalising the figures in the second column with respect to the "non-profiling" times.

In all profilings of Im-Cat reported here, the measured execution comprised model generation and subsequent steady-state analysis with the power method. The profiling results for the mainframe model and for the M/M/1 queueing model (shown in Tables 8.2 and 8.4) turn out to be very similar. In these two profiles, the most time-consuming function is "addMMRecur" which is responsible for the vector-matrix multiplication. It requires 30.95% resp. 27.39% of the total time. In addition to this fact, one has to take into account that almost all calls to the functions "cuddAddApplyRecur" and "Cudd_addPlus", which take 4.17%+3.51% resp. 3.51%+2.48%, are invoked directly or indirectly by "addMMRecur". The function second from the top is "cuddUniqueInter" with 16.27% resp. 18.52%,

[6] We used the standard Unix utility "gprof" to obtain the profiling information.

corr. time [%]	time [%]	time (self) [sec]	no. of calls	name	type
30.95	24.93	1187.21	12156	addMMRecur	apply
–	18.47	879.40		internal_mcount	
16.27	13.11	624.39	398705479	cuddUniqueInter	unique
14.14	11.39	542.35	1394	cuddGarbageCollect	GC
10.85	8.74	416.30	805418550	cuddCacheLookup2	cache
7.32	5.90	280.80	206994095	cuddUniqueConst	unique
5.99	4.83	229.99	639916819	Cudd_RecursiveDeref	
4.50	3.63	172.62	805186491	cuddCacheInsert2	cache
4.17	3.36	160.01	318977473	cuddAddApplyRecur	apply
3.51	2.83	134.83	685066712	Cudd_addPlus	apply
1.85	1.49	71.12	171989955	cuddAllocNode	
–	0.97	46.03		_mcount	
0.17	0.14	6.67		sum of other functions	

Table 8.2: Profiling results (Im-Cat, mainframe example)

corr. time [%]	time [%]	time (self) [sec]	no. of calls	name	type
–	24.69	7.01		internal_mcount	
13.58	10.04	2.85	56	cuddGarbageCollect	GC
12.10	8.95	2.54	2792127	cuddUniqueInter	unique
10.43	7.71	2.19	111	addMMRecur	apply
8.67	6.41	1.82	3594582	cuddCacheLookup2	cache
8.33	6.16	1.75	4766361	Cudd_RecursiveDeref	
6.19	4.58	1.30	1087581	cuddAddApplyRecur	apply
4.76	3.52	1.00	1894256	cuddReclaim	
4.00	2.96	0.84	3547670	Cudd_addPlus	apply
2.85	2.11	0.60	1146771	cuddUniqueConst	unique
2.34	1.73	0.49	2633769	cuddCacheInsert2	cache
–	1.37	0.39		_mcount	
26.76	19.79	5.62		sum of other functions	

Table 8.3: Profiling results (Im-Cat, 2048 state failure-repair model)

corr. time [%]	time [%]	time (self) [sec]	no. of calls	name	type
27.39	22.91	229.81	12486	addMMRecur	apply
–	15.55	156.01		internal_mcount	
18.52	15.49	155.39	82013579	cuddUniqueInter	unique
14.33	11.98	120.21	600	cuddGarbageCollect	GC
11.09	9.28	93.07	71182957	cuddUniqueConst	unique
8.35	6.98	69.97	131289527	cuddCacheLookup2	cache
6.39	5.34	53.61	97947456	Cudd_RecursiveDeref	
3.70	3.10	31.10	130716410	cuddCacheInsert2	cache
3.51	2.94	29.45	48366670	cuddAddApplyRecur	apply
2.79	2.34	23.50	59086329	cuddAllocNode	
2.48	2.08	20.90	114859297	Cudd_addPlus	apply
–	0.82	8.27		_mcount	
1.42	1.19	11.94		sum of other functions	

Table 8.4: Profiling results (Im-Cat, M/M/1 - 2048 example)

corr. time [%]	time [%]	time (self) [sec]	no. of calls	name	type
15.34	15.22	50.43	–	cuddUniqueInter	unique
10.16	10.08	33.38	–	cuddCacheLookup	cache
10.00	9.93	32.90	–	cuddBddAndAbstractRecur	apply
6.74	6.69	22.16	–	ddClearFlag	
5.27	5.23	17.33	–	cuddCacheLookup1	cache
4.69	4.66	15.45	–	cuddBddVarMapRecur	
4.59	4.55	15.07	–	ddCountMintermAux	
4.41	4.38	14.50	–	Cudd_RecursiveDeref	
3.39	3.37	11.18	–	ddDagInt	
3.37	3.34	11.08	–	ddLeavesInt	
3.33	3.30	10.92	–	cuddBddAndRecur	unique
2.96	2.94	9.73	–	cuddCheckCollisionOrdering	
...					
–	0.78	2.59	–	internal_mcount	
25.73	25.53	84.59		sum of other functions	

Table 8.5: Profiling results (NANOTRAV, "rcn25" circuit)

the function responsible for the lookup and manipulation of the unique table. Garbage collection takes 14.14% resp. 14.33%, and the cache lookup and insert functions also require substantial time (10.85% + 4.50% resp. 8.35% + 3.70%). In Tab. 8.3, the picture is somewhat different, i.e. "addMMRecur" is on position three with only 10.43%, but the set of the twelve topmost functions is almost identical with the ones in Tables 8.2 and 8.4. The reason for this difference lies mainly in the small number of iterations performed for the failure-repair model, cf. Tab. 8.1.

The second application which we profiled is NANOTRAV, a simple traversal (i.e. reachability analysis) program for finite state machines that comes as part of the CUDD distribution and works with BDDs. The profile obtained from the analysis of the sequential circuit "rcn25" is shown in Tab. 8.5. Since NANOTRAV is totally different from IM-CAT, so are their profiles, and therefore it is hard to make a comparison. Apart from the results shown in Tab. 8.5, we profiled several other runs of NANOTRAV. The results varied quite a bit, depending strongly on the type of circuit to be analysed by the program and on the variable reordering options. However, an observation common to all these measurements is that the apply-type functions require a much smaller portion of the overall time (only 10.0% in Tab. 8.5) than for the MTBDD application. The second observation is that garbage collection plays only a minor role in NANOTRAV. With NANOTRAV, there was never such a strong weight on one particular function, as we had observed for the function "addMMRecur" in the above IM-CAT profiles.

From these profilings, as well as from several other ones which we carried out, it is clear that the numerical calculation dominates the execution time of the Im-Cat application, which is our main focus of interest. Keeping in mind that compositional model construction and other operations, such as abstraction from action names or reachability analysis, are extremely fast, one can conclude that numerical analysis, in particular the vector-matrix multiplication, is the main performance bottleneck of MTBDD applications such as Im-Cat. Therefore, in future, more time and effort will have to be spent in order to overcome, or at least alleviate, this bottleneck.

8.2 Optimised representations, algorithms and implementations

The results from Sec. 8.1 show that (as expected) the speed of BDD operations, and in particular of MTBDD-based vector-matrix multiplication, is highly dependent on the size of the operands. Therefore it is extremely important to keep the decision diagrams as small as possible. In most cases, applying the heuristics described in Chap. 6 results in compact representations. Of course, it would be possible to apply further techniques, such as the dynamic reordering of Boolean variables during the construction of the decision diagram, or even during its analysis. These techniques have been widely studied for BDDs [130, 200, 282, 35, 34, 191], but it is still an open question whether they could also be applied beneficially, i.e. with reasonable cost-gain ratio, to problems which involve numerical calculations on MTBDDs. Furthermore, it would also be interesting to investigate whether other classes of decision diagrams, such as the class of K*BMDs [111, 109, 110] or modifications thereof, could be useful for problems involving real-valued functions. These are open problems for future research.

Apart from minimising the size of the decision diagrams, one may work on improving the implementation of the time-critical algorithms. Sometimes, improvements can be achieved by carefully adjusting some of the implementation's parameters. For instance, [128] experimented with the size of the cache (i.e. the computed table) used during iterative numerical analysis. Quite against the intuition, it turned out that, for certain combinations of problem size and iterative method, a decrease of the cache size actually led to an acceleration of the computation. However, the outcome of these experiments depended very much on the particular problem at hand, and no general optimisation strategy could be found.

As described in Sec. 4.4.1, various MTBDD-based algorithms have been developed for matrix multiplication (and thus vector-matrix multiplication, which is

our focus since it is the most time-critical part of our application), but it is questionable whether these algorithms can be improved upon very much without changing their basic strategy. Such a change of paradigm has been proposed by Parker [268] and will be described in his forthcoming Ph.D. thesis [269]. The basic idea is as follows: For the iterative solution of a linear system of equations, the iteration matrix is represented by an MTBDD (which is somewhat modified[7]), but the solution vector is represented by a linear data structure, which can lead to a dramatic reduction of the memory requirements for the vector (especially in problem instances with many unreachable states). Parker developed a new, specialised vector-matrix multiplication algorithm for this so-called hybrid approach, and experiments yielded considerable speedups compared to the fully MTBDD-based algorithms.

When one considers the recursive nature of the BDD algorithms, and in particular that of vector-matrix multiplication, another idea comes to the mind. It should be possible to speed up the algorithms through parallel processing, i.e. executing several recursive calls at the same time on different processors. This idea is certainly worth an investigation, but it is clear that the use of common data structures, such as the unique table and possibly the computed table, make the parallelisation of apply-type algorithms non-trivial. Parallelisation is also an attractive idea in combination with specialised, dedicated hardware, which is the topic of the next section.

8.3 Specialised hardware

The high CPU-time consumption of some BDD and MTBDD algorithms is an impeding factor for the success of symbolic methods, especially during the numerical analysis of stochastic transition systems. The use of dedicated hardware offers a substantial potential for speeding up BDD-based applications. Therefore, in this section, we report on a project in which we designed a prototype of a BDD coprocessor [250].

8.3.1 Design of a BDD coprocessor

The aim of this project was to design a coprocessor that performs certain time-intensive functions for BDD manipulation, similar to the way an arithmetic coprocessor performs arithmetic operations.

[7] In this approach, the MTBDD vertices are augmented by integer offsets from which the actual indices of matrix elements in terms of reachable states only can be computed [229].

Since a hardware implementation of a comprehensive set of BDD and MTBDD operations is quite a formidable task, it was decided at the outset to realise only some functions in hardware and leave the remaining parts in software. We decided to follow the design of the MTBDD package CUDD [307], in particular to adopt its data structures, such that the newly developed hardware can easily interact with existing CUDD applications (such as Im-Cat [128, 129] or Prism [102, 231]).

Based on profiling results, such as the one shown in Tab. 8.2, a subset of the CUDD functions was chosen for hardware implementation. The function "addMMRecur", as well as other apply-type functions such as "cuddAddApplyRecur", are very complex, such that we did not find them suitable for immediate hardware realisation. In addition, these functions make extensive use of the cache-type and unique-type functions. If one wanted to realise one of the apply-type functions in hardware, one would have to realise the cache-type and unique-type functions as well, since otherwise the hardware would have to initiate calls to the software, which does not make sense. Since the functions responsible for garbage collection are also quite complex, we chose the unique-type function "cuddUniqueInter" and the two cache-type functions "cuddCacheInsert2" and "cuddCacheLookup2" for hardware realisation. This choice has the advantage that the hardware can be extended by one or several apply-type functions at a later stage, since all functions that are called by "apply" would then already be available in hardware. It is obvious, that the overall performance gain that can be achieved by this choice is rather modest, since the cumulated time percentage of the chosen functions is quite small, e.g. only 31.62% for the profile shown in Tab. 8.2.

One problem that had to be overcome is the fact that the chosen functions contain calls to other, lower-level functions[8], some of which use services of the operating system (e.g. memory allocation). The functions chosen for hardware implementation had to be modified in such a way that calls to other functions take place either before or after — but never during — the hardware operation. While the two cache-type functions required only little modification, the function "cuddUniqueInter" needed to be changed considerably: Garbage collection and the resizing of a hash-table are delayed until after a new vertex has been inserted, which requires a temporary reference to the new vertex. Another critical situation occurs if a new vertex has to be inserted, but the list of free vertices is empty. In this case, the hardware terminates, signalling to the software that the list of free vertices should be refilled (either by garbage collection or by allocation of additional memory). When this has been done, the hardware is restarted.

[8]These are mostly functions for memory management, such as "reclaim", "cacheResize", "rehash" and "garbageCollect".

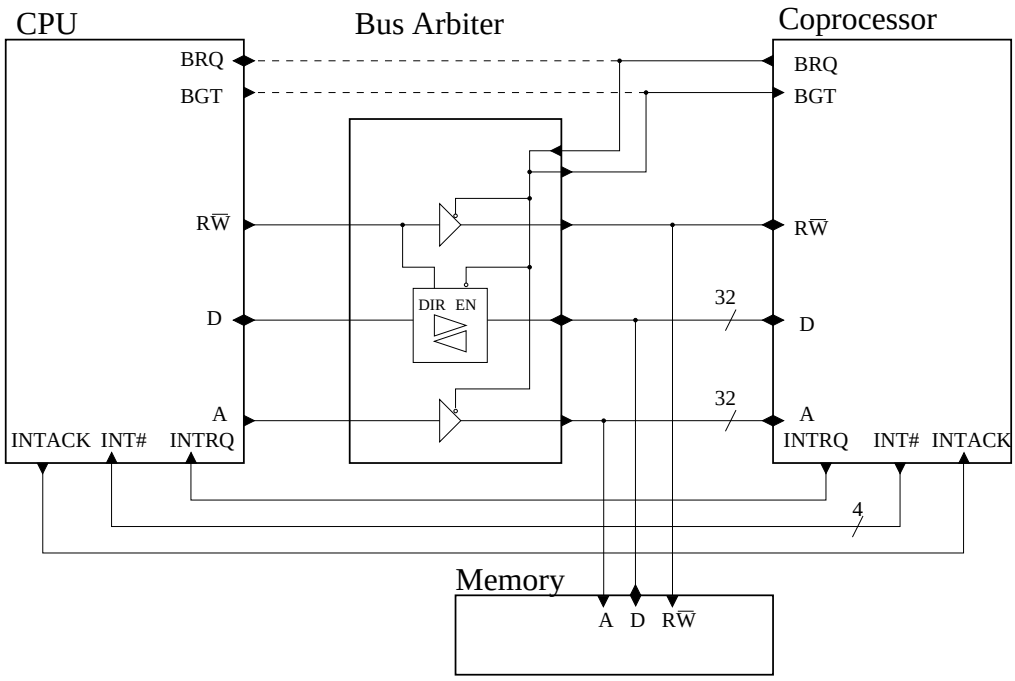


Figure 8.1: Overall configuration, consisting of CPU, coprocessor, bus arbiter and memory

The hashing function employed by CUDD, which is needed both in "cuddUniqueInter" and in the two cache-type functions (albeit with different number of operands), was analysed in detail. It turned out that it can be realised very efficiently in hardware, based on the operations multiplication, masking (selection of the most significant bits) and shifting.

The overall configuration, including the CPU, the BDD coprocessor, as well as a bus arbiter and the main memory, is shown in Fig. 8.1. A bus arbitration unit is needed, since both the CPU and the coprocessor need access to the memory, where the following access modes are possible:

1. The CPU is the bus master, while the memory and the ports[9] of the coprocessors are the slaves.

2. The coprocessor is the bus master, while the memory is the slave.

For CPUs which are capable of isolating themselves from the bus by tristating their bus lines, controlled by bus request (BRQ) and bus grant (BGT) signals,

[9]We use the term "port" to denote storage elements within the coprocessor to which both the CPU and the coprocessor have access.

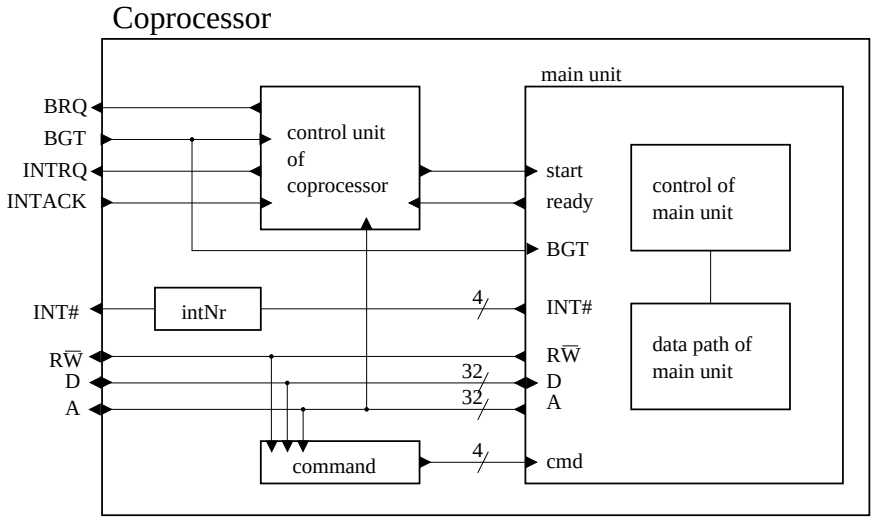


Figure 8.2: Block diagram of the coprocessor

the bus arbitration unit may be omitted. In this case, the BRQ and BGT pins of the CPU and the coprocessor can be directly connected, as indicated by the dashed lines in Fig. 8.1.

The execution of a coprocessor command proceeds as follows: Initially, the CPU is the bus master and therefore allowed to read and write data from/to memory and from/to the coprocessor ports. Thus, operands can be written by the CPU to the ports of the coprocessor. After the CPU has written the code of the command to be performed by the coprocessor to the coprocessor's command port, it puts itself to sleep, waiting in a loop until the coprocessor is finished. The control unit of the coprocessor listens to the address bus, such that it can recognise when a command code is written to the command port. When a new command has been issued, the coprocessor acquires the bus by pulling BRQ to high and waiting for a high on BGT. Now the coprocessor can do its work and as bus master is able to read and write to memory. When finished with its work, the coprocessor pulls BRQ to low, thereby releasing the bus. Now the coprocessor sends an interrupt to the CPU, which reads the interrupt number and sends back an acknowledgement. The wires INTRQ, INT# and INTACK are used for this purpose. Afterwards the CPU, being again the bus master, may read data necessary for further processing from the ports of the coprocessor.

The block diagram of the coprocessor is shown in Fig. 8.2. Of its ports, only the command port and the intNR port are shown explicitly. The control unit realises the high-level interaction with the CPU. Its state machine, shown in Fig. 8.3, works as follows: After a reset, the coprocessor is in state "idle". When

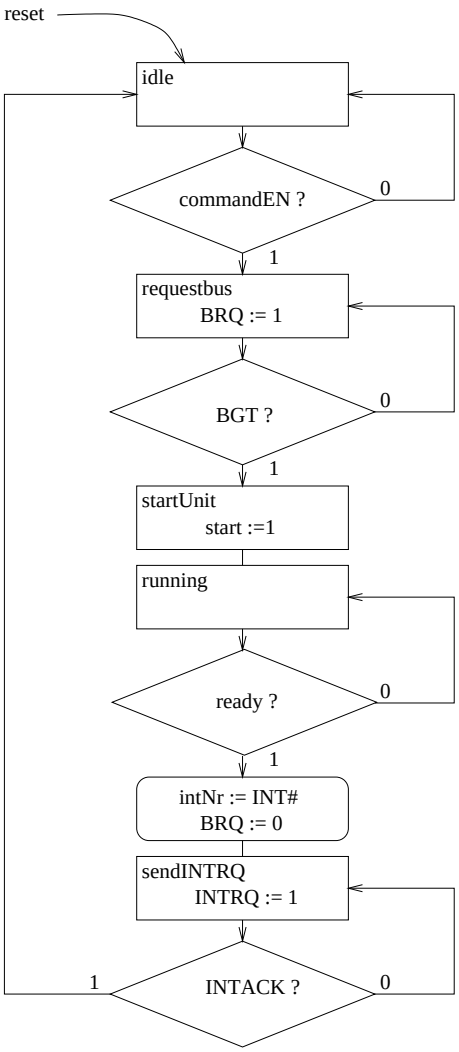


Figure 8.3: State machine of the coprocessor's control unit

a command has been written by the CPU to the command port, the control moves to state "requestbus" where BRQ is pulled to high. When the bus is granted, state "startUnit" is reached, and the main unit is told by the start signal that it can execute the command. The control unit now remains in state "running", waiting for the main unit to complete execution, which is indicated by the ready signal. Then the interrupt number is loaded into the intNr port, BRQ is reset to low, and in the next state ("sendINTRQ") signal INTRQ is pulled to high, after which the control unit waits for the acknowledgement of the interrupt by the CPU, indicated by signal INTACK. Once this signal is received, the control unit returns to state "idle". Note that the control components and the data path within the main unit contain many details which are not shown in Fig. 8.2.

The complete coprocessor was specified in VHDL [72, 134] and simulated with the help of the standard tool SYNOPSIS. For testing the functionality of the coprocessor, and for debugging purposes, a test-bench was written. The test-bench plays the roles of both the CPU and the memory, and can be configured via a command file.

Since the actual hardware realisation of the BDD coprocessor is not yet operational at this time, we cannot present measurement results. Instead, we conducted a performance estimation, which is based on counting the number of states which the coprocessor visits when executing a particular command. This figure is compared to the number of assembler operations of the corresponding software solution. The comparison is based on the assumption that the CPU and the coprocessor work at the same clock frequency, and that the CPU is a RISC processor which executes one assembler operation per clock cycle. These are, of course, rather simplifying assumptions. Since the processing of a particular coprocessor command may contain numerous loops and branches (both in the software and in the hardware realisation), the counting of assembler operations, resp. states, needed to distinguish between many different cases. Analysing the function "cuddCacheInsert2" was simple, since it does not have loops or branches. Here, the speedup achieved through the hardware realisation is 6.8. When analysing the function "cuddCacheLookup2", 14 different combinations of branches had to be considered, and the speedup varied between 3.9 and 5.3. Analysing the function "cuddUniqueInter" was the most complicated, since that function contains a loop that can have different length, depending on the actual data. Software and hardware execution times were thus calculated for different measured parameters. For the relevant cases, the speedup was between 2.2 and 2.7.

We plan to realise the coprocessor by using an FPGA board. Today's FPGAs are large enough to accommodate not only a piece of specialised hardware of the size of our coprocessor, but also additional logic, in some cases even a complete microprocessor. This is our strategy indeed. We will employ the Altera Excalibur Development Kit [9], which includes a configurable RISC processor (called NIOS) that can be placed on the FPGA (an APEX EP20K200E device) together with the coprocessor logic. The advantages of such a configuration are obvious: The CPU executing the software code (in our case the CUDD-based tool ImCat) and the specialised hardware (in our case the BDD coprocessor) actually reside on the same chip, which means that communication between the CPU and the coprocessor is very fast, and the overhead for activating the coprocessor is minimised.

8.3.2 Lessons learned and future research

The software/hardware realisation described in Sec. 8.3.1 resulted in a moderate acceleration of the selected BDD operations. Considerable gain of speed for full BDD and MTBDD applications would be achieved if a comprehensive set of BDD operations, especially including functions such as "addMMRecur", were realised in hardware. This is feasible, but requires substantial effort.

However, it has become clear that, in order to achieve a real performance breakthrough, one must not stick to conventional algorithms and should not use unmodified data structures, as we had done. Reconsidering our software/hardware realisation, note that at any point of time, either the CPU or the coprocessor is active. The CPU and the coprocessor are never active concurrently, nor is there substantial parallelism within the coprocessor (apart from low-level parallelism, such as the concurrent calculation of subexpressions of the hashing function). In principle, much higher speedups could be obtained if one were able to exploit the concurrency which is inherent in the BDD algorithms of recursive nature. Therefore, as we already mentioned in Sec. 8.2, one stream of research should aim at developing new strategies for apply-type algorithms, in particular for matrix multiplication and vector-matrix multiplication, which could exploit the power of parallel hardware.

Part III

Beyond performance analysis

Chapter 9

Verification of stochastic systems

In the previous chapters, we discussed performance and dependability analysis, with a strong focus on compact symbolic representations and algorithms that work on these representations. In the present chapter, we motivate the need to widen the horizon of classical analysis, which will lead us to the formal specification of measures in the form of temporal logic performability properties that should hold for a given system, and algorithms for "checking" whether these properties are indeed satisfied. In Sec. 9.3, we introduce the action-based logic aCSL, a logic that is tailored for specifying properties of stochastic process algebra models, and discuss algorithms for model checking this logic against stochastic labelled transition systems. At the end of the chapter, in Sec. 9.5, we discuss the strong relation between model checking and compact symbolic representations based on BDDs, which brings us back to the main topic of this thesis.

9.1 From performance analysis towards the verification of performability properties

In classical performance evaluation, the system to be analysed is represented by a stochastic model, be it a queueing network, a stochastic Petri net or a stochastic process algebra specification. The aim of analysis, i.e. the measures of interest which the modeller wishes to derive, is described informally or in a tool-dependent way. For instance, the modeller may wish to obtain the mean waiting time of customers at a particular station of a queueing network, the throughput of a particular transition of a SPN, or the probability that at a certain instant of time the sum of the number of tokens in a given subset of places does not exceed

a particular value. In the SPA tool TIPPTOOL, for instance, it is possible to specify three kinds of performance measures:

- State measures, which accumulate the (stationary or transient) state probabilities of all states which match a given regular expression.
- Throughput measures, which yield the frequency with which a certain (timed) action takes place.
- Mean values: In the presence of parameterised processes where one parameter represents a counter, the mean value of that parameter can be determined.

In the area of performability modelling, where the aim is the combined analysis of performance and dependability properties, it is common to work with Markov reward models [158]. In such models, states are associated with reward rates and transitions may be associated with impulse rewards. Typical measures of interest for Markov reward models are the mean stationary reward rate, the accumulated reward up to a certain time, or the accumulated reward up to a special event.

In many situations, the derivation of standard measures such as the ones mentioned above is not enough. Modellers may wish to ask more complicated questions, for example:

- "Once the system is in a state from subset T, what is the probability of reaching a state from subset U, without passing through a state from subset V in between?"
- "Starting from a state in subset T, is the probability to reach a state in subset U within 6.5 time units at least 0.8?"
- "Once event a has taken place, what is the probability of event b taking place after at most 13 time units, without event c taking place in between?"
- "What is the probability, that events a, b and c take place in that order within at most 3.7 time units?"
- "Starting from state s, is the probability that an arbitrary number of a-events followed by a single b-event take place within 39 time units greater than 0.95?"

Such questions can neither be formulated nor answered using the above mentioned classical techniques for measure specification in Markov models and Markov reward models. Although it is, in principle, possible to compute the figures that

characterise the outcome of these questions, this is usually a difficult and tricky task which requires measure-dependent modification of the model and complicated numerical calculations. In the past, such calculations could therefore only be carried out manually by an experienced modeller.

These considerations motivate the use of temporal logics as an expressive specification mechanism for complex performance measures. The general idea of this approach is to specify temporal logic properties (i.e. requirements) which the system under investigation should satisfy, and to employ model checking techniques in order to check whether a given property actually holds. This is the approach which we will follow in this chapter. It leads us from the area of performability evaluation to the area of verification of stochastic systems[1] by means of model checking.

9.2 Model checking of stochastic systems

In this section, we give an overview of model checking techniques for stochastic systems. However, we first have a brief look at classical, i.e. purely functional model checking.

Model checking is a successful technique to establish the correctness of a given model, relative to a set of temporal logic properties which the model should satisfy [87, 88]. The most efficient model checkers use the logics LTL (Linear Temporal Logic, [276]) or CTL (Computation Tree Logic, [116]) which are interpreted over Kripke structures, i.e. transition systems where states are labelled with elementary properties, called atomic propositions, but transitions are unlabelled. Though different in nature, both logics are state-oriented, their basic building blocks being atomic propositions which characterise states. While the model checking of LTL follows an automata-based approach [304], model checking of CTL proceeds by induction on the parse tree of the formula [85].

Classical temporal logics specify purely functional properties that focus on the temporal ordering of events, without taking into account quantified time or probabilities. Real-time logics have been developed, for instance Timed CTL (TCTL) [10], where the quantitative timing behaviour of timed automata [11] is characterised, but these logics are tailored for the specification and verification of hard real-time deadlines and do not comprise stochastic behaviour. Since we are in-

[1] We use the term "verification of stochastic systems" rather than "stochastic verification", since it is the system under investigation which has stochastic features, while the verification methods as such are not stochastic or randomised, but rather deterministic.

terested in performability properties of stochastic systems, we need to develop logics that are extended accordingly.

Probabilistic real-time CTL (PCTL) is an extension of CTL for expressing properties that concern both real-time and probability [148]. PCTL is interpreted over state-labelled DTMCs, a discrete time model where every transition requires one unit of time. Typical examples of properties expressible in PCTL are "With probability at least 0.5, property Φ_1 will hold at some time within the next 20 time units" or "With probability at least 0.99, property Φ_2 will hold continuously for 20 time units".

The Continuous Time Stochastic Logic (CSL) [21] adopts operators of PCTL, but is used to express properties over state-labelled CTMCs where time is continuous. It is based on the equally named logic of Aziz et al. [14], extended with an operator to reason about steady-state properties. A typical requirement that can be formulated with the help of CSL is "The probability that an error state will occur within 17.5 time units is less than 0.001". In general, a time-bounded until operator and a probabilistic operator are used to assert that the probability for a certain event meets given bounds. For model checking time-bounded until-formulas, a system of Volterra integral equations needs to be solved, which is the computationally hardest part of CSL model checking. While the original paper [21] suggested an approximate numerical solution technique which works on discretised distribution functions, it has since been shown that the solution can also be computed by performing standard transient analysis (by means of uniformisation) on a modified CTMC [19], and that the latter technique is numerically superior [173].

We also briefly mention the work of de Alfaro et al. [102], where the probabilistic branching-time temporal logic (PBTL), a derivative of CTL and PCTL, is used for specifying requirements on Markov decision processes, also called concurrent probabilistic systems. This model generalises DTMCs in the sense that there may be a non-deterministic choice between several probability distributions emanating from a given state. As a result, the model checking of such systems involves the solution of a linear optimisation problem [230].

9.3 Action-based logics

Stochastic process algebras represent a formalism to carry out performance and reliability modelling in a compositional way. As opposed to traditional performance modelling techniques, their action-oriented style, where processes are char-

acterised by the sequences of actions which they can perform, supports composition and abstraction in a natural way. However, even though in an action-oriented setting the notion of a state is an auxiliary one, the analysis of stochastic process algebra models is usually carried out in a state-oriented fashion, because standard numerical analysis is based on the calculation of (transient or stationary) state probabilities. This disturbing shift of paradigm, moving from action-oriented specification to state-oriented analysis, hampers the acceptance of the process algebraic approach.

It is therefore mandatory to develop an entirely action-oriented analysis technique for stochastic process algebras. We will do this by following a logic-based approach, where sequences of actions are analysed by means of model checking algorithms. Logics such as LTL, CTL or stochastic extensions thereof do not fit in well with an action-based formalism, since they are based on state labellings. Therefore, we develop an action-based temporal logic for describing behaviours of interest, together with a model checking algorithm to derive the probability with which a stochastic process algebra model exhibits such a behaviour.

In order to enable model checking within action-oriented formalisms, de Nicola and Vaandrager developed an action-based variant of CTL, called aCTL [262, 263][2]. Although aCTL is action-oriented, it naturally corresponds to CTL. There even exists a translation from aCTL to CTL that allows one to perform action-oriented aCTL model checking by means of state-oriented CTL model checking on a transformed model, with only linear overhead. We mention, however, that direct aCTL model checkers have become more popular [119, 242] than the translational approach.

In the remaining part of this chapter, we describe the action-based, branching-time stochastic logic aCSL (action-based Continuous Stochastic Logic), that is strongly inspired by CSL (see above, Sec. 9.2). Similar to CSL, aCSL provides means to reason about CTMCs, but opposed to CSL, it is not state-oriented. Its basic constructors are sets of actions, instead of atomic state propositions (although, in principle, the latter could easily be added, thereby enabling a unified action-oriented and state-oriented approach). The logic provides means to specify temporal properties with quantified continuous time, and means to quantify their probability. As a simple example, aCSL allows one to specify properties such as "Once action *send* has been observed, there is at least a 30% chance that action *ack* will be observed within at most 4.75 time units".

The rest of this section, which is based on [170], is structured as follows: Af-

[2]The logic aCTL should not be confused with the logic ACTL, the restriction of CTL to universal path quantifiers.

ter defining syntax and semantics in Sec. 9.3.1, we develop a dedicated model-checking algorithm for aCSL (Sec. 9.3.2). In Sec 9.3.3, we then show that Markovian bisimulation, an equivalence which can be used to compress SPA specifications compositionally, preserves aCSL properties. Finally, in Sec. 9.3.4, we discuss possible translations of aCSL to state-oriented formalisms.

9.3.1 Syntax and semantics of the logic aCSL

The aCSL model. In the sequel, we consider an SLTS, i.e. an action-labelled Markov chain, $\mathcal{T}$. Its set of states is denoted by S, Act is a set of action labels, and $\rightarrow \subseteq S \times Act \times \mathbb{R}^{>0} \times S$ is the transition relation. $A, B \subseteq Act$ are sets of actions. We assume that $\mathcal{T}$ is finite, i.e., has a finite number of states and is finitely branching. We use the following notation:

$$\begin{aligned} R_A(s,s') &= \sum_{a\in A} \{\, \lambda \mid s \xrightarrow{a,\lambda} s' \,\} \\ E(s) &= \sum_{s'\in S} R_{Act}(s,s') \\ p_A(s,s') &= R_A(s,s')/E(s). \end{aligned}$$

Stated in words, $R_A(s,s')$ denotes the cumulative rate of moving from state s to s' with some action from A, $E(s)$ denotes the total rate with which some transition emanating from s is taken, and $p_A(s,s')$ is the probability of moving from state s to s' by an action in A. For absorbing s, $E(s) = 0$ and $p_A(s,s') = 0$ for any state s' and any set A. Further note that $R_\emptyset(s,s') = p_\emptyset(s,s') = 0$ for any states s, s'.

An infinite path σ is a sequence $s_0 \xrightarrow{a_0,t_0} s_1 \xrightarrow{a_1,t_1} s_2 \xrightarrow{a_2,t_2} \ldots$ with (for $i \in \mathbb{N}$) $s_i \in S$, $a_i \in Act$ and $t_i \in \mathbb{R}^{>0}$, such that $R_{a_i}(s_i, s_{i+1}) > 0$. Let $\sigma[i] = s_i$, the $(i{+}1)$-st state of σ, and $\delta(\sigma, i) = t_i$, the time spent in $\sigma[i]$. For $t \in \mathbb{R}^{\geq 0}$ and i the smallest index with $t \leq \sum_{j=0}^{i} t_j$, let $\sigma@t = \sigma[i]$, the state in σ at time t. A finite path σ is a sequence $s_0 \xrightarrow{a_0,t_0} s_1 \xrightarrow{a_1,t_1} s_2 \ldots s_{l-1} \xrightarrow{a_{l-1},t_{l-1}} s_l$ where s_l is absorbing, and $R(s_i, s_{i+1}) > 0$ for all $i < l$. For finite σ, $\sigma[i]$ and $\delta(\sigma, i)$ are only defined for $i \leq l$; they are defined as above for $i < l$, and $\delta(\sigma, l) = \infty$. For $t > \sum_{j=0}^{l-1} t_j$ let $\sigma@t = s_l$, otherwise $\sigma@t$ is as above. We denote $\sigma[i] \xrightarrow{A} \sigma[i{+}1]$ whenever $\sigma[i]$ can move to $\sigma[i{+}1]$ by performing some action in A, i.e., if $a_i \in A$. Note that $\sigma[i] \not\xrightarrow{\emptyset}$. Let $Path(s)$ denote the set of paths starting in s. A Borel space over $Path(s)$ can be defined in a similar way as in [21] and is omitted here.

Syntax of aCSL. We now describe the action-based stochastic logic aCSL, starting with its syntax.

Definition 9.3.1 Syntax of aCSL
For $p \in [0,1]$ and $\bowtie \in \{\leq, <, \geq, >\}$, the state-formulas Φ of aCSL are defined by the grammar

$$\Phi ::= tt \;\Big|\; \Phi \wedge \Phi \;\Big|\; \neg\Phi \;\Big|\; \mathcal{S}_{\bowtie p}(\Phi) \;\Big|\; \mathcal{P}_{\bowtie p}(\varphi)$$

where path-formulas φ are defined for $t \in \mathbb{R}^{>0} \cup \{\infty\}$ by

$$\varphi ::= \Phi\ {}_A\mathcal{U}^{<t}\ \Phi \;\Big|\; \Phi\ {}_A\mathcal{U}^{\leq t}_B\ \Phi. \qquad \blacksquare$$

Note that atomic propositions are absent. tt stands for the constant "true", $\wedge$ denotes conjunction, and $\neg$ denotes negation. The other Boolean connectives, such as $\vee$ and $\Rightarrow$, are derived in the obvious way. The probabilistic operator $\mathcal{P}_{\bowtie p}(.)$ replaces the CTL path quantifiers $\exists$ and $\forall$. The state-formulas are directly adopted from CSL: $\mathcal{S}_{\bowtie p}(\Phi)$ asserts that the steady-state probability for a Φ-state meets the bound $\bowtie p$, and $\mathcal{P}_{\bowtie p}(\varphi)$ asserts that the probability measure of the paths satisfying φ meets the bound $\bowtie p$.

The path-formula $\Phi_1\ {}_A\mathcal{U}^{<t}\ \Phi_2$ is fulfilled by a path if a Φ_2-state is eventually reached (after at most t time units) via intermediate states which all satisfy Φ_1, while taking only A-transitions. Slightly more discerning, the formula $\Phi_1\ {}_A\mathcal{U}^{\leq t}_B\ \Phi_2$ is fulfilled by a path if a Φ_2-state is eventually reached (after at most t time units) via intermediate states which all satisfy Φ_1, while taking an arbitrary number of A-transitions, followed by a single B-transition. Note the following: Due to the fact that the Φ_2-state must be reached via a B-transition, the formula $\Phi_1\ {}_A\mathcal{U}^{\leq t}_B\ \Phi_2$ is invalid in a $(\neg\Phi_1 \wedge \Phi_2)$-state s: Although the state satisfies Φ_2, it is not able to move from a Φ_1-state to a Φ_2-state via a B-transition as it does not fulfil Φ_1. However, the formula $\Phi_1\ {}_A\mathcal{U}^{<t}\ \Phi_2$ is valid in state s, since for the validity of this formula it is not required that a transition into a Φ_2-state is actually taken. Thus, whereas for $\Phi_1\ {}_A\mathcal{U}^{<t}\ \Phi_2$ it suffices to be currently in a Φ_2-state, this is not the case for $\Phi_1\ {}_A\mathcal{U}^{\leq t}_B\ \Phi_2$.

The major differences with a standard, untimed until-formula $\Phi_1\,\mathcal{U}\,\Phi_2$ of linear and branching temporal logics are that restrictions are put on the action labels of transitions to be taken and on the amount of time that is needed to reach a Φ_2-state. The standard until-formula can be derived in the following way:

$$\Phi_1\,\mathcal{U}\,\Phi_2 = \Phi_1\ {}_{Act}\mathcal{U}^{<\infty}\ \Phi_2.$$

We will use $\Phi_1\ {}_A\mathcal{U}_B\ \Phi_2$ as an abbreviation of $\Phi_1\ {}_A\mathcal{U}^{\leq\infty}_B\ \Phi_2$ and $\Phi_1\ {}_A\mathcal{U}\ \Phi_2$ as an abbreviation of $\Phi_1\ {}_A\mathcal{U}^{<\infty}\ \Phi_2$. These are the untimed versions of the until operators ${}_A\mathcal{U}^{\leq t}_B$ and ${}_A\mathcal{U}^{<t}$.

Derived operators. In aCSL, the following set of next-operators are all derived operators:

$$\begin{aligned} X_A^{<t}\,\Phi &= tt\ {}_\emptyset\mathcal{U}_A^{<t}\,\Phi \\ X_A\,\Phi &= X_A^{<\infty}\,\Phi \\ X\,\Phi &= X_{Act}\,\Phi. \end{aligned}$$

The formula $X_A^{<t}\,\Phi$ asserts that from the current state an A-transition can be made to a Φ-state before time t. Note that the Φ-state must be reached by the first transition, as — due to the empty set of actions — further transitions are disallowed. X_A is the action-labelled next-operator from aCTL, whereas X is the traditional state-based next-operator.

In our logic, the next operator is derived from the until operator. In aCTL the reverse is the case [262]. This stems from the special treatment of internal, i.e., τ-labelled, transitions in aCTL. For instance in aCTL, $X_\emptyset\,\Phi$ allows to reach a Φ-state by an internal transition (but not any other). In our setting, internal transitions are treated as any other transition, and accordingly, $X_\emptyset\,\Phi$ is invalid for any state. We have made this difference deliberately: whereas aCTL is aimed to characterise branching bisimulation – a slight variant of weak bisimulation equivalence – we focus on characterising a strong equivalence like lumping equivalence (since exact weak equivalences on SLTS cannot be obtained [163]).

The temporal operator $\Diamond$, to be read "eventually", and its variants are derived in the following way:

$$\begin{aligned} {}_A\Diamond^{<t}\Phi &= tt\ {}_A\mathcal{U}^{<t}\,\Phi \\ {}_A\Diamond\ \Phi &= {}_A\Diamond^{<\infty}\,\Phi \\ \Diamond^{<t}\Phi &= {}_{Act}\Diamond^{<t}\,\Phi \end{aligned}$$

A path fulfils ${}_A\Diamond^{<t}\Phi$ if it reaches a Φ-state within t time units by only performing A-actions. Formulas ${}_A\Diamond\ \Phi$ and $\Diamond^{<t}\Phi$ denote the generalisations to infinite time and arbitrary actions. Their combination, $\Diamond\ \Phi$, corresponds to the well-known "eventually" operator. Even more discerning $\Diamond$-operators can be defined by

$${}_A\Diamond_B^{<t}\Phi \;=\; tt\ {}_A\mathcal{U}_B^{<t}\,\Phi \qquad \text{and} \qquad {}_A\Diamond_B\Phi \;=\; {}_A\Diamond_B^{<\infty}\,\Phi$$

Here, the path leading to the Φ-state consists of an arbitrary number of A-actions, followed by a single B-action. Dual to these $\Diamond$-operators is the set of $\Box$-operators, of which we only mention the following:

$$\mathcal{P}_{\bowtie p}\,({}_A\Box^{<t}\Phi) \;=\; \neg\mathcal{P}_{\bowtie p}\,({}_A\Diamond^{<t}\neg\Phi) \qquad \text{and} \qquad \mathcal{P}_{\bowtie p}\,({}_A\Box_B^{<t}\Phi) \;=\; \neg\mathcal{P}_{\bowtie p}\,({}_A\Diamond_B^{<t}\neg\Phi)$$

with the obvious generalisations to infinite time and/or arbitrary sets of actions. Existential and universal quantification are introduced as

$$\exists\varphi \;=\; \mathcal{P}_{>0}\,(\varphi) \qquad \text{and} \qquad \forall\varphi \;=\; \mathcal{P}_{\geq 1}\,(\varphi)$$

but note that by this definition formula $\forall\varphi$ holds even if there exists a path that does not satisfy φ, if that path has zero probability mass. For the treatment of such fairness issues see [22]. Finally, we consider the modal operators from Hennessy-Milner logic [160] and the mu-calculus [227] as derived operators. They are obtained as follows:

$$\langle A\rangle\,\Phi \;=\; \mathcal{P}_{>0}\,(X_A\,\Phi) \qquad \text{and} \qquad [A]\,\Phi \;=\; \neg\langle A\rangle\,\neg\Phi.$$

The modal operator $\langle A\rangle\,\Phi$ states that there is some A-transition from the current state to a Φ-state, whereas $[A]\,\Phi$ states that for all A-transitions from the current state a Φ-state is reached.

Semantics of aCSL. aCSL state-formulas are interpreted over the states of an SLTS $(S, A, \longrightarrow)$[3]. Before we are in a position to formally define the semantics of aCSL state-formulas, we must first define when a given path satisfies a path-formula. The meaning of the path-operators is defined by a satisfaction relation, also denoted by $\models$, between a path and a path-formula. We define: $\sigma \models \Phi_1\ {}_A\mathcal{U}^{<t}\,\Phi_2$ if and only if:

$$\exists k \geq 0.\left(\sigma[k] \models \Phi_2 \;\wedge\; \forall i < k.\left(\sigma[i] \models \Phi_1 \wedge \sigma[i] \xrightarrow{A} \sigma[i{+}1]\right) \;\wedge\; t > \sum_{i=0}^{k-1}\delta(\sigma,i)\right)$$

where $\delta(\sigma,i)$ denotes the time spent in state $\sigma[i]$. Thus, $\Phi_1\ {}_A\mathcal{U}^{<t}\,\Phi_2$ is valid for a path if at some time instant before t a Φ_2-state is reached — assume this is the $(k{+}1)$-st state in the path so far — by visiting only Φ_1-states, while taking only A-transitions along the entire path. For the second form of the until-formula we have: $\sigma \models \Phi_1\ {}_A\mathcal{U}^{\leq t}_B\,\Phi_2$ if and only if:

$$\begin{aligned}\exists k > 0.\Big(&\sigma[k] \models \Phi_2 \;\wedge\; (\forall i < k{-}1.\,\sigma[i] \models \Phi_1 \wedge \sigma[i] \xrightarrow{A} \sigma[i{+}1])\\ &\wedge \sigma[k{-}1] \models \Phi_1 \;\wedge\; \sigma[k{-}1] \xrightarrow{B} \sigma[k] \;\wedge\; t > \textstyle\sum_{i=0}^{k-1}\delta(\sigma,i)\Big)\end{aligned}$$

Note the subtle difference with the first form of the until-formula: For $\Phi_1\ {}_A\mathcal{U}^{\leq t}_B\,\Phi_2$ to be valid, there has to be at least one transition, namely the final B-transition leading to a Φ_2-state.

Next we define the steady-state probability $\pi(s, S')$ of being in a state of set S', provided that the system started in s, by means of a probability measure[4] Pr on the set of paths $Path(s)$ emanating from s:

$$\pi(s,S') = \lim_{t\to\infty} \Pr\{\,\sigma \in Path(s) \mid \sigma@t \in S'\,\}$$

[3]The initial state of the SLTS is of no importance here and therefore omitted.

[4]The probability measure Pr is defined by means of a Borel space construction on paths, see [21]. It can be shown that the sets of paths appearing in the next two equations are measurable.

Similarly, the probability $Prob(s, \varphi)$ which denotes the probability measure of all φ-paths emanating from state s, is defined by

$$Prob(s, \varphi) \;=\; \Pr\{\, \sigma \in Path(s) \mid \sigma \models \varphi \,\}.$$

With these ingredients, we can now formally define the semantics of aCSL.

Definition 9.3.2 Semantics of aCSL
Let Φ, Φ_1, Φ_2 be aCSL formulas. Let $\pi(s, S')$ denote the steady-state probability of being in a state of set S' provided that the system started in s, and let $Prob(s, \varphi)$ denote the probability measure of all φ-paths emanating from state s. The satisfaction relation $\models$ between a state s and a given aCSL state-formula is defined as follows:

$$\begin{array}{ll} s \models tt & \text{for all } s \in S \\ s \models \neg\Phi & \text{iff } s \not\models \Phi \end{array} \qquad \begin{array}{ll} s \models \Phi_1 \wedge \Phi_2 & \text{iff } s \models \Phi_1 \text{ and } s \models \Phi_2 \\ s \models \mathcal{S}_{\bowtie p}(\Phi) & \text{iff } \pi(s, Sat(\Phi)) \bowtie p \\ s \models \mathcal{P}_{\bowtie p}(\varphi) & \text{iff } Prob(s, \varphi) \bowtie p \end{array}$$

where $Sat(\Phi) = \{\, s \in S \mid s \models \Phi \,\}$. ■

9.3.2 Model checking aCSL

The general procedure for model checking aCSL is similar as for model checking CTL or CSL: A parse tree is generated from the formula, and subformulas are checked, starting at the leaves of the parse tree and finishing at the root. Checking of the Boolean connectives is standard. Checking of steady-state properties requires the solving of one or more linear systems of equations for which our tool ETMCC [168, 171, 173, 172] employs the iterative schemes of either Jacobi or Gauss-Seidel. As a preprocessing step for steady-state calculation, the strongly connected components of the CTMC are determined with the help of a graph analysis algorithm. Model checking the probabilistic quantifier $\mathcal{P}_{\bowtie p}(\varphi)$ is the crucial difficulty. It relies on the following characterisations of $Prob(s, \varphi)$. We discuss the characterisations by structural induction over φ. For the sake of simplicity, we first treat the simple untimed until-formulas.

Untimed until. For $\varphi = \Phi_1 \, {}_A\mathcal{U} \, \Phi_2$ we have that $Prob(s, \varphi)$ is given by the following equation:

$$Prob(s, \varphi) = \begin{cases} 1 & \text{if } s \models \Phi_2 \\ \sum_{s' \in S} p_A(s, s') \cdot Prob(s', \varphi) & \text{if } s \models \Phi_1 \wedge \neg\Phi_2 \\ 0 & \text{else} \end{cases}$$

For $A = Act$ we obtain the equation for standard until as for DTMCs [148]. Let now $\varphi = \Phi_1 \, {}_A\mathcal{U}_B \, \Phi_2$. For $s \not\models \Phi_1$, the formula is invalid. As for $s \models \Phi_1$ the situation is more involved let us, for the sake of simplicity, assume that A and B are disjoint, i.e. $A \cap B = \emptyset$. Then the only interesting possibilities starting from s are (i) to directly move to a Φ_2-state via a B-transition, in which case the formula φ is satisfied with probability 1, or (ii) to take an A-transition leading to Φ_1-state s' which satisfies φ with probability $Prob(s', \varphi)$. Accordingly, for $A \cap B = \emptyset$ we have

$$Prob(s, \varphi) = \sum_{s' \models \Phi_2} p_B(s, s') + \sum_{s' \models \Phi_1} p_A(s, s') \cdot Prob(s', \varphi).$$

In the general case, we have to take into account that A and B may not be disjoint. One must ensure that an $(A \cap B)$-transition into a state that satisfies both Φ_1 and Φ_2 will not be counted twice. We therefore obtain that $Prob(s, \varphi)$ is the least solution of the following set of equations:

$$\begin{aligned} Prob(s, \varphi) \;=\; & \sum_{s' \models \Phi_2} p_B(s, s') + \sum_{s' \models \Phi_1} p_A(s, s') \cdot Prob(s', \varphi) \\ & - \sum_{s' \models \Phi_1 \wedge \Phi_2} p_{A \cap B}(s, s') \cdot Prob(s', \varphi) \end{aligned}$$

if $s \models \Phi_1$, and 0 otherwise. This last equation may also be written in the following form

$$\begin{aligned} Prob(s, \varphi) \;=\; & \sum_{s' \models \Phi_2} p_B(s, s') + \sum_{s' \models \Phi_1 \wedge \Phi_2} p_{A \setminus B}(s, s') \cdot Prob(s', \varphi) \\ & + \sum_{s' \models \Phi_1 \wedge \neg \Phi_2} p_A(s, s') \cdot Prob(s', \varphi) \end{aligned}$$

which avoids subtraction. Note that

$$Prob(s, X_B \, \Phi) = Prob(s, tt \, {}_\emptyset\mathcal{U}_B \, \Phi) = \sum_{s' \models \Phi} p_B(s, s')$$

which coincides, for $B = Act$, with the characterisation of next for DTMCs [148]. Thus, the probability that s satisfies $X_B \, \Phi$ equals the sum of the probabilities to move to a Φ-state via a single B-transition. Note further that for $B = \emptyset$ there is no state that satisfies $\Phi_1 \, {}_A\mathcal{U}_B \, \Phi_2$ with positive probability.

Timed until. For $\varphi = \Phi_1 \, {}_A\mathcal{U}^{<t} \, \Phi_2$ we have that $Prob(s, \varphi)$ is the least solution of the following set of equations:

$$Prob(s, \varphi) = \begin{cases} 1 & \text{if } s \models \Phi_2 \\ \int_0^t e^{-E(s) \cdot x} \cdot \sum_{s' \in S} R_A(s, s') \cdot Prob(s', \Phi_1 \, {}_A\mathcal{U}^{<t-x} \, \Phi_2) \; dx & \text{if } s \models \Phi_1 \wedge \neg \Phi_2 \\ 0 & \text{else} \end{cases}$$

For state s satisfying $\Phi_1 \wedge \neg\Phi_2$, the probability of reaching a Φ_2-state within t time units from s equals the probability of reaching some direct successor s' of s within x time units, multiplied by the probability of reaching a Φ_2-state from s' within the remaining time $t-x$. Since there may be different paths from s to Φ_2-states, the sum is taken over all these possibilities. (Note that by taking $t = \infty$ we obtain, after some straight-forward calculations, the characterisation for untimed until ${}_A\mathcal{U}$ given before). For $\varphi = \Phi_1 \, {}_A\mathcal{U}_B^{<t} \, \Phi_2$ we have that $Prob(s, \varphi)$ is the least solution of the following set of equations:

$$Prob(s,\varphi) = \int_0^t e^{-E(s)\cdot x} \cdot \left(\sum_{s' \models \Phi_2} R_B(s,s') + \sum_{s' \models \Phi_1} R_A(s,s') \cdot Prob(s', \Phi_1 \, {}_A\mathcal{U}_B^{<t-x} \, \Phi_2) \right.$$
$$\left. - \sum_{s' \models \Phi_1 \wedge \Phi_2} R_{A\cap B}(s,s') \cdot Prob(s', \Phi_1 \, {}_A\mathcal{U}_B^{t-x} \, \Phi_2) \right) \, dx$$

if $s \models \Phi_1$, and $Prob(s, \varphi) = 0$ otherwise[5]. This characterisation can be justified in the same way as for its untimed counterpart, i.e., $\Phi_1 \, {}_A\mathcal{U}_B \, \Phi_2$, given the above explanation for the simpler timed until variant. Let us consider what this yields for $X_B^{<t} \, \Phi$:

$$Prob(s, X_B^{<t} \, \Phi) = Prob(s, tt \; {}_\emptyset\mathcal{U}_B^{<t} \, \Phi) = \int_0^t e^{-E(s)\cdot x} \cdot \sum_{s' \models \Phi} R_B(s,s') \; dx$$

which, after some straight-forward calculations, leads to

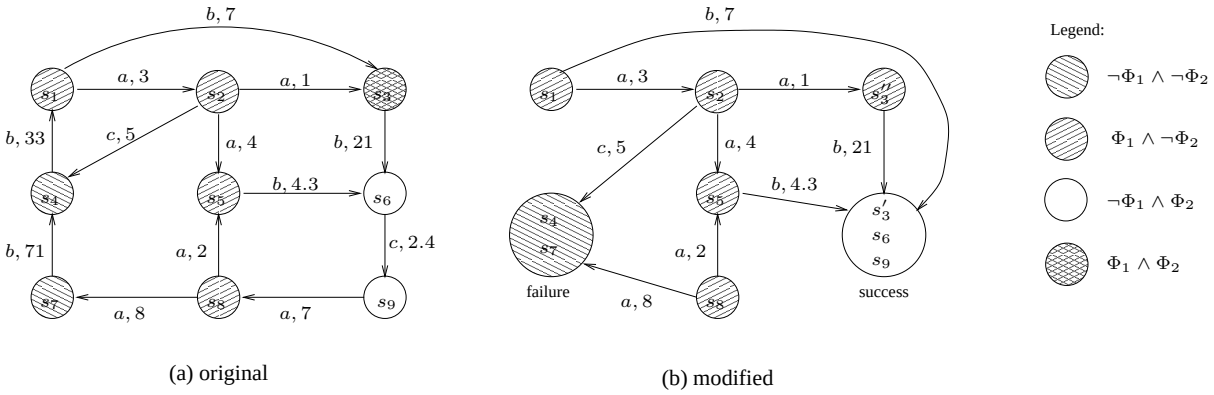
$$\sum_{s' \models \Phi} p_B(s,s') \cdot \left(1 - e^{-E(s)\cdot t}\right).$$

The first term of the product denotes the discrete probability to move via a single B-transition to a Φ-state, whereas the second term denotes the probability to leave state s within t time units.

The solution of the above integral equations can be obtained by two methods: (i) numerical integration, working on discretised representations of distribution functions, and (ii) transient analysis with the help of the uniformisation method, working on a modified CTMC. Both methods are implemented in our tool ETMCC, but as described in [173] (and as we shall see in Chap. 10), transient analysis is usually far superior both concerning the runtime and the accuracy of the results.

Example: We now discuss a small example that shows the various steps that are carried out when checking an until-formula. We discuss the checking of the

[5]The previous formula may be written without the use of subtraction in a similar way as for the untimed case above.



(a) original (b) modified

Figure 9.1: Example action-labelled CTMC

aCSL formula $\Phi = \mathcal{P}_{>0.5}\left(\Phi_1 \ {}_{\{a\}}\mathcal{U}^{<t}_{\{b\}} \ \Phi_2\right)$ for both the untimed case (i.e. $t = \infty$) and the time-bounded case (i.e. for finite $t < \infty$). For checking this formula on the model shown in Fig. 9.1, the following steps are performed:

1. Determine the set of states $\mathcal{E}$ from which there originates a path that functionally satisfies $\Phi_1 \ {}_{\{a\}}\mathcal{U}_{\{b\}} \ \Phi_2$. The set $\mathcal{E}$ is initialised as $\emptyset$. First, Φ_1-states from which there is a b-transition to a Φ_2-state are added to $\mathcal{E}$ (in the example s_1, s_3, s_5). Then those Φ_1-states are added to $\mathcal{E}$ which possess an a-transition to a state already in $\mathcal{E}$ (i.e. s_2 and s_8 are added to $\mathcal{E}$). So finally $\mathcal{E} = \{s_1, s_2, s_3, s_5, s_8\}$.

2. For the untimed case, determine the set of states $\mathcal{A} \subseteq \mathcal{E}$ whose outgoing paths *all* satisfy $\Phi_1 \ {}_{\{a\}}\mathcal{U}_{\{b\}} \ \Phi_2$. Set $\mathcal{A}$ is computed by removing states from $\mathcal{E}$. In the example, s_2 is removed first, since it can can make a c-transition to s_4. Afterwards, s_1 is removed, since it can move to s_2 which is no longer in $\mathcal{A}$. Furthermore, s_8 is removed since it can make an a-transition to s_7 which is not a Φ_1-state. So finally $\mathcal{A} = \{s_3, s_5\}$.

   Next, for all states, we determine the probability with which the state satisfies $\Phi_1 \ {}_{\{a\}}\mathcal{U}^{<\infty}_{\{b\}} \ \Phi_2$, i.e. regardless of the time that passes until reaching the Φ_2-state. States not in $\mathcal{E}$ have probability 0. States from $\mathcal{A}$ have probability 1.0. In general, the probabilities for the states from $\mathcal{E} \setminus \mathcal{A}$ (in the example, $\mathcal{E} \setminus \mathcal{A} = \{s_1, s_2, s_8\}$) are determined by solving a linear system of equations (for which the tool ETMCC again uses Jacobi or Gauss-Seidel). In the example, the probability for state s_2 is simply $1/10 + 4/10 = 0.5$, the probability for state s_8 is $2/10 = 0.2$, and the probability for state s_1 is $7/10$ plus $3/10$ times the probability which was just computed for s_2, i.e. $7/10 + 3/10 * 0.5 = 0.85$.

3. For the time-bounded case, one needs to determine the set of states that satisfy the original formula $\Phi = \mathcal{P}_{>0.5}\left(\Phi_1 \ {}_{\{a\}}\mathcal{U}^{<t}_{\{b\}} \ \Phi_2\right)$ for finite $t < \infty$. Firstly, states not in $\mathcal{E}$ cannot satisfy Φ, so no numerical calculations will have to be performed for states s_4, s_6, s_7, s_9. The CTMC is now modified, such that states which satisfy $\neg\Phi_1 \wedge \neg\Phi_2$ (that is states s_4, s_7) or $\neg\Phi_1 \wedge \Phi_2$ (that is states s_6, s_9) are made absorbing, and states which satisfy $\Phi_1 \wedge \Phi_2$ are duplicated, such that if reached by a b-transition they are made absorbing, while if reached by an a-transition they retain their original outgoing transitions (in the example, s_3 would be duplicated, which yields s_3' and s_3''). All absorbing states which satisfy Φ_2 are collected in a "success" macro state, while those absorbing states which satisfy neither Φ_1 nor Φ_2 are collected in a "failure" macro state. Then transient analysis is performed on this modified model. The probability with which a state s satisfies $\Phi_1 \ {}_{\{a\}}\mathcal{U}^{<t}_{\{b\}} \ \Phi_2$ is calculated as the probability of being in the "success" macro state at time t, provided that the system started in state s at time 0. Note that by a clever cumulation of intermediate results [209] it is possible to calculate these time-dependent probabilities for all states by a single execution of transient analysis, as implemented in our tool ETMCC. Note further, that in case step 2 has been performed before step 3, those states from $\mathcal{E} \setminus \mathcal{A}$ whose probability found in step 2 was less or equal than the margin 0.5 (in the example that is states s_2 and s_8) have no chance of reaching a Φ_2-state in finite time t with high enough probability, so no further calculations need to be performed for them.

9.3.3 Invariance under Markovian bisimulation

In this section, we show that aCSL is invariant under the application of Markovian bisimulation, which (as we mentioned in Sec. 3.7) is a congruence for the stochastic process algebras TIPP [142] and PEPA [192]. In the context of process algebraic composition operators, a congruence relation can be used to compress the state space of components before composition, in order to alleviate the state space explosion problem, under the condition that the relation equates only components obeying the same properties. Hence the question arises whether a Markovian bisimulation $\mathcal{B}$ can be applied to compress models (or model components) prior to model checking aCSL-formulas. In general, this requires that the validity of aCSL-formulas is preserved when moving from an SLTS $\mathcal{T}$ to its quotient SLTS $\mathcal{T}/\mathcal{B}$. In the following we write $\models_{\mathcal{T}}$ for the aCSL satisfaction relation $\models$ on $\mathcal{T}$, and use $[s]_{\mathcal{B}}$ to denote the equivalence class of state s.

Theorem 9.3.1 Invariance of aCSL under Markovian bisimulation
Let $\mathcal{B}$ be a Markovian bisimulation on SLTS $\mathcal{T}$ and s a state in $\mathcal{T}$. Then:

(a) For all aCSL state-formulas Φ: $s \models_{\mathcal{T}} \Phi$ iff $[s]_{\mathcal{B}} \models_{\mathcal{T}/\mathcal{B}} \Phi$

(b) For all aCSL path-formulas φ: $Prob^{\mathcal{T}}(s, \varphi) = Prob^{\mathcal{T}/\mathcal{B}}([s]_{\mathcal{B}}, \varphi)$.

In particular, Markovian bisimilar states satisfy the same aCSL formulas. ■

The proof of Thm. 9.3.1, which proceeds by structural induction on Φ, is sketched in the appendix of [170] and a detailed proof can be found in [169]. This result allows to verify aCSL-formulas on the potentially much smaller SLTS $\mathcal{T}/\mathcal{B}$ rather than on $\mathcal{T}$. Remember from Sec. 3.7.4 that the quotient with respect to Markovian bisimilarity can be computed by a modified version of the partition refinement algorithm for ordinary bisimulation without an increase of complexity. In addition, due to the congruence property of Markovian bisimilarity, a specification can be reduced in a compositional way.

9.3.4 Translating aCSL to CSL

The design of aCSL closely follows the work of De Nicola and Vaandrager on aCTL [262]. For what concerns model checking, they propose a translation $\mathcal{K}$ from aCTL into CTL, and a transformation (also denoted $\mathcal{K}$) from action-labelled to state-labelled transition systems in such a way that for an arbitrary aCTL formula Φ and arbitrary action-labelled transition system $\mathcal{T}$ (with the obvious notation):

$$s \models_{\mathcal{T},\mathrm{aCTL}} \Phi \quad \text{iff} \quad \mathcal{K}(s) \models_{\mathcal{K}(\mathcal{T}),\mathrm{CTL}} \mathcal{K}(\Phi)$$

In this way, aCTL model checking can be reduced to CTL model checking, by checking a translated formula $\mathcal{K}(\Phi)$ on a transformed model $\mathcal{K}(\mathcal{T})$. The bypass via $\mathcal{K}$ blows up the model and the formula, but only by a factor of 2, whence it follows that model checking aCTL has the same worst case complexity as CTL. The key idea of this transformation is to break each transition of $\mathcal{T}$ in two, connected by a new auxiliary state. The new state is labelled with the action label of the original transition, playing the role of an atomic state proposition. (The original source and target states are labelled with a distinguished symbol $\perp$). Formula Φ is manipulated by $\mathcal{K}$ in such a way that starting from some state $\mathcal{K}(s)$ essentially all the labellings of original states ($\perp$) do not matter, while the ones of auxiliary states do. Unfortunately, this approach does not carry over to the Markov chain setting, because splitting a Markovian transition in two

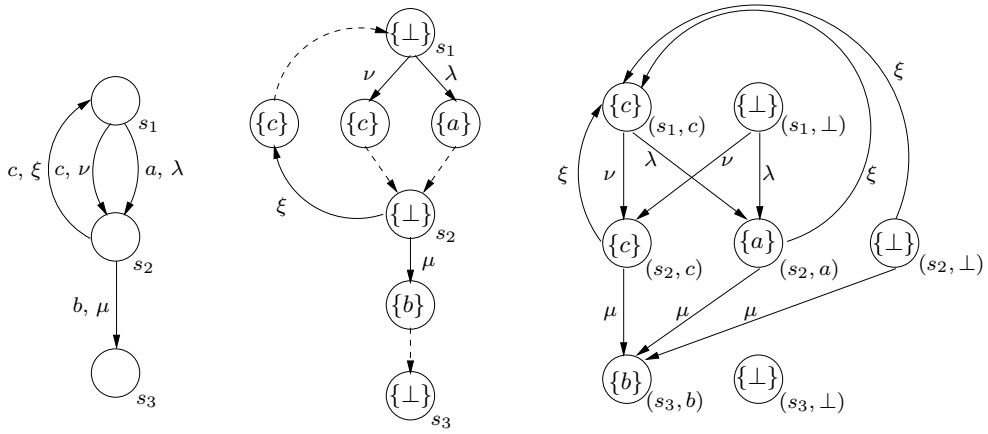


Figure 9.2: Transformation example from an SLTS (left) to an IMC (middle) and a state-labelled CTMC (right)

implies splitting an exponential distribution in two. However, no sequence of two exponential distribution agrees with an exponential distribution. Since aCSL is powerful enough to detect differences in transient probabilities, this approach is infeasible.

Even though a translation in the style of De Nicola and Vaandrager does not allow one to reduce aCSL to CSL, this does not imply that such a reduction is generally infeasible. For the sake of completeness, we remark that it is indeed possible to reduce model checking aCSL to model checking (slight variants of) CSL. We briefly sketch two possibilities:

- Apply the transformation of [262, 263] and map SLTS to Interactive Markov chains (IMC) [162]. This transformation is exemplified in Figure 9.2 (from left to middle), where state labellings appear as sets, and dashed transitions are supposed to be immediate. In general, IMC allow for non-determinism, but this phenomenon is not introduced by the translation. Therefore, the model checking algorithm of [21] can be lifted to this subset of IMC.
- Transform SLTS to state-labelled CTMCs, using a transformation inspired by Emerson and Lei [117]. The main idea is to split each state into a number of duplicates, given by the number of different incoming actions it possesses, and label each duplicate with a different action, and distribute the incoming transitions accordingly. (In order to track the first transition delay correctly, one additional $\bot$-labelled duplicate per state is needed.) To give an intuitive idea, this transformation is depicted in Figure 9.2 (from left to right). A mapping $\mathcal{K}$ from aCSL to a minor variant of CSL exists that ensures $s \models_{\mathcal{T}} \phi$ to hold if and only if $(s, \bot) \models_{\mathcal{K}(\mathcal{T})} \mathcal{K}(\phi)$ holds. (The

satisfaction relation $\models_{\mathcal{K}(\mathcal{T})}$ on CSL requires a subtle – but straight-forward to implement – modification.) Details can be found in [169]. In the worst case, the state space is blown up by a factor given by the maximal number of distinct actions entering a state.

Notice that both translations sketched above require a small modification of the model checking algorithm for CSL [19, 21]. Furthermore, both approaches induce a blow up of the model by a linear factor. To avoid these drawbacks, we decided to develop and implement a direct model checking algorithm, as sketched in Section 9.3.2. This decision was also motivated by the fact that despite the aforementioned translations from aCTL to CTL, dedicated model checkers for aCTL are more popular by now [119, 242].

9.4 Developing more general logics

In this chapter, we have described an action-oriented analysis approach for stochastic process algebras which is based on model checking. This approach closes a disturbing gap in the process algebraic methodology for performance and dependability modelling, because performance engineers are no longer forced to switch from an action-oriented to a state-oriented view when it comes to model analysis.

As we shall see in the application examples in Chap. 10, our logic aCSL allows one to specify a wide range of performability properties. However, there still remain many interesting properties which cannot be expressed by aCSL. For example, with aCSL it is not possible to specify the following requirement: "Starting from a Φ_1-state, the probability that a Φ_2-state is reached within 12 time units, by performing the action sequence $a; b; c$ arbitrarily often, should be at least 0.99?". Note that the only types of action sequences over which aCSL is able to quantify time and probability may be written as regular expressions over the set of actions Act as a^* and $a^*; b$, corresponding directly to the two forms of the until operator. It is, of course, highly desirable to describe more general sequences of actions, such as $(a; b; c)^*$ or $a; b^*; c$.

To this aim, we are currently working on a stochastic extension of propositional dynamic logic (PDL) [125], which we call SPDL [251]. Apart from the capability of specifying complex action sequences, PDL offers a testing operator that can be employed to check whether certain properties are satisfied at intermediate points of the action sequence. In addition to action-oriented properties, PDL also offers state-oriented properties on the basis of atomic propositions, thereby opening the way for combined state-based and action-based model checking. We mention

a related approach in this direction that was described by Sanders et al. [265], which however does not employ temporal logics or model checking techniques. They propose a technique to specify so-called path-based reward variables with the help of a path automaton, which enables the specification of measures over state sequences and avoids the manual tailoring of the model.

9.5 Symbolic model checking

Model checking and the use of symbolic, i.e. BDD-based, techniques for model representation are closely related topics. There has been a lot of successful work on symbolic model checking [243], although mostly for the verification of purely functional, i.e. non-stochastic, systems. In fact, most of the experience with BDDs, including the existing tools and software libraries, has its roots in such areas as the verification of digital circuits [49, 64], the verification of communication protocols [31], or the model checking of concurrent systems in general [84, 210, 243]. As early as 1992, using symbolic techniques, model checking was applied to industrial designs with more than 10^{20} states [65].

BDD-based representation of the model to be checked, and BDD-based representations of satisfaction predicates (i.e. encodings of sets of states that satisfy a given property), are attractive for two reasons: (i) BDDs offer a compact representation, thereby enabling the storage of very large models, and (ii) the various algorithms needed during verification (such as Boolean operations on satisfaction predicates, transition relations and closure operations thereon, etc.) are excellent candidates for efficient BDD-based implementation.

It is therefore obvious that our approach to the compact symbolic representation of Markov chains and their BDD-based analysis, as described in the previous chapters, is an ideal basis for the verification of performability properties by means of model checking. As a general observation, the operations that we used during compositional model construction (namely reachability analysis, hiding of action labels, selection, restriction, abstraction, etc.) are also needed as the basic operations for implementing the model checking algorithms. More specifically, the operations needed for the model checking of stochastic systems as discussed in Sec. 9.3, involving numerical calculations, in particular linear algebra operations, can be realised conveniently with the help of MTBDDs.

However, in this chapter we did not elaborate on the specific BDD-related issues when implementing the model checking algorithms, because the main intention of this chapter is to illustrate the use of model checking as an extension of classical

performability evaluation, and because we do not yet have practical experience with symbolic model checking of stochastic systems. In fact, our model checker ETMCC [168, 171, 173, 172] does not currently employ BDDs or MTBDDs, but is based on sparse storage schemes, although we are planning to develop a symbolic engine for ETMCC in the future. To our knowledge, the tools PROB-VERUS [155] and PRISM [102, 231, 228], which are based on MTBDDs, are the first symbolic model checkers for stochastic systems (cf. Sec. 11.3).

Part IV

Applications

Chapter 10

Analysing complex communication systems

In this chapter, we present three non-trivial case studies: A hospital communication system, a cyclic server polling system and a multiprocessor mainframe system with failures. For each of these, we provide a system description, describe the MTBDD-based state space construction, present some performance measures and use model checking in order to verify some performability properties.

All experiments which involve MTBDDs were carried out with the help of our tool Im-Cat [128, 129]. Im-Cat supports compositional model generation, hiding of actions, elimination of compositionally vanishing states, reachability analysis and calculation of steady-state probabilities. Since Im-Cat does not currently support the specification and computation of performance measures, we used TIPPtool [165] to carry out the performance evaluation experiments in this chapter. The checking of performability properties was done with the help of our tool ETMCC [168, 171, 173, 172].

All experimental results reported in this thesis were obtained on the same reference machine, a SUN 5/10 workstation, equipped with 1GB of main memory and running at 300 MHz.

10.1 A hospital communication system

In this section, we describe the specification and analysis of the hospital communication system (HCS) operated by the University of Erlangen-Nürnberg. The "Universitätsklinikum Erlangen" is a large hospital, spatially distributed over the town of Erlangen, with approximately 1.600 beds, 60.000 in-patients and 140.000 out-patients per year. The study reported here is part of an ongoing performance measurement and modelling project which is being conducted jointly by the computing centre of the hospital and the department of computer science of our University [13, 300, 301].

10.1.1 Global structure of the HCS

The hospital communication system provides a communication infrastructure which is used by medical and administrative subsystems for exchanging information such as patient data, observation results, medical images and accounting data. The system consists of a large number of interacting subsystems, among them the principal business application, the hospital's central laboratory, an observations processing system, the operations documentation system and a great variety of other subsystems associated with different departments and institutions.

Due to historical reasons, these decentralised information processing systems are mostly incompatible. In the past, communication between subsystems was based on proprietary one-to-one relations. Integration efforts have led to the use of standardised message formats (e.g. the Health Level 7 message standard developed for the healthcare sector [194, 195]) and the deployment of a central communication server. In Erlangen, the communication server e*Gate (formerly DataGate) from SeeBeyond Inc. (formerly STC Inc.) is used, whose tasks are the reception, checking, processing, routing and forwarding of (standardised) messages between medical subsystems. Among the subsystems in the Erlangen HCS, the patient management system, a SAP R/3 IS-H product, is the central business application. Beside the patient management system, there is a second large data base, called the communication data base, which mirrors parts of the patient management system and contains additional medical information. The communication database serves as a fast data buffer which, from the point of view of the subsystems, provides data access about 10 times faster than the patient management system itself, and as a side effect also significantly reduces the load of the latter.

Here we only present a rudimentary model of the Erlangen hospital communication system which during our project has been extended and refined in various

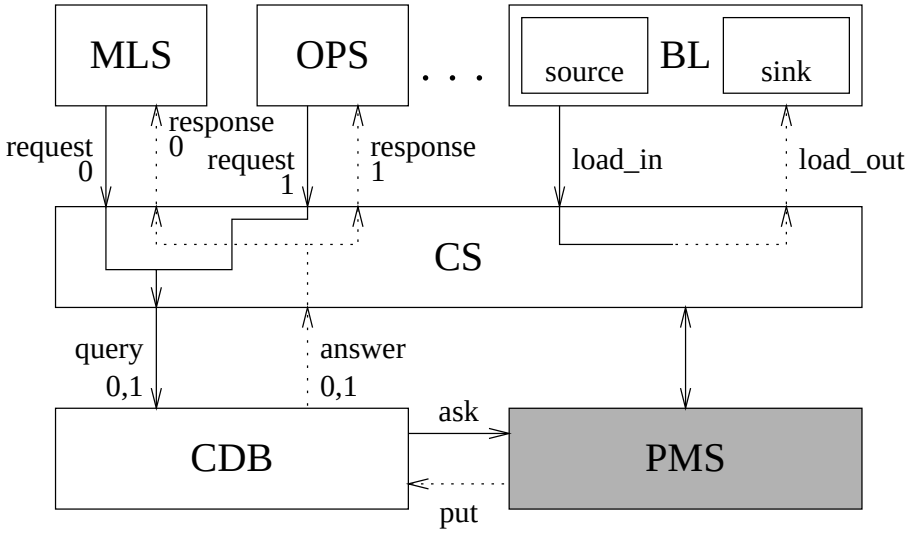


Figure 10.1: The hospital communication system model

directions. Fig. 10.1 shows the basic structure of the model. It consists of the communication server (CS), the communication data base (CDB) and two medical subsystems, the main laboratory system (MLS) and an observations processing system (OPS). Since almost all of the subsystems' demands for data can be satisfied by the CDB, we do not model the patient management system (PMS) in the model presented here (therefore the PMS is drawn grey in the figure). There is an "artificial" subsystem, representing an adjustable background load (BL), caused by those subsystems which are not explicitly modelled (BL actually consists of two subprocesses, a source and a sink which communicate with CS via actions `load_in` and `load_out`).

The top-level specification, using the stochastic process algebra language introduced in Sec. 3.7, is as follows:

```
hide  request,response  in
  (MLS ||| OPS)
  |[request,response]|
  (
  hide  query,answer  in
    (
    hide  load_in,load_out  in
      BL |[load_in,load_out]| CS
    )
  |[query,answer]| CDB
  )
```

We mention that the synchronisation discipline for timed actions (where the product of rates is implemented by the tools TIPPtool and Im-Cat) is irrelevant for this case study, since all synchronisation is carried out over immediate actions.

10.1.2 Specification of components

We now describe a typical communication sequence in the system: The MLS, after some internal processing, needs a patient data record from the CDB. It sends a request message to the CS (action `request`) which is forwarded to the CDB (action `query`). After completing the data base lookup, the CDB generates an answer message which is sent back to the CS (action `answer`) and from there on to the MLS (action `response`). Queries initiated by the OPS subsystem do not request patient data records, they request observation results instead. Apart from that, they follow the same basic pattern as queries initiated by MLS. However, since the answer to a request for observations may consist of a number of different observations, an OPS query does not result in a single answer message, but in a random number of answer messages. Measurements on the real system have shown that this number follows a geometric distribution.

Subsystems MLS and OPS communicate with CS via actions `request` and `response`. In order for these communications to be distinguishable, we use value passing and value matching, as in LOTOS [37, 201], which is supported by the input language of TIPPtool [165]. All actions associated with communications initiated by subsystem MLS carry the value 0, whereas those originating from OPS carry the value 1. The following part of the specification illustrates how value passing and value matching are employed between MLS and CS. Note that MLS is capable of receiving a `response` at any time. This is used to ensure that CS may engage in `response` even though MLS has just decided to induce the next `request!0` (equivalently a sink could be used instead, that runs independently in parallel and just consumes `response!0` actions). Concerning the rate of timed actions such as `time_mls` or `time_cs`, the basic time unit is 1 millisecond.

```
process MLS := (time_mls, lambda); MLS2 [] response!0; MLS
endproc
process MLS2 := request!0; MLS [] response!0; MLS2
endproc
```

The following portion of code specifies the behaviour of subsystem CS. In subprocess `CStodo` a `query` is submitted to the CDB. Depending on the value `x` received

through action `request`, this corresponds to a query for a patient data record or for observation results. In subprocess `CStransmit` a `response` is sent back to either MLS or OPS, again depending on the value `x`.

```
process CS := request?x:int; (time_cs, 0.02); CStodo(x)      []
              answer?x:int; (time_cs, 0.02); CStransmit(x)   []
              load_in; (time_cs, 0.02); load_out; CS
endproc
process CStodo(x) := query!x; CS
endproc
process CStransmit(x) := response!x; CS
endproc
```

Similar value passing mechanisms are employed for the communication between the CS and the CDB. In order to perform the correct type of data base lookup, the CDB has to recall the initiator of each query. To this end, queries are stored in front of the CDB in a queue with multiple job classes. i.e. for each queue position the type of the query is stored. Again, several equivalent ways are possible to represent this data type inside a TIPP specification. The following fragment of code illustrates the concept of a queue with three waiting positions and multiple job classes.

```
process CDB(f,p1,p2,p3) :=
  [f=0] -> (query?x:int; CDB(1,x,0,0))                []
  [f=1] -> (query?x:int; CDB(2,p1,x,0) []
            Lookup(1,p1,0,0))                         []
  [f=2] -> (query?x:int; CDB(3,p1,p2,x) []
            Lookup(2,p1,p2,0))                        []
  [f=3] -> (query?x:int; full; CDB(3,p1,p2,p3) []
            Lookup(3,p1,p2,p3))
endproc
```

Process CDB carries four parameters: Parameter `f` denotes the current queue population. The remaining three parameters are used to store the class of the job in the first, second and third queue position. The notation "`[...]->`" specifies a conditional behaviour that is only possible if the condition inside the brackets is true. Action `query` sets the next free position with the value passed from the CS and increases parameter `f`. (If a `query` action meets a full queue, i.e. in the case where `f=3`, an exception is raised by action `full`). The converse operation, sending (one or several) answer(s) back to the CS and thereafter removing a job

from the queue, is not shown in this fragment. It is part of the `Lookup` process and its subprocesses which can be entered under the condition that the CDB queue is not empty. If the `Lookup` process is processing a `query` originating from the OPS, it generates a geometrically distributed number of answers. This geometric distribution is modelled within subprocess `Count` (see below) by a loop with a non-zero probability of reentry after an answer has been generated. The probability of reentry depends on the ratio of the rates of the two `p`-actions and is given by $\frac{80}{80+20} = \frac{4}{5}$, from which it follows that the mean number of answers sent is five. Note that the rates of the `p`-actions are chosen in such a way that they are more than two orders of magnitude larger than those of the time-consuming actions in order to approximate a probabilistic choice.

```
process Count(f,p1,p2,p3) :=
  (p,80); Send(f,p1,p2,p3) []
  (p,20); CDB(f-1,p2,p3,0)
endproc

process Send(f,p1,p2,p3) :=
  answer!2; Count(f,p1,p2,p3)                          []
  [f=1] -> (query?x:int; Send(2,p1,x,0)                []
  [f=2] -> (query?x:int; Send(3,p1,p2,x)               []
  [f=3] -> (query?x:int; full ; Send(3,p1,p2,p3)
endproc
```

10.1.3 State space construction

We used our tool Im-Cat to construct the MTBDD representation of the HCS model in a compositional fashion. We first generated the elementary transition systems for the subsystems CS, BL, CDB and (MLS ||| OPS) with TIPPtool and encoded them (in that order) as MTBDDs. Afterwards, we employed MTBDD-based parallel composition in order to generate the overall system. There are several alternative ways for constructing (and analysing) a compositional model such as our HCS model, as illustrated in Fig. 10.2: One may construct the overall model, repeatedly using MTBDD-based parallel composition, and afterwards hide all actions in one step, as indicated by the left branch from the start. Alternatively, actions may be hidden as soon as they are not needed any more for further synchronisation, i.e. hiding may be performed directly after every parallel composition step, as indicated by the right branch from the start. The numbers inside the nodes of Fig. 10.2 denote the size of the MTBDDs when building the HCS model. Since Im-Cat uses two separate MTBDDs, one for

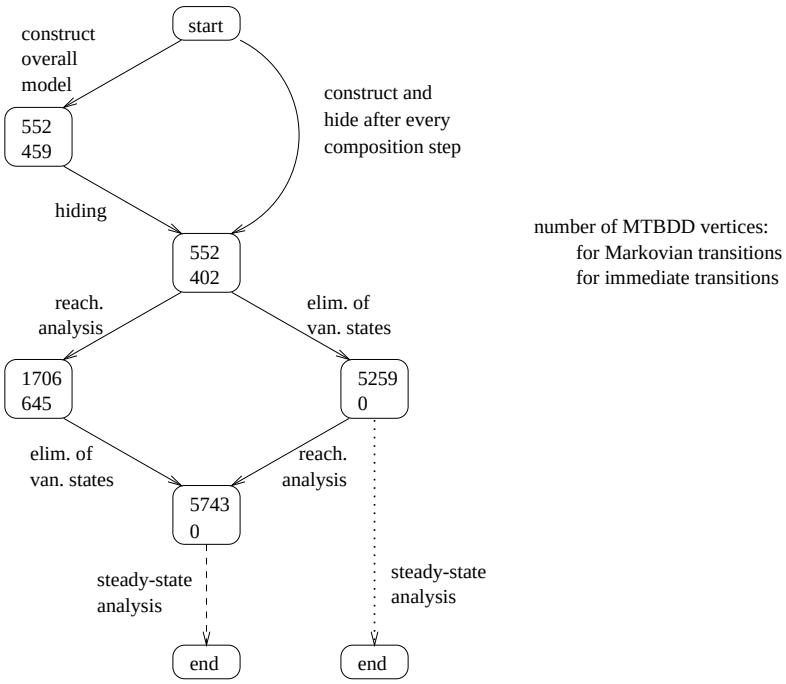


Figure 10.2: Alternative ways for constructing and analysing the compositional HCS model (cf. text)

representing Markovian transitions and one for representing immediate transitions, two figures are given in each node. The case where MTBDD-based hiding of immediate actions was employed immediately after every construction step is shown in full detail in Fig. 10.3, which gives the number of MTBDD vertices at every stage of the construction (again, two figures are given at every stage). The resulting MTBDDs are the same, no matter whether hiding is delayed or not, but by following the second alternative, the intermediate MTBDDs are kept slightly smaller (in the HCS case by up to 13%). Note that hiding of actions that are no longer needed for synchronisation is important; in particular, the hiding of immediate actions is a prerequisite for the elimination of vanishing states and construction of a CTMC.

There are different ways of how to proceed from this point:

1. Following the left branch from the middle of Fig. 10.2, one may first perform MTBDD-based reachability analysis. The number of reachable states for the HCS model turned out to be 4951, and the number of transitions is 11285. However, reducing the transition system to its reachable part increases the MTBDD sizes to 1709 vertices for Markovian transitions and 645 vertices for immediate transitions. The next step is the elimination

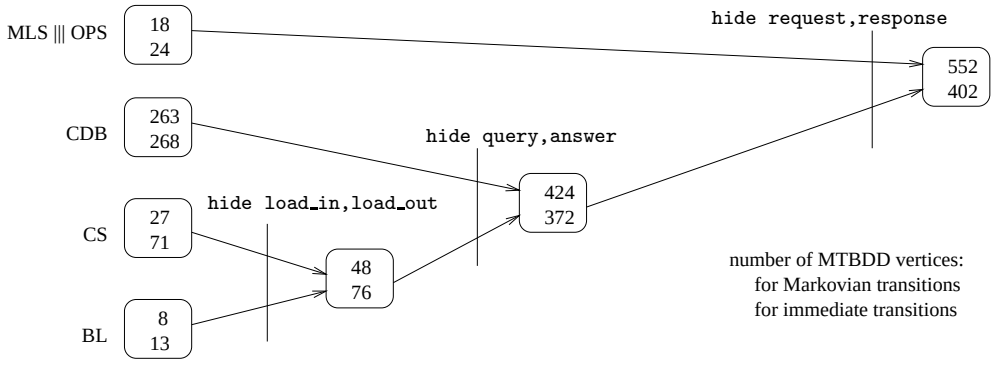


Figure 10.3: Compositional MTBDD construction for the hospital communication system

of vanishing states. Afterwards there are 2644 reachable states and 10178 transitions left, the MTBDD for the Markovian transitions has 5743 vertices, and the MTBDD for the immediate transitions is empty.

2. Alternatively, following the right branch from the middle of Fig. 10.2, one may delay reachability analysis and first eliminate vanishing states (which yields an intermediate MTBDD with 5259 vertices), but the final result is the same as in the first alternative.

In summary, one can say that compositional model construction yields reasonably compact MTBDDs (cf. Fig. 10.3), but that compactness greatly suffers from reachability analysis and elimination of vanishing states. It should also be emphasised that compositional model construction is essential when working with MTBDDs: Taking the overall transition system of the HCS model as generated by TIPPtool (comprising 4951 states and 11285 transitions), and encoding it symbolically, yields an MTBDD with 21240 vertices for representing Markovian transitions, and an MTBDD with 6932 vertices for representing immediate transitions.

Once the full model has been generated and the vanishing states have been eliminated, MTBDD-based steady-state analysis can be performed, in order to determine the steady-state probabilities, from which various performance measures may be calculated in the usual way. Starting from the MTBDD which encodes the reachable transitions of the HCS model, we employed the power method (as indicated by the dashed arrow in Fig. 10.2), whose iteration matrix is represented by an MTBDD with 6132 vertices that was constructed in 6 seconds. The solution was found after iteration step 840, and the mean CPU time consumption per iteration step was 2.6 seconds, which is, of course, quite slow for solving a system

with a mere 2644 states. As a second alternative, we also performed steady-state analysis on the potential state space, i.e. without ever performing reachability analysis (as indicated by the dotted arrow in Fig. 10.2). In this case, the iteration matrix for the power method was represented by an MTBDD with 5905 vertices, whose construction took 7 seconds. The solution was found after iteration step 460, and the mean CPU time per iteration step was 0.86 seconds. So, working with the potential state space yielded better performance for the HCS model (although still not comparable to state-of-the-art sparse linear solvers[1]), and this reflects our general experience, i.e. that reachability analysis destroys regularity, thereby increasing the size of the MTBDD representation and slowing down the computation.

It is possible to construct an aggregated state space for the HCS model in a compositional fashion, by applying stepwise aggregation with respect to bisimulation equivalence. However, since our MTBDD-tool IM-CAT does not support bisimulation algorithms at this stage, the following results were obtained with the help of TIPPTOOL: As shown in Fig. 10.4, the state space of CDB could be aggregated from 80 to 73 states, and the parallel composition of CS and BL could be reduced from 22 to 20 states, after abstraction of `load_in` and `load_out`. Combining these intermediate state spaces and hiding `query` as well as `answer` we obtained 1132 states which, again, were aggregated to 764 states. Finally, we obtained 3056 states for the overall system specification which could be aggregated to an equivalent specification with 2294 states, instead of the original 4951 states[2]. All aggregations were based on the notion of weak Markovian bisimulation.

10.1.4 Performance evaluation

We calculated a variety of performance measures for this model. There are no queues in front of process CS, i.e. subsystems wishing to communicate via the CS may have to wait until the CS is ready for synchronisation. This waiting time can become quite significant if the CS is very heavily loaded. For instance, experiments revealed that under heavy background load the subsystem MLS spends up to 11% of total time waiting for the CS (cf. Fig. 10.5, left). In this as in other experiments, the offered background load was increased exponentially, starting with an initial value $load_0 = 1$ request/sec which was doubled in every step.

[1] Using TIPPTOOL, building the generator matrix and solving the linear system took about 10 seconds (using the Gauss-Seidel iteration scheme).

[2] This very last aggregation step took more than 15 hours, indicating that the implementation of TIPPTOOL's bisimulation algorithms needs to be improved. Recently, the interface between TIPPTOOL and ALDEBARAN [122] has been successfully used to circumvent this bottleneck and to compositionally analyse specifications with more than 10^7 states [174].

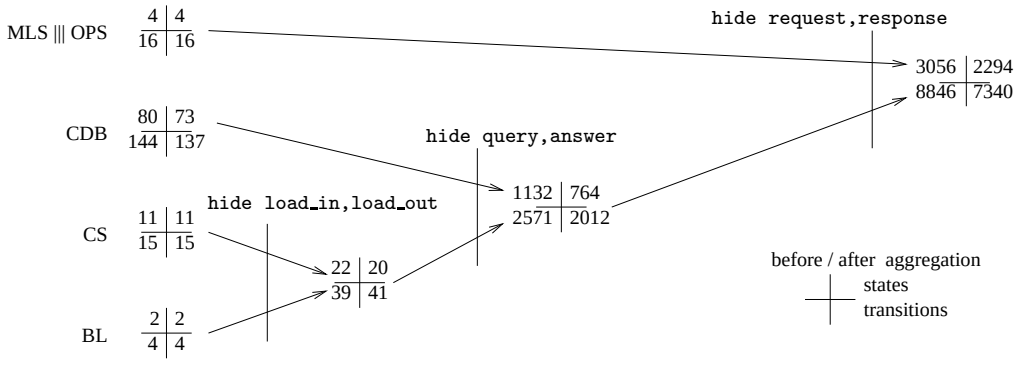


Figure 10.4: Compositional aggregation of the state space of the hospital communication system

This raises the question of the utilisation of CS which of course depends on the offered load. The traffic generated by subsystems MLS and OPS is constant, namely 1 request/sec for MLS and 0.25 requests/sec for OPS. The offered background load, as mentioned, is increased dramatically from $load_0 = 1$ request/sec to 1024 requests/sec. Fig. 10.5 (right) shows the proportion of time the CS is idle, depending on the offered background load. It should be noted that due to the average message processing time of 50 ms/message, a maximum of 20 messages/sec can be carried by the CS, no matter how much background load is offered.

We now briefly discuss the size of the queue in front of the CDB and its implications. In the real system, an almost unlimited number of queries can be queued in front of the CDB, the only limitation being the size of physical memory of the machine. In order to avoid state space explosion, we can only model very small queue sizes. The diagram in Fig. 10.6 (left) shows that for a queue size of 3 waiting positions the probability of the queue being full is between 5.9% and 8.3%, depending on the offered background load.

The rate at which (MLS and OPS) queries get lost is 0.077 to 0.097 queries/sec, see Fig. 10.6 (right). This corresponds to a query loss probability of 6.3% (low background load) to 8.6% (high background load). In real life such high loss probabilities would of course not be acceptable. On the other hand, we observe that even for modelling a queue size of 3 one already obtains quite reasonable estimates for the performance measures.

The parameters used in the model were derived from measurements on the real system. We found that exponential assumptions were justified for the request inter-arrival times and even for the message processing time in the CS. The data

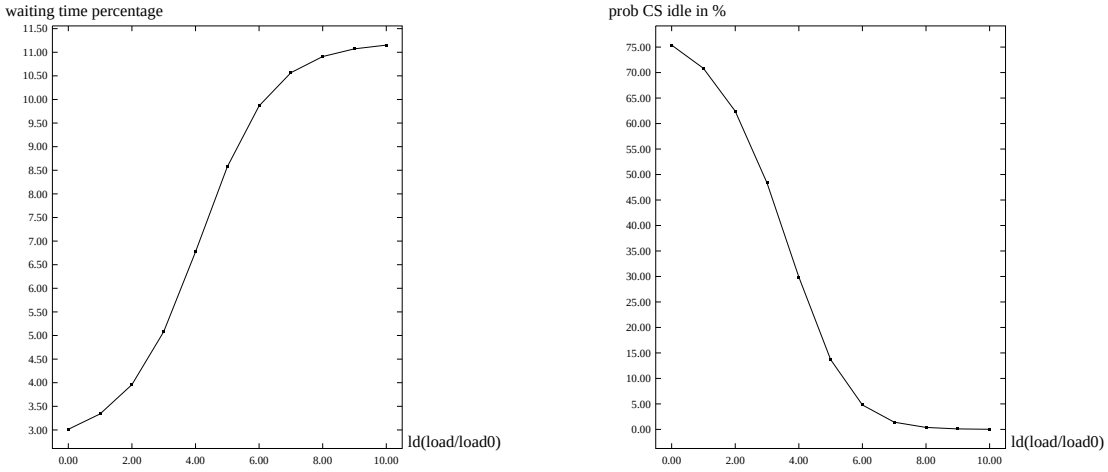


Figure 10.5: Left: waiting time percentage in subsystem MLS dependent on background load. Right: probability of CS being idle. (Note the logarithmic scale.)

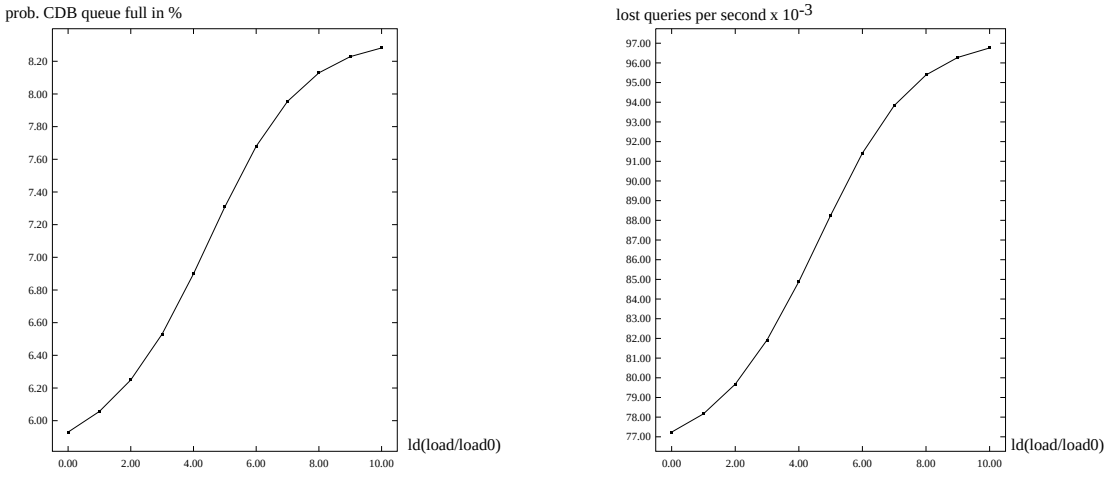


Figure 10.6: Left: probability of the CDB queue being full. Right: rate of lost queries.

base lookup times in the CDB were modelled as Erlang-2 distributions (as a sequence of two exponential phases), with different mean, depending on whether a query originated from the MLS or OPS subsystem.

As mentioned above, the model described in this section was our first rudimentary model, which we extended in various directions. For instance, we explicitly modelled the PMS and the fact that queries which cannot be satisfied by the CDB have to be forwarded to the PMS (see Fig. 10.1, where it is indicated that synchronisation between CDB and PMS is performed via actions `ask` and `put`). We conducted experiments with varying CDB "hit rates", i.e. varying probability of forwarding a query form the CDB to the PMS. We also studied the question whether it is beneficial to enable parallel queries to the CDB by employing a separate "query" process for each subsystem generating queries. In some of these investigations we dealt with state space sizes of several hundreds of thousands of states.

10.1.5 Verification of performability properties

In this section, some performability properties, which are of interest for the HCS model, are specified and checked with the help of the model checker ETMCC.

The HCS model was generated by TIPPtool from a SPA specification which contains both Markovian and immediate actions. Since the model checker ETMCC cannot deal with immediate transitions yet, we had to modify the specification, replacing immediate actions by "fast" Markovian actions, i.e. Markovian actions with large rate. As a consequence, the state space became larger; it now has 5548 states and 18584 transitions compared to the previous 4951 states and 11285 transitions[3]. Value passing, which had been employed in the original specification, and which TIPPtool allows only for immediate transitions, could therefore not be used any more. Instead, the values were coded explicitly into the action names, e.g. we used `request_1` instead of `request!1`. At the time when we conducted this study, ETMCC only supported CSL model checking (as opposed to aCSL)[4]. Therefore the aCSL properties given below had to be translated to equivalent CSL properties. The state labelling information, specifying the atomic propositions which hold in a given state, was generated from TIPPtool's state-description file.

[3]The larger state space for the purely Markovian model is due to the fact that TIPPtool implements a maximal progress semantics, i.e. does not generate Markovian transitions emanating from states with outgoing internal immediate transitions.

[4]An aCSL extension of ETMCC has recently been developed [44].

For each of the properties to be checked, a description in plain English, its aCSL formulation and some explanation is given below. We specify some purely functional requirements, as well as some performability requirements which the system should satisfy. For a set of actions $A \subseteq Act$ we use $\overline{A}$ to denote its complement, i.e. $Act \setminus A$. We omit brackets for singleton sets. We use the following sets of actions: $Request = \{request\_0, request\_1, \ldots\}$, $Query = \{query\_0, query\_1, \ldots\}$, etc.. We checked the following properties:

Φ_1: Whenever a request occurs, no further request must be accepted by the communication server before the query (corresponding to the request) occurs.

$$[request\_x] \forall \left(\overline{\overline{Request}}\Diamond_{query\_x} tt\right)$$

This property was found to hold for all states of the HCS model.

Φ_2: With probability at least 0.95, a request will cause the corresponding query within 70 ms.

$$[request\_x] \mathcal{P}_{\geq 0.95}\left(\overline{\overline{query\_x}}\Diamond^{<70}_{query\_x} tt\right)$$

This property was found to hold for all states of the HCS model.

Φ_3: Once answer_1 (i.e. an answer to a query from the OPS) has occurred, with probability at least $0.8 \cdot 0.9 = 0.72$ another answer_1 will occur within 55 ms (note that the probability of generating another answer_1 is 0.8, according to the geometric distribution).

$$[answer\_1] \mathcal{P}_{\geq 0.72}\left(\overline{\overline{answer\_1}}\Diamond^{<55}_{answer\_1} tt\right)$$

This property was found not to hold for all states of the HCS model, since there are situations where the generation of another answer_1 is slower than required by property Φ_3.

Φ_4: In steady state, the probability of the CDB queue being empty is at least 40 percent.

$$\mathcal{S}_{\geq 0.4}\left(\Phi_{empty}\right)$$

where $\Phi_{empty} = \neg\exists\left(\overline{\overline{Query}}\Diamond_{Answer} tt\right)$, i.e. the empty queue is characterised by the fact that it is not possible to observe an answer without first observing a query. This property was found to hold for the HCS model. Since this model consists of a single strongly connected component, the initial state is irrelevant for steady-state properties. The actual steady-state property of Φ_{empty} was calculated as 49.46 percent.

Φ_5: In steady state, the probability of the CDB queue containing three queries (and thus being full) is less than 8 percent.

$$\mathcal{S}_{<0.08}\left(\Phi_{full}\right)$$

property	verification runtime (in seconds)
Φ_1	0.02
Φ_2	43.03
Φ_3	3.41
Φ_4	4.47
Φ_5	4.45
Φ_6	4.41

Table 10.1: Verification runtimes for the HCS model

where $\Phi_{full} = \exists\,(\overline{Query}\Diamond_{Answer}(\exists\,(\overline{Query}\Diamond_{Answer}(\exists\,(\overline{Query}\Diamond_{Answer})))))$, i.e. the full queue is characterised by the fact that it is possible to observer three answers without ever observing a query. This property was found to hold for the HCS model. The actual steady-state property of Φ_{full} was found to be 7.38 percent, which agrees with the results presented in Sec. 10.1.4[5].

Φ_6: In steady state, the probability of OPS not being able to submit a request because CS is busy is smaller than 1 percent.

$$\mathcal{S}_{<0.01}\,(\Phi_{OPS\_ready}\ \wedge\ \Phi_{CS\_busy})$$

where $\Phi_{OPS\_ready} = \neg\langle time\_ops\rangle tt$ and $\Phi_{CS\_busy} = \neg\langle Request\rangle tt$. This property was found not to hold for the HCS model. The steady-state property for $\Phi_{OPS\_ready}\ \wedge\ \Phi_{CS\_busy}$ was found to be 2.55 percent.

The runtimes for checking the above properties with the help of ETMCC are given in Table 10.1. Checking property Φ_2, involving the time-bounded until operator, turns out to be by far the most time consuming. Property Φ_3 also involves a time-bounded until, but can be checked relatively quickly, since the paths which functionally satisfy the property are very short[6]. Property Φ_1 which is purely functional, is checked extremely fast, while the three steady-state properties Φ_4, Φ_5 and Φ_6 all require the same moderate amount of CPU time.

[5]The background load parameter used during model checking was 32 request/sec, which corresponds to the point $ld(load/load_0) = 5$ in Fig. 10.6.

[6]This is not obvious but requires quite a detailed analysis of the state space.

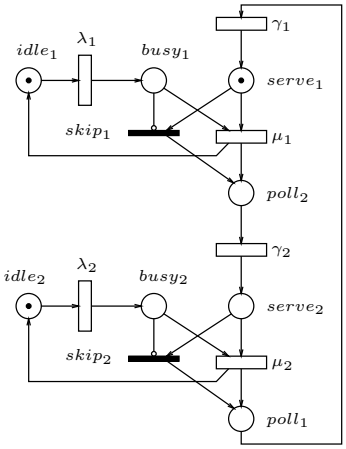


Figure 10.7: A cyclic server polling system with 2 stations

10.2 Data networks with polling

Polling mechanisms play an important role in the area of data networks and multiprocessor systems with shared communication medium. In general, polling is used to operate multi-access channels, over which several users (terminals, workstations, processors, mobile stations, ...) wish to transmit data packets. As an early example for polling we mention multidrop telephone lines, where one primary station polls a set of secondary stations in some pre-specified order. In LANs and MANs, token passing mechanisms have evolved, which can be viewed as an improved version of polling where the communication overhead for polling is avoided by directly passing a token between stations. In today's cellular networks, polling also plays an important role. The currently active mobile stations, for instance, may be polled in a cyclic fashion by the base station.

10.2.1 Description of the polling system

In this section, we consider a cyclic server polling system consisting of d stations and a server, modelled as a GSPN[7], since it is taken from [199]. For $d = 2$, i.e. a two-station polling system, the GSPN model is depicted in Fig. 10.7. For a d-station polling system, the Petri net is extended in the obvious way. Place $idle_i$ represents the condition that station i is idle, and place $busy_i$ represents the condition that station i has generated a request. The server visits the sta-

[7]We refer to [1, 2] for details on the semantics of GSPNs. In particular, immediate transitions (drawn black in the figure) lead to so-called vanishing markings in the reachability graph.

tions in a cyclic fashion. After polling station i (place $poll_i$) the server serves station i (place $serve_i$), and then proceeds to poll the next station. The times for generating a message, for polling a station and for serving a request are all distributed exponentially with parameters λ_i, γ_i and μ_i, respectively. In case the server finds station i idle, the service time is zero which is modelled by the immediate transition $skip_i$ and the inhibitor arc from place $busy_i$ to transition $skip_i$. In this study we consider polling systems with $d = 3$, 5, 7 and 10 stations (like in [199]). In addition, we consider the cases $d = 15$ and $d = 20$. The polling systems are assumed to be symmetric, i.e. all λ_i have the same numerical values, and the same is true for all $\gamma_i = 200$ and all $\mu_i = 1$. For fixed d, we choose $\lambda_i = \mu_i/d$ as the default value for λ_i. In Sec. 10.2.3, the parameter λ_i is varied in a performance evaluation experiment.

10.2.2 State space construction

The MTBDD representation of the overall polling model was constructed compositionally from $d+1$ elementary transition systems[8], one for the server and one for each station, which were encoded as individual MTBDDs Server and $\mathsf{Station}_i$ ($i = 1, \ldots, d$). The order in which the component MTBDDs are generated turned out to be of great importance concerning the resulting MTBDD size, since it determines the ordering of the MTBDD variables. In particular, we considered two orderings:

(1) The MTBDD for the server is generated first, followed by the MTBDDs for the stations. This leads to an overall MTBDD where the server is closest to the root.

(2) The d MTBDDs for the stations are generated first, followed by the server. This leads to an overall MTBDD where the server is closest to the leaf vertices.

In both cases, the MTBDD for the overall system was then computed by applying MTBDD-based parallel composition d times. In the case of ordering (1), composition was performed according to the scheme

$$(\ldots((\mathsf{Server}|[S_1]|\mathsf{Station}_1)|[S_2]|\mathsf{Station}_2)\ldots)|[S_d]|\mathsf{Station}_d$$

[8]The elementary transition systems were generated by TIPPtool. We used an equivalent model without immediate transitions, since our model checker ETMCC, to be used in Sec. 10.2.4, cannot deal with immediate transitions yet.

d	reach. states	transitions	MTBDD size compositional before reachability	MTBDD size compositional after reachability	MTBDD size monolithic
3	36	84	169	203	351
5	240	800	387	563	1,888
7	1,344	5,824	624	1,087	9,056
10	15,360	89,600	1,163	2,459	69,580
15	737,280	6.144e+6	2,191	6,317	–
20	3.14573e+07	3.40787e+08	3,704	13,135	–

Table 10.2: Statistics for the polling system, ordering (1)

d	MTBDD size compositional before reachability	MTBDD size compositional after reachability
3	163	200
5	386	623
7	658	1,734
10	1,281	10,598
15	2,641	297,942
20	4,632	9,507,435

Table 10.3: Statistics for the polling system, ordering (2)

with appropriate synchronisation sets S_i, i.e. starting with MTBDD Server, the overall system was obtained by adding the stations one by one. In the case of ordering (2), composition was performed according to

$$(\mathsf{Station}_1 ||| \ldots ||| \mathsf{Station}_d) |[S]| \mathsf{Server}$$

i.e. an MTBDD Station representing *all* stations was generated first, and subsequently composed in parallel with Server[9].

In Tab. 10.2, the sizes of the resulting MTBDDs are given for different values of d, provided that ordering (1) is employed. The first column of the table contains the number of stations d, the 2nd (3rd) column contains the number of reachable states (the number of reachable transitions), and the remaining columns give the number of vertices of the corresponding MTBDDs. The MTBDD generated compositionally according to the above scheme represents all transitions which are possible within the product of the $d+1$ components' state spaces. As can be observed from the 5th column of Tab. 10.2, determining the set of reachable states and "deleting" the transitions which originate in unreachable states considerably increases the size of the MTBDDs (we had observed the same phenomenon for the hospital communication system in Sec. 10.1). Therefore, in general, it is recommended to work with MTBDDs which represent the "potential" rather than the actual state space. The last column of Tab. 10.2 shows the number of MTBDD vertices which one would obtain if one took the monolithic transition system of

[9]The symbol ||| abbreviates $|[\emptyset]|$, i.e. denotes parallel composition without synchronisation.

the overall model, i.e. the overall model as generated by TIPPtool which does not contain unreachable states, and directly encoded it as an MTBDD. Clearly, this method cannot be recommended: Apart from the fact that the transition system of the overall model may not be available due to its size, the growth of the MTBDD sizes is prohibitive. As expected from Thm. 5.1.4, the figures in column 4 grow linearly, whereas the ones in column 6 grow exponentially. Tab. 10.2 shows that even for an extremely large state space, the MTBDD representation can be very compact, if it is constructed in a compositional fashion. The "–" entries in the last column denote that the numbers could not be determined because the monolithic transition system of the overall model was never explicitly constructed due to excessive runtime and memory requirements.

Tab. 10.3 contains the sizes of the MTBDDs which were constructed compositionally according to ordering (2). This ordering obviously yields much larger MTBDDs than ordering (1). In particular, it is interesting to observe that, while the figures in the column "before reachability" are still close to the corresponding ones in Tab. 10.2 (only up to 1.25 times larger), the figures in the column "after reachability" are dramatically worse for ordering (2) (up to 724 times larger). Unfortunately, it is by no means obvious a priori which of the two component orderings yields smaller MTBDDs.

10.2.3 Performance evaluation

In this section, we briefly present the calculation of some typical performance measures for the polling system. The probability of a particular station being idle is of course strongly dependent on the rate λ_i at which messages are generated. The higher that rate, the shorter the idle times will be. This is illustrated by the graphs shown in Fig. 10.8 (left) for two instances of our polling model, namely for $d = 7$ stations (solid line) and $d = 10$ stations (dotted line). In the case $d = 7$, λ_i is varied between 0.1 and 1.5, while in the case $d = 10$, λ_i is varied between 0.07 and 1.05. The graphs on the right of the figure depict the probability of a particular station having generated a message and being waiting for the server. The waiting probability increases with increasing message generation rate, since the generation of more messages causes more work to be done by the server, which in turn increases the time for performing one service cycle.

Note that for carrying out an experiment with varying parameters, TIPPtool only needs to explore the state space once, which took about 7 seconds for the $d = 7$ stations case and about 95 seconds for the $d = 10$ stations case. For each value of λ_i, the generation of the Gauss-Seidel iteration matrix, the calculation of the steady-state probabilities and the computation of the measures took between 6 and 15 seconds for the $d = 7$ stations case, and between 55 seconds and 27

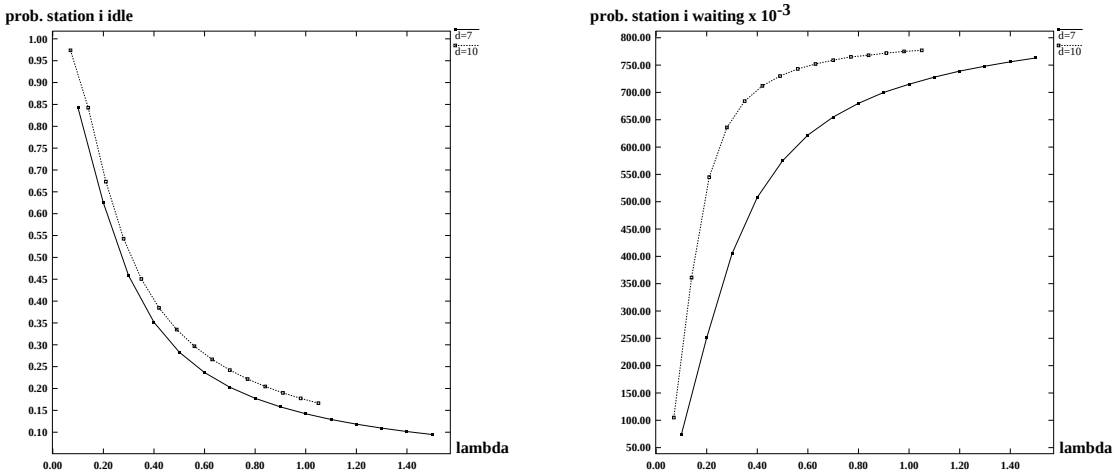


Figure 10.8: Left: probability of station i being idle. Right: probability of station i being waiting for the server.

minutes (!) for the $d = 10$ stations case. This large variation of the solution times is mostly due to the convergence behaviour of the Gauss-Seidel method, which depends heavily on the actual value of λ_i. As one specific instance, for $d = 7$ and $\lambda_i = 1/7$, 105 Gauss-Seidel iterations were required, and one iteration took 0.0013 seconds at the average.

As a comparison, for the $d = 7$ station model, $\lambda_i = 1/7$ and working on the potential state space, the MTBDD representing the power iteration matrix, as generated by our tool Im-Cat, has 806 vertices and takes 0.8 seconds to construct. One iteration of the power scheme takes 0.122 seconds, but it takes a ridiculous 8070 power iterations to converge. The MTBDD representing the Jacobi iteration matrix for the same system is larger, it has 1639 vertices and takes 18.94 second to construct, but one Jacobi iteration takes only 0.101 seconds and convergence is reached after 240 iterations. We also performed MTBDD-based steady-state analysis for the $d = 10$ station model: The MTBDD representing the Jacobi iteration matrix now has 4073 vertices but takes 4436 second to construct (!). However, one Jacobi iteration takes only 0.181 seconds.

10.2.4 Verification of performability properties

In this section, we describe the model checking of some performability properties of the polling system. In the context of GSPNs, it is natural to identify the set of places that possess a token in a given marking — i.e. a state of the CTMC — with the set of atomic propositions valid in this state. Therefore, for the polling

system discussed in this section, we decided not to employ action-based model checking, but state-based model checking, where properties are expressed in the continuous stochastic logic CSL [21], a stochastic variant of CTL. Note that, since CSL properties are based on states labelled by atomic propositions (and not on transitions labelled by action names), its until-operator $\mathcal{U}$ and (derived) eventually-operator $\Diamond$ are not indexed by sets of actions. Having said this, even though we did not formally introduce CSL, the properties given below should be understandable.

Based on the set of atomic propositions $AP = \bigcup_{i=1}^{d} \{idle_i, busy_i, serve_i, poll_i\}$, we checked the following properties on the polling system:

Φ_1: The server never polls two stations at the same time.

$$\neg(poll_i \wedge poll_j)$$

where $i \neq j$.

Φ_2: With probability $\bowtie p$, station i will be served before station j, where $\bowtie \in \{\leq, <, \geq, >\}$ and $p \in]0, 1[$.

$$\mathcal{P}_{\bowtie p}\,(\neg serve_j\,\mathcal{U}\,serve_i)$$

The execution time for model checking Φ_2 is dependent on the instantiation of i and j, and the verification result depends, of course, on the choice of $\bowtie$ and p.

Φ_3: Once station i has become busy, it will eventually be polled, i.e. no station will be starved.

$$busy_i \Rightarrow \mathcal{P}_{\geq 1}\,(\Diamond poll_i)$$

Φ_4: Once station i has become busy, with probability $\bowtie p$ it will be polled within t time units (we let $t = 1.5$).

$$busy_i \Rightarrow \mathcal{P}_{\bowtie p}\left(\Diamond^{\leq t} poll_i\right)$$

The remaining two properties are steady-state formulas:

Φ_5: In steady state, the probability of station i being waiting for the server is $\bowtie p$.

$$\mathcal{S}_{\bowtie p}\,(busy_i \wedge \neg serve_i)$$

Φ_6: In steady state, the probability of station i being idle is $\bowtie p$.

$$\mathcal{S}_{\bowtie p}\,(idle_i)$$

d	# states	$\Phi_1 = \neg(poll_i \wedge poll_j)$ time (in *sec*)	$\Phi_2 = \mathcal{P}_{\bowtie p}(\neg serve_j \,\mathcal{U}\, serve_i)$ time (in *sec*)	$\Phi_3 = busy_i \Rightarrow \mathcal{P}_{\geq 1}(\Diamond poll_i)$ time (in *sec*)
3	36	0.002	0.031	0.005
5	240	0.002	0.171	0.009
7	1344	0.005	1.220	0.011
10	15360	0.037	16.14	0.080

	$\Phi_4 = busy_i \Rightarrow \mathcal{P}_{\bowtie p}(\Diamond^{\leq 1.5} poll_i)$			$\Phi_5 = \mathcal{S}_{\bowtie p}(busy_i \wedge \neg serve_i)$		$\Phi_6 = \mathcal{S}_{\bowtie p}(idle_i)$	
	numerical integration		transient analysis				
d	# iter.	time (in *sec*)	time (in *sec*)	# iter.	time (in *sec*)	# iter.	time (in *sec*)
3	8	2.308	0.068	39	0.044	39	0.038
5	12	30.92	0.233	61	0.103	61	0.102
7	14	308.5	1.430	80	0.677	80	0.658
10	18	7090	17.67	107	11.28	107	11.29

Table 10.4: Verification runtimes for checking CSL-formulas on the polling system

The execution times for checking these properties with our tool ETMCC are given in Tab. 10.4. All path-properties were checked with precision $\varepsilon = 10^{-6}$, and the steady-state properties were checked with precision $\varepsilon = 10^{-8}$. As mentioned before, the tool ETMCC implements two different algorithms for checking properties of type time-bounded until (such as property Φ_4): One is an iterative numerical integration algorithm which works on discretised representations of distribution functions[10], while the other is transient analysis (following the well-known uniformisation method) of a Markov chain which has been modified according to the particular formula at hand [19]. In general, as in the particular case of property Φ_4, the latter method is far superior, both with respect to numerical accuracy and to execution time [173].

10.3 Multiprocessor mainframe with software failures

10.3.1 System description

In this section, we consider a multiprocessor mainframe which was first introduced in [190] and has since then served as a standard stochastic process algebra example, see e.g. [166, 99]. Here we only briefly repeat the main features of the model.

[10] For checking property Φ_4, the number of interpolation points for numerical integration was set to "only" 64.

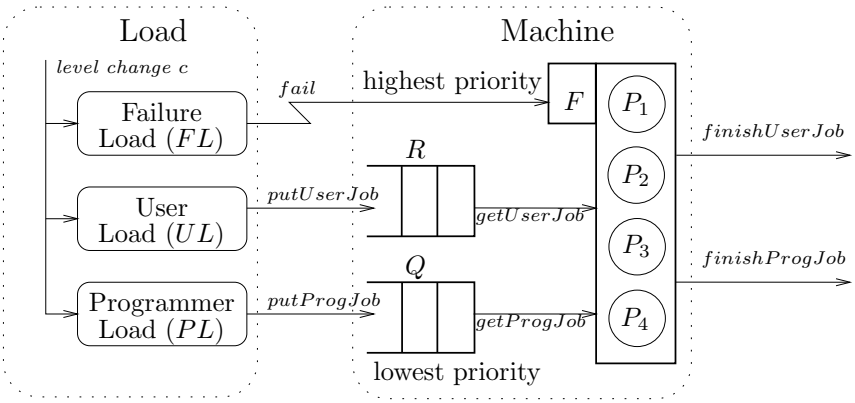


Figure 10.9: Mainframe model structure

The multiprocessor mainframe serves two purposes: It has to process database transactions submitted by users, and it provides computing capacity to programmers maintaining the database. The system is subject to software failures which are modelled as special jobs. On the top level, the system is composed of two processes (cf. Fig. 10.9).

$$System := Load \,|[putUserJob, putProgJob, fail|]\, Machine$$

Process *Load* represents the system load caused by the database users, the programmers and the failures. The mainframe itself is modelled by the *Machine* process.

The three different system load components are modelled as Markov modulated Poisson processes, see [190]. The intensity of the load changes between different levels. To realize a synchronous change of load level, a synchronising action c is used.

$$Load := ProgLoad \,|[c]|\, UserLoad \,|[c]|\, FailLoad$$

The *Machine* component consists of two finite queues (Q and R), a failure handling component (F) and four identical processors. The queues buffer incoming jobs. They are controlled by a priority mechanism to ensure that programmer jobs have the lowest priority, while failures have the highest priority. The priority mechanism is realised by appropriate synchronisation of the queue processes. For instance, process Q can only deliver a job to a processor if queue R is empty and no failure is present. Furthermore, the insertion of new jobs into the system is prohibited once a failure has occurred, until the system is repaired.

Each of the four processors executes user or programmer jobs waiting in the respective queues, unless a failure occurs. As failures have preemptive priority over the other two job classes, all processors stop working once action $fail$ has

occurred and then wait until the system will recover (via action *repair*). The actual repair is controlled by the failure handling component.

10.3.2 State space construction

In this section, we consider a multiprocessor mainframe model with 40 (10) programmer (user) buffer places (the number of buffer places of the finite queues Q and R is the major factor influencing the size of the state space). With this choice, the model has 110946 reachable states and 761989 transitions. Note that this is a model where all transitions are Markovian, i.e. there are no immediate transitions.

We generated the MTBDD representing the overall multiprocessor mainframe system from 10 elementary components in a compositional fashion with the help of our tool Im-Cat. As a preparation step, we used TIPPtool to generate the transition systems for the three load components ($UserLoad$, $ProgLoad$ and $FailLoad$), for the two queues (Q and R), for the failure handling component (F), and for the four processors (P_1, P_2, P_3 and P_4). These 10 transition systems were then encoded as MTBDDs (in that order). Afterwards, MTBDD-based parallel composition was applied in a step by step fashion, as depicted in Fig. 10.10. That figure gives the size of the MTBDDs at each step of the construction. In contrast to Fig. 10.3, only one number is given at each stage, since the mainframe model only contains Markovian transitions and thus the MTBDD representing immediate transitions is always zero.

The resulting MTBDD representing the overall multiprocessor mainframe system has 985 vertices, which number is increased to 1206 vertices after reachability analysis (MTBDD-based reachability analysis took less than one second). Obviously, the MTBDD is very compact, considering that it encodes 110946 reachable states and 761989 transitions.

We tried to calculate the steady-state probabilities for this model on the basis of its reachable state space. The power iteration matrix has 2797 MTBDD vertices but took 7.5 hours to generate. Iteration was slowed down severely by swapping activities, such that after 16.5 hours, when not even 10 iterations had been performed, the experiment was aborted. Next, we tried the power method on the potential state space, which has more than 33 million states. The iteration matrix now has 21153 MTBDD vertices and took almost 12 hours to generate. However, in this case the iteration converged after only 10 steps, and one iteration took only 30.63 seconds.

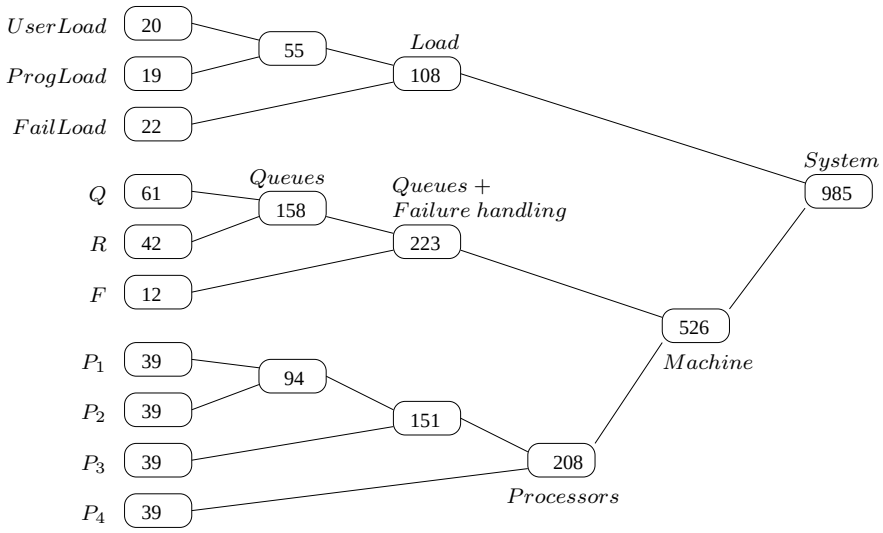


Figure 10.10: Compositional MTBDD construction for the multiprocessor mainframe system: Number of MTBDD vertices (for Markovian transitions)

We also wish to remind the reader of the results presented earlier in Sec. 8.1.1, where we considered a smaller instance of the multiprocessor mainframe model featuring 2640 states and 12295 transitions. There we had used a space-inefficient monolithic MTBDD encoding of the transition system, but had found that the construction of the iteration matrix and the actual iteration had worked reasonably fast.

In summary, this example shows how unpredictably MTBDD-based numerical analysis may behave. For instance, it is contrary to our general experience that the power iteration matrix based on the actual state space is smaller than the one based on the potential state space. Furthermore, it is astonishing that iteration with the matrix of 2797 vertices is so slow, while iteration with the matrix of 21153 vertices is relatively fast.

10.3.3 Performance evaluation

We do not discuss the calculation of performance results for the multiprocessor mainframe model here, since a rather comprehensive set of such results has already been presented before, and since our main concern is MTBDD-based representation. The interested reader is referred to [190], where the derivation of several performance measures, both of steady-state and transient type, is discussed in detail.

10.3.4 Verification of performability properties

This section contains some example properties which are of interest for the multiprocessor mainframe model. For each property, a description in plain English, its aCSL formulation and some explanation are given. We first introduce some purely functional requirements to ensure that the priority mechanism is properly realised by the model. Then we check some performance and reliability requirements which the system should satisfy. As before, for $A \subseteq Act$ we let $\overline{A}$ denote $Act \setminus A$, and we omit brackets for singleton sets. We use the following sets of actions: $Get := \{getUserJob, getProgJob\}$, $Put := \{putUserJob, putProgJob\}$, $Fin := \{finishUserJob, finishProgJob\}$ and $FailRep := \{fail, repair\}$.

Φ_1: If there are user jobs waiting, the processors will not start programmer jobs.

$$\Phi_{UserJobWaiting} \;\Rightarrow\; \neg\langle getProgJob\rangle\; tt$$

where $\Phi_{UserJobWaiting}$ is defined by $\exists\, (\overline{_{putUserJob}}\Diamond_{getUserJob}\; tt)$, characterising at least one user job waiting in the queue.

Φ_2: Whenever a failure occurs, no jobs can be inserted into the queues until the system is repaired.

$$[fail]\; \forall\, (\overline{_{Put}}\Diamond_{repair}\, tt)$$

Φ_3: Whenever a failure occurs, the processors will be blocked until the system is repaired.

$$[fail]\; \forall\, (\overline{_{Get \cup Fin}}\Diamond_{repair}\, tt)$$

Φ_4: After a repair, both queues are empty.

$$[repair]\; (\; \neg\, \Phi_{UserJobWaiting} \;\wedge\; \neg\, \Phi_{ProgJobWaiting}\;)$$

where $\Phi_{ProgJobWaiting}$ characterises at least one waiting programming job, defined in a similar way as $\Phi_{UserJobWaiting}$. This is an example of a property which is not true, since a failure does not cause the queues to be flushed.

Φ_5: In steady state, the probability of low priority programming jobs having to wait because of user jobs being served is smaller than 0.01.

$$\mathcal{S}_{<0.01}\, (\langle finishUserJob\rangle tt \;\wedge\; \Phi_{ProgJobWaiting})$$

Φ_6: At least two processors are occupied by user jobs.

$$\langle finishUserJob\rangle\; \langle finishUserJob\rangle\; tt$$

states (original)	3690	13530	110946
(compressed)	720	2640	21648

property	verification runtimes (in seconds)		
Φ_1	0.012	0.037	0.268
Φ_2	0.008	0.049	0.864
Φ_3	0.008	0.039	0.319
Φ_4	0.003	0.005	0.036
Φ_5	0.642	2.371	18.750
Φ_6	0.001	0.002	0.014
Φ_7	0.558	2.122	18.814
Φ_9	0.554	2.009	18.819
Φ_{10}	0.387	1.570	11.603

Table 10.5: Verification runtimes for the multiprocessor mainframe system

Φ_7: In steady state, the probability that at least two processors are occupied by user jobs is greater than 0.002.

$$\mathcal{S}_{>0.002}\left(\Phi_6\right)$$

Φ_8: There is at least a 30% chance that some job will be finished within at most 4 time units.

$$\mathcal{P}_{\geq 0.3}\left(\overline{_{Fin}}\Diamond^{<4}_{Fin}\, tt\right)$$

Φ_9: In steady state, the probability of the system being unavailable (i.e. waiting for repair) is at most 0.05.

$$\mathcal{S}_{\leq 0.05}\left(\exists(\overline{_{FailRep}}\Diamond_{repair}\, tt)\right)$$

Φ_{10}: After a system failure, there is a chance of more than 90% that it will come up again within the next 5 time units.

$$[fail]\ \mathcal{P}_{>0.9}\left(\overline{_{repair}}\Diamond^{<5}_{repair}\, tt\right)$$

The fact that the above property holds for all states can be expressed by $\forall \Box\ \Phi_{10}$. Slightly weaker, one might require the above property to hold on the long run, formulated as $\mathcal{S}_{\geq 1}\left(\Phi_{10}\right)$.

We now report on our experience with the verification of the above properties. As mentioned in Sec. 10.1.5, the aCSL model checking component of our tool

ETMCC has only recently been completed [44] and was not yet available during this case study. Therefore, the results presented here were obtained by translating the above aCSL properties to CSL[11].

The verification runtimes are given in Table 10.5. We checked three models: A small model with 4 (2) programmer (user) buffer places, an intermediate model with 10 (4) programmer (user) buffer places and a large model with 40 (10) programmer (user) buffer places. The small model has 3690 states and 24009 transitions, the intermediate model has 13530 states and 91069 transitions and the large model has 110946 states and 761989 transitions. However, we did not perform model checking on the original models but on models with compressed state spaces which we gained through the application of Markovian bisimilarity (in the example multiprocessor system, the main potential for reduction stems from the symmetry of the four identical processors). By Thm. 9.3.1, the compressed models satisfy the same properties as the original ones. After bisimilarity compression, the small model has 720 states and 3219 transitions, the intermediate model has 2640 states and 12295 transitions and the large model has 21648 states and 103471 transitions. All steady-state properties given in the table were double checked with TIPPtool.

[11] We did not follow the translation procedure from aCSL to CSL which involves splitting of states as described in Sec. 9.3.4, but simply translated the properties in an ad-hoc fashion without modifying the structure of the model. For that reason, translation of property Φ_8 was not possible, and therefore Φ_8 is not checked here.

Chapter 11

Discussion and conclusions

11.1 Future role of performance analysis and verification

Communication plays a vital role in our society, with communication systems of increasing complexity penetrating virtually all areas of our daily life. The development and management of such distributed systems, which are expected to meet the highest quality standards, is difficult and poses many challenges. Model-based performability analysis and verification of not only functional properties are highly valuable methods which can help solving these problems, and it is generally agreed on that the importance of these methods will increase in the future. These considerations clearly justify the research into new methods for performance analysis and verification.

11.2 Assessment of the BDD-based approach

State space explosion remains one of the most prominent problems of analytical performance and dependability modelling, and it is also a major problem in the areas of formal verification and model checking. In Chap. 1 we gave an overview of techniques that have been developed in order to avoid or tolerate large state spaces. Our own focus in this thesis was on symbolic techniques based on decision diagrams, which we consider to be very promising to successfully alleviate the state space explosion problem.

We have shown that with the help of BDDs and related data structures it is possible in many instances to achieve extremely compact representations of enormous state spaces. This usefulness of BDDs to encode transition systems had been observed before. However, we have stressed the important point that a naïve encoding of state spaces or transition systems usually does not lead to memory-efficient representations. We showed that heuristics for encodings are needed, exploiting the structure of the specification. In particular, we saw that the realisation of parallel composition on BDDs is extremely successful, since an exponential blow-up can be reduced to a linear growth. This leads us to the conclusion that the use of BDDs is only beneficial if they are employed in a compositional framework, where large models are constructed from small components.

All conventional algorithms for building, manipulating and analysing transition systems (LTS, SLTS or ESLTS) have a corresponding symbolic algorithm which works directly on the symbolic representation, such that the whole modelling and evaluation process can be carried out in a symbolic setting. Most of these symbolic algorithms are also time-efficient, provided that the size of the decision diagrams is kept reasonably small. However, as an exception to this general rule, it turned out that numerical calculations based on symbolic representations are slow. In particular, MTBDD-based matrix multiplication and vector-matrix multiplication constitute a performance bottleneck in the symbolic performance analysis process. Therefore, in the future we will need to accelerate these operations by techniques such as the ones suggested in Chap. 8.

11.3 Tools for BDD-based modelling

The current state-of-the-art of any approach or methodology is always reflected and documented by the existing software tools that support the approach. It is therefore worth mentioning the existing software implementations of the BDD-based approach to performability modelling and verification which we described in this thesis. Therefore, in this section, we briefly survey some software tools which are related to the symbolic representation and analysis of stochastic models.

The tool DNBDDtool was developed from scratch at the University of Erlangen-Nürnberg as a proof of concept of the DNBDD data structure [43]. Its main capabilities are the representation of SLTS, DNBDD-based parallel composition and reachability analysis, and the minimisation of the state space on the basis of Markovian bisimulation.

Im-Cat is a tool for the MTBDD-based generation and analysis of ESLTSs. It was developed at the University of Erlangen-Nürnberg [128, 129, 133] and uses

the library CUDD [307] as its underlying MTBDD engine. Apart from model representation, parallel composition, hiding and reachability analysis, Im-Cat supports a flexible mechanism for the elimination of compositionally vanishing states (cf. Sec. 5.2.2). Im-Cat also supports MTBDD-based numerical steady-state analysis, where the power method, Jacobi method, different versions of the Gauss-Seidel method and the Bi-CGSTAB method are realised.

The probabilistic model checker Prism [102, 231, 228], developed at the University of Birmingham, has its own module-based system description language and uses MTBDDs as its underlying data structure. Prism supports modular model generation, the checking of PCTL [148] properties of DTMCs or Markov decision processes, and is currently being extended towards the checking of CSL [21] properties on state-labelled CTMCs.

A tool-set for the symbolic performance analysis of GSPNs was developed at the University of Cape Town [100]. It generates MTBDDs directly from the GSPN specification, aims at compact representation without considering composition of submodels (but taking into account structural properties of the Petri net at hand) and performs MTBDD-based numerical analysis.

The latter two tools were developed as part of academic research projects similar to our own (actually, close communication exists between our group and the groups of Birmingham and Cape Town), and the general findings of these projects, especially concerning the compactness of symbolic representations and the unsatisfactory speed of linear algebra operations, are also similar to our own, although the context of their work is somewhat different.

11.4 Future research

Although quite a lot has been achieved, there remain many open problems and questions. Therefore, in this concluding section, we point out a few interesting topics for future research.

Compact encodings: We have shown that the use of decision diagrams may lead to extremely compact representations of transition systems, provided that they are applied in a compositional context. We have not investigated techniques for finding good or even optimal encodings for monolithic transition systems. In [100], techniques for the compact representation of the reachability information of GSPNs have been developed, by exploiting structural properties of Petri nets. It would be interesting to develop similar heuristics for other modelling formalisms

or specific fields of applications. A related topic is the problem of optimising the variable ordering within a BDD by heuristic methods, based on information from the high-level model specification.

Speeding up MTBDD-based numerical analysis: As we have pointed out, the speed of the linear algebra operations on matrices represented as MTBDDs is a major bottleneck of the symbolic approach. As already said in Sec. 8.3.2, more effort should be put into the development of dedicated hardware for efficiently performing these operations on MTBDDs. There is a high potential for parallel processing which could help to speed up the operations. Parallelisation (or distribution) of apply-type BDD algorithms should also be studied and could also be implemented without dedicated hardware. Hybrid algorithms, such as the one proposed in [268, 269], are a promising alternative to purely MTBDD-based solutions, but the details of these techniques have not yet been published and will certainly need to be investigated further.

Verification of stochastic systems using symbolic methods: In Chap. 9 we presented an approach to the verification of performability properties by model checking of stochastic systems. The logic we discussed, called aCSL, is currently being extended such that it will be possible to specify a broader class of properties. The development of a general logic for performability is still a hot topic for future research. Our model checker ETMCC, which is capable of checking properties specified with the help of the logics CSL and aCSL, uses its own sparse data structure for storing the model under investigation. It would certainly be worth-while to realise a symbolic version of ETMCC which is based on MTBDDs as the underlying data structure, and compare its memory requirements and performance to the current one. This could be combined with the direct generation of MTBDD representations from process algebraic specifications as described below.

BDD semantics for stochastic process algebras: We have seen that symbolic parallel composition realises the semantics of the parallel composition operator of (stochastic) process algebras. In Sec. 5.2.2, we have also described a symbolic realisation of the hiding operator (which, by the way, could easily be extended to a general renaming of actions). It would be interesting to develop a semantics for SPA which is fully BDD-based, i.e. which generates MTBDDs directly from an SPA specification without the need to generate an SLTS or ESLTS first. To this end, one would need to develop symbolic realisations of the other SPA operators (namely choice, prefixing, recursion, process instantiation). There already exists some work in this direction: The paper [118] describes operators on BDDs which realise the CCS operators parallel composition, channel-hiding and renaming. The papers [305] and [306] describe an algorithm for building BDD representations from LOTOS specifications, also exploiting the compositionality

of process algebras. The paper [112] describes a general method to efficiently generate reduced BDDs from (non-stochastic) process algebraic descriptions, where the operational semantics of the process algebra is given by a so-called GSOS system [29] (which is further constrained such that finiteness of the state space is guaranteed). Its main idea is to symbolically encode the parse tree of the process term at hand, as well as the current progress of the process. All of the mentioned investigations consider only purely functional process algebraic models. In [280], however, a denotational matrix semantics is presented which — although not being concerned with symbolic representations — realises a mapping from a restricted stochastic process algebra to matrices instead of the usual transition systems. These matrices, of course, could be represented by MTBDDs.

Bibliography

[1] M. Ajmone Marsan, G. Balbo, and G. Conte. A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems. *ACM Transactions on Computer Systems*, 2(2):93–122, May 1984.

[2] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with generalized stochastic Petri nets.* Wiley, 1995.

[3] S.B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, C-27:509–516, June 1978.

[4] S.B. Akers. Functional testing with binary decision diagrams. In *Proc. 8th Annual Conf. on Fault-Tolerant Computing*, pages 75–82, 1978.

[5] S. Allmaier, S. Dalibor, and D. Kreische. Parallel Graph Generation Algorithms for Shared and Distributed Memory Machines. In *Parallel Computing Conference (ParCo'97)*, Bonn, 1997.

[6] S. Allmaier and G. Horton. Parallel shared-memory state-space exploration in stochastic modelling. In G. Bilardi, A. Ferreira, B. Lüling, and J. Rolim, editors, *Solving Irregularly Structured Problems in Parallel.* Springer, LNCS 1253, 1997.

[7] S. Allmaier, M. Kowarschik, and G. Horton. State-Space Construction and Steady-State Solution of GSPNs on a Shared-Memory Multiprocessor. In *IEEE Int. Workshop Petri Nets and Performance Models (PNPM'97)*, St. Malo, France, 1997. IEEE Computer Science Press.

[8] S. Allmaier and D. Kreische. Parallel Approaches to the Numerical Transient Analysis of Stochastic Reward Nets. In *IEEE 20th Int. Conf. on Application and Theory of Petri Nets (ICATPN'99)*, Williamsburg, Virginia, 1999.

[9] Altera Corporation. *Developer's Guide for the Altera Excalibur Development Kit*, 2000.

[10] R. Alur, C. Courcoubetis, and D. Dill. Model checking for real-timed systems. In *5th IEEE Symp. on Logic in Computer Science*, pages 414–425, 1990.

[11] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

[12] H.R. Andersen. An Introduction to Binary Decision Diagrams. Technical report, Department of Computer Science, Technical University of Denmark, December 1994.

[13] B. Aures. Modellierung des Erlanger Klinikkommunikationssystems mit Hilfe von stochastischen Prozeßalgebren und TIPPtool. Studienarbeit (student's thesis), Universität Erlangen–Nürnberg, October 1998 (in German).

[14] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying continuous time Markov chains. In *Proc. CAV'96*, pages 269–276. Springer, LNCS 1102, 1996.

[15] F. Baccelli, A.M. Makowski, and A. Shwartz. The Fork–Join Queue and Related Systems with Synchronization Constraints: Stochastic Ordering and Computable Bounds. *Advances in Applied Probability*, 21:629–660, 1989.

[16] F. Baccelli, W.A. Massey, and D. Towsley. Acyclic Fork–Join Queuing Networks. *Journal of the Association for Computing Machinery*, 36(3):615–642, 1989.

[17] R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and their Applications. In *ICCAD-93: Int. Conference on Computer-Aided Design*, pages 188–191, Santa Clara, CA, 1993. ACM/IEEE.

[18] R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and their Applications. *Formal Methods in System Design*, 10(2/3):171–206, April/May 1997.

[19] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Model Checking Continuous Time Markov Chains by Transient Analysis. In *Computer-aided Verification (CAV)*, pages 358–372. Springer, LNCS 1855, 2000.

[20] C. Baier and H. Hermanns. Weak Bisimulation for Fully Probabilistic Processes. In *"CAV'97: 9th International Conference on Computer Aided Verification"*, pages 199–130. Springer, LNCS 1254, November 1997.

[21] C. Baier, J.-P. Katoen, and H. Hermanns. Approximate symbolic model checking of continuous-time Markov chains. In *Concurrency Theory*, pages 146–162. Springer, LNCS 1664, 1999.

[22] C. Baier and M. Kwiatkowska. On the verification of qualitative properties of probabilistic processes under fairness constraints. *Inf. Proc. Letters*, 66(2):71–79, 1998.

[23] R. Barrett, M. Berry, T.F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, Philadelphia, PA, 1994.

[24] 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] F. Bause and P. Buchholz. Queueing Petri nets with product form solution. *Performance Evaluation*, 32(4):265–299, 1998.

[26] J.A. Bergstra, A. Ponse, and S.A. Smolka, editors. *Handbook of Process Algebra*. Elsevier, Amsterdam, 2001.

[27] J. Bern, C. Meinel, and A. Slobodova. Global rebuilding of OBDDs avoiding memory requirement maxima. In *Proc. CAV'95*, pages 4–15. Springer, LNCS 939, 1995.

[28] M. Bernardo and R. Gorrieri. A Tutorial on EMPA: A Theory of Concurrent Processes with Nondeterminism, Priorities, Probabilities and Time. *Theoretical Computer Science*, 202:1–54, 1998.

[29] B. Bloom. *Ready simulation, bisimulation and the semantics of CCS-like languages*. PhD thesis, MIT, Cambridge, Mass., October 1990.

[30] H. Bohnenkamp and B. Haverkort. Semi-Numerical Solution of Stochastic Process Algebra Models. In J.-P. Katoen, editor, *ARTS'99*, pages 228–243. Springer, LNCS 1601, 1999.

[31] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In R. Alur and T. Henzinger, editors, *Computer-Aided Verification (CAV'96)*, pages 1–12. Springer, LNCS 1102, 1996.

[32] G. Bolch and S. Greiner. Modeling and Performance Evaluation of Production Lines Using the Modeling Language MOSEL. In *Proceedings of the 2nd IEEE/ECLA/IFIP Int. Conf. on Architectures and Design Methods for Balanced Automation Systems*, pages 163–174, June 1996.

[33] G. Bolch, M. Roessler, R. Zimmer, H. Jung, and S. Greiner. Leistungsbewertung mit PEPSY-QNS und MOSES. *Informationstechnik und Technische Informatik*, (3/95):28–33, Juni 1995 (in German).

[34] B. Bollig, M. Löbbing, , and I. Wegener. Simulated annealing to improve variable orderings for OBDDs. In *Int. Workshop on Logic Synthesis*, 1995.

[35] B. Bollig, P. Savicky, and I. Wegener. On the improvement of variable orderings for OBDDs. In *IFIP Workshop on Logic and Architecture Synthesis*, pages 71–80, Grenoble, 1994.

[36] B. Bollig and I. Wegener. Improving the Variable Ordering of OBDDs is NP-Complete. *IEEE Transactions on Computers*, 45(9):993–1006, 1996.

[37] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. In P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 23–73. North-Holland, Amsterdam, 1989.

[38] A. Bouali. Weak and branching bisimulation in FCTOOL. Rapports de recherche 1575, INRIA Sophia Antipolis, 1992.

[39] A. Bouali and R. de Simone. Symbolic Bisimulation Minimisation. In *Computer Aided Verification*, pages 96–108, 1992. Springer, LNCS 663.

[40] M. Bozga and O. Maler. On the Representation of Probabilities over Structured Domains. In N. Halbwachs and D. Peled, editors, *International Conference on Computer-Aided Verification (CAV'99)*, pages 261–273. Springer, LNCS 1633, July 1999.

[41] K.S. Brace, R.L. Rudell, and R.E. Bryant. Efficient Implementation of a BDD Package. In *27th ACM/IEEE Design Automation Conference*, pages 40–45, 1990.

[42] H. Brinksma, J.-P. Katoen, R. Langerak, and D. Latella. A stochastic causality-based process algebra. *The Computer Journal*, 38(7):552–565, December 1995. Special issue: Proc. of the 3rd Workshop on Process Algebras and Performance Modelling.

[43] H. Bruchner. Symbolische Manipulation von stochastischen Transitionssystemen. Studienarbeit (student's thesis), Universität Erlangen–Nürnberg, 1998 (in German).

[44] H. Bruchner. Entwicklung und Implementierung von Verfahren für das Model Checking stochastischer Systeme. Diplomarbeit (Masters thesis), Universität Erlangen–Nürnberg, September 2001 (in German).

[45] R.E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE ToC*, C-35(8):677–691, August 1986.

[46] R.E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[47] R.E. Bryant and Y. Chen. Verification of Arithmetic Functions with Binary Moment Diagrams. Technical Report CMU-CS-94-160, CMU, 1994.

[48] R.E. Bryant and Y. Chen. Verification of Arithmetic Functions with Binary Moment Diagrams. In *32nd ACM/IEEE Design Automation Conference*, pages 535–541, 1995.

[49] R.E. Bryant and C.-J.H. Seger. Formal Verification of Digital Circuits by Symbolic Ternary System Models. In E.M. Clarke and R.P. Kurshan, editors, *Computer-Aided Verification (CAV'90)*, pages 33–43. Springer, LNCS 531, 1990.

[50] P. Buchholz. *Die strukturierte Analyse Markovscher Modelle*. PhD thesis, Universität Dortmund, 1991 (in German).

[51] P. Buchholz. A Hierarchical View of GCSPNs and Its Impact on Qualitative and Quantitative Analysis. *Journal of Parallel and Distributed Computing*, 15(3):207–224, July 1992.

[52] P. Buchholz. Aggregation and Reduction Techniques for Hierarchical GCSPNs. In *Proceedings of PNPM '93*, pages 216–225, Tolouse, October 1993.

[53] P. Buchholz. Hierarchies in Colored GSPNs. In M. Ajmone Marsan, editor, *14th Int. Conf. on Application and Theory of Petri Nets*, pages 106–125. Springer, LNCS 691, 1993.

[54] P. Buchholz. A class of hierarchical queueing networks and their analysis. *Queueing Systems*, 15:59–80, 1994.

[55] P. Buchholz. Exact and Ordinary Lumpability in Finite Markov Chains. *Journal of Applied Probability*, (31):59–75, 1994.

[56] P. Buchholz. Markovian Process Algebra: Composition and Equivalence. In U. Herzog and M. Rettelbach, editors, *Proc. of the 2nd Workshop on Process Algebras and Performance Modelling*, pages 11–30, Regensberg/Erlangen, July 1994. Arbeitsberichte des IMMD, Universität Erlangen-Nürnberg.

[57] P. Buchholz. A Framework for the Hierarchical Analysis of Discrete Event Dynamic Systems. Habilitation thesis, Universität Dortmund, 1996.

[58] P. Buchholz. Projection methods for the analysis of stochastic automata networks. In B. Plateau, W.J. Stewart, and M. Silva, editors, *Numerical Solution of Markov Chains*, pages 149–168, Zaragoza, Spain, Sept. 1999. Prensas Universitarias de Zaragoza.

[59] P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper. Complexity of Kronecker Operations on Sparse Matrices with Applications to Solution of Markov Models. Technical Report ICASE Report No. 97-66, ICASE, Langley, VA, 1997.

[60] P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper. Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models. *INFORMS Journal of Computing*, 12(3):203–222, Summer 2000.

[61] P. Buchholz and P. Kemper. On Generating a Hierarchy for GSPN Analysis. *Performance Evaluation Review (ACM Sigmetrics)*, 26(2):5–14, August 1998.

[62] P. Buchholz and P. Kemper. A Toolbox for the Analysis of Discrete Event Dynamic Systems. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification*, pages 483–486. Springer, LNCS 1633, 1999.

[63] J.R. Burch, E.M. Clarke, and D. Long. Symbolic Model Checking with partitioned transition relations. In *VLSI'91*, Edinburgh, Scotland, 1991.

[64] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, and D. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on CAD*, 13:401–424, 1994.

[65] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, (98):142–170, 1992.

[66] J.P. Buzen. *Queueing Network Models of Multiprogramming*. PhD thesis, Harvard University, Division of Engineering and Applied Sciences, Cambridge, Mass., 1971.

[67] J.P. Buzen. Computational Algorithms for Closed Queueing Networks with Exponential Servers. *Communications of the ACM*, 16:527–531, 1973.

[68] W.L. Cao and W.J. Stewart. Iterative Aggregation/Disaggregation Techniques for Nearly Uncoupled Markov Chains. *Journal of the ACM*, 32(3):702–719, July 1985.

[69] S. Caselli, G. Conte, F. Bonardi, and M. Fontanesi. Experience on SIMD Massively Parallel GSPN Analysis. In G. Haring and G. Kotsis, editors, *7th Int. Conf. on Modelling Techniques and Tools for Comoputer Performance Evaluation (TOOLS'94)*, pages 266–283, Wien, 1994. Springer, LNCS 794.

[70] S. Caselli, G. Conte, and P. Marenzoni. Parallel state space exploration for GSPN models. In *16th Int. Conf. on the Application and Theory of Petri Nets*, pages 181–200. Springer, LNCS 935, 1995.

[71] E. Çinlar. *Introduction to Stochastic Processes*. Prentice-Hall, 1975.

[72] K.C. Chang. *Digital Systems Design with VHDL and Synthesis: An Integrated Approach*. IEEE, Los Alamitos, CA, 1999.

[73] G. Chiola. GreatSPN 1.5 Software Architecture. In G. Balbo and G. Serazzi, editors, *Proceedings of the 5th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Torino, Feburary 1991*, pages 117–132. Elsevier Science Publisher B.V., 1992.

[74] G. Chiola, J. Campos, J.M. Colom, M. Silva, and C. Anglano. Operational analysis of timed Petri nets and application to the computation of performance bounds. In *th Intern. Workshop on Petri Nets and Performance Models*. IEEE Computer Society, 1993.

[75] G. Chiola, S. Donatelli, and G. Franceschinis. GSPNs versus SPNs: what is the actual role of immediate transitions? In *Proceedings of the 4th International Workshop on Petri Nets and Performance Models*, pages 20–31, Melbourne, December 1991. IEEE.

[76] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. On Well-Formed Coloured Nets and their Symbolic Reachability Graph. In *Proceedings of the 11th International Conference on Application and Theory of Petri Nets*, pages 387–410, Paris, June 1990. Reprinted in High-level Petri Nets, K. Jensen, G. Rozenberg, eds.

[77] G. Ciardo. *Analysis of Large Stochastic Petri Net Models*. PhD thesis, Duke University, Durham, NC, USA, 1989.

[78] G. Ciardo, J. Gluckman, and D. Nicol. Distributed state space generation of discrete-state stochastic models. *INFORMS Journal on Computing*, 10(1):82–93, 1998.

[79] G. Ciardo and A.S. Miner. Storage alternatives for large structured state spaces. In *Proc. 9th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 44–57, St. Malo, France, 1997. Springer, LNCS 1245.

[80] G. Ciardo and A.S. Miner. A data structure for the efficient Kronecker solution of GSPNs. In P. Buchholz and M. Silva, editors, *PNPM'99*, pages 22–31. IEEE computer society, 1999.

[81] G. Ciardo and M. Tilgner. Parametric State Space Structuring. Technical Report ICASE Report No. 97-67, ICASE, 1997.

[82] G. Ciardo and K.S. Trivedi. A Decomposition Approach for Stochastic Petri Net Models. In *Proceedings of the Fourth International Workshop on Petri nets and Performance Models*, pages 74–83, Melbourne, December 1991.

[83] G. Ciardo and K.S. Trivedi. A decomposition approach for stochastic reward net models. *Performance Evaluation*, 18(1):37–59, July 1993.

[84] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In R. Alur and T. Henzinger, editors, *Computer-Aided Verification (CAV'96)*, pages 419–422. Springer, LNCS 1102, 1996.

[85] E.M. Clarke, E.A. Emerson, and A. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *10th ACM Annual Symp. on Principles of Programming Languages*, pages 117–126, 1983.

[86] E.M. Clarke, M. Fujita, P. McGeer, K. McMillan, J. Yang, and X. Zhao. Multi-terminal Binary Decision Diagrams: An efficient data structure for matrix representation. In *IWLS: International Workshop on Logic Synthesis*, Tahoe City, May 1993.

[87] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[88] E.M. Clarke and R.P. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, 1996.

[89] E.M. Clarke, K.L. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral Transforms for Large Boolean Functions with Applications to Technology Mapping. In *30th Design Automation Conference*, pages 54–60. ACM/IEEE, 1993.

[90] A.E. Conway and N.D. Georganas. *Queueing Networks – Exact Computational Algorithms*. MIT Press, 1989.

[91] D. Coppersmith and S. Winograd. Matrix Multiplication via Arithmetic Progressions. In *Proc. 19th ACM Symposium on Theory of Computing*, 1987.

[92] P.J. Courtois. *Decomposability, queueing and computer system applications.* ACM monograph series, 1977.

[93] P.J. Courtois. On Time and Space Decomposition of Complex Structures. *Communications of the ACM*, 28(6):590–603, June 1985.

[94] J. Couvillion, R. Freire, R. Johnson, W.D. Obal II, M.A. Qureshi, M. Rai, W.H. Sanders, and J. Tvedt. Performability modeling with UltraSAN. *IEEE Software*, 8(5):69–80, September 1991.

[95] D.R. Cox and W.L. Smith. *Queues.* Methuen, London, 1961.

[96] W. Damm, H. Hungar, P. Kelb, and R. Schlör. Statecharts. In C. Lewerentz and T. Lindner, editors, *Formal Development of Reactive Systems*, chapter VIII, pages 131–149. Springer, LNCS 891, 1995.

[97] P.R. D'Argenio, J.-P. Katoen, and E. Brinksma. Specification and analysis of soft real-time systems: Quantity and quality. In *Proc. IEEE Real-Time Systems Symposium*, pages 104–114. IEEE CS Press, 1999.

[98] P.R. D'Argenio, J.P. Katoen, and E. Brinksma. A compositional approach to generalised semi–Markov processes. In A. Guia and R. Smedinga, editors, *Workshop on Discrete–Event Systems.* IEEE Press, 1998.

[99] P.R. D'Argenio, J.P. Katoen, and E. Brinksma. General purpose discrete–event simulation using Spades. In C. Priami, editor, *6th Int. Workshop on Process Algebras and Performance Modelling*, pages 85–102. Universita di Verona, 1998.

[100] I. Davies. Symbolic techniques for the performance analysis of generalised stochastic Petri nets. Master's thesis, University of Cape Town, Department of Computer Science, January 2001.

[101] M. Davio. Kronecker Products and Shuffle Algebra. *IEEE Transactions on Computers*, C-30(2):116–125, February 1981.

[102] L. de Alfaro, M. Kwiatkowska, G. Norman, D. Parker, and R. Segala. Symbolic Model Checking for Probabilistic Processes using MTBDDs and the Kronecker Representation. In S. Graf and M. Schwartzbach, editors, *TACAS'2000*, pages 395–410, Berlin, 2000. Springer, LNCS 1785.

[103] D.D. Deavours and W.H. Sanders. An efficient disk-based tool for solving large Markov models. *Performance Evaluation*, 33(1):67–84, June 1998.

[104] H. Van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13:631–644, 1992.

[105] J. Desel. Basic Linear Algebraic Techniques for Place/Transition Nets. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, pages 257–308. Springer, NCS 1491, 1998.

[106] E. Doberkat. *Stochastic Automata: Stability, Nondeterminims and Prediction*. Springer, LNCS 113, 1981.

[107] S. Donatelli. Superposed stochastic automata: a class of stochastic Petri nets with parallel solution and distributed state space. *Performance Evaluation*, 18(1):21–36, July 1993.

[108] S. Donatelli. Superposed Generalized Stochastic Petri Nets: Definition and Efficient Solution. In M. Silva, editor, *15th Int. Conf. on Application and Theory of Petri Nets*, Zaragoza, Spain, June 1994.

[109] R. Drechsler and B. Becker. *Binary Decision Diagrams: Theory and Implementation*. Kluwer Academic Publishers, 1998.

[110] R. Drechsler and B. Becker. *Graphenbasierte Funktionsdarstellung: Boolesche und Pseudo-Boolesche Funktionen*. Leitfäden der Informatik. Teubner, Stuttgart, 1998 (in German).

[111] R. Drechsler, B. Becker, and S. Ruppertz. K*BMDs: A new data structure for verification. In *European Design and Test Conference*, pages 2–8, 1996.

[112] A. Dsouza and B. Bloom. Generating BDD models for process algebra terms. In *Computer Aided Verification*, pages 16–30, 1995. Springer, LNCS 939.

[113] W. Dulz, S. Gruhl, L. Kerber, and M. Söllner. Early Performance Prediction of SDL/MSC-specified Systems by Automated Synthetic Code Generation. In R. Dssouli, G. v. Bochmann, and Y. Lahav, editors, *SDL'99: The Next Millenium*, pages 457–471. Elsevier, 1999.

[114] H. El-Sayed, D. Cameron, and M. Woodside. Automated Performance Modeling from Scenarios and SDL Designs of Distributed Systems. In *Int. Symp. on Software Engineering for Parallel and Distributed Systems*, pages 127–135, Kyoto, Japan, 1998. IEEE Press.

[115] S.E. Elmaghraby. *Activity Networks: Project Planning and Control by Network Models*. John Wiley and Sons, New York, 1977.

[116] E.A. Emerson and E.M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Automata, Languages and Programming*, pages 169–181. Springer, LNCS 85, 1980.

[117] E.A. Emerson and C.-L. Lei. Temporal model checking under generalized fairness constraints. In *Proc. Hawaii Int. Conf. on System Sc.*, pages 277–288, 1985.

[118] R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for symbolic model checking in CCS. *Distributed Computing*, (6):155–164, 1993.

[119] A. Fantechi, S. Gnesi, and G. Ristori. Model checking for action-based logics. *Formal Methods in System Design*, 4:187–203, 1994.

[120] P. Fernandes, B. Plateau, and W.J. Stewart. Efficient Descriptor-Vector Multiplications in Stochastic Automata Networks. *Journal of the ACM*, 45(3):381–414, May 1998.

[121] J.C. Fernandez. An Implementation of an Efficient Algorithm for Bisimulation Equivalence. *Science of Computer Programming*, 13:219–236, 1989.

[122] J.C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: A Protocol Validation and Verification Toolbox. In *Computer-Aided Verification (CAV)*, pages 437–440,, 1996.

[123] D. Ferrari. Considerations on the Insularity of Performance Evaluation. *IEEE Transactions on Software Engineering*, SE–12(6):678–683, June 1986.

[124] D. Ferrari, G. Serazzi, and A. Zeigner. *Measurement and Tuning of Computer Systems*. Prentice Hall, Inc., Englewood Cliffs, 1983.

[125] M. Fischer and R. Ladner. Propositional dynamic logic of regular programs. *J. Comput. System Sci.*, 18:194–211, 1979.

[126] B.L. Fox and P.W. Glynn. Computing Poisson Probabilities. *Comm. of the ACM*, 31(4):440–445, 1988.

[127] G. Franceschinis and M. Ribaudo. Efficient Performance Analysis Techniques for Stochastic Well-Formed Nets and Stochastic Process Algebras. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets II: Applications*, pages 386–437. Springer, LNCS 1492, 1998.

[128] E. Frank. Codierung und numerische Analyse von Transitionssystemen unter Verwendung von MTBDDs. Studienarbeti (student's thesis), Universität Erlangen–Nürnberg, October 1999 (in German).

[129] E. Frank. Erweiterung eines MTBDD-basierten Werkzeugs für die Analyse stochastischer Transitionssysteme. Technical Report Inf 7, 01/00, Universität Erlangen–Nürnberg, Supervisor: M. Siegle, January 2000 (in German).

[130] M. Fujita, Y. Matsunaga, and T. Kakadu. On variable ordering of binary decision diagrams for the application of multi-valued logic synthesis. In *Proc. EDAC'91*, pages 50–53, 1991.

[131] M. Fujita, P. McGeer, and J.C.-Y. Yang. Multi-terminal Binary Decision Diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10(2/3):149–169, April/May 1997.

[132] M. Fujita and P.C. McGeer, editors. *Special Issue on Multi-Terminal Binary Decision Diagrams*. Formal Methods in System Design, 10(2/3), 1997, 1997.

[133] A. Gajdzik. Erstellung einer Programmoberfläche für das Tool LaTransCAT. Studienarbeit (student's thesis), Universität Erlangen–Nürnberg, January 2001 (in German).

[134] D.D. Gajski. *Digital Design*. Prentice Hall, 1997.

[135] R. German. *Performance Analysis of Communication Systems — Modelling with Non-Markovian Stochastic Petri Nets*. Wiley, 2000.

[136] R. German and C. Lindemann. Analysis of Stochastic Petri Nets by the Method of Supplementary Variables. In G. Iazeolla and S.S. Lavenberg, editors, *Performance '93*, pages 320–338, Roma, September 1993.

[137] S. Gilmore, J. Hillston, and M. Ribaudo. An efficient algorithm for aggregating PEPA models. *IEEE Trans. on Software Engineering*, 27(5), May 2001.

[138] P. Glynn. (ed.), Generalized semi-Markov processes. Special Issue of Discrete Event Dynamic Systems: Theory and Applications 6, 1, 1996.

[139] P. Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-space explosion problem*. Springer, LNCS 1032, 1996.

[140] W.J. Gordon and G.F. Newell. Closed Queueing Systems with Exponential Servers. *Operations Research*, 15:254–265, 1967.

[141] N. Götz. *Stochastische Prozeßalgebren – Integration von funktionalem Entwurf und Leistungsbewertung Verteilter Systeme*. PhD thesis, Universität Erlangen–Nürnberg, Martensstraße 3, 91058 Erlangen, April 1994 (in German).

[142] N. Götz, U. Herzog, and M. Rettelbach. Multiprocessor and Distributed System Design: The Integration of Functional Specification and Performance Analysis Using Stochastic Process Algebras. In *Proc. of the 16th International Symposium on Computer Performance Modelling, Measurement and Evaluation, PERFORMANCE 1993, Tutorial*, pages 121–146. Springer, LNCS 729, 1993.

[143] W. Grassmann. Transient Solutions in Markovian Queues. *European Journal of Operational Research*, 1:396–402, 1977.

[144] J.F. Groote and F.W. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In *Proc. ICALP'90*, pages 626–638. Springer, LNCS 443, 1990.

[145] G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Probabilistic Analysis of Large Finite State Machines. In *31st Design Automation Conference*, pages 270–275, San Diego, CA, June 1994. ACM/IEEE.

[146] G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Symbolic Algorithms to Calculate Steady-State Probabilities of a Finite State Machine. In *European Design Automation Conference*, pages 214–218, Paris, February 1994. IEEE.

[147] G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Markovian Analysis of Large Finite State Machines. *IEEE Trans. on CAD*, 15(12):1479–1493, Dec. 1996.

[148] H. Hansson and B. Jonsson. A logic for reasoning about time and probability. *Formal Aspects of Computing*, (6):512–535, 1994.

[149] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[150] P. Harrison and J. Hillston. Exploiting quasi-reversible structures in Markovian process algebra models. *The Computer Journal*, 38(7):510–520, December 1995. Special issue: Proc. of the 3rd Workshop on Process Algebras and Performance Modelling.

[151] F. Hartleb. Graph Models for Performance Evaluation of Parallel Programs. In A. Bode and M. Dal Cin, editors, *Paralle Computer Architectures: Theory, Hardware, Software, Applications*, pages 80–86. Springer, LNCS 732, 1993.

[152] F. Hartleb, P. Dauphin, R. Klar, A. Quick, and M. Siegle. Modellierung und Meßunterstützung mit dem Werkzeug PEPP. *it+ti — Informationstechnik und Technische Informatik*, 37(3):41–48, Juni 1995 (in German).

[153] F. Hartleb and V. Mertsiotakis. Bounds for the Mean Runtime of Parallel Programs. In R. Pooley and J. Hillston, editors, *Proceedings of the Sixth International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 197–210, Edinburgh, 1992.

[154] F. Hartleb and A. Quick. Performance Evaluation of Parallel Programms — Modeling and Monitoring with the Tool PEPP. In B. Walke and O. Spaniol, editors, *Proceedings der 7. GI-ITG Fachtagung "Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen", Aachen, 21.–23. September 1993*, pages 51–63. Informatik Aktuell, Springer, 1993.

[155] V. Hartonas-Garmhausen, S. Campos, and E. Clarke. ProbVerus: Probabilistic Symbolic Model Checking. In J.-P. Katoen, editor, *Formal Methods for Real-Time and Probabilistic Systems (ARTS'99)*, pages 96–110. Springer, LNCS 1601, 1999.

[156] C. Harvey. Performance Engineering as an Integral Part of System Design. *British Telecom Technology Journal*, 4(3):142–147, July 1986.

[157] B. Haverkort, A. Bell, and H. Bohnenkamp. On the Efficient Sequential and Distributed Generation of Very Large Markov Chains from Stochastic Petri Nets. In P. Buchholz and M. Silva, editors, *Petri Nets and Performance Models (PNPM'99)*, pages 12–21. IEEE Computer Society, 1999.

[158] B.R. Haverkort and K.S. Trivedi. Specification Techniques for Markov Reward Models. *Discrete Event Systems: Theory and Applications*, 3:219–247, 1993.

[159] B. Henderson. Performance analysis: When do we give up on product form solutions. *Computer Networks and ISDN Systems*, 20(1-5):271–275, 1990.

[160] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *J. of the ACM*, 32(1):137–161, 1985.

[161] H. Hermanns. Performance and reliability model checking and model construction. In S. Gnesi, I. Schieferdecker, and A. Rennoch, editors, *5th Int. ERCIM Workshop on Formal Methods for Industrial Critical Systems*, pages 11–27, Berlin, April 2000. GMD Report 91.

[162] H. Hermanns. *Interactive Markov Chains*. PhD thesis, Universität Erlangen-Nürnberg, September 1998. Arbeitsberichte des IMMD No. 32/7.

[163] H. Hermanns, U. Herzog, and J.-P. Katoen. Process algebra for performance evaluation. *Th. Comp. Sci.*, 274(1-2):43–87, 2002.

[164] H. Hermanns, U. Herzog, U. Klehmet, V. Mertsiotakis, and M. Siegle. Compositional Performance Modelling with the TIPPtool. In R. Puigjaner, N. Savino, and B. Serra, editors, *10th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS '98)*, pages 51–62, Palma de Mallorca, September 1998. Springer, LNCS 1469.

[165] H. Hermanns, U. Herzog, U. Klehmet, V. Mertsiotakis, and M. Siegle. Compositional performance modelling with the TIPPtool. *Performance Evaluation*, 39(1-4):5–35, January 2000.

[166] H. Hermanns, U. Herzog, and V. Mertsiotakis. Stochastic Process Algebras as a Tool for Performance and Dependability Modelling. In *Proc. of IEEE International Computer Performance and Dependability Symposium*, pages 102–111, Erlangen, April 1995. IEEE Computer Society Press.

[167] H. Hermanns, U. Herzog, and V. Mertsiotakis. Stochastic Process Algebras - Between LOTOS and Markov Chains. *Computer Networks and ISDN systems (CNIS)*, 30(9-10):901–924, 1998.

[168] H. Hermanns, J.-P. Katoen, J. Meyer-Kayser, and M. Siegle. A Markov Chain Model Checker. In S. Graf and M. Schwartzbach, editors, *TACAS'2000*, pages 347–362, Berlin, 2000. Springer, LNCS 1785.

[169] H. Hermanns, J.-P. Katoen, J. Meyer-Kayser, and M. Siegle. Model checking stochastic process algebra. Technical Report Tech. Rep. IMMD 7–2/00, University of Erlangen-Nürnberg, 2000.

[170] H. Hermanns, J.-P. Katoen, J. Meyer-Kayser, and M. Siegle. Towards model checking stochastic process algebra. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *2nd Int. Conference on Integrated Formal Methods*, pages 420–439, Dagstuhl, November 2000. Springer, LNCS 1945.

[171] H. Hermanns, J.-P. Katoen, J. Meyer-Kayser, and M. Siegle. A Markov Chain Model Checker. In P. Kemper, editor, *MMB-PNPM-PAPM-PROBMIV 2001 Tool Proceedings*, pages 1–6. Universität Dortmund, Informatik IV, Bericht 760/2001, 2001.

[172] H. Hermanns, J.-P. Katoen, J. Meyer-Kayser, and M. Siegle. Implementing a Model Checker for Performability Behaviour. In R. German, J. Luethi, and M. Telek, editors, *Fifth Int. Workshop on Performability Modelling of Computer and Communication Systems (PMCCS5)*, pages 110–115. Universität Erlangen-Nürnberg, Arbeitsberichte des Instituts für Informatik, Band 34 Nummer 13, September 2001, 2001.

[173] H. Hermanns, J.-P. Katoen, J. Meyer-Kayser, and M. Siegle. A tool for model checking Markov chains. *Software Tools for Technology Transfer (STTT)*, 2002. (accepted for publication).

[174] H. Hermanns and J.P. Katoen. Automated compositional Markov chain generation for a plain-old telephony system. *Science of Computer Programming*, 36(1):97–127, December 1999.

[175] H. Hermanns and M. Lohrey. Priority and Maximal Progress are completely axiomatisable. In D. Sangiorgi and R. de Simone, editors, *CONCUR'98 Concurrency Theory*, pages 237–252. Springer, LNCS 1446, 1998.

[176] H. Hermanns, V. Mertsiotakis, and M. Siegle. TIPPtool: Compositional Specification and Analysis of Markovian Performance Models. In N. Halbwachs and D. Peled, editors, *Compter Aided Verification (CAV)*, pages 487–490, Trento, July 1999. Springer, LNCS 1633.

[177] H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi Terminal Binary Decision Diagrams to Represent and Analyse Continuous Time Markov Chains. In B. Plateau, W.J. Stewart, and M. Silva, editors, *3rd Int. Workshop on the Numerical Solution of Markov Chains*, pages 188–207. Prensas Universitarias de Zaragoza, 1999.

[178] H. Hermanns and M. Rettelbach. Markovian Processes go Algebra. Technical Report 10/94, Universität Erlangen-Nürnberg, IMMD 7, Martensstr. 3, 91058 Erlangen, March 1994.

[179] H. Hermanns and M. Rettelbach. Syntax, Semantics, Equivalences, and Axioms for MTIPP. In U. Herzog and M. Rettelbach, editors, *Proc. of the 2nd Workshop on Process Algebras and Performance Modelling*, pages 71–88, Erlangen-Regensberg, July 1994. IMMD, Universität Erlangen-Nürnberg.

[180] H. Hermanns and M. Rettelbach. A Superset of Basic LOTOS for Performance Prediction. In *Proc. of the 4th Workshop on Process Algebras and Performance Modelling*, Torino, July 1996. Dpto. di Informatica, Universita di Torino.

[181] H. Hermanns, M. Rettelbach, and T. Weiß. Formal characterisation of immediate actions in SPA with nondeterministic branching. *The Computer Journal*, 38(7):530–541, December 1995. Special issue: Proc. of the 3rd Workshop on Process Algebras and Performance Modelling.

[182] H. Hermanns and M. Ribaudo. Exploiting Symmetries in Stochastic Process Algebras. In R. Zobel and D. Moeller, editors, *Simulation-Past, Present and Future. 12th European Simulation Multiconference*, pages 763–770. SCS International, June 1998.

[183] H. Hermanns and M. Siegle. Bisimulation Algorithms for Stochastic Process Algebras and their BDD-based Implementation. In J.-P. Katoen, editor, *ARTS'99, 5th Int. AMAST Workshop on Real-Time and Probabilistic Systems*, pages 144–264. Springer, LNCS 1601, 1999.

[184] U. Herzog. Leistungsbewertung und Modellbildung für Parallelrechner. *Informationstechnik (it)*, 31(1):31–38, 1989 (in German).

[185] U. Herzog. Formal Description, Time and Performance Analysis. A Framework. In T. Härder, H. Wedekind, and G. Zimmermann, editors, *Entwurf und Betrieb Verteilter Systeme*, pages 172–190. Springer Verlag, Berlin, IFB 264, 1990.

[186] U. Herzog. Performance Evaluation and Formal Description. In V.A. Monaco and R. Negrini, editors, *Advanced Computer Technology, Reliable Systems and Applications, Proceedings*, pages 750–756. IEEE Comp. Soc. Press, 1991.

[187] U. Herzog. A Concept for Graph-Based Stochastic Process Algebras, Generally Distributed Activity Times and Hierarchical Modelling. In *Proc. of the 4th Workshop on Process Algebras and Performance Modelling*, pages 1–20, Torino, July 1996. Dpto. di Informatica, Universita di Torino.

[188] U. Herzog and W. Hofmann. Synchronization Problems in Hierarchically Organized Multiprozessor Computer Systems. In M. Arato, A. Butrimenko, and E. Gelenbe, editors, *Performance of Computer Systems — Proc. of the 4th Int. Symposium on Modelling and Performance Evaluation of Computer Systems*, Vienna, Austria, February 1979.

[189] U. Herzog, W. Hofmann, and W. Kleinöder. Performance Modeling and Evaluation for Hierarchically Organized Multiprocessor Computer Systems. In *Int. Conf. on Parallel Processing*, Bellaire, USA, August 1979.

[190] U. Herzog and V. Mertsiotakis. Applying Stochastic Process Algebras to Failure Modelling. In U. Herzog and M. Rettelbach, editors, *Proc. of the 2nd Workshop on Process Algebras and Performance Modelling*, pages 107–126, Regensberg/Erlangen, July 1994. Arbeitsberichte des IMMD, Universität Erlangen-Nürnberg.

[191] A. Hett, R. Drechsler, and B. Becker. Fast and Efficient Construction of BDDs by Reordering Based Synthesis. In *IEEE European Design and Test Conference*, 1997.

[192] J. Hillston. *A Compositional Approach to Performance Modelling.* Cambridge University Press, 1996.

[193] J. Hillston and V. Mertsiotakis. A Simple Time Scale Decomposition Technique for Stochastic Process Algebras. *The Computer Journal*, 38(7):566–577, December 1995. Special issue: Proc. of the 3rd Workshop on Process Algebras and Performance Modelling.

[194] Health Level Seven: An Application Protocol for Electronic Data Exchange in Healthcare Environments. Health Level Seven, Inc., Michigan, USA, 1992.

[195] Health Level Seven, Version 2.2. Final Standard, 1994. URL http://www.mcis.duke.edu/standards/HL7/hl7.htm.

[196] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[197] R. Hofmann, R. Klar, B. Mohr, A. Quick, and M. Siegle. Distributed Performance Monitoring: Methods, Tools, and Applications. *IEEE Transactions on Parallel and Distributed Systems*, 5(6):585–598, June 1994.

[198] T. Huynh and L. Tian. On some Equivalence Relations for Probabilistic Processes. *Fundamenta Informaticae*, 17:211–234, 1992.

[199] O.C. Ibe and K.S. Trivedi. Stochastic Petri Net Models of Polling Systems. *IEEE Journal on Selected Areas in Communications*, 8(9):1649–1657, December 1990.

[200] N. Ishiura, H. Sawada, and S. Yajima. Minimization of binary decision diagrams based on exchange of variables. In *Int. Conf. on CAD*, pages 472–475, 1991.

[201] ISO. *LOTOS : A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. ISO, Geneve, 1989.

[202] J.R. Jackson. Queueing systems with phase type service. *Operations Research Quart.*, 5:109–120, 1954.

[203] J.R. Jackson. Networks of Waiting Lines. *Operations Research*, 5:518–521, 1957.

[204] J.R. Jackson. Jobshop-like Queueing Systems. *Management Science*, 10(1):131–142, 1963.

[205] A. Jensen. Markoff Chains as an Aid in the Study of Markoff Processes. *Skand. Aktuarietiedskr.*, 36:87–91, 1953.

[206] T. Kam. *State Minimization of Finite State Machines using Implicit Techniques*. PhD thesis, University of California at Berkeley, 1995.

[207] P. Kanellakis and S. Smolka. CCS Expressions, Finite State Processes, and Three Problems of Equivalence. *Information and Computation*, 86:43–68, 1990.

[208] J. P. Katoen. *Quantitative and Qualitative Extensions of Event Structures*. PhD thesis, Centre for Telematics and Information Technology, Enschede, No. 96-09, 1996.

[209] J.-P. Katoen, M. Kwiatkowska, G. Norman, and D. Parker. Faster and Symbolic CTMC Model Checking. In *Process Algebra and Probabilistic Methods (Proc. of PAPM/PROBMIV'01)*, pages 23–38. Springer, LNCS 2165, 2001.

[210] P. Kelb, D. Dams, and R. Gerth. Practical symbolic model checking of the full mu-calculus using compositional abstractions. Technical Report 95-31, Eindhoven University of Technology, Department of Computer Science, 1995.

[211] C.T. Kelley. *Iterative Methods for Linear and Nonlinear Systems.* Series on Frontiers in Applied Mathematics. SIAM, 1995.

[212] J.G. Kemeny and J.L. Snell. *Finite Markov Chains.* Springer, 1976.

[213] P. Kemper. Reachability analysis based on structured representation. In J. Billington and W. Reisig, editors, *Application and Theory of Petri Nets*, pages 269–288. Springer, LNCS 1091, 1999.

[214] L. Kerber. Scenario-based Performance Evaluation of SDL/MSC-Specified Systems. In R. Dumke, A. Schmietendorf, and A. Scholz, editors, *Performance Engineering*, pages 185–201. Springer, LNCS 2047, 2001.

[215] C. Kim and A.K. Agrawala. Analysis of the Fork–Join Queue. *IEEE Transactions on Computers*, 38(2):250–255, 1989.

[216] R. Klar, P. Dauphin, F. Hartleb, R. Hofmann, B. Mohr, A. Quick, and M. Siegle. *Messung und Modellierung paralleler und verteilter Rechensysteme.* Teubner, Stuttgart, 1995. (in German).

[217] U. Klehmet and V. Mertsiotakis. TIPPtool: Timed Processes and Performability Evaluation — User's Guide, Version 2.3. Technical Report 1/98, Universität Erlangen–Nürnberg, IMMD 7, Martensstrasse 3, D 91058 Erlangen, Januar 1998. Draft.

[218] W. Kleinöder. *Stochastische Bewertung von Aufgabenstrukturen für hierarchische Mehrrechnersysteme.* PhD thesis, Universität Erlangen–Nürnberg, IMMD 7, Arbeitsbericht Bd. 15, Nr. 10, August 1982 (in German).

[219] L. Kleinrock. *Queueing Systems*, volume 1: Theory. John Wiley & Sons, 1975.

[220] A. Klingler. *Über komponentenbasierte Architekturen von medizinischen Informationssytemen.* PhD thesis, Ruprecht-Karls-Universität Heidelberg, Institut für Medizinische Biometrie und Informatik, 2001 (in German).

[221] O. Kluge. Binäre Entscheidungsdiagramme für stochastische Prozeßalgebren. Studienarbeit (student's thesis), Universität Erlangen–Nürnberg, Dezember 1996 (in German).

[222] W. Knaup. A New Iterative Numerical Solution Algorithm for Markovian Queueing Networks. In H. Beilner and F. Bause, editors, *Quantitative Evaluation of Computing and Communication Systems. Combined Proc. of 8th Int.*

Conf. on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'95) and 8th GI/ITG Conference on Measuring, Modelling and Evaluating Computing and Communication Systems (MMB'95), pages 194–208, Heidelberg, 1995. Sringer, LNCS 977.

[223] W. Knottenbelt. Generalized Markovian Analysis of Timed Transition Systems. Master's thesis, University of Cape Town, June 1996.

[224] W.J. Knottenbelt. *Parallel Performance Analysis of Large Markov Models*. PhD thesis, University of London, Imperial College, Dept. of Comupting, 1999.

[225] W.J. Knottenbelt and P.G. Harrison. Distributed Disk-based Solution Techniques for Large Markov Models. In B. Plateau, W.J. Stewart, and M. Silva, editors, *Numerical Solution of Markov Chains (NSMC'99)*, pages 58–75. Prensas Universitarias de Zaragoza, 1999.

[226] W.J. Knottenbelt, M.A. Mestern, P.G. Harrison, and P.S. Kritzinger. Probability, parallelism and the state space exploration problem. In R. Puigjaner, N.N. Savino, and B. Serra, editors, *10th Int. Conf. on Modelling, Techniques and Tools (TOOLS'98)*, pages 165–179, Palma de Mallorca, Spain, 1998. Springer, LNCS 1469.

[227] D. Kozen. Results on the propositional mu-calculus. *Th. Comp. Sc.*, 27:333–354, 1983.

[228] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic Symbolic Model Checker. In P. Kemper, editor, *MMB-PNPM-PAPM-PROBMIV Tool Proceedings*. Universität Dortmund, Informatik IV, Bericht 760/2001, 2001.

[229] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic Symbolic Model Checking with PRISM: A Hybrid Approach. Technical Report CSR-01-10, School of Computer Science, University of Birmingham, October 2001.

[230] M. Kwiatkowska, G. Norman, D. Parker, and R. Segala. Symbolic model checking of concurrent probabilistic systems using MTBDDs and Simplex. Technical Report CSR-99-1, School of Computer Science, University of Birmingham, January 1999.

[231] M. Kwiatkowska, G. Norman, and R. Segala. Automated Verification of a Randomised Distributed Consensus Protocol Using Cadence SMV and PRISM. Technical Report CSR-01-1, School of Computer Science, University of Birmingham, January 2001.

[232] Y.-T. Lai and S. Sastry. Edge-Valued Binary Decision Diagrams for Multi-Level Hierarchical Verification. In *29th Design Automation Conference*, pages 608–613. ACM/IEEE, 1992.

[233] K. Lampka. A stochastic process algebra interface for DNAmaca. Studienarbeit (student's thesis), Universität Erlangen–Nürnberg, 2000.

[234] K. Larsen and A. Skou. Bisimulation through Probabilistic Testing. *Information and Computation*, 94(1):1–28, 1991.

[235] A.A. Lazar and T.G. Robertazzi. Markovian Petri Net Protocols with Product Form Solution. *Performance Evaluation*, 12(1):67–77, 1991.

[236] E. Lazowska, J. Zahorjan, G. Graham, and K. Sevcik. *Quantitative System Performance*. Prentice Hall, 1984.

[237] C.Y. Lee. Representation of switching circuits by binary decision diagrams. *Bell System Technical Journal*, 38:985–999, July 1959.

[238] F. Lemmen and R. Hofmann. Spezifikationsgesteuertes Monitoring an TCP/IP. In D. Baum, N. Müller, and R. Rödler, editors, *MMB'99*, pages 213–220, Trier, September 1999. VDE Verlag.

[239] C. Lindemann. An improved numerical algorithm for calculating steady-state solutions of deterministic and stochastic Petri net models. *Performance Evaluation*, 18(1):79–95, July 1993.

[240] A.D. Malony, V. Mertsiotakis, and A. Quick. Automatic Scalability Analysis of Parallel Programs Based on Modeling Techniques. In G. Kotsis G. Haring, editor, *Proceedings of the 7th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, Vienna, May 1994. Springer Verlag.

[241] A.D. Malony, V. Mertsiotakis, and A. Quick. Stochastic Modeling of Scaled Parallel Programs. In *Proceedings of the International Conference on Parallel and Distributed Systems*, Hsinchu, Taiwan, December 1994. IEEE Press.

[242] R. Mateescu and M. Sighireanu. Efficient on-the-fly model checking for regular alternation-free mu-calculus. In *Proc. Workshop on Form. Meth. for Industrial Critical Sys.*, pages 65–86. GMD/FOKUS, 2000.

[243] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[244] V. Mertsiotakis. Time Scale Decomposition of Stochastic Process Algebra Models. In E. Brinksma and A. Nymeyer, editors, *Proc. of 5th Workshop on Process Algebras and Performance Modelling*. CTIT Technical Report series, No. 97-14, University of Twente, June 1997.

[245] V. Mertsiotakis. *Approximate Analysis Methods for Stochastic Process Algebras*. PhD thesis, Universität Erlangen-Nürnberg, 1998.

[246] V. Mertsiotakis and M. Silva. A Throughput Approximation Algorithm for Decision Free Processes. In *Proc. of the 4th Workshop on Process Algebras and Performance Modelling*, Torino, July 1996. Dpto. di Informatica, Universita di Torino.

[247] V. Mertsiotakis and M. Silva. Throughput Approximation of Decision Free Processes Using Decomposition. In *Proc. of the 7th Int. Workshop on Petri Nets and Performance Models*, pages 174–182, St. Malo, June 1997. IEEE CS-Press.

[248] J.F. Meyer. On Evaluating the Performability of Degradable Computing Systems. *IEEE Transactions on Computer Systems*, C-29(8):720–731, August 1980.

[249] J.F. Meyer. Performability: A retrospective and some pointers to the future. *Performance Evaluation*, 14(3-4):139–156, February 1992.

[250] M. Meyer. Entwurf eines spezialisierten Coprozessors für die Manipulation von Binären Entscheidungsdiagrammen. Students thesis, Universität Erlangen–Nürnberg, January 2001 (in German).

[251] J. Meyer-Kayser. Action-based Model Checking of Markov Chains. Talk at kick-off meeting of NWO-DFG project "Validation of Stochastic Systems (VOSS)", June 29 2001.

[252] R. Milner. *Communication and Concurrency*. Prentice Hall, London, 1989.

[253] A. Miner. Efficient solution of GSPNs using matrix diagrams. In R. German and B. Haverkort, editors, *Petri Nets and Performance models (PNPM'01)*, pages 101–110. IEEE Computer Society Press, 2001.

[254] A. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. In H. Kleijn and S. Donatelli, editors, *Application and Theory of Petri Nets 1999*, pages 6–25, Williamsburg, VA, USA, 1999. Springer, LNCS 1639.

[255] A.S. Miner, G. Ciardo, and S. Donatelli. Using the exact state space of a Markov model to compute approximate stationary measures. *Performance Evaluation Review*, 28(1):207–216, June 2000. Proc. of ACM SIGMETRICS 2000.

[256] A. Mitschele-Thiel. *Engineering Performance-Critical Communicating Systems with SDL*. Wiley, 2001.

[257] A. Mitschele-Thiel and B. Müller-Clostermann. Performance Engineering of SDL/MSC Systems. *Computer Networks*, 31(17):1801–1815, June 1999.

[258] M.K. Molloy. Performance Analysis Using Stochastic Petri Nets. *IEEE Transactions on Computers*, C-31:913–917, September 1982.

[259] R. Nelson and A.N. Tantawi. Approximate Analysis of Fork/Join Synchronization in Parallel Queues. *IEEE Transactions on Computers*, 37(6):739–743, 1988.

[260] K. Neumann. *Stochastic Project Networks – Temporal Analysis, Scheduling and Cost Minimization*. Springer, Lecture Notes in Economics and Mathematical Systems 344, 1990.

[261] D. Nicol and G. Ciardo. Automated parallelization of discrete state-space generation. *Journal of Parallel and Distributed Computing*, 47:153–167, 1997.

[262] R. De Nicola and F.W. Vaandrager. Action versus state based logics for transition systems. In *Semantics of Concurrency*, pages 407–419. Springer, LNCS 469, 1990.

[263] R. De Nicola and F.W. Vaandrager. Three logics for branching bisimulation. *J. of the ACM*, 42:458–487, 1995.

[264] G. Norman. Divide and conquer algorithms for matrices. Private communication, Dec. 1999.

[265] W.D. Obal and W.H. Sanders. State-space support for path-based reward variables. *Performance Evaluation*, 35:233–251, 1999.

[266] R. Paige and R. Tarjan. Three Partition Refinement Algorithms. *SIAM Journal of Computing*, 16(6):973–989, 1987.

[267] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Fifth GI Conference on Theoretical Computer Science*. Springer, LNCS 104, 1981.

[268] D. Parker. A hybrid approach to MTBDD-based vector-matrix multiplication. Private communication, Dec. 2000.

[269] D. Parker. *Implementation of symbolic model checking for probabilistic systems*. PhD thesis, School of Computer Science, University of Birmingham, 2002. (to appear).

[270] H. Perros. *Queueing Networks with Blocking: Exact and Approximate Solutions*. Oxford University Press, 1994.

[271] B. Plateau. On the Synchronization Structure of Parallelism and Synchronization Models for Distributed Algorithms. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 147–154, Austin, TX, August 1985.

[272] B. Plateau and K. Atif. Stochastic Automata Network for Modeling Parallel Systems. *IEEE Transactions on Software Engineering*, 17(10):1093–1108, 1991.

[273] B. Plateau and J.-M. Fourneau. A Methodology for Solving Markov Models of Parallel Systems. *Journal of Parallel and Distributed Computing*, 12:370–387, 1991.

[274] B. Plateau, J.-M. Fourneau, and K.-H. Lee. PEPS: A Package for Solving Complex Markov Models of Parallel Systems. In *Proceedings of the 4th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 341–360, Palma (Mallorca), September 1988.

[275] G. Plotkin. A structural approach to operational semantics. technical report, Computer Science Department FN-19, DAIMI, Aarhus University, September 1981.

[276] A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[277] M. Reiser and S. Lavenberg. Mean Value Analysis of Closed Multichain Queueing Networks. *Journal of the ACM*, 27(2):313–322, 1980.

[278] M. Rettelbach. Probabilistic Branching in Markovian Process Algebras. *The Computer Journal*, 38(7):590–599, December 1995. Special issue: Proc. of the 3rd Workshop on Process Algebras and Performance Modelling.

[279] M. Rettelbach. *Stochastische Prozeßalgebren mit zeitlosen Aktivitäten und probabilistischen Verzweigungen*. PhD thesis, Universität Erlangen–Nürnberg, 1996 (in German).

[280] M. Rettelbach and M. Siegle. Compositional Minimal Semantics for the Stochastic Process Algebra TIPP. In U. Herzog and M. Rettelbach, editors, *Proc. of the 2nd Workshop on Process Algebras and Performance Modelling*, pages 89–106, Regensberg/Erlangen, July 1994. Arbeitsberichte des IMMD, Universität Erlangen-Nürnberg 27 (4).

[281] J.A. Rolia and K.C. Sevcik. The Method of Layers. *IEEE Trans. on Software Engineering*, 21(8):689–700, August 1995.

[282] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. IEEE ICCAD'93*, pages 42–47, 1993.

[283] W.H. Sanders and J.F. Meyer. Reduced Base Model Construction Methods for Stochastic Activity Networks. *IEEE Journal on Selected Areas in Communications*, 9(1):25–36, January 1991.

[284] T. Sasao and M. Fujita, editors. *Representation of Discrete Functions*. Kluwer Academic Publishers, 1996.

[285] C. Sauer and E. McNair. The Evolution of the Research Queueing Package RESQ. In *Modeling Techniques and Tools*, Paris, 1984.

[286] I. Schieferdecker and A. Rennoch. Usage of Timed MSC for Test Purpose Definition. In A. Mitschele-Thiel and B. Müller-Clostermann, editors, *Workshop on Performance and Time in SDL and MSC*, pages 81–89. University of Erlangen-Nürnberg, Interner Bericht IMMD 7-1/98, 1998.

[287] W.G. Schneider. Job Shop Scheduling with Stochastic Job Precedence Constraints. Technical report, University of Karlsruhe, Report WIOR-448, February 1995.

[288] M. Sczittnick. Techniken zur funktionalen und quantitativen Analyse von Markoffschen Rechensystemmodellen. Diplomarbeit, Universität Dortmund, August 1987 (in German).

[289] M. Sereno. Towards a product form solution for stochastic process algebras. *The Computer Journal*, 38(7):622–632, December 1995. Special issue: Proc. of the 3rd Workshop on Process Algebras and Performance Modelling.

[290] M. Sereno and G. Balbo. Mean Value Analysis of Stochastic Petri Nets. *Performance Evaluation*, 29(1):35–62, 1997.

[291] M. Siegle. Structured Markovian Performance Modelling with Automatic Symmetry Exploitation. In G. Haring and H. Wabnig, editors, *Short Papers and Tool Descriptions Proc. of the 7th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 77–81, Vienna, Austria, May 1994.

[292] M. Siegle. Using Structured Modelling for Efficient Performance Prediction of Parallel Systems. In G.R. Joubert, D. Trystram, F.J. Peters, and D.J. Evans, editors, *Parallel Computing: Trends and Applications, Proceedings of the International Conference ParCo93*, pages 453–460. North-Holland, 1994.

[293] M. Siegle. *Beschreibung und Analyse von Markovmodellen mit großem Zustandsraum.* PhD thesis, Universität Erlangen–Nürnberg, 1995 (in German).

[294] M. Siegle. BDD extensions for stochastic transition systems. In D. Kouvatsos, editor, *Proc. of 13th UK Performance Evaluation Workshop*, pages 9/1 – 9/7, Ilkley/West Yorkshire, July 1997.

[295] M. Siegle. Compact representation of large performability models based on extended BDDs. In *Fourth International Workshop on Performability Modeling of Computer and Communication Systems (PMCCS4)*, pages 77–80, Williamsburg, VA, September 1998.

[296] M. Siegle. Technique and tool for symbolic representation and manipulation of stochastic transition systems. TR IMMD 7 2/98, Universität Erlangen-Nürnberg, March 1998.

[297] M. Siegle. Technique and Tool for Symbolic Representation and Manipulation of Stochastic Transition Systems. In *IEEE International Computer Performance and Dependability Symposium (IPDS)*, page 272, Durham, NC, September 1998.

[298] M. Siegle. Compositional Representation and Reduction of Stochasitic Labelled Transition Systems based on Decision Node BDDs. In D. Baum, N. Müller, and R. Rödler, editors, *MMB'99*, pages 173–185, Trier, September 1999. VDE Verlag.

[299] M. Siegle. Advances in model representation. In L. de Alfaro and S. Gilmore, editors, *Process Algebra and Probabilistic Methods, Joint Int. Workshop PAPM-PROBMIV 2001*, pages 1–22. Springer, LNCS 2165, September 2001.

[300] M. Siegle, D. Kraska, M. Simon, and B. Wentz. Analyse des Erlanger Klinikkommunikationssystems mit Hilfe von Leistungsmessungen. In E. Greiser and M. Wischnewsky, editors, *43. Jahrestagung der Deutschen Gesellschaft für Medizinische Informatik, Biometrie und Epidemiologie (GMDS)*, pages CD–ROM C24, Bremen, September 1998 (in German). MMV Medien & Medizin Verlag.

[301] M. Siegle, B. Wentz, A. Klingler, and M. Simon. Neue Ansätze zur Planung von Klinikkommunikationssystemen mittels stochastischer Leistungsmodellierung. In

R. Muche, G. Büchele, D. Harder, and W. Gaus, editors, *42. Jahrestagung der Deutschen Gesellschaft für Medizinische Informatik, Biometrie und Epidemiologie (GMDS)*, pages 188 – 192, Ulm, September 1997 (in German). MMV Medien & Medizin Verlag.

[302] M. Silva and J. Campos. Structural performance analysis of stochastic Petri nets. In *IEEE International Computer Performance and Dependability Symposium (IPDS'95)*, pages p. 61–70. IEEE Computer Society, 1995.

[303] H.A. Simon and A. Ando. Aggregation of Variables in Dynamic Systems. *Econometrica*, 29:111–138, 1961.

[304] A.P. Sistla, M.Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49(2):217–237, 1987.

[305] R. Sisto. A method to build symbolic representations of LOTOS specifications. In *Protocol Specification, Testing and Verification*, pages 323–338, 1995.

[306] R. Sisto. Using binary decision diagrams for representation and analysis of communication protocols. *Computer Networks*, 32:81–98, 2000.

[307] F. Somenzi. CUDD: Colorado University Decision Diagram Package, Release 2.3.0. User's Manual and Programmer's Manual, September 1998.

[308] F. Sötz. A Method for Performance Prediction of Parallel Programs. In H. Burkhart, editor, *CONPAR 90–VAPP IV, Joint International Conference on Vector and Parallel Processing. Proceedings*, pages 98–107, Zürich, Switzerland, September 1990. Springer, LNCS 457.

[309] F. Sötz and G. Werner. Lastmodellierung mit stochastischen Graphen zur Verbesserung paralleler Programme auf Multiprozessoren mit Fallstudie. In *11. ITG/GI–Fachtagung Architektur von Rechensystemen*, pages 231–243. vde–Verlag, Berlin und Offenbach, Januar 1990 (in German).

[310] A. Srinivasan, T. Kam, S. Malik, and R.K. Brayton. Algorithms for discrete function manipulation. In *Int. Conf. on CAD*, pages 92–95. IEEE Computer Society, 1990.

[311] P.H. Starke. *Analyse von Petri-Netz-Modellen*. Teubner, Stuttgart, 1990.

[312] W. Stewart, K. Atif, and B. Plateau. The Numerical Solution of Stochastic Automata Networks. Rapport Apache 6, Institut IMAG, LGI, LMC, Grenoble, November 1993.

[313] W.J. Stewart. MARCA: Markov Chain Analyzer, A Software Package for Markov Modeling. In W.J. Stewart, editor, *Numerical Solution of Markov Chains*. Marcel Dekker, 1991.

[314] W.J. Stewart. *Introduction to the numerical solution of Markov chains*. Princeton University Press, 1994.

[315] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969.

[316] S. Tani, K. Hamaguchi, and S. Yajima. The Complexity of Optimal Variable Ordering of a Shared Binary Decision Diagram. In *Proc. 45th ISAAC*, pages 389–398. Springer, LNCS 762, 1993.

[317] H.J. Touati, H. Savoj, B. Lin, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDDs. In *International Conf. on Computer Aided Design*, 1990.

[318] K.S. Trivedi and M. Malhotra. Reliability and Performability Techniques and Tools: A Survey. In B. Walke and O. Spaniol, editors, *Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen*, pages 27–48, Aachen, September 1993. Springer.

[319] M. Veran and D. Potier. QNAP2: A Portable Environment for Queueing Systems Modelling. In *Proceedings of the First International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, Paris, May 1984.

[320] H. Wabnig, G. Kotsis, and G. Haring. Performance Prediction of Parallel Programs. In B. Walke and O. Spaniol, editors, *Proceedings der 7. GI–ITG Fachtagung "Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen", Aachen, 21.–23. Spetember 1993*, pages 64–76. Informatik Aktuell, Springer, 1993.

[321] C. Wagner. Analyse kritischer Pfade in Geschäftsprozessen. Diplomarbeit, Universität Erlangen–Nürnberg, Informatik 7, February 1996 (in German).

[322] V.L. Wallace and R.S. Rosenberg. Markovian models and numerical analysis of computer systems behaviour. In *Proc. of the ACM-IEEE Comp. Soc. Spring Joint Comp. Conf.*, pages 141–148, 1966.

[323] V.L. Wallace and R.S. Rosenberg. RQA–1, The recursive queue analyzer. Technical report, University of Michigan, Ann Arbor, 1966.

[324] M. Woodside, J.E. Neilson, D.C. Petriu, and S. Majumdar. The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software. *IEEE Trans. on Computers*, 44(1):20–34, January 1995.

Index