Symbolic Semantics and Verification of Stochastic Process Algebras

Symbolische Semantik und Verifikation stochastischer Prozessalgebren

Der Technischen Fakultät der Universität Erlangen-Nürnberg
zur Erlangung des Grades

DOKTOR-INGENIEUR

vorgelegt von

Georg Wolfgang Matthias Kuntz

Erlangen - 2006
Als Dissertation genehmigt von der Technischen Fakultät der Universität Erlangen-Nürnberg

Tag der Einreichung: 14.10.2005
Tag der Promotion: 02.02.2006
Dekan: Prof. Dr.-Ing. Alfred Leipertz
Berichterstatter: Prof. Dr.-Ing. Markus Siegle
Prof. Dr.-Ing. Reinhard German
In the realm of performance and reliability analysis, high-level specification languages like stochastic process algebras (SPA) or generalised stochastic Petri nets (GSPN) have turned out to be extremely useful. During the analysis of systems, being specified with either of the two methods, two main problems can be identified:

1. **State space explosion**: This problem occurs on generation and storage of the semantic models of complex systems on which analysis is carried out. This semantic model is normally some kind of a continuous time Markov chain (CTMC) or stochastic labelled transition system (SLTS).

2. **The specification and automatic verification of complex system requirements**

The state space explosion problem arises mainly in the context of the analysis of concurrent systems. This is the type of systems, for which we want to devise analysis methods in this thesis. If the system is specified using SPAs, it is composed of many parallel processes. The overall state space of such a model is then the Cartesian product of the state spaces of the respective subprocesses. Thus, the memory requirements for storing the SLTSs of complex systems are often prohibitive. To combat this problem several steps must be taken. At first, data structures that allow the compact storage of even huge state spaces must be applied. Multi-terminal binary decision diagrams (MTBDD) turned out to be advantageous in the storage of large SLTSs. Secondly, upon generation of the SLTSs, the fact that SPA specifications are normally composed of many smaller subprocesses has to be taken into account. The traditional semantics of SPAs supports this only in a very limited way. These semantics only support a monolithic way of state space generation which turned out to have exponential memory requirements. To avoid this, we devise a new semantics that utilises the structure of a given SPA specification upon generation of its underlying SLTS. For every subprocess of a given specification we derive the MTBDD based representation of its corresponding SLTS. According to the syntactic structure of the specification at hand, the overall MTBDD representation can be derived from the subprocesses’ MTBDDs. It can be shown that the memory requirement for an semantic approach that utilises the compositional nature of process algebras is only linear, opposed to an exponential memory requirement in the case of traditional SPA semantics.

In the second part of this thesis we address the problem of automatically verifying performance and reliability requirements. Here, three subproblems can be identified:

1. **Complexity of requirements**
2. **Abstraction**
VIII Summary

3. Efficient numerical analysis

Usually, in traditional performance and reliability analysis only quite simple requirements are expressible. For example:

- Mean number of jobs in a system,
- the probability, that a system is operational over a period of time,
- the mean time to failure, mean time to repair.

Because of this in the recent years, efforts have been made to transfer the means that are provided by functional model checking to the stochastic world (model checking of stochastic systems). We will further refine this approach in this thesis, which can be best explained in context with the problem of abstraction. The problem of abstraction stems from the fact that both in traditional performance and reliability analysis and model checking of stochastic systems, the requirements have to be formulated with the semantic model in mind. To define useful requirements detailed knowledge about the structure of the underlying state space is required. This contradicts both the usage of high-level specification languages in general and the usage of SPAs in particular. In general, the application of high-level methods shall free the user from the need to have knowledge about low-level details, such as the semantic model. In particular the usage of SPAs as specification language makes it possible to think of the system behaviour as a sequence of actions, states do not play a role in this context. Thus, referring to states here, is not only a violation of the principle of abstraction but also means a disturbing shift of paradigms from an action- to a state oriented view of the modelled system. To avoid these problems, i.e. violation of the abstraction principle and the shift of paradigms, we develop a new stochastic logic in this thesis. This logic is a stochastic extension of the logic PDL (propositional dynamic logic), called SPDL. With SPDL it is possible to express complex performance and reliability requirements and to automatically verify them. The logic SPDL allows the specification of requirements by means of extended regular expressions, thereby conserving an abstract and action-oriented view on the system’s behaviour. The extensions of regular expressions SPDL provides, make it possible to apply programming language constructs such as if-then-else and while. This allows the specification and verification of very complex requirements.

To compute performance and reliability requirements the data structures that are applied to represent the system’s SLTS must allow efficient numerical computations. To this end, we apply the MTBDD based numerical algorithms that are described in [121].

In the tool CASPA we have implemented both the novel semantics for SPA and the model checking algorithms for SPDL and demonstrate their efficiency by a number of case studies.
# Contents

## Part I Introduction and Foundations

1 **Introduction** .................................................. 1  
  1.1 Motivation ................................................... 1  
    1.1.1 Choice of Specification Language .................. 3  
    1.1.2 Modelling of Complex Systems .................... 4  
    1.1.3 Specification of Qualitative and Quantitative Measures .. 6  
  1.2 Contributions of the Thesis .................................. 9  
    1.2.1 Compositional Symbolic Semantics .................. 10  
    1.2.2 Automatic Verification of Stochastic Systems ...... 11  
  1.3 Related Work .................................................. 12  
    1.3.1 Symbolic Semantics for Process Algebras .......... 13  
    1.3.2 Temporal Logics ........................................ 13  
  1.4 Outline ....................................................... 14  

2 **Foundations** .................................................. 17  
  2.1 Continuous-Time Markov Chains .............................. 17  
    2.1.1 Transient Analysis .................................. 19  
    2.1.2 The Uniformisation Method .......................... 19  
    2.1.3 Steady State Analysis ................................ 21  
  2.2 Boolean and Pseudo-Boolean Functions ...................... 22  
    2.2.1 Boolean Algebra ..................................... 22  
    2.2.2 Boolean Formulae .................................... 24  
    2.2.3 Switching Formulae ................................... 24  
  2.3 Binary Decision Diagrams .................................... 25  
    2.3.1 Motivation .............................................. 25  
    2.3.2 Representation of Boolean Formulae ................. 25  
    2.3.3 Representation of Boolean Formulae as Graphs - BDDs .. 27  
    2.3.4 Basic BDD Algorithms ................................ 33  
    2.3.5 Extending BDDs to Represent Pseudo-Boolean Formulae .. 39  
    2.3.6 Representing Transition Systems by Binary Decision Diagrams .. 40  
  2.4 Process Algebras ............................................... 43  
    2.4.1 Syntax, Semantics, and Bisimulation for LOTOS ....... 44  
    2.4.2 Syntax, Semantics, and Equivalence for YAMPA ..... 49  
  2.5 The Logic PDL ................................................ 52
**Contents**

2.5.1 Syntax of PDL ................................................. 52
2.5.2 Semantics of PDL ........................................... 53
2.6 The Logic CTL .................................................. 55
  2.6.1 Introduction ............................................... 55
  2.6.2 Syntax and Semantics ..................................... 55
2.7 The Logic CSL ................................................ 56
  2.7.1 State-Labelled Continuous-Time Markov Chains .......... 56
  2.7.2 Syntax and Semantics of CSL ............................. 58
  2.7.3 Model Checking CSL ....................................... 60
  2.7.4 Markov-AP Bisimulation ................................. 62

**Part II  Semantics and Verification of Stochastic Process Algebras**

3 Symbolic Semantics for Stochastic Process Algebras ................. 65
  3.1 Introduction .................................................. 65
  3.2 Symbolic Denotational Semantics for YAMPA .................. 66
    3.2.1 General Idea ............................................ 67
    3.2.2 Semantic Rules .......................................... 68
    3.2.3 Parallel Composition at Arbitrary Levels ............ 73
    3.2.4 Example ................................................. 75
  3.3 Correctness of Semantics ................................... 77
    3.3.1 Outline of the Proof .................................... 77
    3.3.2 Construction of the Transition System ............... 78
    3.3.3 Correctness Proof of the Semantics ................. 79
  3.4 Towards a Minimal Symbolic Semantics for YAMPA ............... 79

4 Stochastic Propositional Dynamic Logic ............................... 83
  4.1 Introduction .................................................. 83
  4.2 General Idea ................................................ 84
  4.3 Stochastic PDL: Syntax, Semantics and Automata ............... 85
    4.3.1 Action- and State-Labelled Continuous-Time-Markov Chains .. 85
    4.3.2 Syntax of SPDL .......................................... 86
    4.3.3 Programs and Automata .................................. 87
    4.3.4 Semantics of SPDL ...................................... 90
    4.3.5 Derived Operators ...................................... 92
  4.4 Automata Construction ........................................ 92
    4.4.1 Building the NPA $N_\rho$ from $N'_\rho$ ................ 93
    4.4.2 Determinisation of $N_\rho$ .............................. 94
    4.4.3 On the Relation between NPA and DPA for SPDL Programs ... 98
  4.5 Model Checking SPDL .......................................... 99
    4.5.1 Model Checking Probabilistic Path Formulae by Solving Integral Equations ......................... 100
    4.5.2 Model Checking Probabilistic Path Formulae by Transient Analysis .................. 101
    4.5.3 Correctness of Model Transformation ................... 104
  4.6 Example: System Model and Requirements ........................ 104
    4.6.1 The System Model ....................................... 104
    4.6.2 Performance Requirements .............................. 105
    4.6.3 Building the Product Markov Chain .................... 105
### Part III Implementation and Case Studies

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>CASPA: A Symbolic Performance Evaluation and Stochastic Model Checking Tool</td>
<td>139</td>
</tr>
<tr>
<td></td>
<td>Tool Architecture</td>
<td>139</td>
</tr>
<tr>
<td>6.2</td>
<td>CASPA's Specification Language</td>
<td>140</td>
</tr>
<tr>
<td></td>
<td>The System Specification Language</td>
<td>141</td>
</tr>
<tr>
<td>6.3</td>
<td>Requirement Specification: Definition and Data Structures</td>
<td>142</td>
</tr>
<tr>
<td></td>
<td>State Measures</td>
<td>142</td>
</tr>
<tr>
<td></td>
<td>Mean Values</td>
<td>143</td>
</tr>
<tr>
<td></td>
<td>Throughput Measures</td>
<td>144</td>
</tr>
<tr>
<td></td>
<td>SPDL Formulae</td>
<td>144</td>
</tr>
<tr>
<td>6.4</td>
<td>Performance Requirements and Process Instances</td>
<td>145</td>
</tr>
<tr>
<td>6.5</td>
<td>The State Space Manager</td>
<td>145</td>
</tr>
<tr>
<td>6.6</td>
<td>Symbolic Product Markov Chain Generation for SPDL Path Formulae</td>
<td>148</td>
</tr>
<tr>
<td>6.7</td>
<td>Numerical Analysis</td>
<td>150</td>
</tr>
<tr>
<td></td>
<td>Methods of Numerical Analysis</td>
<td>150</td>
</tr>
<tr>
<td></td>
<td>Hybrid Matrix-Vector Multiplication</td>
<td>151</td>
</tr>
<tr>
<td></td>
<td>Offset-labelled MTBDDs</td>
<td>151</td>
</tr>
<tr>
<td></td>
<td>Hybrid MTBDDs</td>
<td>152</td>
</tr>
<tr>
<td></td>
<td>Pseudo Gauss-Seidel</td>
<td>152</td>
</tr>
<tr>
<td>6.7.6</td>
<td>Integration in CASPA</td>
<td>152</td>
</tr>
</tbody>
</table>
## Case Studies

### 7.1 Introduction
- Model Statistics and Construction Times
- Steady State Computation
- Model Checking Probabilistic SPDL State Formulae

### 7.2 M/M/1/N-Queue
- Introduction
- Model Statistics
- Model Construction Times
- Steady State Probability Computation

### 7.3 Buffer with Erroneous Arrivals
- Model Statistics
- Model Checking of Probabilistic SPDL State Formulae

### 7.4 Tandem Queueing System with Blocking
- Introduction
- Model Statistics
- Model Construction Time
- Comparison with PRISM
- Steady State Probability Computation
- Model Checking of Probabilistic SPDL State Formulae

### 7.5 Polling System
- Introduction
- Model Statistics
- Model Construction Times
- Comparison with PRISM
- Steady State Probability Computation
- Model Checking of Probabilistic SPDL State Formulae

### 7.6 Kanban System
- Introduction
- Model Statistics
- Model Construction Times
- Comparison with PRISM
- Steady State Probability Computation
- Model Checking Probabilistic SPDL State Formulae

### 7.7 Flexible Manufacturing System
- Introduction
- Model Statistics
- Model Construction Times
- Comparison with PRISM
- Steady State Probability Computation

### 7.8 Mainframe with Software Errors
- Introduction
- Model Statistics
- Model Construction Times
- Steady State Probability Computation

### 7.9 Fault Tolerant Multi Computer System (FTMCS)
- Introduction
- Model Statistics
- Model Construction Times
- Transient Probability Computation
Part I

Introduction and Foundations
Introduction

1.1 Motivation

It is commonplace that distributed, concurrent hard- and software systems have become part of our daily life. Because of our high dependency on these systems, it becomes more and more important that we can rely on the functioning of these systems. We will distinguish three principle expectations one can have towards the functionality of a system:

1. Correctness: We normally expect that a system always does the “right” things. To put this expectation on a more concrete base, we consider the plain old telephone system (POTS). If we call a number, we expect that finally a connection will be established (in general, such a property is referred to as a liveliness property). Further, we expect that we are connected to the called number, i.e. if we call telephone number \( X \), it is not desired to be routed to number \( Y \). (This is a representative of so-called safety properties). It is also desirable that any input of the user leads to a reaction of the system. (Such a property is referred to as deadlock-freeness). In the sequel, we will refer to properties that describe the correct functionality of a system as functional or qualitative requirements.

2. Performance: In many areas it is of great importance that a system does not only work correctly in the sense of functional correctness, but that it also respects time constraints, i.e. a system has to react timely to inputs. Such systems are referred to as real-time systems.

We can distinguish between real-time systems where a guarantee for the compliance of a strict time bound has to be given (hard real-time) and such systems, where only a probability can be given that time bounds are respected (weak real-time).

Hard real-time systems typically interact with hardware, e.g. in embedded systems. For example, a car engine control system or a car brake control system has to be a hard real-time system, as delayed signals may cause damages. Other examples of hard real-time embedded systems are medical systems and industrial process controllers.

Soft real-time systems are typically used, where multiple, concurrent access to restricted resources is an issue, examples of such systems are computer networks such as the Internet, video streaming systems, etc. These systems have in common that delayed data may cause loss of quality but no real damages. In this thesis, we are interested only in soft-real time systems. Typical questions that arise in connection with this type of systems are:

- In the mean, how long has a customer to wait, until a request is completely processed?
2 1 Introduction

- What is the mean number of jobs, clients, etc. in a system, what is their sojourn time?
- What is the utilisation of a server, of channels, etc.?

3. Reliability: In an environment, where the functionality is critical for the safety of the environment, e.g. flight controllers, electronic process controllers in industry or power plants, we have to assess that the system is operational over a certain period of time. For example, for disaster-tolerant computer systems there must be a 99.99999% chance that they are operational, i.e. their down-time must be less than three seconds per year [134].

The second and third aspect can be referred to as \textit{quantitative} requirements.

To assess, or to validate that a system satisfies both qualitative and quantitative requirements, it is possible to proceed in different ways:

- **Testing:** Roughly, testing requires an implemented system, for which the correct functioning is checked. The advantage of testing is that the results are realistic. The disadvantages of testing are, that it can be performed only if the system is already implemented, at least as a prototype, and a negative result, i.e. the system does not behave as required may lead to costly redesigns. Furthermore, it cannot be guaranteed that a system is error-free if the testing procedure did not yield an error, since testing can never be exhaustive.

- **Simulation:** For simulation, we need a model of the system that is to be analysed. This model is an abstract, somehow simplified description of the real system’s behaviour. On this model experiments are conducted to gain insight into the system’s behaviour. The data for the experiments may be drawn from similar, existing systems or the data itself may be models of real data. The advantage of simulation lies in the fact that it can be easily applied at all stages of system design. The disadvantages of simulation are: The results of the simulation are only as good as the model, if the model is either wrong or not detailed enough, the significance of the results is limited. Furthermore, like in testing, error-freeness cannot be guaranteed, as rare events, like system failure, may not be covered by the experiments.

- **Formal Verification:** In formal verification it is possible to prove by using mathematical-logical methods that a given system specification is correct. To do formal verification, we need the following things:
  - a specification language, to model the system that is to be analysed (system specification),
  - a formal language, to describe the desired properties (requirement specification),
  - a verification method, to check whether the system specification satisfies the requirement specification.

The existing methods of formal verification can roughly be classified as follows:

- **proof-based verification:** Here, the system \( \Gamma \) is described as a set of logical formulae \( \{ \Gamma_1, \ldots, \Gamma_n \} \), the properties we want to verify are also logical formulae \( \Psi_j \). By means of a logical calculus we have to prove that the behaviour as described by \( \Psi_j \) is possible in the system as described by \( \Gamma \), more formally we have to prove that \( \Psi_j \) is a (logical) consequence of \( \Gamma \).

- **model-based verification:** The system that is to be verified is described by a finite model \( M \), for now we can think of a model as an entity that possesses states, transitions between states, action names attached to transitions, and logical properties that are attached to states. Such an entity is often referred to as a \textit{labelled transition system} (LTS). The property \( \Psi \) that is to be verified has to be described by means of
a logic. Then we have to check whether $M$ satisfies $\Psi$ or not. As $M$ is finite, this check can be done automatically.

In this thesis we exclusively concentrate on the model-based verification approach. The disadvantages of formal verification with respect to the model are the same as for simulation. The advantage of formal verification lies in the fact that we can rely on its results, i.e. if the system is reported error-free with respect to a particular property, we can rely that this error does not occur (in the model).

Both testing and simulation can be applied to assess a system's performance and reliability, but similar to functional verification there exist more formal methods to do so:

- **Analytical Performance Evaluation**: Using this approach, a system is modelled, for example by means of queues and queueing networks. Complex systems can be modelled by a number of queues that are interconnected appropriately. The performance measures of interest can be calculated exactly.

- **Numerical Performance Evaluation**: In this case, the analysis is done on the base of Markov or semi-Markov chains. Using this Markov chain, numerical algorithms for the solution of large linear or differential equation systems can be applied to compute the performance and reliability measures of interest.

In this thesis we advocate an approach that combines model-based formal verification and numerical stochastic performance analysis in a single approach. The reasons for such a unified approach are manifold:

- With respect to decreasing time-to-market and life-cycles, it is not realistic to assume that both a model for qualitative and another model for quantitative analysis can be developed.

- Using two different models for functional and quantitative verification bears the problem to make them consistent among each other.

- Using different formalisms for qualitative and quantitative analysis requires expertise in both areas.

Pursuing the approach of combined functional and quantitative model-based system verification leads to the following important problems:

1. Choice of an appropriate specification language
2. Modelling of complex systems
3. Specification and verification of complex qualitative and quantitative system requirements

These three problems will be addressed in the next three subsections.

### 1.1.1 Choice of Specification Language

As it is impossible to specify large and complex systems using the low-level model on which analysis is carried out, we have to choose an appropriate modelling language. This language has to provide means to allow the combined qualitative and quantitative system analysis.

For performance analysis we need means to express timing information within the system specification. In the seventies, Petri nets and process algebras have been extended with deterministic real time [131, 140, 11]. In the recent years industrial standard specification languages like UML [26, 143] and SDL [44, 45, 142] offer means to model deterministic
real time behaviour [46, 116, 115]. For UML no commonly accepted standard for real-time modelling exists, for SDL a standardisation is currently on its way.

In the eighties, Petri nets were extended with stochastic times [1, 114, 117], the so called stochastic Petri nets. In the nineties, stochastic process algebras, i.e. process algebras with stochastic timing were introduced. The three most prominent representatives of this class of timed modelling formalisms are TIPP [61], PEPA [76], and EMPA [22, 20]. In the QUEST project [103] SDL was extended with means to express stochastic timing. Very recently for statecharts, as part of the UML, extensions to stochastic timing have been proposed [82]. The OMG profiles for schedulability, performance and time [63] and for modelling quality of service and fault tolerance [64] also address stochastic behaviour.

It is beyond the scope of this thesis to discuss the advantages and disadvantages of the various formalisms. Instead of this, we only justify our choice of stochastic process algebras as specification language used in this thesis. In our opinion stochastic process algebras have the following advantages:

- **Structured design**: Composition operators make it possible to build large systems systematically out of smaller ones.
- **Abstraction**: In the higher-level model details of its submodels can be hidden. This allows the construction of very detailed models on the lower level, without knowing the details of their behaviour at higher levels.
- **Mathematical Calculus**: Like in ordinary algebra, in process algebras a mathematical calculus allows the manipulation, simplification etc. of process algebraic terms. For example, we can replace subterms of a given term by equivalent ones and the entire transformed term is equivalent to the original term.

These three aspects can be summarised by the notion of **constructivity**. Another advantage of process algebras, which it has in common with Petri nets, is that process algebras have a rigorous formal semantics.

In the sequel we will discuss the consequences of choosing stochastic process algebras as modelling formalism with respect to the remaining two problems we have identified.

### 1.1.2 Modelling of Complex Systems

Using process algebras (both functional and stochastic) as modelling language means that the analysis of qualitative and quantitative requirements is done on the level of a labelled transition system (LTS). LTSs can be considered the standard semantic model of process algebras. From a given process algebra specification its underlying semantic model can be derived automatically by applying a set of semantic rules (cf. section 2.4).

In case of stochastic process algebras the semantic model is basically a kind of a state and transition-labelled continuous time Markov chain (cf. section 2.1), which is commonly referred to as stochastic LTS (SLTS). An SLTS is an LTS where the transition labels consist of both an action name and a rate, at which a state is left. SLTSs as semantic model for stochastic process algebras can be used both for functional and quantitative analysis.

On the one hand, analysis based on (S)LTSs is quite simple and flexible, on the other hand, the semantic models tend to become very large, which is in fact a major problem:

1. If the verification results shall be reliable, a model whose behaviour is close to that of the real system is needed. Such models normally have to be detailed, such that the (S)LTSs are very large.
2. The systems that are to be analysed are typically concurrent and distributed systems, i.e. systems that are composed of many smaller (sub-)systems that can interact with each other and work concurrently. The (S)LTS of the overall system is composed of the (S)LTSs of its constituent subsystems. Thus, the number of states of the large overall (S)LTS is the product of the number of states of the (S)LTSs of the subsystems. In the worst case we face an exponential growth of the overall state space. For example let us consider a concurrent system consisting of only three components. Let us assume each component has an LTS of only 100 states, then the overall system’s LTS consists of $100^3$, i.e. 1,000,000 states, if all of these states are reachable.

This central problem of state space based analysis became notorious as the so-called State Space Explosion Problem: The state space explosion problem is well-studied and a number of approaches have been suggested to combat this problem.

The mathematical calculus provided by process algebras, can be applied to combat the state space explosion problem. The transformations such a calculus allows can be used to replace subsystems of a larger system by equivalent ones that have a smaller semantic model. The disadvantage of this approach is that it is both specific to the modelling formalism at hand and that it can be computationally hard to derive a smaller system from a given specification.

But there exist other approaches that avoid these problems and that do not depend on a specific high-level specification formalism. These methods are applicable on the semantic level. The existing methods of this class can roughly be divided into two sub-classes: Approaches that generate the entire state space and methods that only generate portions of the state space

1. The class of approaches that generate and store the entire state space, can be divided into approaches that utilise mass storage and/or distributed hardware, e.g. [68, 86, 106] and those that represent the state space in an efficient and compact way, e.g. [33, 74, 130].
2. The methods in this second class are the so-called partial analysis methods. In contrast to the methods in the first class, only fractions of the state space are generated.

The two most prominent representatives of this class are partial order and probabilistic methods. Partial order techniques exploit the fact that the major contribution to the state space explosion stems from the huge number of possible interleavings of execution traces of concurrent processes. It can be observed that not all interleavings are relevant for system analysis, because some traces are equivalent. Generally spoken, partial order methods aim to prune away such interleavings. This can be achieved for example, by defining an equivalence relation on system behaviour [60].

Using probabilistic methods, it is possible that only a fraction of all reachable states of a system being modelled is stored and analysed, thus the probability that states are omitted is greater than zero. For a comprehensive overview on probabilistic methods see our paper [90].

It should be noted that in contrast to probabilistic methods partial order methods still guarantee full coverage of the state space to be analysed.

In the realm of quantitative analysis, a further problem arises: The chosen technique to combat the state space explosion problem must not only allow the storage and functional analysis of huge state spaces but must also allow efficient numerical analysis [37, 35, 121].

In this thesis we concentrate on the approach, where the state space is stored in a compact way. To this end, we employ a special type of data structures, the so called binary decision diagrams (BDDs) and extensions thereof that are suited for the needs of both
qualitative and quantitative analysis, these extensions are referred to as multi-terminal BDDs (MTBDDs). The representation of state spaces by BDDs and its variant is also called symbolic representation.

### 1.1.3 Specification of Qualitative and Quantitative Measures

Here, we distinguish between techniques to specify and verify functional, i.e. qualitative and performance and reliability, i.e. quantitative requirements.

**Functional Requirements:** Besides the possibility to check very simple properties of a system like the absence of deadlocks, there exist powerful means to specify and verify complex requirements, using a formal language for the specification of requirements and efficient techniques for their verification. The technique for the verification of functional requirements we want to cover in greater detail is **model checking**.

Model checking is an automatic verification method that checks the validity of functional requirements with respect to a given model, i.e. the semantic model of a system’s specification. Model checking is applicable to all software and hardware systems that have finite state-transition systems as their underlying semantic model.

In principle, model checking works as shown in figure 1.1. The model checker requires two inputs:

1. A model of the system that describes its possible behaviour
2. A description of the requirements that shall be satisfied by the system, i.e. a description of the system’s desired behaviour.

After the model checker was run, we obtain as result whether the requirement is satisfied or not. In case of a negative result, we also obtain a counter example that provides additional information about how and where in the system the property is violated. Counter examples are a help in the debugging procedure. Model checking is an iterative process, in case of errors, i.e. a property violation, the model can be corrected until the requirement is satisfied by the modified model. If no errors occur the system description can be refined, and the verification process can be continued and so on.

The model checking procedure is basically an exhaustive state space search, i.e. for each state of the model it is checked whether this state satisfies the requirement that is to be verified. The model has to be a finite state-transition system that is either provided directly by the user or system modeller, or that can be uniquely derived by applying the rules of a formal semantics from some high level specification, like Petri nets or process algebra.

The second input, i.e. the specification of the requirements is given in the formal language of a temporal logic. With temporal logics it is possible to express two classes of properties that are especially important in the realm of verification of distributed, reactive systems:

- Safety properties comprise requirements of the kind:
  \[ \text{Now, and in any future state or situation property } p \text{ holds, or equivalently its negation } \neg p \text{ will never be valid.} \]

- Liveness properties comprise requirements of the kind:
  \[ \text{At some point in the future, property } p \text{ will be satisfied.} \]

Temporal logics are composed of the following three main components, that make it possible to reason about safety and liveness properties:

- Propositional logic, i.e. atomic propositions, disjunction, and negation, to describe properties of states.
- Temporal operators describe paths of a given LTS, by specifying the order in which state properties have to hold along a path of the LTS. For example we can express that a property shall hold in the next state with respect to a given path and initial state, or it has to hold in all states of a path.
- Path quantifiers are used to express that a certain property shall hold on all paths (universal path operator) or at least at one path (existential path operator).

There exists a number of temporal logics, of which LTL (linear temporal logic) [4] and CTL [40] (computation tree logic) are the two most prominent representatives. For these two logics, many model checking tools exist, of which we only mention two, SPIN [78] for LTL-based model checking and SMV [105] for CTL-based model checking. It is worth mentioning that the automatic verification of functional requirements has become industrial standard, and many large companies have incorporated model checking into their design process.

**Quantitative Requirements**: For numerical performance evaluation we can identify the following approaches to the validation of requirements:

2. Derivation of Markov reward models from the high-level specification, such as stochastic process algebras or stochastic Petri nets.
3. Extending and adapting the technique of functional model checking to model checking of stochastic system.

All these methods have in common that the specification of measures is mainly state oriented (cf. figure 1.2). No advantage is taken from the fact, that using process algebras, the system’s behaviour can be easily described by sequences of actions instead of sequences of states.

As an example, TIPP is a representative of the first class of performance measure specification methods:

- In TIPP three different types of measures can be specified:
  - State measures: Such measures define the probability that the system is in a specific set of states. An example of such a measure is the probability that in a queue, \( n \) jobs are waiting for service.
  - Mean value measures: With this type of measure, the mean value of certain parameters of a process can be computed. An example of this kind of measure is the mean number of jobs in a queue.
  - Throughput measures: With this kind of measure, the mean number of times a certain action can be performed per time unit can be computed. An example of such a measure is the mean number of jobs served in a certain amount of time.

TIPP allows the computation of both transient and stationary state and mean value measures.

The process algebra EMPA, as realised in the tool TwoTowers [21], allows the specification of transient and stationary measures like utilisation and throughput by using state and impulse (transition) rewards. The reward measures are defined by specifying actions that are of interest and associating to them state or transition rewards. The respective rewards
1.2 Contributions of the Thesis

In section 1.1 we have identified three major problems that arise in the context of formal modelling and verification of distributed systems. This thesis aims to contribute to the solution of the second and third problem in the following ways:

1. We propose a new semantics for stochastic process algebras that exploits the compositional nature of process algebras, thereby gaining a linear growth of the memory required for the storage of the state space instead of an exponential one. By this semantics, the specification is directly mapped to a compact, BDD-based representation of its underlying semantic model.

2. We extend the existing possibilities of specifying performance and reliability measures by means of stochastic temporal logics, by devising a new logic and its corresponding model checking algorithms. The new logic allows the specification of qualitative and quantitative measures on the behaviour-oriented level.
1.2.1 Compositional Symbolic Semantics

In figure 1.4 we find a pictorial overview of our approach to semantics for stochastic process algebras. Process algebras are a compositional modelling language, i.e. it is possible...
to construct large systems out of smaller ones by using a rich set of composition operators, such as parallel composition, choice, sequential composition, etc.

Applying the usual structural operational semantics [122] (SOS) to derive the model of the system description leads to the loss of the information about how a system is composed, i.e. we have a flat semantic model. This non-compositional (monolithic) approach of deriving the semantic model imposes big problems, especially if the overall system is composed of many interacting parallel processes. The size of the model of such processes is the product of the sizes of the state spaces of the respective subprocesses. In the monolithic case, the storage of the state space requires memory that is in the worst case exponential in the number of processes, i.e. the size of systems that can be modelled and analysed is very restricted.

Our new semantics avoids this drawback by having the following features:

- Our semantics is symbolic, i.e. it directly generates the MTBDD based representation of the semantic model out of the system description.
- Our semantics exploits the compositional nature of process algebras during the derivation of the semantic model of the system description at hand. We define semantic rules for each composition operator that define how to construct the semantic model of the overall process out of the models of the operands of the current operator. This is especially useful in case of parallel composition as it can be shown that such a compositional approach leads only to a linear growth of the overall memory requirements [130], as opposed to the exponential growth in the monolithic case.

1.2.2 Automatic Verification of Stochastic Systems

We want to apply the methods of automatic verification of qualitative properties to the stochastic context, thereby having the possibility to express the measures of interest on the abstract action-/behaviour-oriented level of process algebras. To this purpose we devise both a new logic, which we call SPDL (stochastic propositional dynamic logic) to express complex requirements a stochastic system has to satisfy, and verification algorithms to automatically check the validity of these requirements. SPDL is the stochastic extension of the modal logic PDL (propositional dynamic logic) [53].

It should be noted that the applicability of the logic SPDL and its corresponding verification algorithms is not restricted to stochastic process algebras. In fact, it is applicable to all stochastic modelling formalisms, whose semantic model is a continuous time Markov chain and whose behaviour is described by sequences of actions. For example, generalised stochastic Petri nets are such a formalism. Here, sequences of actions can be interpreted as firing sequences of transitions.

SPDL is capable of expressing weak real-time properties, the satisfaction of such properties also depends on the time at which a satisfying state is reached. SPDL provides operators which define the probability, transient or stationary, with which a property shall be fulfilled. Examples of such probabilistic measures are:

- Transient requirement:
  After the reception of a data packet the receiver will send within 10 time units an acknowledge message with probability 0.8.

- Steady state requirement:
  In equilibrium, the system will work correctly with probability 0.99

In figure 1.5 the approach to the verification of stochastic systems we propose in this thesis can be found. We can see that our approach extends the other approaches by al-
Fig. 1.5. Automatic verification of performance measures

allowing the specification of performance and reliability measures not only on the state- but also on the more abstract level of the system’s specification. This makes it possible to define complex requirements without knowledge about the underlying SLTS.

Using temporal logics, and especially SPDL, as property specification language has the following advantages:

- Properties can be defined using the same abstract, action-oriented view on the system as for the specification.
- The formal semantics of SPDL guarantees that the properties to be verified are interpreted in a unique and unambiguous way.
- The user does not need knowledge about the structure of the semantic model, i.e. unnecessary details are hidden from the user (abstraction).
- SPDL offers powerful means to express very precisely and concisely system runs that must be possible to satisfy the performability requirement specified. This is achieved by the possibility to use regular expressions and constructs familiar from programming languages, such as loops and other control structures, to define desired system runs.
- The possibility to characterise satisfying system runs very precisely allows the modeller to apply SPDL on different levels of the system design, thereby gaining new insights into the system’s behaviour.

1.3 Related Work

In this section we describe related work in the area of symbolic semantics for process algebras and automatic verification of performance measures.
1.3 Related Work

1.3.1 Symbolic Semantics for Process Algebras

For purely functional process algebras we are aware of three previous approaches to derive from a given process algebraic specification the BDD-based representation of its underlying semantic model.

In [50] the authors describe for the process algebra CCS [109] procedures how to derive BDD based representations of the transition system that results from a subset of the CCS language operators. Here, it is assumed that the semantic models of the operands are already available as BDDs.

In [132, 133] the process algebra LOTOS [25, 81] is considered. Here, the overall system specification is decomposed into smaller parts, the so-called basic-building blocks. For these blocks small transition systems are derived, which are transformed into their corresponding BDD representation. From these small BDDs the BDD representation of the semantic model of the overall specification is generated by combining the BDDs of the basic building blocks in an appropriate way.

The approach presented in [47] is applicable to a large number of process algebras, but the concepts are exemplified by using the process algebra CCS. Here, the process algebra specification is encoded as a vector consisting of 0’s and 1’s, i.e. it is encoded binarily. The behaviour of the process is captured by changing the encoding of the original process according to a set of rules that are available for every operator of the process algebra. This encoding and interpretation of a process and its dynamics makes it possible to directly represent the vector as a BDD. But, this approach has several drawbacks that hamper its applicability with respect to our goal of having a compact representation of even huge transition systems. The major problem is that for parallel composition the BDD of the overall process is not derived from the BDDs of its constituents, but in a monolithic way. This contradicts the findings of [50, 42, 127, 128, 129], where it is shown that the key to compact representations lies in the non-monolithic generation of the symbolic representation of the transition system.

1.3.2 Temporal Logics

There exist several extensions of the logic CTL for the stochastic case. At first we should mention the logic PCTL [65], a probabilistic logic, that is interpreted over discrete time Markov chains (DTMCs). In PCTL the CTL path quantifiers A and E are replaced by a probabilistic operator \( P_{\infty} p(\phi) \), that expresses that the probability that the path formula \( \phi \) is satisfied meets the bounds expressed in \( \infty p \), where \( \infty \in \{ <, \leq, >, \geq \} \).

Widely used in model checking of stochastic systems is the logic CSL, introduced in [6, 7] that is interpreted over a continuous time Markov chain (CTMC). This logic is extended in [15] by a steady-state operator \( S_{\infty} p(\Phi) \), that allows one to reason about steady-state probabilities, i.e. to reason about the probability that the system, considered on the long run is in a certain set of states.

CSL is state-oriented, i.e. to specify measures of interest, it offers the possibility to characterise states via simple logical formulas, so-called atomic propositions.

From CSL an action-oriented version, aCSL [73, 108] has been developed. Similar to aCTL (action based CTL) [51] the characterisation of satisfying paths is done by specifying sets of actions \( A \) that have to occur to satisfy a given path formula. aCSL is completely action-oriented, i.e. it has no means to simply characterise states by atomic propositions. But, as states are present as auxiliary means in the semantic model of stochastic process
algebras, the possibility to characterise them can ease the definition of performance measures. The lack of aCSL of using simple state formulae as auxiliary means to define performance measures hampers the applicability of aCSL. Another weakness of aCSL are the limited possibilities to characterise the order of actions on satisfying paths. For instance, it is not possible to state that the actions $a, b, c$ have to appear in this order and each of this actions exactly once. Therefore, aCSL has been extended to aCSL+ [107, 108] to overcome these problems. In aCSL+ states can be characterised by atomic propositions and paths are characterised via regular expressions.

In [14] we have devised the logic asCSL, which is similar to SPDL. The logic asCSL allows to characterise satisfying paths by means of regular expressions. The executability of a regular expression can be made dependent on test formulae, like in SPDL. Compared to SPDL, asCSL has a different syntax and semantics, and the model checking procedure is different from that of SPDL. In contrast to SPDL and its extensions, asCSL possesses no means to reason about systems that have both timed and untimed transitions. Also, SPDL is able to express a richer class of timed properties than asCSL.

1.4 Outline

This thesis is organised as follows: In chapter 2 we introduce the foundations that are necessary for the rest of the thesis. At first we repeat some basics of stochastic processes, then we present the foundations of Boolean and pseudo-Boolean functions. In the sequel we explain in quite detail the basic ideas of binary decision diagrams (BDDs) and the symbolic representation of transition systems by means of BDDs and variants thereof. The next two subsections of chapter 2 are dedicated to process algebras and their stochastic extensions. We conclude this chapter with an introduction to the logics PDL and CSL.

In chapter 3 we present a new denotational semantics for stochastic process algebras, which directly maps process algebraic system descriptions to the symbolic representation of their underlying labelled Markov chain. This semantics is defined with respect to a Markovian process algebra, derived from TIPP, which we will call YAMPA (yet another Markovian process algebra). We prove the semantics to be correct by showing that the semantic model generated by our new denotational semantics is equivalent with the models generated by the standard SOS semantics. Finally, we present heuristic algorithms that aim to keep the Markov chain minimal at every construction step by computing on the fly its minimised coefficient with respect to Markov bisimulation.

In chapter 4 we introduce the new logic SPDL, which is a stochastic extension of the logic PDL. We introduce its syntax and semantics and propose model checking algorithms for it. We also provide proofs that the model checking procedure is correct and that the validity of SPDL formulae is invariant with respect to a variant of Markov bisimulation. Furthermore, we provide a bound of both the space and time complexity of model checking SPDL formulae. We conclude chapter 4 with two extensions of the basic logic SPDL.

In chapter 5 we compare the expressive power of various stochastic logics. We prove that the logic SPDL is strictly more expressive than the logics CSL, aCSL, and aCSL+.

In chapter 6 we introduce the software tool CASPA, which is a stochastic model checker in which the denotational semantics from chapter 3 and the model checking algorithms from chapter 4 are implemented.

In chapter 7 we show the applicability of our theoretical concepts, implemented in CASPA, by a number of case studies. Some of these case studies are widespread and can be regarded as benchmarks for performance analysis. Among the benchmark case studies
considered in this thesis we have the Kanban system [38] and the flexible manufacturing system (FMS) [39]. Case studies, which cannot be seen as benchmark studies are: a fault-tolerant computer system and the handover procedure of a wireless communication network.

Chapter 8 concludes this thesis with a summary and pointers to future research.
2 Foundations

2.1 Continuous-Time Markov Chains

In this section we briefly review some basic notions from the theory of stochastic processes. We will also fix some notation that is used in the following chapters of the thesis. For details on stochastic processes, especially Markov processes, we refer to [88, 138].

Definition 1 (Stochastic Processes). A stochastic process is a set of random variables:

\[ \{ X_t | t \in T \} \]

where \( T \) is a set of parameters or indices; \( T \) is often interpreted as time. The random variables \( X_t \) can take values on a state space \( S \). If \( T \) is interpreted as time, then \( t \in T \) are time points. In this case, \( X_t \) is the state of the process at time point \( t \).

The state space \( S \) and the index set \( T \) can be either discrete or continuous. This yields four classes of stochastic processes:

1. State space continuous, time continuous
2. State space discrete, time continuous
3. State space continuous, time discrete
4. State space discrete, time discrete

In cases where the state space is discrete, i.e. it is finite or countably infinite, a stochastic process is often called a stochastic chain. If \( T \) is finite or countable, i.e. discrete time points \( t_i \in T \) are considered, such a process is called a discrete time stochastic process. If \( T \) is continuous, we speak of a continuous time stochastic process. In this thesis we only consider time continuous stochastic chains, which have the memoryless or Markov property. The Markov property states that the future behaviour of a process does not depend on its past, only the current state is decisive for the future behaviour.

Definition 2 (Markov Property). A stochastic process has the Markov property, if

\[
P(X_{t_{n+1}} = i_{n+1} | X_{t_n} = i_n, ..., X_{t_0} = i_0) = P(X_{t_{n+1}} = i_{n+1} | X_{t_n} = i_n)
\]

for all \( t_i \in T \), with \( t_0 \leq t_1 \leq ... \leq t_n \leq t_{n+1} \), for all \( n \), for all \( i_0, ..., i_n, i_{n+1} \).

The right side of the equation above can be read as:

The probability to be at time point \( t_{n+1} \) in state \( i_{n+1} \) depends on the probability to be at time point \( t_n \) in state \( i_n \).
Stochastic processes having this property are called Markov processes.

Continuous time stochastic chains with this property are referred to as continuous time Markov chains (CTMC).

The Markov property implies that the remaining sojourn time \( T_i \) in state \( i \in S \) does not depend on the time already spent in \( i \), i.e.

\[
P(T_i \leq t + t' | T_i > t') = P(T_i \leq t)
\]

The only distribution that satisfies these constraints is the exponential distribution, i.e. for the sojourn time we obtain the following distribution:

\[
P(T_i \leq t) = 1 - e^{-\lambda_i t}
\]

**Definition 3 (State Probabilities).** The probability that a Markov chain \( \mathcal{M} \) is in state \( i \) at time point \( t \) is defined by

\[
\pi_i(t) = P(X_t = i) \quad (2.1)
\]

The probability to transit within time interval \( \Delta t \) from state \( i \) to \( j \) is defined by

\[
p_{ij}(\Delta t) = P(X_{t+\Delta t} = j | X_t = i) \quad (2.2)
\]

If in \( \mathcal{M} \) these probabilities do not depend on \( t \) but only on \( \Delta t \), we call \( \mathcal{M} \) a homogeneous Markov chain.

All transition probabilities of a Markov chain \( \mathcal{M} \) can be written in matrix form:

\[
P(\Delta t) = [p_{ij}(\Delta t)] \in [0, 1]^{|S| \times |S|}
\]

where \( |S| \) is the number of states of \( \mathcal{M} \). Using equations 2.1 and 2.2 the state probabilities at time point \( t + \Delta t \) can be written as

\[
\pi_j(t + \Delta t) = \sum_{i \in S} \pi_i(t) \cdot p_{ij}(\Delta t)
\]

or equivalently:

\[
\vec{\pi}(t + \Delta t) = \vec{\pi}(t) \cdot P(\Delta t) \quad (2.3)
\]

where \( \vec{\pi} \) is the state probability vector of dimension \( |S| \). To compute the state probabilities at arbitrary time points, we proceed as follows:

\[
\frac{d}{dt} \vec{\pi}(t) = \lim_{\Delta t \to 0} \frac{\vec{\pi}(t + \Delta t) - \vec{\pi}(t)}{\Delta t} = \vec{\pi}(t) \cdot \lim_{\Delta t \to 0} \frac{P(\Delta t) - I}{\Delta t}
\]

(2.4)

where \( I \) is the identity matrix.

\[
Q := \lim_{\Delta t \to 0} \frac{P(\Delta t) - I}{\Delta t}
\]

\( Q \) is called the infinitesimal generator matrix with entries \( q_{ij} \):

\[
q_{ij} = \lim_{\Delta t \to 0} \frac{p_{ij}(\Delta t)}{\Delta t} \quad (i \neq j)
\]
where \( q_{ij} \) are the transition rates from state \( i \) to \( j \). For the diagonal elements of \( Q \) we have the following relation:

\[
q_{ii} = -\sum_{i \neq j} q_{ij}
\]

Equation 2.4 can be rewritten as:

\[
\frac{d \pi(t)}{dt} = \pi(t) \cdot Q
\]

**Definition 4.** In this definition we will fix some notation that is used throughout the thesis.

- **\( R \):** defines the transition rate matrix, \( R(s, s') \) denotes the transition rate from state \( s \) to state \( s' \), where it holds that for all states \( s \): \( R(s, s) = 0 \)
- **\( E \):** A vector of dimension \( |S| \), that denotes the sum of all rates of the transitions emanating from \( s \):
  
  \[
  \text{for all states } s: E(s) = \sum_{s' \in S} R(s, s')
  \]
- **\( P \):** transition probability matrix between states:
  
  \[
  \text{for all states } s: P(s, s') = \frac{R(s, s')}{E(s)}
  \]

\( P(s, s') \) is thus the probability to reach \( s' \) from \( s \) with a single transition.

### 2.1.1 Transient Analysis

To compute the vector of state probabilities at time point \( t, \pi(t) \), the Chapman-Kolmogorov differential equation system

\[
\frac{d \pi(t)}{dt} = \pi(t) \cdot Q
\]

has to be solved. The analytic solution is given by:

\[
\pi(t) = \pi(0) \cdot e^{Q \cdot t} = \pi(0) \cdot \sum_{k=0}^{\infty} \frac{Q^k \cdot t^k}{k!}
\]

where \( \pi(0) \) is the initial probability distribution. To solve this equation one often uses the uniformisation method [83, 137].

### 2.1.2 The Uniformisation Method

In this section we give a brief introduction to transient analysis via the so-called uniformisation method. The matrix exponential \( e^{Q \cdot t} \) can be rewritten as follows (Taylor series expansion):

\[
e^{Q \cdot t} = \sum_{k=0}^{\infty} \frac{(Q \cdot t)^k}{k!}
\]

The attempt to solve the matrix exponential using the Taylor expansion is not satisfactory, because ([113]):
• the truncation point of the series can not be computed efficiently
• the round-off errors are note negligible, because $Q$ contains both negative and non-negative entries.
• where $Q$ is sparse, it is the case that $(Q \cdot t)^k$ becomes non-sparse.

Therefore more appropriate means have to be used to solve the equation.

**Uniformisation:**

For uniformisation we define a stochastic matrix $P$, i.e. a matrix having entries that range from 0 to 1. $P$ is derived from $Q$:

$$P := I + \frac{Q}{\lambda}$$

$I$ is the identity matrix. $\lambda$ is chosen as the maximum absolute value of the diagonal entries of the generator matrix $Q$, i.e. $\lambda \geq \max(|Q(i,i)|)$. $P$ is a discrete time Markov chain (DTMC). We rewrite $Q$:

$$Q = \lambda \cdot (P - I)$$

We obtain:

$$\pi(t) = \pi(0) \cdot e^{Q \cdot t} = \pi(0) \cdot e^{(\lambda \cdot (P - I)) \cdot t}$$

Using a series expansion we have

$$\pi(t) = \pi(0) \cdot e^{-\lambda \cdot t} \cdot \sum_{k=0}^{\infty} \frac{(\lambda \cdot t)^k}{k!} \cdot P^k$$

$e^{-\lambda \cdot t} \cdot \{\{(\lambda \cdot t)^k / (k!\})\}$ are Poisson probabilities.

This Taylor-series now can be solved more efficiently. We write the equation above as follows:

$$\pi(t) = \sum_{k=0}^{\infty} e^{-\lambda \cdot t} \frac{(\lambda \cdot t)^k}{k!} \cdot P^k = \sum_{k=0}^{\infty} e^{-\lambda \cdot t} \frac{(\lambda \cdot t)^k}{k!} \cdot (\pi(0) \cdot P^k)$$

$\pi_k$ is the distribution of state probabilities in the DTMC $P$ after $k$ steps and can be computed recursively:

$$\pi_0 = \pi(0) \quad \pi_k = \pi_{k-1} \cdot P$$

Now, we have reduced the problem to a number of vector-matrix multiplications. The question is, how large this ‘number’ is, i.e. we have to determine the truncation point of the series. We compute $\pi_{approx}$ instead of $\pi$, because the series looks like this:

$$\pi_{approx}(t) = \sum_{k=0}^{n_{approx}} e^{-\lambda \cdot t} \frac{(\lambda \cdot t)^k}{k!} \cdot \pi_k$$
This truncation point $n_{approx}$ can be computed efficiently. It has to be the least value for $n_{approx}$ that satisfies the following condition:

$$
\sum_{k=0}^{n_{approx}} \frac{(\lambda \cdot t)^k}{n!} \geq (1 - \epsilon) \cdot e^{-\lambda \cdot t}
$$

Where $\epsilon$ is the maximum round-off error we allow. The Poisson probabilities are computed using the Fox-Glynn-algorithm [56].

### 2.1.3 Steady State Analysis

Steady state analysis examines the behaviour of the CTMC on the long-run, i.e. $t \to \infty$. This leads to the stationary (steady state) probability vector $\pi = \lim_{t \to \infty} \pi(t)$. This vector exists for homogeneous CTMCs with finite state space, but depends on the initial distribution, if the CTMC is reducible. A Markov chain is irreducible, if its underlying state graph is strongly connected (cf. definition 6).

For irreducible Markov chains, the computation of $\pi$ is quite simple. As $\pi$ does not depend on $t$ the differential quotient $\frac{d\pi(t)}{dt}$ is equal to 0, it suffices to solve the linear equation system

$$
\pi \cdot Q = 0 \quad (2.5)
$$

For solving this linear equation system efficient methods exist [137].

To compute stationary probabilities of not irreducible Markov chains we have to compute the absorbing (or bottom) strongly connected components of its underlying state graph.

**Definition 5 (Paths).** An infinite path $\sigma$ is a sequence of transitions of the form $s_0 \rightarrow s_1 \rightarrow s_2 \ldots$ where $s_i \in S$, $\sigma[i] = s_i$ is the $i+1$st state of path $\sigma$. A finite path $\sigma$ is a finite sequence of transitions of the form: $s_0 \rightarrow s_1 \rightarrow s_2 \ldots s_{n-1} \rightarrow s_n$, where the set of all finite or infinite paths with initial state $s$ is called $\text{PATH}(s)$

$$
\text{PATH}(s) := \{\sigma | \sigma[0] = s\}
$$

**Definition 6 (Reachability set, Connected Components).** The set $\text{Reach}(s)$ contains all states $s'$ that are reachable from $s$ with finitely many transitions:

$$
\text{Reach}(s) = \{s' \in S | \exists \sigma \in \text{PATH}(s) (\exists i \in \mathbb{N} (\sigma[i] = s'))\}
$$

A set $B \subseteq S$ is a strongly connected component (SCC), if all states of $B$ are mutually reachable after a finite number of transitions:

$$
\forall s, s' \in B (s' \in \text{Reach}(s))
$$

An SCC is maximal if, adding an extra state to an SCC destroys the SCC property; more formally:

$$
\not\exists s \in S \setminus B (\exists s' \in B ((s' \in \text{Reach}(s)) \land s \in \text{Reach}(s')))
$$

A Markov chains whose states are all within one single maximal SCC is irreducible.
Example 1 (SCC, Maximal SCC). Consider the CTMC in fig. 2.1. The set of states \( S = \{ s_3, s_4 \} \) form an SCC. In fact this is also a maximal SCC as, adding \( s_1 \) to \( S \) would destroy the SCC property of \( S \).

Definition 7 (Bottom Strongly Connected Component). An SCC \( B \) is called bottom strongly connected component (BSCC) if \( B \) is a maximal SCC and Reach\( (s) = B \) for all \( s \in B \), i.e. from no state \( s \) in \( B \) there is a transition to a state \( s' \) which is not in \( B \).

The stationary distribution of a BSCC \( B \) can be computed using equation 2.5. The distribution of the entire CTMC is computed as follows:

- If the initial state \( s \) is in a BSCC \( B_i \) only for \( B_i \) the stationary distribution has to be computed. Otherwise \( s \) is a transient state and we have to compute the distribution for all BSCCs. The vectors \( \pi_{B_i} \) have to be weighted with the probabilities \( p_{B_i} \) to reach BSCC \( B_i \) from \( s \).

2.2 Boolean and Pseudo-Boolean Functions

In this section we will briefly introduce the theory of Boolean functions. Boolean functions are of particular interest in this thesis, as we can use them to encode and compactly represent labelled transition systems\(^1\).

2.2.1 Boolean Algebra

Here, we will present some of the basic ideas of Boolean algebra. A more thorough textbook-style introduction can be found in [104]. All definitions and theorems apply to finite carrier sets \( C \). We are especially interested in the case where \( |C| = 2 \). This carrier set has its own symbol, \( B \) and has only the elements 0 and 1. This special type of Boolean algebras is so important, for example in the realm of digital circuit design, that it has its own name, switching algebra.

Definition 8 (Boolean Variables, Boolean Functions). The elements \( a, b, c, ... \) are called Boolean variables. Functions \( f : C^m \rightarrow C^n \) are called Boolean functions.

Definition 9 (Boolean Algebra). A finite set \( C \) with two special elements 0 and 1, two binary functions + (sum) and \( \cdot \) (product) and a unary function \( \bar{\cdot} \) (complement) is called Boolean algebra \( B = (C, +, \cdot, \bar{\cdot}, 0, 1) \) if for all \( a, b \) the following axioms are true:

\(^1\) As we will see, CTMCs can be interpreted as a special kind of labelled transition systems.
2.2 Boolean and Pseudo-Boolean Functions

- **Commutativity:**
  
  \[ a + b = b + a \]
  
  \[ a \cdot b = b \cdot a \]

- **Distributivity:**
  
  \[ a \cdot (b + c) = (a \cdot b) + (a \cdot c) \]
  
  \[ a + (b \cdot c) = (a + b) \cdot (a + c) \]

- **Identity:**
  
  \[ a + 0 = a \]
  
  \[ a \cdot 1 = a \]

- **Complement:**
  
  \[ a + \overline{a} = 1 \]
  
  \[ a \cdot \overline{a} = 0 \]

\( C \) is the carrier set, \( 0 \) the zero-element and \( 1 \) the one-element of the algebra.

**Example 2.** Let \( S \) be a finite set and \( 2^S \) its corresponding power set, i.e. the set of all subsets of \( S \). Let \( A \subset S \), then \( \overline{A} = S \setminus A \). With the two binary operations of set union (\( \cup \)) and set intersection (\( \cap \)), we obtain a Boolean algebra:

\( (2^S, \cup, \cap, \overline{\cdot}, \emptyset, S) \)

The empty set is the zero-element and the entire set \( S \) is the one-element of this algebra. Set union can be interpreted as sum, and set intersection can be interpreted as product.

Let \( S = \{A, B\} \), then \( 2^S = \{\emptyset, \{A\}, \{B\}, \{A, B\}\} \). For example, the axioms of identity are satisfied, as:

\[ \{A\} \cup \emptyset = \{A\}, \{A\} \cap \emptyset = \{A\} \]

likewise for all other elements of \( 2^S \).

The axioms of complement are also satisfied, as:

\[ \{A\} \cup \overline{\{A\}} = S, \{A\} \cap \overline{\{A\}} = \emptyset \]

\( \boxdot \)

**Theorem 1 (Laws of Boolean Algebras).** For elements \( a, b, c \in C \) of a Boolean algebra \( B \) we have the following arithmetic laws.

- **Idempotency:**
  
  \[ a + a = a \]
  
  \[ a \cdot a = a \]

- **Properties of 0 and 1:**
  
  \[ a + 1 = 1 \]
  
  \[ a \cdot 0 = 0 \]
Absorption:

\[ a + (a \cdot b) = a \]
\[ a \cdot (a + b) = a \]

Associativity:

\[ a + (b + c) = (a + b) + c \]
\[ a \cdot (b \cdot c) = (a \cdot b) \cdot c \]

DeMorgan:

\[ \overline{a + b} = \overline{a} \cdot \overline{b} \]
\[ \overline{a \cdot b} = \overline{a} + \overline{b} \]

Involution:

\[ \overline{\overline{a}} = a \]

### 2.2.2 Boolean Formulae

Boolean formulae can be constructed inductively from Boolean variables and Boolean functions.

**Definition 10 (Boolean Formulae).** Let \( B \) be a Boolean algebra. A Boolean formula over \( n \) variables \( a_1, \ldots, a_n \), function symbols \( +, \cdot, \neg \) and elements from the carrier set \( C \) is defined recursively as follows:

- All elements from \( C \) are Boolean formulae
- Variable symbols \( a, b, \ldots \) are Boolean formulae
- If \( F \) and \( G \) are Boolean formulae, so are
  - \( F + G \)
  - \( F \cdot G \)
  - \( \overline{F} \)

**Pseudo-Boolean Functions**

Pseudo-Boolean functions extend Boolean functions in the way that their range is a finite subset \( D \) of the real numbers, i.e. for any Pseudo-Boolean function \( f \), we have:

\[ f : \mathbb{C}^n \rightarrow D \]

All results on Boolean functions remain valid.

### 2.2.3 Switching Formulae

In the sequel we will draw our attention to the case where \( |C| = 2 \). For this special carrier set we use the symbol \( B \) and let \( B = \{0, 1\} \).

**Definition 11 (Switching Algebra).** Let the functions \(+, \cdot, \neg\) over the set \( B \) be defined as follows:
2.3 Binary Decision Diagrams

- \( a + b = \max\{a, b\} \)
- \( a \cdot b = \min\{a, b\} \)
- \( 0 = 1, \quad 1 = 0 \)

This defines a Boolean algebra, called switching algebra.

**Definition 12 (Switching Function).** An \( n \)-ary function \( f : B^n \rightarrow B \) is a switching function.

**Definition 13 (Literal, Product Term, . . .).** In the sequel we define some important notions needed in subsequent chapters of the thesis.

- **Literal:** A literal is a variable, or the complement of a variable.
- **Product term:** A product term is a single literal or a product of two or more literals.
- **Sum term:** A sum term is a single literal or a sum of two or more literals.
- **Normal term:** A normal term is a product or sum term in which no variable appears more than once.
- **Minterm:** An \( n \)-variable minterm is a normal product term with \( n \) literals.
- **Maxterm:** An \( n \)-variable maxterm is a normal sum term with \( n \) literals.

As in the remainder of the thesis all Boolean formulae are defined over \( B \), we will use the term ‘Boolean’ also for switching functions.

---

**2.3 Binary Decision Diagrams**

In this section the central notion of binary decision diagrams (BDDs), some extensions of the basic concepts and their application in formal verification of stochastic models are introduced.

**2.3.1 Motivation**

Boolean formulae play a central role in many areas of electronical engineering, such as digital circuit design. But it is also the case that in the important area of formal verification of distributed systems, Boolean formulae can be applied with great success. For example, many formal specification languages have as semantic model labelled transition systems (LTS) that can be expressed by means of Boolean formulae. This is especially interesting for this work, as we concentrate on (stochastic) process algebras as modelling formalisms, which also have (S)LTSs as their semantic model.

The notion of binary decision diagrams goes back to early works of Lee [101] and Akers [2]. But the break-through of binary decision diagrams as important and efficient data structures in the areas indicated above is due to the seminal works of Bryant [30] and Brace [29], that led the way to the efficient application of binary decision diagrams.

**2.3.2 Representation of Boolean Formulae**

If Boolean functions shall be applied successfully for circuit design, formal verification, etc., the central issue is how to represent and manipulate Boolean formulae efficiently. All possible representations have to fulfill the following requirements:

A) **Compact representation:** Due to memory constraints, it is required that also large formulae with possibly tens of thousands of variables can be represented compactly.
B) Efficient manipulation: It is obvious that different operations on a given formula have to be performed efficiently. Examples of such operations are negation, conjunction, test for satisfiability, etc.

We will consider three well-known methods to represent Boolean formulae and check whether they allow compact representation and efficient manipulation of formulae:

1. truth table
2. normal forms
3. graph-based representation

Truth Tables

Truth tables form the easiest and most straight-forward way to represent Boolean formulae. Let us assume the Boolean formula has \( n \) distinct variables, then we have a truth table of size \( 2^n \), i.e. each of the \( 2^n \) lines represents one of the possible value assignments to the \( n \) variables and its corresponding outcome. Truth tables have the advantage that they are a canonical representation of Boolean formulae, i.e. semantically equivalent formulae are represented by the same truth table (modulo permutation of the order in which lines appear).

However, truth tables do not satisfy either of the two requirements A) and B). As argued for \( n \) variables, the truth table has \( 2^n \) entries, i.e. its size grows exponentially in the number of variables, this violates requirement A). Furthermore, requirement B) is violated, as many important operations are not efficiently executable, e.g. test of satisfiability requires in the worst case the inspection of all \( 2^n \) lines of the table.

Example 3. Let \( F \) be a Boolean formula with three variables \( x_0, x_1, \) and \( x_2 \) that yields as result 1, if the binary number \( [x_0 x_1 x_2]_2 \) is a prime number. This is the case for 010, 011, 101, and 111, The corresponding truth table for \( F \) looks as follows:

<table>
<thead>
<tr>
<th>( x_0 )</th>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( F )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Normal Forms - Propositional Logic Formulae

Boolean formulae coincide with formulae of propositional logic. In this case we interpret complement as logical negation (\( \neg \)), sum as disjunction (\( \lor \)) and product as conjunction (\( \land \)).

We will briefly introduce the notion of a normal form.

\(^2\) This interpretation will be used in the subsequent sections.
Definition 14 (Disjunctive Normal Form). A formula $\phi$ is in disjunctive normal form (DNF), if it is a disjunction of conjunction terms. A conjunction term is built by the conjunctive connection of literals. Literals are negated or non-negated Boolean variables. A formula in disjunctive normal form is defined as follows:

$$\bigvee_i \bigwedge_j (-) x_{ij}$$

Example 4 (Disjunctive Normal Form). Let $F$ be a Boolean formula in three variables $x_0$, $x_1$, and $x_2$ in DNF that yields as result 1, if the binary number $[x_0x_1x_2]_2$ is a prime number. This is the case for 010, 011, 101, and 111. The DNF of $F$ is defined as follows:

$$DNF(F) = (\neg x_0 \land x_1 \land \neg x_2) \lor (\neg x_0 \land x_1 \land x_2) \lor (x_0 \land \neg x_1 \land x_2) \lor (x_0 \land x_1 \land x_2)$$

Definition 15 (Conjunctive Normal Form). A formula $\phi$ is in conjunctive normal form (CNF), if it is a conjunction of disjunction terms. A disjunction term is built by the disjunctive connection of literals. Literals are negated or non-negated Boolean variables. A formula in conjunctive normal form is defined as follows:

$$\bigwedge_i \bigvee_j (-) x_{ij}$$

Example 5 (Conjunctive Normal Form). Let $G$ be a Boolean formula $G$ that yields 1, if $[x_0x_1x_2]_2$ is a prime number. The CNF of formula $G$ is defined as follows:

$$CNF(G) = (x_0 \lor x_1 \lor x_2) \land (\neg x_0 \lor x_1 \lor x_2) \land (x_0 \lor x_1 \lor \neg x_2) \land (\neg x_0 \lor \neg x_1 \lor x_2)$$

We will assume that the formulae are given in any kind of normal form (disjunctive, conjunctive normal form), which is very important in practice, as only normal form guarantee canonicity of the representation, i.e. two semantically equivalent formulae have the same normal form representation. Propositional formulae can often meet requirement A), as they can often be represented very compactly.

Unfortunately they do not meet the efficiency requirements, as many operations are not executable efficiently. For example negation of a formula normally destroys the normal form property, the formula’s new normal form has to be recomputed.

2.3.3 Representation of Boolean Formulae as Graphs - BDDs

In this section we will describe in some detail the representation of Boolean formulae by means of graphs, especially by binary decision diagrams (BDDs). In this section we will describe in a more informal way how to derive from a given binary decision tree that represents a Boolean formula a compact graph-based representation. In section 2.3.4 we will put this on a more formal base.

---

3 Minterms in terminology of Boolean formulae
4 Maxterms in terminology of Boolean formulae
Binary Decision Trees

Definition 16 (Binary Decision Trees). Binary decision trees (BDTs) are trees whose non-terminal nodes are labelled with Boolean variables. Their terminal nodes are either labelled with zero (0) or one (1). Each non-terminal node has exactly two outgoing edges, representing the two different possible value assignments to its associated Boolean variable. For the graphical representation of BDTs we decide that dotted lines represent value 0 and solid lines value 1 of a Boolean variable.

Definition 17 (Path, Level). A path in a tree is defined as a non-empty sequence of nodes, such that the ith node on the path is the parent node of the i + 1st node. A path with nodes $x_0, \ldots, x_k$ has length $k$.

The level of a node $x$ in a tree $T$ is the length of the unique path in $T$ from its root to the node $x$. E.g., the root of $T$ is at level zero and the roots of the subtrees of $T$ with the root node as parent node are at level one.

Definition 18 (Boolean Formula Representation by Binary Decision Trees). Let $T$ be a finite binary decision tree. $T$ determines uniquely a Boolean formula $f$, whose variables are determined by the labels of $T$’s non-terminal nodes. To determine the outcome of $f$ with respect to a specific value assignments to its variables, $T$ is traversed from the root node to bottom by taking on each level the outgoing edge that represents the value of the Boolean variable, associated to the current node.\(^5\) Note, that for all paths it holds, that no variable can occur more than once.

Example 6. For a given Boolean formula $f = (x_1 \land \neg x_2) \lor (\neg x_1 \land \neg x_2)$ its corresponding BDT can be found in fig. 2.2.

\[
\begin{align*}
  f(0,0) &= 1 \\
  f(0,1) &= 0 \\
  f(1,0) &= 1 \\
  f(1,1) &= 0
\end{align*}
\]

Fig. 2.2. Boolean formula and its BDT

With respect to size, BDTs have no advantages over truth tables, as a BDT is a complete binary tree, i.e. it requires $2^{n+1} - 1$ nodes to represent an $n$-ary Boolean formula.

The good message is, that we can do better, as a BDT is normally highly redundant, i.e. its size can normally be reduced significantly by exploiting these redundancies.

Binary Decision Diagrams

In this section we describe how to exploit redundancies in BDTs to obtain smaller representations. These redundancy reduction strategies destroy the tree property of the directed graph, we will refer to the non-redundant graph structures as binary decision diagrams (BDDs). Before we formally define the reductions that can be applied, we will introduce them by means of examples:

\(^5\) The algorithm that constructs a BDT from a given Boolean formula $f$ can be found in section 2.3.4.
1. BDT terminal nodes are either zero or one, therefore it suffices to have only exactly one terminal node for each of the possible values, see fig. 2.3:

![Fig. 2.3. BDT with only two distinct terminal nodes](image)

2. To further reduce the size a second reduction rule can be applied. Isomorphic sub-trees can be superposed (cf. fig. 2.4). The two $x_2$-rooted sub-graphs are isomorphic, i.e. they can be replaced by a single BDD, incoming edges of the deleted sub-trees are redirected accordingly.

In fig. 2.4 we can see that applying one reduction rule can lead to the applicability to other reduction rules, here a node whose outgoing edges lead to the same successor node, a so-called don’t-care-node, is generated, which could be removed.

3. As a third reduction it is possible to remove redundant non-terminal nodes (fig. 2.5): The $x_1$ node is redundant, as both outgoing edges lead to the $x_2$ successor-node. This node can be removed and its incoming edges are redirected to the successor node of the node to be deleted. Such nodes are also called don’t-care-nodes.

To summarise, we obtain the following reduction rules:

**O1) Removing redundant terminal nodes:** If a BDD possesses more than one 0 (1) terminal node then all occurrences of 0 (1) nodes except exactly one of each type are deleted. All
edges leading to a 0 (1) terminal node, that is to be deleted are redirected accordingly to the remaining terminal node.

O2) Removing isomorphic sub-trees: If two nodes \(m\) and \(n\) are root nodes of structural identical sub-BDDs, then one of them say, the one rooted at \(m\), can be deleted, its incoming edges are redirected to \(n\).

O3) Removing don’t care nodes: If both outgoing edges of a node \(m\) are leading to the same successor \(n\), then all incoming edges of \(m\) are redirected to \(n\) and \(m\) is deleted.

This leads us to the following definition of (reduced) BDDs.

**Definition 19 (Reduced BDD).** A BDD is a directed acyclic graph (DAG), having one special node, the root node. All terminal nodes are either labelled with 0 or 1. All non-terminal nodes are labelled with a Boolean variable, the variable associated to node \(n\) is denoted by \(\text{var}(n) = x\), as in the case of BDTs, variables can occur at most once on any path. Each non-terminal node has exactly two outgoing edges, the 0-edge and the 1-edge. A BDD is called reduced if none of the rules O1) to O3) is applicable.

In the sequel we always assume, unless otherwise stated, that BDDs are reduced.

**Variable Order**

The reduction rules may lead to compact representations of Boolean formulae such that requirement A) of section 2.3.2 is met. But up to now, requirement B) may be violated. This stems from the fact that the definition of BDDs does not forbid that on different paths variable labellings appear at different levels. This leads both to inefficiencies that can be overcome, if we introduce a variable order on paths which is defined over the Boolean variables resp. node labels.

**Definition 20 (Variable Order).** Let \(S = \{x_1, x_2, \ldots, x_n\}\) be an ordered set of Boolean variables. Let \(B\) be a BDD whose labellings are among the elements of \(S\). \(B\) has order type \(S\), if for any occurrence of \(x_i\) which is followed by \(x_j\) it holds that level \(i \prec j\) and therefore \(x_i \prec x_j\), i.e. the level at which \(x_i\) appears is closer to the root than the level of \(x_j\). A BDD is called ordered (OBDD) if it possesses an order as described above.

**Example 7.** The BDD in fig. 2.6 has order \(x_1 \prec x_2 \prec x_3\).  ❖

![Fig. 2.6. BDD with order \(x_1 \prec x_2 \prec x_3\)](image-url)
In the sequel we assume that, unless stated differently, BDDs are both reduced and ordered, i.e. they are reduced ordered BDDs (ROBDDs). Performing operations on two BDDs, e.g. to generate a new BDD out of two given BDDs requires that their respective variable orders are compatible, i.e. if in BDD \( B_1 \) \( x \prec y \) it must hold that in BDD \( B_2 \) \( x \prec y \). The introduction of orders to reduced BDDs leads to the following fundamental theorem [30], which we will cite without proof.

\begin{align*}
\textbf{Theorem 2.} & \text{ The ROBDD representing Boolean formula } f \text{ is unique. Let } B_1 \text{ and } B_2 \text{ be two ROBDDs representing the same Boolean formula with compatible variable order, then both are structural identical, i.e. they are isomorphic.}
\end{align*}

**Influence of Variable Orders on BDD Size**

The size of a BDD is defined as the number of its nodes. The chosen variable order for a BDD strongly influences its size. There are formulae for which the size is either exponential or linear in the size of the number of variables, depending on the chosen variable order. Therefore, the task of finding a good variable order for the BDD representation of a given Boolean formula is crucial. However, it is known that finding an optimal order is NP-complete [24]. Therefore, we have to rely on heuristics. In this work especially one heuristic variable order is of interest, which turned out to yield good results, i.e. compact BDD representations, namely an interleaved order (see example 8 and section 2.3.5).

**Example 8 (Influence of Variable Order on BDD Size).** The formula

\[ F := (x_1 \leftrightarrow y_1) \land (x_2 \leftrightarrow y_2) \]

is a good example to demonstrate the influence of the variable order on the size of a BDD. In fig. 2.7\(^6\) the chosen order is \( x_1 \prec x_2 \prec y_1 \prec y_2 \). This BDD possesses 9 non-terminal nodes. In fig. 2.8 we see the BDD for the same formula \( F \), but with order \( x_1 \prec y_1 \prec x_2 \prec y_2 \).

![BDD for F with order x1≺x2≺y1≺y2](image)

This BDD has only 6 non-terminal nodes. Generally, the second type of order is called an interleaved order, whereas the first type is referred to as non-interleaved order. For the

---

\(^6\) Both in fig. 2.7 and in fig. 2.8 we have omitted edges that lead to the 0 terminal node
generalised version of $F$, i.e.

$$F' := \bigwedge_{i=0}^{n} (x_i \leftrightarrow y_i)$$

the size of the non-interleaved order of the resulting BDD is exponential in the number of variables, whereas in case of an interleaved ordering of the variables the BDD size is only linear.

**Significance of Canonicity**

The fact that BDDs are a canonical representation of Boolean formulae is of great importance, as this allows the efficient execution or implementation of many important operations such as:

- Test on semantic equivalence: If $f, g$ are Boolean formulae and $B_f$ and $B_g$ their respective BDDs, with compatible variable order then $B_f$ and $B_g$ must be isomorphic if $f$ and $g$ are equivalent.
- Satisfiability: If $f$ is a Boolean formula and $B_f$ its BDD representation, then for $f$ to be satisfiable, $B_f$ must be different from the BDD consisting only of the 0 terminal node.
- Validity: If a formula $f$ is valid, i.e. its outcome is 1 regardless of the variable assignments, then its corresponding BDD representation is the 1 BDD, i.e. the BDD consisting only of the 1 terminal node.

We conclude this section with a formal definition of BDDs that also introduces some notation used throughout this thesis:

**Definition 21.** An ordered binary decision diagram (OBDD) over a set of variables $Vars$ and an order relation $\prec$ is a rooted DAG $B = (\text{Nod}, \text{var}, \text{lo}, \text{hi})$ defined by

- a finite set of nodes $\text{Nod} = T \cup NT$, where $T$ ($NT$) is the set of terminal (non-terminal) nodes, $|\text{Nod}| \geq 1$, $T \subseteq B$,
- a function $\text{var} : NT \mapsto Vars$, where $Vars = \{x_1, \ldots, x_n\}$ is a set of Boolean variables with a fixed order relation $\prec \subset Vars \times Vars$,
• a function $hi : NT \mapsto Nod$ and a function $lo : NT \mapsto Nod$, with the following properties:

$$\forall x \in NT : hi(x) \in T \lor \text{var}(x) \prec \text{var}(hi(x))$$

$$\forall x \in NT : lo(x) \in T \lor \text{var}(x) \prec \text{var}(lo(x))$$

where $lo(x)$ denotes the 0-successor of node $x$ and $hi(x)$ denotes the 1-successor of node $x$.

An OBDD is called reduced (ROBDD) iff

1. $\forall x \in NT (lo(x) \neq hi(x))$
2. $\forall x, y \in NT (\text{var}(x) \neq \text{var}(y) \lor lo(x) \neq lo(y) \lor hi(x) \neq hi(y))$

Condition 1 states that redundant nodes, i.e. don’t-care nodes are skipped, the second condition states that no pair of isomorphic nodes exist.

### 2.3.4 Basic BDD Algorithms

In this section we will describe three algorithms that are crucial for working with BDDs:

1. Construction: Given a Boolean formula $f$, generate its corresponding ordered BDT.
2. Reduction: Given an ordered BDT, transform it to its corresponding ROBDD.
3. Apply: Given two ROBDDs $B_f$ and $B_g$, construct with respect to a given binary operation $\star$ the ROBDD $B_{f \star g}$ of formula $f \star g$.

**Construction Algorithm**

The construction algorithm is based on co-factors and Shannon’s expansion theorem.

**Definition 22 (Co-factors).** $f(x_1, x_2, ..., x_i, ..., x_n)$ is called co-factor of $f$ with respect to $x_i$ (positive co-factor), for short $f\mid_{x_i}$, $f(x_1, x_2, ..., x_i-1, 1, x_i+1, ..., x_n)$ is called co-factor of $f$ with respect to $\neg x_i$ (negative co-factor), for short $f\mid_{\neg x_i}$.

Without proof, we cite Shannon’s expansion theorem:

**Theorem 3 (Shannon’s Expansion Theorem).** For Boolean formulae $f$ the following relation holds:

$$f(x_1, ..., x_n) = (x_i \land f\mid_{x_i}) \lor (\neg x_i \land f\mid_{\neg x_i})$$

A formula in $n$ variables is replaced by two formulae in $n-1$ variables. Co-factorisation can be interpreted as partial evaluation of a formula $f$.

We apply the Shannon expansion theorem to transform a given Boolean function $f$ to its corresponding ordered BDT:

C1) Generate for each variable in $f$ its co-factor, starting with the minimal variable in the given order.

C2) Each co-factor $f\mid_{x}$ resp. $f\mid_{\neg x}$ results in a partial BDT with:

- root node $x$
- $f\mid_{x}$ resp. $f\mid_{\neg x}$ as child nodes

C3) The algorithm terminates if the result of the co-factorisation with respect to all possible variable assignments is only the constant 0 or 1.

**Example 9.** For $f(x_1, x_2, x_3, x_4) = x_1 \land x_2 \land x_3 \lor \neg x_2 \land x_4 \lor \neg x_3 \land x_4$. We want to compute its corresponding BDT. Let the variable order $x_2 < x_3 < x_4 < x_1$ be given. ☐
\[ f \left( x_1, x_2, x_3, x_4 \right) = x_1 \land x_3 \lor \neg x_3 \land x_4 \]

\[ f \left( \neg x_1, x_2, x_3, x_4 \right) = x_4 \lor \neg x_3 \land x_4 \]

The Reduction Algorithm

Any practical application relies on ROBDDs, i.e. we have to transform the ordered BDT from the construction algorithm to ROBDDs.

Let the order \( \{ x_1, ..., x_l \} \) of Boolean variables on a BDD \( B \) be given. \( B \) has at most \( l + 1 \) levels. The reduction algorithm traverses \( B \) level by level, starting at the terminal nodes. On traversal each node \( n \) is assigned an integer label \( \text{id}(n) \). This labelling helps to identify nodes that can be deleted, such that each label \( \text{id}(n) \) appears exactly once in a BDD.

R0) All 0 terminal and all 1 terminal are replaced by a single 0 resp. 1 terminal node and incoming edges are redirected accordingly.

R1) If \( \text{id(lo}(n)) = \text{id(hi}(n)) \) then \( \text{id}(n) \) is either \( \text{id(lo}(n)) \) or \( \text{id(lo}(n)) \), as in this case the Boolean formula represented by the BDD rooted at \( n \) is identical with the one represented by the successor node of \( n \), i.e. \( n \) can be removed as it is a don’t care node according to rule O3).
R2) If there are nodes \( n \) and \( m \) with the same variable label \( x_i \) and \( id(lo(n)) = id(lo(m)) \) and \( id(hi(n)) = id(hi(m)) \) then \( id(n) = id(m) \), as \( m \) and \( n \) are roots of isomorphic sub-BDDs, which can be superposed according to rule R2).

R3) If none of the rules R0) to R2) is applicable to node \( n \), then \( n \) is assigned a new identification number \( id(n) \).

Example 10. In fig. 2.9 we find the result of both the labelling procedure of the reduction algorithm and the application of reduction rule R0. In fig. 2.10 we find the result of applying the reduction rules with respect to variable \( x_1 \). In fig. 2.11 we find the result of applying the reduction rules with respect to variable \( x_4 \). Finally, in fig. 2.12 the result of applying the reduction rules on variable \( x_3 \) is displayed.

The Apply Algorithm

Another central algorithm is the apply algorithm, which generates a new BDD \( B_h \) from two given BDDs \( B_f \) and \( B_g \) with respect to a given binary Boolean operation \( \star \).

The algorithm is recursive and starts at the roots \( r_f \) and \( r_g \) of the respective BDDs \( B_f \) and \( B_g \) to compute \( B_h = B_f \star B_g \). After the top-level call \( apply(\star, r_f, r_g) \), the algorithm proceeds as follows:
A1) If \( r_f \) and \( r_g \) are terminal nodes with value \( v_f \) resp. \( v_g \), then \( v_f \star v_g \) can be computed directly. \( B_h \) is then either 0 or 1 depending on the result of \( v_f \star v_g \).

A2) If \( \text{var}(r_f) = \text{var}(r_g) = x_i \), in \( B_h \) an \( x_i \) node with 0 edge to the sub-BDD \( \text{apply}(\star, \text{lo}(r_f), \text{lo}(r_g)) \) and with 1 edge to the sub-BDD \( \text{apply}(\star, \text{hi}(r_f), \text{hi}(r_g)) \) is generated.

A3) If \( \text{var}(r_f) \prec \text{var}(r_g) \) or \( r_g \) a terminal node, then in \( B_h \) an \( \text{var}(r_f) \) node with 0 edge to \( \text{apply}(\star, \text{lo}(r_f), r_g) \) and 1 edge to \( \text{apply}(\star, \text{hi}(r_f), r_g) \) is generated.

A4) The case in which \( \text{var}(r_g) = x_i \) and \( r_f \) is either a terminal node or \( \text{var}(r_g) \prec \text{var}(r_f) \) is treated symmetrically to case A3).

The apply algorithm generates a non-reduced ordered BDT, which has to be reduced applying the reduction algorithm. In practice, we proceed differently: A clever implementation generates an ROBDD without detour over an unreduced BDT. Details on efficient implementation can be found in [29].

Example 11. Consider the two BDDs \( B_1 \) and \( B_2 \) in fig. 2.13. We want to compute \( B = B_1 \lor B_2 \). For better readability, we have added labels \( R_i \) in \( B_1 \) resp. \( S_i \) in \( B_2 \) to each node. In fig. 2.14 each node is labelled with a pair \((R_i, S_j)\). We have added the rules A1) to A4) according to which a node \((R_i, S_j)\) and its corresponding variable label has been generated. For example:
• Node \((R2, S3)\) is labelled with \(x_2\), as \(B_1\) depends on \(x_2\), whereas \(B_2\) does not, therefore we have to generate this node according to rule \(A3\).

• According to rule \(A2\) we have labelled node \((R3, S2)\) with \(x_3\) as the 1-successor of \((R1, S1)\) is in both BDDs a node labelled with \(x_1\).

The BDT that results from \(\text{apply}(\lor, B_1, B_2)\) is shown in fig. 2.14, the resulting ROBDD can be found in fig. 2.15.

**The Restrict Operator**

The basic idea of this operator is to restrict a variable \(x\) to a specific value \(v\), i.e. to 0 or 1. To restrict \(x\) to \(v\) we simply redirect any incoming edge of node \(n\), \(\text{var}(n) = x\), to \(lo(x)\), if \(v = 0\) or to \(hi(x)\), if \(v = 1\). More formally, this can be expressed as follows:

\[
\text{Restrict}(B, x, v) := B\big|_{x=v}
\]
Restriction with respect to more than one variable can be defined recursively as follows:

\[
\text{Restrict}(B, \langle x_i, \ldots, x_m \rangle, \langle v_i, \ldots, v_m \rangle) := \text{Restrict}(B, \langle x_i, \ldots, x_{m-1} \rangle, \langle v_i, \ldots, v_{m-1} \rangle)|_{x_m := v_m}
\]

**Example 12.** Let the BDD \(B\) from fig. 2.16 (a) be given. In fig. 2.16 (b) we find the intermediate resulting graph after having restricted variable \(x_2\) to value 1. All nodes labelled with \(x_2\) are bypassed to their respective 1-successors, 0-successors are omitted. The type of the edges (0- or 1-edge) is determined by the type of the incoming edges of \(x_2\). In fig. 2.16 (c) the resulting ROBDD of this restrict-operation can be found.

![Fig. 2.16. (a) ROBDD \(B\), (b) Intermediate result of Restricting \(x_2\) to value 1, (c) Final ROBDD after restriction](image)

**The Abstract Operator**

Abstraction can be seen as the combination of all possible restrictions to a set of variables combined by an associative binary operator \(*\). Formally, this can be defined as follows:

\[
\text{Abstract}(B, \langle x_i, \ldots, x_m \rangle, *) := B|_{x_i:=0,...,0} * \ldots * B|_{x_i:=1,...,1}
\]
The operator $\star$ must be associative as the order in which the co-factors and the variables are chosen shall not influence the outcome of the operation.

### 2.3.5 Extending BDDs to Represent Pseudo-Boolean Formulae

In section 2.3.6 we will see that stochastic LTS (SLTS), i.e. LTS where the transitions between states are timed, are representable by pseudo-Boolean formulae. Therefore we have to extend the BDD data structure to cope with this type of formulae.

For pseudo-Boolean formulae, we interpret complement of a variable $x_i$ as $(1 - x_i)$, the conjunction of two variables $x_i$ and $x_{i+1}$ as multiplication, $x_i \star x_{i+1}$, and the disjunction of two variables $x_i$ and $x_{i+1}$ as addition, $(x_i + x_{i+1})$.

In fact a large number of extensions exist, that can cope with pseudo Boolean functions, among these are edge-valued BDD (EVBDDs) [100], decision node BDDs (DNBDDs) [128, 130], matrix diagrams (MDs) [36, 37, 111, 112, 110], based on multi-valued decision diagrams (MVDDs) [133, 136], multi-terminal BDDs (MTBDDs) [59], algebraic decision diagrams (ADDs) [12, 13], that coincide with MTBDDs.

MTBDDs are the most straightforward way to extend BDDs to deal with Pseudo-Boolean functions. Although they are quite simple, they have turned out to be very efficient means to compactly store and manipulate even huge stochastic labelled transition systems.

#### Properties of Multi-Terminal BDDs

As the name multi-terminal BDDs suggests, MTBDDs may have more than two terminal nodes: one for each of the possible outcomes of the pseudo-Boolean formula that is to be represented by an MTBDD. In this thesis we concentrate on MTBDDs for two reasons:

1. MTBDDs are well-understood and have demonstrated their usefulness for compact representation of huge SLTSs in many case studies.
2. For MTBDDs efficient manipulation and numerical analysis algorithms exist.

**Definition 23.** An (ordered) multi-terminal BDD (MTBDD) over an ordered set of variables $\text{Vars}$, an order relation $\prec$ and a finite range $\mathbb{D} \subset \mathbb{R}$ is a rooted DAG $M = (\text{Nod}, \text{var}, \text{lo}, \text{hi}, \text{value})$ defined by

- a finite set of nodes $\text{Nod} = T \cup \text{NT}$, where $T$, $(\text{NT})$ is the set of terminal (non-terminal) nodes, $|\text{Nod}| \geq 1$,
- a function $\text{var} : \text{NT} \mapsto \text{Vars}$, where $\text{Vars} = \{x_1, \ldots, x_n\}$ is a set of Boolean variables with a fixed order relation $\prec \subseteq \text{Vars} \times \text{Vars}$,
- a function $\text{hi} : \text{NT} \mapsto \text{Nod}$ and a function $\text{lo} : \text{NT} \mapsto \text{Nod}$ with the following properties:
  \[ \forall x \in \text{NT} : \text{hi}(x) \in T \lor \text{var}(x) \prec \text{var(} \text{hi}(x) \text{)} \]
  \[ \forall x \in \text{NT} : \text{lo}(x) \in T \lor \text{var}(x) \prec \text{var(} \text{lo}(x) \text{)} \]
  $\text{hi}(x)$ denotes the 1-successor of variable node $x$ and $\text{lo}(x)$ denotes the 0-successor of node $x$.
- a function $\text{value} : T \mapsto \mathbb{D}$

An MTBDD is reduced iff

1. $\forall x \in \text{NT} (\text{lo}(x) \neq \text{hi}(x))$
2. $\forall x, y \in \text{NT} (\text{var}(x) \neq \text{var}(y) \lor \text{lo}(x) \neq \text{lo}(y) \lor \text{hi}(x) \neq \text{hi}(y))$
3. $\forall x, y \in T (x \neq y \Rightarrow \text{value}(x) \neq \text{value}(y))$
Operations on MTBDDs

The algorithms and operations on BDDs from section 2.3.4 apply with small changes also to MTBDDs. For example, the operator $\star$ from apply is now interpreted as an arbitrary binary arithmetical operator. Some new operations are possible, among these are:

- Conversion of MTBDDs to BDDs
- Maximum terminal node value
- Scalar multiplication
- Matrix multiplication
- Matrix-vector multiplication

Details on these operators and their actual implementation can be found in [130].

2.3.6 Representing Transition Systems by Binary Decision Diagrams

In this section we will describe how labelled transition systems can be represented by means of BDDs and how stochastic labelled transition systems can be represented by means of MTBDDs.

Representing LTSs by ROBDDs

Definition 24 (Labelled Transition System (LTS)). Let $S$ be a finite set of states, let $s_0 \in S$ be the initial state and let $\text{Act}$ be a finite set of action labels. Let $\rightarrow \subseteq S \times \text{Act} \times S$ be the transition relation. We call $T = (S, \text{Act}, \rightarrow, s_0)$ a labelled transition system. If $(s, a, t) \in \rightarrow$ we write $s \xrightarrow{a} t$.

Definition 25 (Minterm function). Given $n$ distinct Boolean variables $a_1, \ldots, a_n$ and a Boolean vector $(b_1, \ldots, b_n)$ of length $n$, $MT(a_1, \ldots, a_n, b_1, \ldots, b_n)$ denotes the minterm consisting of $n$ literals, i.e.

$$MT(a_1, \ldots, a_n, b_1, \ldots, b_n) := a_1^* \land \ldots \land a_n^*$$

where $a_i^* = a_i$ if $b_i = 1$ and $a_i^* = (\neg a_i)$ if $b_i = 0$.

Definition 26 (Encoding function). Let $M$ be an arbitrary finite set. $Enc_M(m)$ denotes the injective encoding function that maps $m \in M$ to its binary encoding (a Boolean vector) of length $n$, i.e. $Enc_M : M \rightarrow \mathbb{B}^n$, $n \geq \lceil \log_2 |M| \rceil$. If $M$ is obvious from the context, the index of the encoding function can be omitted. We write $Enc_M(m) = m = (m_{n-1}, \ldots, m_0)$.

Definition 27 (Encoding sets). Let the length $n$ of an encoding be given. $PC$ is the set of all possible binary encodings, i.e. $PC := \mathbb{B}^n$. The set of used encodings $UC$ contains those elements of $PC$ that were already used to encode elements of a given set $M$, i.e. $UC := \{c \mid c \in PC \land \exists m \in M : (Enc_M(m) = c)\}$. The set of free encodings $FC$ contains those elements of $PC$ that are not in $UC$, i.e. $FC := PC \setminus UC$.

Definition 28 (Extension of a set of encodings by a leading binary digit). Let $C$ be a set of Boolean vectors of length $n$. $Ext_0(C)$ is obtained by adding a leading zero to the elements of $C$, i.e.:

$$Ext_0(C) = \{c' \mid c = 0 \circ c' \land c' \in C\}$$

Analogously we obtain $Ext_1(C)$ from $C$ by adding a leading one. The function $Ext(C)$ adds an arbitrary leading digit to the vectors in $C$, i.e. $Ext(C) = Ext_0(C) \cup Ext_1(C)$. 

Definition 29 (Choice of encoding). An element $c$ of a given set of encodings $C$ is chosen with respect to a total ordering relation $\preceq \succeq$ by the function $\text{Ch}(C, \preceq \succeq) := c \in C$ such that $\forall c' \in C : (c \preceq \succeq c')$.

Definition 30 (Transition encoding function). A transition $x \xrightarrow{a} y$ of an LTS can be encoded using the minterm function:

$$\text{TR}(x \xrightarrow{a} y) := \text{MT}(s, \text{Enc}_S(x)) \land \text{MT}(a, \text{Enc}_{\text{Act}}(a)) \land \text{MT}(t, \text{Enc}_S(y))$$

where $a$ denotes the vector of Boolean variables encoding the action, and $s$ and $t$ denote the vectors of Boolean variables encoding the source and target state of the transition. In the sequel $\text{TR}(x \xrightarrow{a} y)$ will be written as $\text{TR}(x, a, y)$.

Definition 31 (Symbolic Encoding of an LTS). Let $T = (S, \text{Act}, \rightarrow, s_0)$ be an LTS. Its symbolic encoding $T_{\text{symb}} = (\rightarrow_{\text{symb}}, s_{\text{symb}}^0)$ is defined as follows:

- $\rightarrow_{\text{symb}} := \bigvee_{(x \xrightarrow{a} y) \in \rightarrow} \text{TR}(x, a, y)$
- $s_{\text{symb}}^0 := \text{Enc}(s_0)$

In the sequel we assume that the BDD variables have the following ordering, denoted by $\prec$:

At the first $n_a \geq \lceil \log_2 |\text{Act}| \rceil$ levels from the root are the variables $a_i$ encoding the action. On the remaining levels we have $2 \ast n_s \geq 2 \ast \lceil \log_2 |S| \rceil$ variables encoding the source and target state of a transition. The source state variables ($s_i$) and the target state variables ($t_i$) are ordered in an interleaved fashion, which yields the following overall variable ordering$^7$:

$$a_0 \prec \ldots \prec a_{n_a-1} \prec s_0 \prec t_0 \prec \ldots \prec s_{n_s-1} \prec t_{n_t-1}$$

We will show by means of an example how to use BDDs to encode LTSs.

Example 13. Let the LTS from figure 2.17 be given. This system has the actions $\text{arr}$, $\text{serve}$,

![Fig. 2.17. Example LTS](#)

$\text{fail}$, $\text{repair}$, therefore we need two binary variables, $a_1$ and $a_0$, to encode them binarily:

$^7$ This interleaved ordering is the commonly accepted heuristics for obtaining small MTBDD sizes, see for instance [50, 59, 129].
The five states are encoded as follows:

- \( \text{Enc}(z_0) = 000 \)
- \( \text{Enc}(z_1) = 001 \)
- \( \text{Enc}(z_2) = 011 \)
- \( \text{Enc}(z_4) = 010 \)
- \( \text{Enc}(z_3) = 100 \)

For example the transition \( z_1 \xrightarrow{\text{fail}} z_4 \) is encoded binarily using function \( TR \) as follows:

\[
TR(z_1 \xrightarrow{\text{fail}} z_4) = MT(s, z_1) \land MT(a, \text{fail}) \land MT(t, z_4) = \\
MT(s, 000) \land MT(a, 10) \land MT(t, 010) = \\
\neg s_0 \land \neg s_1 \land \neg s_2 \land a_0 \land \neg a_1 \land \neg t_0 \land t_1 \land \neg t_2
\]

The BDD representation of the given LTS is shown in figure 2.18. ☐
Representing SLTSs by MTBDDs

Definition 32 (Stochastic Labelled Transition System (SLTS)). Let \( S \) be a finite set of states, let \( s_0 \in S \) be the initial state and let \( \text{Act} \) be a finite set of action labels. Let \( 
rightarrow \subseteq S \times \text{Act} \times \mathbb{R}^{>0} \times S \). We call \( T = (S, \text{Act}, \nrightarrow, s_0) \) a stochastic labelled transition system. If \( (s, a, \lambda, t) \in \nrightarrow \) we write \( s \xrightarrow{a, \lambda} t \).

An SLTS is essentially a CTMC, where the transitions are labelled by actions and rates. The notion of a path in an SLTS is defined as follows:

Definition 33 (Paths in SLTSs). A finite path \( \sigma \) in an SLTS \( T = (S, \text{Act}, \rightarrow, s_0) \) is a sequence \( s_0 \xrightarrow{a_1, \lambda_1} s_1 \xrightarrow{a_2, \lambda_2} \ldots s_{l-1} \xrightarrow{a_l, \lambda_l} s_l \) with \( l \in \mathbb{N}, s_i \in S, a_i \in \text{Act} \). Such a path has length \( l \) and \( \sigma[i] = s_i \) denotes its \((i+1)\)-st state. Let \( \text{Path}(s) \) denote the set of paths originating in \( s \).

Definitions 26 to 29 also apply to the MTBDD case, we will only give those definitions that are not identical to the ROBDD case.

The only difference is that since we are working with MTBDDs we use \( 1 - x \) instead of \( \neg x \) and multiplication \( \ast \) instead of conjunction \( \land \).

Definition 34 (Transition encoding function). A transition \( x \xrightarrow{a, \lambda} y \) of an SLTS can be encoded using the minterm function:

\[
\text{TR}(x \xrightarrow{a, \lambda} y) := MT(s, \text{Enc}_S(x)) \ast MT(a, \text{Enc}_\text{Act}(a)) \ast MT(t, \text{Enc}_S(y)) \ast \lambda
\]

where \( a \) denotes the vector of Boolean variables encoding the action, and \( s \) and \( t \) denote the vectors of Boolean variables encoding the source and target state of the transition. In the sequel \( \text{TR}(x \xrightarrow{a, \lambda} y) \) will be written as \( \text{TR}(x, a, \lambda, y) \).

Example 14. Let the SLTS from figure 2.19 be given. All encodings of states and actions

![Fig. 2.19. Example SLTS](image)

are the same as in example 13, therefore we restrict ourselves to show the final MTBDD representation of the given SLTS (cf. figure 2.20). ☐

2.4 Process Algebras

In this section we will briefly introduce syntax, semantics, and equivalence relations for both functional and stochastic process algebras. As a representative of functional process
algebras we briefly introduce LOTOS [25, 81], as this formalism is the basis on which the stochastic process algebra TIPP [61, 69] was developed. Other functional process algebras are CCS [62, 109], ACP [10, 55, 19], CSP [77, 124, 126]. From TIPP we derived a stochastic process algebra which serves as specification language for the stochastic model checker CASPA (see chapter 6). We will refer to this process algebra as YAMPA (Yet Another Markovian Process Algebra).

2.4.1 Syntax, Semantics, and Bisimulation for LOTOS

LOTOS is a very powerful process algebra of which two versions exist:
- Basic LOTOS, using the usual process algebraic operators
- Full LOTOS, which also comprises interprocess communication and data types that are definable in ASN.1 (Abstract Syntax Notation [48, 141]).

We will concentrate on Basic LOTOS, in the sequel shortly LOTOS, as we do not make use of interprocess communication and data types in this thesis.

Syntax of Process Algebras

The syntax of LOTOS is defined by the following grammar:

Definition 35 (Syntax of LOTOS). Let $X$ be a process variable, $\text{Act}$ a finite set of action names/labels, $a \in \text{Act}$ an action name, $L \subseteq \text{Act}$ a set of action names. LOTOS has two special actions:
• \( \tau \) is a special action, the so-called silent or invisible action. \( \tau \) is not element of Act.
• \( \delta \) denotes the successful termination of a process. \( \delta \) is not element of Act.
• \( \text{Act}^* := \text{Act} \cup \{ \tau \} \cup \{ \delta \} \).

LOTOS processes are defined as follows:

\[
P := \text{stop} \mid \text{exit} \mid a; P \mid P + P \mid P >> P \mid P|> P \mid P||P \mid \text{hide } a \text{ in } P \mid \text{rec } X : P \mid P[a/b]
\]

where the operators have the following meaning:
• \( P := \text{stop} \): describes deadlocking, i.e. inactive behaviour
• \( P := \text{exit} \): describes successful termination of a process
• \( P := a; Q \): Process \( P \) can perform action \( a \) and then behaves like process \( Q \).
• \( P := Q + R \): Process \( P \) behaves either as \( Q \) or \( R \).
• \( P := Q >> R \): \( P \) describes the sequential composition of \( Q \) and \( R \). At first, \( P \) behaves as \( Q \), after successful termination of \( Q \), which is denoted by action \( \delta \), \( P \) behaves like \( R \).
• \( P := Q|> R \): \( P \) describes interruption of process \( Q \) by \( R \). The execution of \( Q \) can be interrupted by \( R \) after any action performed by \( Q \), if \( R \) takes over control, \( Q \) cannot be resumed. In case of stop or successful termination of \( Q \), \( R \) is not executed.
• \( P := Q||R \): Process \( P \) describes synchronous parallel composition of processes \( Q \) and \( R \). Actions from \( L \) must be performed by both processes at the same execution step. Actions not in \( L \) can be performed by both processes independently.
• \( P := \text{hide } a \text{ in } Q \): In \( P \) occurrences of action \( a \) are hidden from the environment, i.e. each occurrence of \( a \) in \( Q \) is replaced by the special silent action \( \tau \). The purpose of this operation is to prevent synchronisation over \( a \) on parallel composition of processes. One can interpret this as hiding of internal behaviour from the environment (abstraction).
• \( P := \text{rec } X : Q \): This operator describes cyclic, infinite behaviour of process \( P \). In \( Q \) each occurrence of \( X \) is replaced by the definition of \( Q \) such that \( Q \) can be executed infinitely often. In practice, this is often described by having a process variable on both sides of a defining equation, for example:

\[
P := a; b; P
\]

This means that process \( P \) can perform actions \( a \) followed by \( b \) infinitely often
• \( P := Q[a/b] \): \( P \) describes relabelling, i.e. in process \( Q \) any occurrence of action \( b \) is replaced by \( a \).

**Semantics of Process Algebras**

Process Algebras have the advantage that they possess a rigorous formal semantics. Several types of semantics exist. We will give the semantics of Basic LOTOS in the usual structural operational semantics (SOS) style [122] (cf. fig. 2.21). All these rules have a common structure:

\[
\text{Premise}_1 \ldots \text{Premise}_n \quad \text{Conclusion} \quad \text{Sidecondition}_1 \ldots \text{Sidecondition}_m
\]

In words, this can be expressed as follows:

If all premises and all side conditions are satisfied, we can draw the given conclusion.
Applied to the semantic rules, premises and conclusions make claims about the possible process behaviour. We will clarify this by means of three examples:

**Example 15.**
- **Prefix rule:**
  \[
  \frac{a; P}{P} \quad \text{or} \quad \frac{a; P}{P'}
  \]
  Here, the set of premises and side conditions is empty, i.e. any process \( P \) that is capable of performing an action \( a \) can do this.
- **Choice:**
  \[
  \frac{P}{P + Q} \quad \text{or} \quad \frac{Q}{P + Q'}
  \]
  Whenever process \( P \) (\( Q \)) is able to perform action \( a \) and behaves afterwards as \( P' \) (\( Q' \)), then process \( P + Q \) can do the same.
- **Hiding:**
  \[
  \frac{a; P}{P'} \quad \text{or} \quad \frac{a; Q}{Q'}
  \]
  Here, we have a rule that possesses besides its premise also a side condition: If process \( P \) can do action \( a \) and action \( a \neq b \), then process \( \text{hide } b \) in \( P \) can perform action \( a \). As \( a \neq b \), \( a \) is not replaced by \( \tau \).

Applying these rules, any process algebraic specification can be mapped to an LTS, which is its semantic model. On this model functional verification is carried out.
Example 16. Let \( P := a; (b; c; \text{stop} + d; P) \) be given. We want to derive \( P \)'s semantic model, i.e. its underlying LTS:

1. In the first derivation step, prefix is applicable:

\[
a; (b; c; \text{stop} + d; P) \xrightarrow{a} (b; c; \text{stop} + d; P)
\]

In fig. 2.22 we find the corresponding LTS:

Fig. 2.22. LTS for \( a; (b; c; \text{stop} + d; P) \)

2. For \( P' := (b; c; \text{stop} + d; P) \) choice is applicable, which leads to two possible transitions in the LTS (cf. fig. 2.23):

\[
\begin{align*}
(b; c; \text{stop} &\xrightarrow{b} c; \text{stop} \\
d; P &\xrightarrow{d} P
\end{align*}
\]

Fig. 2.23. LTS for \( (b; c; \text{stop} + d; P) \)

3. In the third step, for \( P'' := c; \text{stop} \), only prefix is applicable:

\[
c; \text{stop} \xrightarrow{c} \text{stop}
\]

The final LTS can be found in fig. 2.24

An Equivalence Relation for Process Algebras

For reasoning about systems it is useful to have a notion of equivalence. In fact, there exists a huge number of equivalences for process algebras. The most commonly used are trace equivalence, failure equivalence, testing equivalence, simulation and strong and weak bisimulation, see [139] for a comprehensive treatment of notions of equivalence.

In this thesis we concentrate on strong bisimulation. Bisimulation is used in this thesis for two purposes:
1. Checking whether two processes are equivalent with respect to strong bisimulation
2. Reducing the state space, by replacing several states that are equivalent with respect to strong bisimulation by a single one.

For the first application of bisimulation, checking whether two processes are equivalent, a bisimulation is defined as follows:

**Definition 36 (Strong Bisimulation).** A binary relation $B \subseteq P \times P$, where $P$ is the set of all possible processes is a strong bisimulation, if $(P, Q) \in B$ implies:

- if $P \xrightarrow{a} P'$ then there is $Q'$ such that $Q \xrightarrow{a} Q'$ and $(P', Q') \in B$
- if $Q \xrightarrow{a} Q'$ then there is $P'$ such that $P \xrightarrow{a} P'$ and $(P', Q') \in B$

Two processes are bisimilar (written $P \sim Q$), if they are contained in the same bisimulation.

**Lemma 1 (Properties of $\sim$).** The relation $\sim$ has the following properties:

- Reflexivity: $P \sim P$
- Transitivity: $P \sim Q \land Q \sim R \implies P \sim R$
- Symmetry: $P \sim Q \implies Q \sim P$.

i.e. $\sim$ is an equivalence relation.

**Example 17.** Let the processes $P$ and $Q$ be defined as follows:

$$
P := a; (b; \text{stop} + c; \text{stop})
$$

$$
Q := a; b; \text{stop} + a; c; \text{stop}
$$

To check whether $P \sim Q$ we have to try to give a binary relation $B$ that is a bisimulation. Assume $(P, Q) \in B$, this implies:

- if $P \xrightarrow{a} (b; \text{stop} + c; \text{stop})$ then there is $Q'$ such that $Q \xrightarrow{a} Q'$ and $(P', Q') \in B$
- if $Q \xrightarrow{a} b; \text{stop}$ then there is $P'$ such that $P \xrightarrow{a} P'$ and $(P', Q') \in B$

Obviously, $(P', Q') = ((b; \text{stop} + c; \text{stop}), c; \text{stop})$. For $(P', Q')$ to be in $B$ it must hold:

- if $(b; \text{stop} + c; \text{stop}) \xrightarrow{b} P''$ then there is $Q''$ such that $c; \text{stop} \xrightarrow{b} Q''$

This condition is obviously violated, as $c; \text{stop}$ is not capable of any $b$ transition, i.e. $P$ and $Q$ are not bisimilar. ☒
2.4 Process Algebras

The second purpose of bisimulation equivalence yields a slightly different interpretation and therefore a different definition of bisimulation equivalence: Bisimulations are useful for reducing the state space of a given transition system, by replacing each class of equivalent states by a single macro state. In that case, the carrier set of the bisimulation relation is the state space \( S \) of the transition system to be reduced.

**Definition 37.** A binary relation \( B \) on a set of states of an LTS is a strong bisimulation, if \((s_1, s_2) \in B\) implies:

- if \( s_1 \xrightarrow{a} s'_1 \) then there is a \( s'_2 \) such that \( s_2 \xrightarrow{a} s'_2 \) and \((s'_1 s'_2) \in B\)
- if \( s_2 \xrightarrow{a} s'_2 \) then there is a \( s'_1 \) such that \( s_1 \xrightarrow{a} s'_1 \) and \((s'_1 s'_2) \in B\)

Two states are bisimilar if they are contained in the same bisimulation. Let \( s_1 \) and \( s_2 \) be two bisimilar states, we then write \( s_1 \sim_M s_2 \).

**Example 18.** In fig. 2.25 (a) we can find the LTS of process \( Q \), in fig. 2.25 (b) its minimised variant. States \( s_2 \) and \( s_3 \) are bisimilar, as we can find for any transition emanating from them a pair of states that is bisimilar: We assume \((s_2, s_3) \in B\):

- For \( s_2 \xrightarrow{c} s_3 \), and \( s_3 \xrightarrow{c} s_2 \), we have \((s_2, s_3) \in B\), by initial assumption.
- For \( s_2 \xrightarrow{b} s_1 \), and \( s_3 \xrightarrow{b} s_1 \), we have \((s_1, s_1) \in B\), which trivially holds.

\( \implies \) \( s_2 \sim s_3 \).

Analoguously, for states \( s_4 \) and \( s_5 \) it holds that \( s_4 \sim s_5 \). In fig. 2.25 (b) we have replaced the bisimilar states \( s_2 \) and \( s_3 \) and \( s_4 \) and \( s_5 \) by a representative state \( s_2 \) resp. \( s_4 \).

2.4.2 Syntax, Semantics, and Equivalence for YAMPA

Stochastic process algebras are an extension of functional process algebras in the same way as stochastic Petri nets form an extension of Petri nets to reason about the performance of a modelled system.

The purpose of stochastic process algebras is threefold:

- stochastic process algebras shall provide a unified framework to reason both about functional and performance aspects of a system model
they may help to integrate performance evaluation into early stages of system design
they exploit the advantages of functional process algebras such as abstraction, constructivity, rigorous formal semantics etc.

In the last decade a number of stochastic process algebras have been developed such as MPA [32], EMPA [22], PEPA [76] and TIPP [61, 69]. In this thesis we focus on YAMPA.

**Syntax of Stochastic Process Algebras**

We will now present the syntax of YAMPA:

**Definition 38 (Syntax of YAMPA).** For a set of actions \( \text{Act} \), let \( a \in \text{Act}^* \) and \( b \in \text{Act} \). Let \( L \subseteq \text{Act} \) be a set of visible actions, let \( \lambda \in \mathbb{R}_{>0} \) be a rate, and let \( X \in \text{Var} \) be a process variable.

YAMPA is the language whose terms are given by the following grammar:

\[
P := \text{stop} | \text{exit} | (a, \lambda); P | P + P | P > > P | P| > P \\
P||[L]|P | P|||P | \text{hide} \ b \ \text{in} \ P | \text{rec}X : Q | P[a/b]
\]

The only differences between definition 38 and 35 are \((a, \lambda); P\), i.e. the prefix operator, and parallel composition, their meaning can be described as follows:

- **P := (a, \lambda); Q:** After an exponentially distributed delay, governed by rate \( \lambda \), action \( a \) can be performed instantaneously, afterwards the process behaves as \( Q \).
- **The rule of synchronising parallel composition is different from the functional case.** Here, we have to take the timed and stochastic behaviour into account. So, we need both a plausible synchronisation policy and a policy that preserves the Markov property of the new process. In the case of YAMPA the synchronisation policy is multiplication\(^8\).

**Semantics of Stochastic Process Algebras**

The semantics of YAMPA can be defined in SOS-style. Here, we restrict ourselves to the semantic rules that are different from that of LOTOS, see fig. 2.26

We have to note, that the semantic model of stochastic process algebras, SLTSs (cf. def. 32), has to be adopted, such that the transition relation is a multi-set, that allows several entries:

\((s_1, a, \lambda_1 t_1)\) and \((s_2, a, \lambda_2 t_2)\)

with \( s_1 = s_2 \) and \( t_1 = t_2 \). It is both possible that \( \lambda_1 = \lambda_2 \) and \( \lambda_1 \neq \lambda_2 \). This means we extend SLTSs to so called multi-transition systems. We have to require this, because an SPA process of the kind \( P := (a, \lambda); Q + (a, \lambda); Q \) must have a semantic model that represents both transitions, to preserve the stochastic behaviour. The exit rate from \( P \) is \( 2 \ast \lambda \), not \( \lambda \) as it would be the case if we do not allow multiple transitions in the SLTS.

\(^8\) In other stochastic process algebras different policies are chosen.
An Equivalence Relation for Stochastic Process Algebras

Here, we will introduce the important equivalence relation of strong Markovian Bisimulation. We will give a definition that suits the application of bisimulation in this thesis: Minimising the state space of a given process. Informally, two states are Markovian bisimilar if from both states the same equivalence classes can be reached in one step by the same actions and with the same “cumulative rate”. Markovian bisimulation can be seen as a refinement of Markov chain lumpability [31], by distinguishing between different action names. Formally:

**Definition 39 (Cumulative rate \(\gamma\)).** The cumulative rate from a state \(s \in S\) by action \(a \in \text{Act}\) to a set of states \(C \subseteq S\) is defined by the function

\[
\gamma(s,a,C) = \sum \{ |\lambda | s \xrightarrow{a\lambda} s' \wedge s' \in C \}
\]

where \(\{\}\) and \(\[]\) denote multiset brackets. This notation was taken from [61].

**Definition 40 (Markovian bisimulation).** A binary relation \(B\) on the set of states \(S\) of an SLTS is a (strong) Markovian bisimulation, if \((s_1, s_2) \in B\) implies that for all equivalence classes \(C\) of \(B\) and all actions \(a\) it holds that

\[
\gamma(s_1, a, C) = \gamma(s_2, a, C)
\]

Two states \(s_1\) and \(s_2\) are Markovian bisimilar (written \(s_1 \sim_M s_2\)) if they are contained in a Markovian bisimulation.

If \(C_1 = \{s_1, s_2, \ldots\}\) and \(C_2\) are equivalence classes of a Markovian bisimulation \(B\) we sometimes write \(\gamma(C_1, a, C_2)\) instead of \(\gamma(s_i, a, C_2)\), knowing that the cumulative rate is the same for all \(s_i \in C_1\). Given all Markovian bisimulations \(B_1, B_2, \ldots\) on the same SLTS, one is typically interested in the largest (i.e. coarsest) one, namely \(B = \bigcup_i B_i\).

**Example 19.** In fig. 2.27 (a) we find the SLTS of process \(Q\), and in fig. 2.27 (b) we can see its minimised variant.

![Diagram](image-url)
2.5 The Logic PDL

PDL (propositional dynamic logic) [54, 87, 66] is a representative of logics of programs, i.e. of logics with which one can reason about programs. The purpose of such logics for example is formalisation of correctness specifications and proving that such specifications are met by a particular program. Further examples of applications of logics of programs are proving that two programs are equivalent, comparing the expressive power of various program operators etc.

PDL is a modal logic (in fact it is even a multimodal logic). The basic idea behind all modal logics is to give up the “static” interpretation of truth of classical first order logic, where truth of a formula is considered to be immutable. In modal logic a “dynamic” notion of truth is considered, i.e. the truth of a formula can change over time.

Formulæ of first order logic are interpreted over a single, fixed semantic model (structure or “world” or state), in which a formula is true or not. In modal logic semantic interpretation is over a set of possible worlds. One can think of such worlds as states, in which certain formulæ are true, others not, and the set of true formulæ usually can vary from state to state. As modal logics have several possible worlds and truth is considered to be mutable, the notion of an accessibility relation between the various worlds is crucial, i.e. how can one transit from one world to the other. In fact modal logics (among other criteria, such as axiomatisation) can be distinguished by the type of accessibility or transition relation they allow (reflexive, symmetric, transitive, linear, i.e. each state has a single successor, branching, i.e. each state can have several successors). In general, one can say that two worlds $s$ and $t$, are connected, if they are in the transition relation.

How do modal logics interplay with logics of programs? The dynamic interpretation of truth is familiar from program execution, where normally each execution step of a program changes its states, i.e. the values of variables, the program’s control structures, etc. The set of states of modal logics can be seen as the set of all possible execution steps of a program. Over this set a ternary accessibility relation $R$ can be defined. If $(s, \rho, t) \in R$, where $s$ and $t$ are states and $\rho$ is a program then execution of $\rho$ starts in $s$ and one possible final state of $\rho$ is $t$, as we consider also non-deterministic programs there may be several such final states.

Syntactically, each program is a modality of the multimodal logic. Programs are written inside the classical modal operators $\Box$ (necessity) and $\Diamond$ (possibility): $[\rho]$, $<\rho>$.

- $[\rho] \phi$ can be read as: every terminating execution of $\rho$ ends in a state that satisfies $\phi$
- $<\rho> \phi$ can be read as: it is possible to execute $\rho$ and execution of $\rho$ ends in a state that satisfies $\phi$.

Besides operators to construct complex programs from simpler ones, PDL possesses means to model the behaviour of programming language constructs, such as if-then-else, while etc.

2.5.1 Syntax of PDL

The elements of PDL are: propositional logic, modal logic, and algebra of regular expressions. PDL has two types of expressions: formulæ and programs. The operators of PDL can be divided into four categories:

- propositional logic operators: $\neg$ (negation), $\lor$ (disjunction)
- regular expression operators: $;$ (sequence), $+$ (choice), $*$ (finite looping)
- modal operators: $<[>, (possibility, enriched with programs)$
• test operator: ? (test)

Formally, the syntax of PDL is defined as follows.

Definition 41 (Syntax of PDL). Let $AP$ be a set of atomic propositions, $Act$ a finite set of atomic programs or actions. Let $p \in AP$ be an atomic proposition, $a \in Act$ an atomic program, then PDL formulae are defined by the following grammar:

$$\phi := p \mid \neg \phi \mid \phi \lor \phi \mid \langle \rho \rangle \phi$$

where $\rho$ is a program defined by the following grammar:

$$\rho := \epsilon \mid a \mid \rho; \rho \mid \rho^* \mid \phi?; \rho$$

$\epsilon$ is the empty program.

2.5.2 Semantics of PDL

Informal Semantics of PDL

Before we give the formal semantics of PDL, we will describe the intuitive meaning of some of the PDL-constructs.

• $\rho_1; \rho_2$: Execute $\rho_1$ and $\rho_2$ sequentially
• $\rho_1 + \rho_2$: Choose nondeterministically $\rho_1$ or $\rho_2$ and execute the chosen program
• $\rho_1^*$: Execute $\rho_1$ a non-deterministically chosen finite number of times, including zero times
• $\langle \rho_1 \rangle \Phi$: It is possible to execute $\rho_1$ and halt in state satisfying $\Phi$
• $[\rho_1] \Phi$: Although not explicitly present in PDL, the box is the dual of the diamond and can be expressed as follows:

$$[\rho_1] \Phi \equiv \neg \langle \rho_1 \rangle \neg \Phi$$

This means it is not possible to execute $\rho_1$ and end up in a state that does not satisfy $\Phi$. Equivalently, for every terminating computation of $\rho_1$ it holds that $\rho_1$ halts or stops in a state satisfying $\Phi$. Note, that for the satisfaction of a formula of this kind no terminating computation at all needs to exist.

• $\Phi?; \rho_1$: Test, if $\Phi$ holds in the current state, if so, execute $\rho_1$ otherwise fail. The formula $\Phi$ can also be referred to as test formula.

Formal Semantics of PDL

PDL formulae and programs are interpreted over a Kripke-structure $M$, which is defined as follows.

Definition 42 (Kripke Structure). A Kripke structure is a quadruple:

$$M = (S, Act, L, R)$$

where

• $S$ is a finite set of states
• $Act$ is a finite set of actions
The semantics of PDL programs is defined using the generalised meaning function \( \mathcal{I} \):

**Definition 43 (Formal Semantics of PDL).** Let \( \mathcal{M} \) be a Kripke structure, \( p \) an atomic formula, \( \phi, \psi \), PDL formulae, \( a \) an atomic program, let \( \rho, \rho_1, \rho_2 \) be PDL programs, then the formal semantics of PDL formulae is given by the following relation:

\[
\begin{align*}
\mathcal{I}_f(p) & := \{ s \mid p \in L(s) \} \\
\mathcal{I}_p(\epsilon) & \subseteq \{ (s,s) \mid s \in S \} \\
\mathcal{I}_p(a) & \subseteq \{ (s,s') \mid (s,a,s') \in R \} \\
\mathcal{I}_f(\neg \phi) & := S \setminus \mathcal{I}_f(\phi) \\
\mathcal{I}_f(\phi \lor \psi) & := \mathcal{I}_f(\phi) \cup \mathcal{I}_f(\psi) \\
\mathcal{I}_f((p)\phi) & := \{ s \mid \exists s' \in S((s,s') \in \mathcal{I}_p(\rho) \land s' \in \mathcal{I}_f(\phi)) \} \\
\mathcal{I}_p(\rho_1;\rho_2) & := \mathcal{I}_p(\rho_1) \circ \mathcal{I}_p(\rho_2) := \{ (s,s') \mid \exists \rho \in \mathcal{I}_p(\rho_1) \land (w,s') \in \mathcal{I}_p(\rho_2) \} \\
\mathcal{I}_p(\rho^*) & := \bigcup_{n \geq 0} \mathcal{I}_p(\rho^n) \\
\mathcal{I}_p(\phi?) & := \{ (s,s) \mid s \in \mathcal{I}_f(\phi) \}
\end{align*}
\]

where:

- \( \mathcal{I}_p \) is the interpretation function for programs
- \( \mathcal{I}_f \) is the interpretation function for formulae

We can think of the interpretation function \( \mathcal{I}_f(\phi) \) as the set of states that satisfy \( \phi \) in \( \mathcal{M} \) and of \( \mathcal{I}_p(\rho) \) as the set of initial and final states of the program \( \rho \). The operator \( \circ \) denotes relational composition.

**Examples for PDL**

**Example 20 (Program Operators).** We can derive the following program operators, where we assume that \( \text{true} \in \mathcal{AP} \land \text{false} \in \mathcal{AP} \).

- **skip** := \( \text{true} \); \( \epsilon \)
- **fail** := \( \text{false} \); \( \epsilon \)
- **case** \( \phi_1 \rightarrow \rho_1 \mid \ldots \mid \phi_n \rightarrow \rho_n := \phi_1 ?; \rho_1 + \ldots + \phi_n ?; \rho_n \)
- **do** \( \phi_1 \rightarrow \rho_1 \mid \ldots \mid \phi_n \rightarrow \rho_n \text{od} := (\phi_1 ?; \rho_1 + \ldots + \phi_n ?; \rho_n)^*; (\neg \phi_1 \land \ldots \land \neg \phi_n)?; \epsilon \)
- **if** \( \phi \text{ then } \rho_1 \text{ else } \rho_2 \) := \( \phi?; \rho_1 + \neg \phi?; \rho_2 \)
- **while** \( \phi \text{ do } \rho \text{ od} := (\phi?; \rho)^*; \neg \phi?; \epsilon \)
- **repeat** \( \rho \text{ until } \phi := \rho; (\neg \phi?; \rho)^*; \phi?; \epsilon \)

We will now illustrate the “informal” semantics of if-then-else-operator and the while-loop. Let the model in figure 2.28 be given. Some example PDL formulae are

- \( \Phi_1 = \langle \text{while } \neg \phi \text{ do } (a + b); c \text{ od } \rangle \phi \) To satisfy \( \Phi_1 \) there must be a path that visits states in which \( \neg \phi \) is satisfied at least one \( a \) action or one \( b \) followed by a \( c \) action must be executable. The states \( s_4 \) to \( s_7 \) and \( s_0 \) do not satisfy this formula, as in these states neither \( a \) nor the sequence \( b;c \) is executable.
• \( \Phi_2 = < \text{if } \xi \text{ then } d; \neg \phi \text{? else } \phi \text{?; e } \text{fi} > \text{true} \): To satisfy \( \Phi_2 \) there must be a path that starts in a state in which \( \xi \) is satisfied, and terminates in a state in which \( \neg \phi \) is satisfied after executing the atomic program \( d \). This part (if-branch) of the program is satisfied by state \( s_8 \). The else-branch of the formula is satisfied by state \( s_{10} \), as \( \xi \) is not satisfied, but \( \phi \) and action \( e \) is executable.

2.6 The Logic CTL

In this very brief section we will only give a short introduction to the logic CTL to make section 2.7 more readable. For an exhaustive overview on CTL we refer to [85].

2.6.1 Introduction

The logic CTL (computation tree logic) is a very important representative of the class of temporal logics. Although the term temporal suggests it, it is not possible to describe the quantitative timing behaviour, this is done using real-time logics, of a system but to describe the order of properties of states that occur along a path through the LTS.

2.6.2 Syntax and Semantics

Here, we will define the syntax of CTL and give an informal interpretation of the meaning of CTL formulae.

Syntax of CTL

CTL contains propositional logic, temporal and path operators.
**Definition 44 (Syntax of CTL).** Let \( q \in \text{AP} \) be an atomic formula. CTL state formulae \( \Phi \) are defined as follows:

\[
\Phi := q \mid \neg \Phi \mid \Phi \lor \Phi \mid E\phi \mid A\phi
\]

CTL path formulae are defined using the following grammar:

\[
\phi := X\Phi \mid F\Phi \mid G\Phi \mid \Phi U \Psi
\]

**Informal Semantics of CTL**

Informally the respective operators have the following meaning:

- **E\( \phi \):** The formula \( E\phi \) is satisfied if, given a specific model, and an initial state \( s \), at least one path emanating from \( s \) in the given model satisfies \( \phi \).
- **A\( \phi \):** The formula \( A\phi \) is satisfied if, given a specific model, and an initial state \( s \), all paths emanating from \( s \) in the given model satisfy \( \phi \).
- **X\( \Phi \):** Given a specific model, and a path \( \sigma \) with initial state \( s \), \( \sigma \) satisfies this formula, if a successor state of \( s \) satisfies \( \Phi \).
- **F\( \Phi \):** Given a specific model, and a path \( \sigma \) with initial state \( s \), \( \sigma \) satisfies this formula, if there is a state \( s' \) on \( \sigma \), that satisfies \( \Phi \).
- **G\( \Phi \):** Given a specific model, and a path \( \sigma \) with initial state \( s \), \( \sigma \) satisfies this formula, if all states on \( \sigma \) satisfy \( \Phi \).
- **\( \Phi U \Psi \):** Given a specific model, and a path \( \sigma \) with initial state \( s \), \( \sigma \) satisfies this formula, if there is an \( s' \) on \( \sigma \) that satisfies \( \Psi \), and all states before \( s' \) satisfy \( \Phi \).

2.7 The Logic CSL

In this section we will introduce the logic CSL [6, 15] (Continuous Stochastic Logic). CSL is a formal language that provides means to express and verify weak real-time properties. CSL extends CTL in various ways:

- The semantic model is a state-labelled CTMC.
- It possesses time-bounded next (\( X^I \)) and time-bounded until (\( U^I \)) operators. \( I = [t, t'] \) is a time interval that determines the time bounds within which path formulae like \( \phi = \Phi U^I \Psi \) have to be satisfied. A path satisfies \( \phi \) if at a time point that lies within \( I \), the current state of the path is a \( \Psi \) state. All preceeding states have to satisfy \( \Phi \).
- The path quantifiers A and E are replaced by a probabilistic path quantifier \( P_{\circ p} \), where \( \circ p \in \{ <, \leq, >, \geq \} \). A formula \( P_{\circ p}(\phi) \) is valid in a state \( s \), if the probability mass of all paths satisfying \( \phi \) is within \( \circ p \).
- CSL provides a steady-state operator \( S_{\circ p} \) to reason about the the stationary system behaviour. \( S_{\circ p}(\Phi) \) is satisfied if the stationary probability to reside in a state that satisfies \( \Phi \) is within \( \circ p \).

2.7.1 State-Labelled Continuous-Time Markov Chains

In this section we will introduce the semantic model of CSL which is a state-labelled continuous-time Markov chain (SMC).
Definition 45 (State-Labelled Continuous-Time Markov Chains). A state labelled CTMC (SMC) $M$ is a triple:

$$M = (S, R, L)$$

where:

- $S$ is a finite set of states
- $R \subseteq S \times \mathbb{R} \times S$ is the transition relation
- $L : S \mapsto 2^{2^{\mathcal{AP}}}$ a state labelling function that associates with each state a set of atomic propositions that are true in this state.

In the sequel it is useful to assume that each state $s$ can be characterised/identified by an atomic proposition $q_s$ that is true only in $s$. If $S' \subseteq S$, then $q_{S'}$ is defined as the disjunction over all $q_{s'}$, $s' \in S'$, i.e.

$$q_{S'} = \bigvee_{s' \in S'} q_{s'}$$

Example 21 (A 4-Place Buffer with Erroneous Arrivals). We will model a 4-place buffer $M$ as an SMC. Let the labelling function $L$ be defined as follows:

- $L(s_1) = \{\text{empty}, \text{errorfree}\}$
- $L(s_2) = L(s_3) = L(s_4) = \{\text{errorfree}\}$
- $L(s_5) = \{\text{full}\}$
- $L(s_6) = L(s_7) = L(s_8) = L(s_9) = \{\text{error}\}$
- $L(s_{10}) = \{\text{failure}\}$

The functionality of $M$ can roughly be described as follows:

The system, as modelled in fig. 2.29 receives four data packets, then it processes all four packets. Data packets can be either error-free or erroneous. Error-free packets have all arrival rate $\lambda$. The arrival rate of erroneous packets is $\mu$. An error in a data packet can be corrected (transitions with rate $\gamma$), or not (rate $\delta$). If the error is not corrigible the buffer is emptied and all data packets received so far have to be retransmitted (transitions with rate $\kappa$). If all data packets are error free or contain only corrigible errors, the received data can be processed with rate $\omega$. Now, the system can again receive four data packets.

Fig. 2.29. State labelled Markov model of a 4-place buffer
Paths in SMCs

**Definition 46 (Paths in SMCs).** Let $\mathcal{M}$ be an SMC. An infinite path $\sigma$ is a sequence $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots s_i \xrightarrow{t_i} \ldots$, where for all $i \in \mathbb{N}$, $s_i \in S$, $t_i \in \mathbb{R}$, such that $R(s_i, s_{i+1}) > 0$ for all $i$. A finite path is a sequence $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots s_n \xrightarrow{t_n}$ such that $s_n$ is absorbing and $R(s_i, s_{i+1}) > 0$ for all $i < n$.

Let $\sigma$ be an infinite path, $i \in \mathbb{N}$, then $\sigma[i] = s_i$ is the $i$-th state on $\sigma$. $\tau(\sigma, i) = t_i$ is the sojourn time in $s_i$. For $t \in \mathbb{R}_{\geq 0}$ and $i$ the smallest index for which $t \leq \sum_{j=0}^{i-1} t_j$ let $\sigma[t] = \sigma[i]$ be the state of $\sigma$ at time point $t$.

For finite $\sigma$, ending in state $s_n$, $\sigma[i]$, $\tau(\sigma, i)$ are only defined for $i \leq n$. For $i < n$ the definitions are just like in the case of infinite paths. For $i = n$ $\tau(\sigma, n) = \infty$, for $t > \sum_{j=0}^{n-1} t_j$ we have $\sigma[t] = s_n$.

$\text{PATH}^M$ is the set of all finite and infinite paths in $\mathcal{M}$, $\text{PATH}^M(s)$ is the set of all paths with initial state $s$.

Stationary and Transient State Probabilities

On SMCs we can define both transient and stationary state probabilities.

**Definition 47 (Transient State Probabilities).** Transient probabilities on SMC $\mathcal{M}$ are defined via the probability measure $\Pr$ over paths $\text{Path}^M(s)$:

$$\pi^M(\alpha, s', t) = \Pr_\alpha(\sigma \in \text{Path}^M) | \sigma[t] = s'$$

$\pi^M(\alpha, s', t)$ is the probability to be in state $s'$ at time point $t$, given initial distribution $\alpha$. If $\alpha := (1, 0, \ldots, 0)$, we simply write $\pi^M(s, s', t) = \Pr(\sigma \in \text{Path}^M(s)) | \sigma[t] = s'$, given $s$ is the state with initial probability 1. $\pi^M$ is the state probability vector of dimension $|S|$.

To compute the transient state probabilities at arbitrary time points we have to solve the Chapman-Kolmogorov differential equation system, cf. section 2.1.

**Definition 48 (Steady State Probabilities).** Stationary or steady state probabilities are characterised as follows:

$$\pi^M(\alpha, s') = \lim_{t \to -\infty} \pi^M(\alpha, s', t)$$

Let $S' \subseteq S$, $\pi^M(\alpha, S', t) = \sum_{s \in S'} \pi^M(\alpha, s', t)$, then

$$\pi^M(\alpha, S') = \lim_{t \to -\infty} \Pr(\sigma \in \text{Path}^M) | \sigma[t] \in S')$$

Stationary state probabilities can then be computed as described in section 2.1.

2.7.2 Syntax and Semantics of CSL

Syntax of CSL

**Definition 49 (Syntax of CSL).** Let $p \in [0, 1]$, $q \in AP$, and $\in \{ \leq, <, \geq, > \}$. State formulae of CSL are defined by the following grammar:

$$\Phi ::= q \mid \neg \Phi \mid \Phi \lor \Phi \mid S_{\text{cop}}(\Phi) \mid P_{\text{cop}}(\phi)$$

where $\phi$ is a path formula that is defined as follows:

$$X^1 \phi \mid \Phi U^1 \Psi$$

$I = [t, t']$ is a closed time interval with $t \geq 0$ and $t' \neq 0$. 

Intuitively, the CSL formulae have the following meaning:

- $S_{\text{cop}}(\Phi)$: The stationary state probability to be in a state that satisfies $\Phi$ is within $\gg p$.
- $P_{\text{cop}}(\Phi)$: The probability measure of all paths, satisfying $\phi$ lies within the bounds imposed by $\gg p$. For a discussion on this topic see [15].
- $X^I\phi$ is the time bounded variant of the next-operator known from CTL. The path formula $X^I\phi$ expresses that a state that satisfies $\phi$ must be reached after a passage of at least $t$ and at most $t'$ time units, if $I = [t, t']$.
- $\Phi U^I \Psi$ is the time bounded variant of the CTL until-operator. $\Phi U^I \Psi$ expresses that a state satisfying $\Psi$ must be reached at a time point that lies within $I$. All states visited before such a $\Psi$ state must satisfy $\Phi$.

**Definition 50 (Length of CSL Formulae).** Let $q \in AP$, $\Phi$ and $\Psi$ be state formulae, and $\phi$ a path formula. Then, the length of a CSL formula is defined as follows:

$$
\begin{align*}
|q| & := 1 \\
\lnot \Phi & := |\Phi| + 1 \\
|\Phi \lor \Psi| & := |\Phi| + |\Psi| + 1 \\
S_{\text{cop}}(\Phi) & := |\Phi| + 1 \\
P_{\text{cop}}(\phi) & := |\phi| + 1 \\
|\phi| & = |\Phi U^I \Psi| := |\Phi| + |\Psi| + 1 \\
|\phi| & = |X^I \phi| := |\Phi| + 1
\end{align*}
$$

**Semantics of CSL**

The semantics of CSL is defined over a binary satisfaction relation “$|$”.

**Definition 51 (Semantics of CSL).** Let $M$ be an SMC, $s \in S$, $q \in AP$, the semantics of CSL state formulae is defined as follows:

$$
\begin{align*}
M, s & | = q \iff q \in L(s) \\
M, s & | = \lnot \Phi \iff M, s \not| \Phi \\
M, s & | = \Phi \lor \Psi \iff M, s | = \Phi \lor M, s | = \Psi \\
M, s & | = S_{\text{cop}}(\Phi) \iff \pi^M(s, \text{Sat}(\Phi)) \gg p \\
M, s & | = P_{\text{cop}}(\phi) \iff \text{Prob}^M(s, \phi) \gg p
\end{align*}
$$

where $\text{Sat}(\Phi) = \{ s \in S | M, s | = \Phi \}$ is the set of all states that satisfy $\Phi$. A state $s$ satisfies $S_{\text{cop}}(\Phi)$, if the probability $\pi^M(s, \text{Sat}(\Phi))$ to reside in equilibrium in a $\Phi$-state is within $\gg p$.

$\text{Prob}^M(s, \phi)$ is defined as follows:

$$
\text{Prob}^M(s, \phi) := \text{Pr}(\sigma | \Phi \in L(s) | M, \sigma | = \phi)
$$

The satisfaction relation for path formulae is defined as follows:

$$
\begin{align*}
M, s & | = X^I \phi \iff \exists t_2 \in I(M, \sigma \oplus t_2 | = \Phi \land \sigma \oplus t_2 = \sigma[1]) \\
M, s & | = \phi U^I \Psi \iff \exists t_2 \in I(M, \sigma \oplus t_2 | = \Psi \land \forall t_1 \in [0, t_2)(M, \sigma \oplus t_1 | = \Phi))
\end{align*}
$$

The temporal operators $F^I$ (finally) and $G^I$ (globally) can be derived using the $U^I$ operator:

$$
\begin{align*}
F^I \Phi & := \text{true} U^I \Phi \\
G^I \Phi & := \lnot F^I \lnot \Phi
\end{align*}
$$
Fig. 2.30. Model Checking Algorithm for CSL

Example 22 (Properties of the 4-Place Buffer). Here, we will present some example properties of the model from example 21

- \( \Phi_1 \): \( P_{<0.03}(\neg \text{full} \cup [2,10] \text{full}) \): Is the probability to reach state \( s_5 \) in at least 2 and at most 10 time units at most 0.3?
- \( \Phi_2 \): \( P_{>0.95}(X^{[0,3]} \text{error}) \): Is the probability to reach within at most 3 time units in one step an error state greater than 5 percent?
- \( \Phi_3 \): \( P_{>0.75}(\neg \text{full} \land \neg \text{error} \cup [2.8,4] \text{full}) \): Is the probability to reach state \( s_5 \) between 2 and 8.4 time units greater than 0.75, given all data packets are error free?
- \( \Phi_4 \): \( S_{>0.95}(\Phi_3) \): When residing in a steady state the system must satisfy property \( \Phi_3 \) with a probability of at least 95 percent.

\( \Box \)

2.7.3 Model Checking CSL

The model checking core algorithm (fig. 2.30) is similar to the CTL procedure [85]. The state formula \( \Phi \) is split into its subformulas \( \Phi_i \), for each of these subformulas the satisfaction set \( Sat(\Phi_i) \) is computed. To check \( S_{\text{scp}}(\Phi) \) and \( P_{\text{scp}}(\phi) \) for each state of \( M \) the probability to satisfy the respective formulae is computed. \( Sat(...) \) then denotes the set for which the probabilities to satisfy the respective formulae lies within \( \approx p \). To compute \( P_{\text{scp}}(\Phi \cup \Psi) \) two methods exist:

1. Solving a system of integral equations. This method is of low practical interest, as the numerical stability is not satisfactory. We will not discuss it here, for details see [15].
2. Computing transient state probabilities of a modified Markov chain by using the method of uniformisation.

Definition 52 (Subformulae of CSL Formulae). Let \( \Phi, \Psi \) be CSL formulae, the set \( SF \) of subformulae is recursively defined as follows:

\[
SF(q) = \{q\} \\
SF(\neg \Phi) = SF(\Phi) \cup \{\neg \Phi\} \\
SF(\Phi \lor \Psi) = SF(\Phi) \cup SF(\Psi) \cup \{\Phi \lor \Psi\} \\
SF(S_{\text{scp}}(\Phi)) = SF(\Phi) \cup \{S_{\text{scp}}(\Phi)\} \\
SF(P_{\text{scp}}(X^{i} \Phi)) = SF(\Phi) \cup \{P_{\text{scp}}(X^{i} \Phi)\} \\
SF(P_{\text{scp}}(\Phi \cup \Psi)) = SF(\Phi) \cup SF(\Psi) \cup \{P_{\text{scp}}(\Phi \cup \Psi)\}
\]
Model Checking Steady State Formulae

To compute the stationary probabilities for a formula of the type $\mathcal{S}_{\bigcirc p}(\Phi)$ we can proceed as follows:

1. Compute $\text{Sat}(\Phi)$
2. Compute the set of BSCCs of $\mathcal{M}$: $B(\mathcal{M})$
3. forall $B \in B(\mathcal{M})$
4. Compute the probability to be in a $\Psi$-state: $\pi^\Psi(B)$
5. endforall
6. $\pi^\Psi(M)(s, \text{Sat}(\Psi)) = \sum_{B \in B(M)} \text{Prob}^M(s, F[0,\infty]q_B) \cdot \pi^\Psi(B)$

To compute $\pi^\Psi(B)$ from line (4) a system of linear equations has to be solved, details can be found in [137]. Line (6) deserves some words of explanation:

- $\pi^\Psi(M)(s, \text{Sat}(\Psi))$ is the probability to be in stationary state in a $\Psi$ state, given $s$ is the initial state.
- $\text{Prob}^M(s, F[0,\infty]q_B)$ is the probability to finally reach from $s$ a state in $B$, which is indicated by the characteristic formula $q_B$. To compute $\text{Prob}^M(s, F[0,\infty]q_B)$ we proceed as follows: At first, recall that $F\Phi := \text{true}_U\Phi$, then:
  $$\text{Prob}^M(s, F[0,\infty]q_B) := \begin{cases} 
1 & \text{if } \mathcal{M}, s \models q_B \\
\sum_{s' \in \text{Sat}(\Phi)} P(s, s') \cdot \text{Prob}^M(s', F[0,\infty]q_B) & \text{if } \mathcal{M}, s \models \neg q_B
\end{cases}$$

Model Checking Next Formulae

To check formulae of the type $\mathcal{P}_{\bigcirc p}(X^I\Phi)$ it suffices to consider the discrete transition probabilities of the SMC. The probability to reach a $\Phi$-state from $s$ with a single step can be computed by:

$$\text{Prob}^M(s, X^I\Phi) = (e^{E(s)}t - e^{E(s)}t') \cdot \sum_{s' \in \text{Sat}(\Phi)} P(s, s')$$

Model Checking Time-Bounded Until Formulae

Here, we will only describe the basic idea of model checking formulae of the type $\mathcal{P}_{\bigcirc p}(\phi)$, with $\phi = \Phi U^I\Psi$ by using the method of transient analysis, for details see [16, 15]:

The SMC $\mathcal{M} = (S, \mathcal{R}, L)$ is transformed into an SMC $\mathcal{M}' = (S, \mathcal{R}', L)$, where in $\mathcal{M}'$ the following types of states are made absorbing:

1. $\neg \Phi \land \neg \Psi$ states are made absorbing, as in this case the formula $\phi = \Phi U^I\Psi$ is not satisfiable, regardless of the future behaviour of $\mathcal{M}$.
2. $\Psi$ states can be made absorbing. If we assume that a $\Psi$ state is reached on a path on which all preceeding states satisfy $\Phi$, then the future behaviour of $\mathcal{M}$ is also of no interest.

$\mathcal{R}'$ is identical to $\mathcal{R}$ with exception of transitions that lead to absorbing states of $\mathcal{M}'$.

To compute $\text{Prob}^M(s, \Phi U^I\Psi)$, with respect to $I$, we have to distinguish three cases:

1. $I = [0, t]$
2. $I = [t, t']$
3. $I = [t, t']$, with $t \neq 0$
For case 1 we have to solve the following equation system:

\[
\text{Prob}^{M'}(s, \Phi U^t \Psi) = \sum_{s' \models \Psi} \pi^{M'}(s, s', t)
\]

where \(\pi^{M'}(s, s', t)\) is the probability to reach \(s'\), with \(s' \in \text{Sat}(\Psi)\), from \(s\) within \(t\) time units.

For a thorough discussion of all cases, see [15].

### 2.7.4 Markov-AP Bisimulation

To combat the notorious state space explosion problem equivalence relations like bisimulation in the functional world or Markov bisimulation in the Markovian world have turned out to be extremely useful. In the context of SMC a variant of Markov bisimulation, Markov-AP bisimulation can be defined. We will show that Markov-AP bisimulation can be used to reduce the state space of SMCs thereby preserving the validity of CSL formulae that are valid on the unreduced SMC.

**Definition 53 (Markov-AP bisimulation).** Given the SMC \(M\), an equivalence relation \(B\) is a Markov-AP bisimulation over \(M\) if

\[
(s, s') \in B \implies L(s) = L(s') \quad R(s, C) = R(s', C)\]

for all \(C \in S / B\)

with \(R(s, C) = \sum_{s' \in C} R(s, s')\), \(S / B = \{C_1, C_2, ..., C_n\}\) is the partitioning of \(S\) into its equivalence classes \(C_i\) with respect to \(B\).

Two states \(s\) and \(s'\) are Markov-AP bisimilar, if there is an equivalence relation \(B\) such that \((s, s') \in B\).

**Definition 54.** Let \([s]_B\) be the equivalence class of \(s\) with respect to \(B\), then the SMC \(M / B\) is defined as follows:

\[
M / B = (S / B, R_B, L_B)
\]

where

- \(S / B = \{C_1, C_2, ..., C_n\}\)
- \(R_B([s]_B, C) = R(s, C)\)
- \(L_B([s]_B) = L(s)\)

The following theorem [15], that we cite without proof states that \(M / B\) satisfies the same CSL properties as \(M\).

**Theorem 4.** Let \(B\) be a Markov-AP bisimulation and \(s\) a state of \(M\), then:

- for all state formulae \(\Phi\): \(M, s \models \Phi \iff M / B [s]_B \models \Phi\)
- for all path formulae \(\psi\): \(\text{Prob}^{M}(s, \psi) = \text{Prob}^{M / B}([s]_B, \psi)\)
Part II

Semantics and Verification of Stochastic Process Algebras
3.1 Introduction

In this chapter we describe a new symbolic semantics for stochastic process algebras [91, 92, 89].

The purpose of the use of formal modelling languages like stochastic process algebras is to have means to perform both qualitative and quantitative analysis. To this end we have to derive from a given SPA specification, its underlying semantic model, which is normally an SLTS. The data structures for representing these SLTSs have to satisfy the following constraints:

1. Memory-efficiency: This is of special interest, as, if the system consists of $n$ parallel processes, then the overall number of states is the size of the Cartesian product of the number of states of parallelly composed processes. Such a system specification often occurs when distributed reactive systems are modelled.
2. Time-efficiency: This incorporates the following aspects: Firstly, the generation of the data structures must be efficient, secondly, the data structure must allow efficient numerical computations.

The data structures we apply in this thesis of which we will demonstrate that they satisfy both constraints are MTBDDs and extensions thereof (cf. chapters 6 and 7). To derive SLTSs and their corresponding representations from a given stochastic process algebraic specification, three different approaches are possible:

1. In the most naive approach (cf. fig. 3.1), the specification is considered to be a monolithic entity, to which the SOS rules are applied to derive its underlying SLTS. The derived SLTS is then stored explicitly, for example using data structures that represent the transition between states together with the respective rates as matrices. From tools like TIPPtool [70] and PEPA [76] it is known that such an approach is inefficient with respect to both time and memory. Furthermore, if the SLTS shall finally be represented as an MTBDD it is very likely not to find a compact, i.e. small MTBDD representation for this monolithic SLTS [130].
2. The second approach tries to avoid the above-mentioned drawbacks by decomposing a given SPA specification in the following way: If the system is composed of $n$ parallel processes, then for each of these $n$ processes, an explicit SLTS representation is generated. Each of these SLTS can then be represented as a separate MTBDD. Using the algorithms from [50, 130] these MTBDDs can be combined parallelly on the
symbolic level. This method has the following drawbacks: At first, this method is still slow, and secondly, it is not guaranteed that the SLTSs of the involved processes can be represented compactly. The tool latranscat [57], which is an extension of the TIPP tool, utilises this approach (cf. fig. 3.1).

3. The new semantics we propose in this thesis avoids all these drawbacks, by circumventing the intermediate explicit SLTS representation, by providing semantic rules for all operators of the SPA from section 2.4.2, that allow the direct translation from any SPA term to its behavioural model, represented as an MTBDD (cf. fig. 3.2). As we shall demonstrate in chapter 7 this semantics allows both time- and memory-efficient generation of MTBDD-based representations of SLTSs.

This chapter is organised as follows: In section 3.2 we present the general idea and the detailed rules of our new semantics, in section 3.3 we prove that the MTBDDs generated by the new semantics are Markov bisimilar to those, derived by traditional SOS semantics. In section 3.4 we present some ideas how the state space can be kept minimal on generation, i.e. on-the-fly, with respect to Markov bisimulation.

3.2 Symbolic Denotational Semantics for YAMPA

In this section we describe the symbolic semantics for the process algebra YAMPA. We provide the semantic rules for the operators described in chapter 2.4. We can consider our semantics to be denotational, as the meaning of the entire expression (process) is composed by the meaning of its constituent sub-processes, i.e. the semantics of the entire process is built inductively from the semantics of its sub-processes.
3.2 Symbolic Denotational Semantics for YAMPA

### 3.2.1 General Idea

The general idea behind our semantics is to encode the transitions of a given process algebraic term \( P \) symbolically by an MTBDD. The symbolic representation \([P]\) is constructed from the parse tree of \( P \) which is processed in a depth first manner, thereby constructing \([P]\) inductively from simpler terms. Finally, we have the pure MTBDD-based representation of the transitional behaviour of \( P \).

**Definition 55 (Symbolic representation of process algebra terms).** The symbolic representation of a process algebra term \( P \) is denoted \([P]\). It consists of the following parts:

- an MTBDD \( B(P) \) which encodes the transition relation,
- a list of encodings of the process variables that appear in \( P \), denoted \( \text{Enc}_S(X) \)\(^1\),
- the encoding of the initial state \( \text{Enc}_S(s_D^{DS}) \).

The list of action encodings \( \text{Enc}_\text{Act}(a) \) is globally valid for all processes and therefore not included in \([P]\).

In the following we describe how to obtain \([P]\) from the symbolic representations of its constituents.

### Parallel Transitions

Two transitions \( s_1 \xrightarrow{a_1, \lambda_1} t_1 \) and \( s_2 \xrightarrow{a_2, \lambda_2} t_2 \) are called parallel if \( s_1 = s_2 \) and \( a_1 = a_2 \) and \( t_1 = t_2 \) (note that, in principle, both \( \lambda_1 \neq \lambda_2 \) and \( \lambda_1 = \lambda_2 \) is possible, although the latter

---

\(^1\) Process variables correspond to states, therefore we use \( \text{Enc}_S \) for both states and process variables.
case is ruled out if we only consider ordinary transition systems, as opposed to multi-
transition systems). Parallel transitions can be created by applying the choice or hiding
operators, or by applying the recursion operator in combination with choice. As we will
see in lemma 2, our MTBDD semantics does not represent parallel transitions separately,
but cumulates their rates, which is correct by lemma 2.

**Lemma 2 (Cumulation of parallel transitions).** Let $T$ be an SLTS and let transition system
$T'$ be constructed from $T$ by cumulating parallel transitions, i.e. by replacing each set of parallel
transitions $\{s \xrightarrow{a_i,\lambda} t | i = 1, \ldots, n\}$ by a single transition $s \xrightarrow{a,\lambda} t$, where $\lambda = \sum_{i=1}^{n} \lambda_i$. Then $T \sim_M T'$.

The proof is straightforward by comparing the cumulative rates, the details are omitted.

**Example 23.** Consider the following process

$$P := (a,\lambda)\cdot \text{stop} + (b,\mu)\cdot \text{stop}$$

and its corresponding SLTS (cf. fig. 3.3 (a)) These transitions are not parallel, as $a \neq b$ Now,

<table>
<thead>
<tr>
<th>$a,\lambda$</th>
<th>$b,\mu$</th>
<th>$\tau,\lambda$</th>
<th>$\tau,\mu$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td></td>
<td>$s_0$</td>
<td>$s_0$</td>
</tr>
<tr>
<td>$s_1$</td>
<td></td>
<td>$s_1$</td>
<td>$s_1$</td>
</tr>
</tbody>
</table>

**Fig. 3.3.** (a) SLTS without parallel transitions, (b) SLTS with parallel transitions

consider

$$P' := \text{hide}[a, b] \in P$$

and its corresponding SLTS shown in fig. 3.3 (b). Here, the transitions are parallel, as both $a$
and $b$ were replaced by $\tau$. Thus we can replace them by a single transition with rate $\lambda + \mu$,
as displayed in the rightmost figure.

3.2.2 Semantic Rules

The notations used for defining the binary encoding of states and transitions were intro-
duced in chapter 2.3.6.

In the sequel we assume that all sub-processes are sequential processes, i.e. no parallel
operator occurs in them.

**Process Variables and the stop Process**

**Verbal description:** A (guarded) process variable $X$ specifies a reference state within a
surrounding rec$X$ operator. Therefore, process variables are encoded in a similar fashion
as states, i.e. their encodings are taken from PC (the set of possible encodings). Within each
sequential component process variables having the same name get the same encoding.

$^2$ A sequential component is a process term which does not include the parallel composition operator.
Upon first appearance of a process variable $X$, the MTBDD associated with $X$ is the 0-MTBDD (that is the MTBDD consisting of only the terminal vertex 0).

**Formal description:**

```plaintext
if not first appearance of $X$ within present seq. component then
    skip /* do nothing */
endif
if $FC = \emptyset$ then /* need to extend the set of possible encodings */
    $PC := \text{Ext}(PC); UC := \text{Ext}_0(UC); FC := PC\setminus UC$
endif
Enc_S(X) := Ch(FC, <)
B(X) := 0
```

The stop process is a special case of a process variable (a process constant). It has no emanating behaviour, i.e. it remains inactive forever. As far as the symbolic representation is concerned, no state variables and so no symbolic transition representations are associated with the stop process. That means, the stop process has no specific encoding. Therefore, the stop process is associated with the 0-MTBDD.

**Formal description:**

```plaintext
if $FC = \emptyset$ then /* need to extend the set of possible encodings */
    $PC := \text{Ext}(PC); UC := \text{Ext}_0(UC); FC := PC\setminus UC$
endif
Enc_S(stop) := Ch(FC, <)
B(stop) := 0
```

**The exit Process**

We apply here an adopted interpretation for the exit process that suffices in our context: The exit process is treated like the stop process. The exit process is a process that has no emanating behaviour and remains inactive forever, i.e. we do not require the special $\delta$ action that indicates termination.

**Verbal Description:** The exit process is associated with the 0-MTBDD.

**Formal Description:**

```plaintext
if $FC = \emptyset$ then /* need to extend the set of possible encodings */
    $PC := \text{Ext}(PC); UC := \text{Ext}_0(UC); FC := PC\setminus UC$
endif
Enc_S(exit) := Ch(FC, <)
B(exit) := 0
```

**Prefix $P := (a, \lambda); Q$**

**Verbal description:** To generate $\llbracket P \rrbracket = \llbracket (a, \lambda); Q \rrbracket$ from $\llbracket Q \rrbracket$ an additional transition has to be inserted in $B(Q)$. The transition leads from the encoding of a new initial state to the encoding of the initial state of $Q$. Two cases can be distinguished when the new state is introduced:

1. The set $PC$ of free encodings is not empty: In this case one of the unused encodings is chosen and used as the new initial state of the overall process. The path that encodes the new transition is added to the existing MTBDD.
2. The set $PC$ of free encodings is empty: In this case as a first step the set of possible encodings has to be extended. In the MTBDD $B(Q)$ a new source- and target-variable $(s_n$ and $t_n)$ have to be introduced, whose values remain constant.

**Formal description:**

```
if $FC = \emptyset$ then
    $PC := \text{Ext}(PC)$; $UC := \text{Ext}_0(UC)$; $FC := PC \setminus UC$
    $B(Q) := B(Q) + (1 - s_n) * (1 - t_n)$
endif
```

$Enc_S(s_{DS}^P) := Ch(FC, <)$

$B(P) := B(Q) + TR_S(s_{DS}^P, a, \lambda, s_Q^D_S)$

**Enabling:** $P = Q >> R$

**Verbal Description:** When deriving the symbolic representation $\llbracket P \rrbracket$, we assume that $\llbracket Q \rrbracket$ and $\llbracket R \rrbracket$ are already given. The initial state $s_{DS}^P$ of the overall process $P$ is that of $Q$. All transitions in $Q$, with exit as target state are redirected to the initial state of $R$. All transitions in $Q$, leading to stop remain unchanged.

**Formal Description:**

```
Enc_S(s_{DS}^P) := Enc(s_{DS}^Q)
B(Q') := B(Q) * (1 - MT(t, Enc_S(\text{exit}))) + B(Q)_{t=Enc_S(\text{exit})} * MT(t, Enc_S(s_{DS}^S))
B(P) := B(Q') + B(R)
```

**Disabling:** $P = Q[> R$

**Verbal Description:** $\llbracket P \rrbracket$ is derived from $\llbracket Q \rrbracket$ and $\llbracket R \rrbracket$. We copy all transitions in $Q$ to $R$’s initial state. Finally, we take account of the fact that it is possible that $R$ interrupts $Q$ before $Q$ can take any transition by adding $B(R)$ to $B(P)$ and the fact that $Q$ is never interrupted. 

This can be accomplished by applying two times the appropriate algorithm for choice. Thus, the initial state of $B(P)$ is determined by the second call of the choice algorithm

**Formal Description:**

```
B(R') := 0
B(R'') := 0
forall $s_Q$
    if $Enc_S(s_Q) \neq Enc_S(\text{exit})$ and $Enc_S(s_Q) \neq Enc_S(\text{stop})$
        $B(R') := B(R) * (1 - MT(s, Enc_S(s_{DS}^S))) + B(R)_{s=Enc_S(s_{DS}^S)} * MT(s, Enc_S(s_Q))$
        $B(R'') := B(R') + B(R'')$
    $B(P') := \text{choice}(B(R'), B(R))$ /*Symbolic choice, as described before*/
    $B(P) := \text{choice}(B(P'), B(Q))$ /*Here, the initial state is determined*/
```

**Choice** $P = Q + R$

**Verbal description:** When deriving the symbolic representation $\llbracket Q + R \rrbracket$ from $\llbracket Q \rrbracket$ and $\llbracket R \rrbracket$, four cases have to be distinguished:
1. In both $Q$ and $R$ there exists a path $\sigma$ of the form $s_0 \xrightarrow{a_1,\lambda_1} s_1 \xrightarrow{a_2,\lambda_2} s_2 \ldots s_{l-1} \xrightarrow{a_1,\lambda_1} s_0$. In this case a new initial state has to be introduced for $Q + R$. All transitions emanating from the initial states of the subprocesses $Q$ and $R$ have to be copied, as they may also take place in the initial state of the overall process.

2. Only subprocess $Q$ has a loop back to its initial state. In this case $R$’s initial state can be used as the initial state of the overall process. All transitions starting in the initial state of $Q$ have to be copied such that they can also originate from the initial state of the overall process.

3. Only subprocess $R$ has a loop. This case is symmetric to 2.

4. None of the two processes has a loop. In this case one of the two initial states can be chosen to become the initial state of the overall process. The MTBDD construction method is analogous to that given in 2. resp. 3. (At the end in the overall MTBDD the transitions emanating from the starting state that does not represent the starting state of the overall process can be removed, as this state is no longer reachable.)

If the set $FC$ of free encodings is empty the set of possible encodings $PC$ has to be extended first, and the existing MTBDDs $B(Q)$ and $B(R)$ have to be adjusted accordingly.

Formal description:

1. Case $(\exists \sigma_Q : \sigma_Q = s_{Q}^{DS} \rightarrow \ldots \rightarrow s_{Q}^{DS}) \land (\exists \sigma_R : \sigma_R = s_{R}^{DS} \rightarrow \ldots \rightarrow s_{R}^{DS})$
   
   if $FC = \emptyset$ then
   
   $PC := Ext(PC); UC := Ext_0(UC); FC := PC \setminus UC$
   
   $B(Q) := B(Q) \ast (1 - s_n) \ast (1 - t_n)$
   $B(R) := B(R) \ast (1 - s_n) \ast (1 - t_n)$
   
   endif
   
   $Enc(s_{Q+R}^{DS}) := Ch(FC, <)$
   
   $B(Q') := B(Q)\big|_{s = Enc(s_{Q}^{DS})} \ast MT(s, Enc(s_{Q+R}^{DS}))$
   $B(R') := B(R)\big|_{s = Enc(s_{R}^{DS})} \ast MT(s, Enc(s_{Q+R}^{DS}))$
   $B(Q + R) := B(Q) + B(R) + B(Q') + B(R')$

2. Case $(\exists \sigma_Q : \sigma_Q = s_{Q}^{DS} \rightarrow \ldots \rightarrow s_{Q}^{DS}) \land (\exists \sigma_R : \sigma_R = s_{R}^{DS} \rightarrow \ldots \rightarrow s_{R}^{DS})$

   $Enc(s_{Q+R}^{DS}) := Enc(s_{Q}^{DS})$
   
   $B(Q') := B(Q)\big|_{s = Enc(s_{Q}^{DS})} \ast MT(s, Enc(s_{Q+R}^{DS}))$
   $B(Q + R) := B(Q) + B(R) + B(Q')$

3. Case $(\exists \sigma_Q : \sigma_Q = s_{Q}^{DS} \rightarrow \ldots \rightarrow s_{Q}^{DS}) \land (\exists \sigma_R : \sigma_R = s_{R}^{DS} \rightarrow \ldots \rightarrow s_{R}^{DS})$

   $Enc(s_{Q+R}^{DS}) := Enc(s_{Q}^{DS})$
   
   $B(R') := B(R)\big|_{s = Enc(s_{R}^{DS})} \ast MT(s, Enc(s_{Q+R}^{DS}))$
   $B(Q + R) := B(Q) + B(R) + B(R')$

4. Case $(\exists \sigma_Q : \sigma_Q = s_{Q}^{DS} \rightarrow \ldots \rightarrow s_{Q}^{DS}) \land (\exists \sigma_R : \sigma_R = s_{R}^{DS} \rightarrow \ldots \rightarrow s_{R}^{DS})$

   choose $Enc(s_{Q+R}^{DS}) := Enc(s_{Q}^{DS})$: similar to case 2, or
   
   choose $Enc(s_{Q+R}^{DS}) := Enc(s_{Q}^{DS})$: similar to case 3.
At this point we observe that this procedure will cumulate parallel or multiple transitions correctly: In case $Q$ contains a transition $s^D_{Q} \xrightarrow{a_{1}} t$ and $R$ contains a transition $s^D_{R} \xrightarrow{a_{2}} t$ (for any common target state $t$, and for either $\lambda_{1} \neq \lambda_{2}$ or $\lambda_{1} = \lambda_{2}$), these two transitions will be cumulated, since they are represented in $B(Q')$ and $B(R')$ as parallel transitions emanating from $s^D_{P}$ and leading to $t$, and since the MTBDD addition on the last line realises the addition of rates.

**Parallel composition $P = Q][L][R$**

**Verbal description:** For symbolic parallel composition we follow the same basic strategy as described e.g. in [50, 42, 127, 128, 129], where it had been found that this scheme ensures that the size of the symbolic representation of the composed process is linear in the size of its components. $[P] = [Q][L][R]$ can be constructed from $[Q]$ and $[R]$ as follows\(^3\): The MTBDD which represents the transitions in which both processes participate is constructed by combining those parts of $B(Q)$ and $B(R)$ which correspond to transitions labelled by actions from $L$ (we use $L$ to denote the BDD which encodes the actions in $L$). The MTBDD which represents the transitions which $Q (R)$ performs independently of $R (Q)$ is constructed by multiplying the part of $B(Q) (B(R))$ with a BDD $Id_R$ ($Id_Q$) which denotes stability\(^4\) of process $R (Q)$.

**Formal description:**

\[
Enc_S(s^D_{P}) := Enc_S(s^D_{Q}) \circ Enc_S(s^D_{R})
\]

\[
B(P) := (B(Q) \ast L) \ast (B(R) \ast L) + B(Q) \ast (1 - L) \ast Id_R + B(R) \ast (1 - L) \ast Id_Q
\]

**Recursion $P = \text{rec}X : Q$**

**Verbal description:** When constructing $[P] = [\text{rec}X : Q]$ from $[Q]$ we can distinguish two cases:

1. $X$ does not appear (unbound) in $Q$: In this case we simply identify the symbolic representation of $\text{rec}X : Q$ with that of $Q$.
2. $X$ appears in $Q$: In this case the process variable $X$ is identified with the encoding of the initial state of $Q$.

**Formal description:**

1. Case $X$ does not appear in $Q$

\[
Enc_S(s^D_{P}) := Enc_S(s^D_{Q})
\]

\[
B(P) := B(Q)
\]

2. Case $X$ appears in $Q$

\[
Enc_S(s^D_{P}) := Enc_S(s^D_{Q})
\]

\[
B(P) := B(Q) \ast (1 - MT(t, Enc_S(X))) + B(Q) \bigg|_{t = Enc_S(X)} \ast MT(t, Enc_S(s^D_{Q}))
\]

---

\(^3\) It is assumed that $B(Q)$ depends on the vectors of Boolean variables $a, s^Q, t^Q$ and $B(R)$ depends on $a, s^R, t^R$, i.e. their sets of state variables are disjoint.

\(^4\) BDD $Id_Q$, depending on the vectors of Boolean variables $s^Q$ and $t^Q$, encodes the identity matrix of appropriate size, and has a very compact representation under the interleaved variable ordering. Similar for $Id_R$. 

Note that recursion (in combination with the choice operator appearing within the scope of the recursion) may lead to parallel transitions which are cumulated correctly by the above procedure: In case process $Q$ contains two transitions $s \xrightarrow{b_1} t_1$ and $s \xrightarrow{b_2} t_2$ (for any source state $s$ and any action $b$), the latter of them will be redirected to the target state $t_2$ and the two transitions will be cumulated into the single transition $s \xrightarrow{b_1+b_2} t_1$ by the addition of the two MTBDDs.

### Hiding $P = \text{hide } b$ in $Q$

**Verbal description:** For constructing $[[P]] = [[\text{hide } b \text{ in } Q]]$ from $[[Q]]$, the MTBDD $B(Q)$ is first cofactorised with respect to the encoding of action $b$. The result is multiplied with the minterm encoding the internal action $\tau$. Finally, the part of the original MTBDD $B(Q)$ that does not correspond to action $b$ is added.

Note: Some languages provide the operator $\text{[hide } L \text{ in } P\text{]}$ where $L$ is a set of visible actions. Note that this is equivalent to hiding of all elements of $L$ sequentially.

**Formal description:**

\[
\begin{align*}
\text{Enc}_S(s^{DS}_P) &:= \text{Enc}_S(s^{DS}_Q) \\
B(P) &:= B(Q) \bigg|_{a=\text{Enc}_\text{Act}(b)} \ast MT(a, \text{Enc}_\text{Act}(\tau)) + B(Q) \ast (1 - MT(a, \text{Enc}_\text{Act}(b)))
\end{align*}
\]

Again, this procedure cumulates parallel transitions correctly. For any pair of states $s$ and $t$, a transition $s \xrightarrow{b \lambda_1} t$ (which will be turned into an internal $\tau$-transition) and an existing $\tau$-transition $s \xrightarrow{\tau \lambda_2} t$ will be cumulated by the addition of the two MTBDDs, leading to a single transition $s \xrightarrow{b \lambda_1 + \tau \lambda_2} t$.

### Relabelling $P = Q[a/b]$

**Verbal description:** For constructing $[[P]] = [[Q[a/b]]]$ from $[[Q]]$, the MTBDD $B(Q)$ is first cofactorised with respect to the encoding of action $b$. The result is multiplied with the minterm encoding the action $a$. Finally, the part of the original MTBDD $B(Q)$ that does not correspond to action $b$ is added.

**Formal description:**

\[
\begin{align*}
\text{Enc}_S(s^{DS}_P) &:= \text{Enc}_S(s^{DS}_Q) \\
B(P) &:= B(Q) \bigg|_{a=\text{Enc}_\text{Act}(b)} \ast MT(a, \text{Enc}_\text{Act}(a)) + B(Q) \ast (1 - MT(a, \text{Enc}_\text{Act}(b)))
\end{align*}
\]

Again, this procedure cumulates parallel transitions correctly. For any pair of states $s$ and $t$, a transition $s \xrightarrow{a \lambda_1} t$ (which will be turned into an $a$-transition) and an existing $a$-transition $s \xrightarrow{a \lambda_2} t$ will be cumulated by the addition of the two MTBDDs, leading to a single transition $s \xrightarrow{a \lambda_1 + \lambda_2} t$.

### 3.2.3 Parallel Composition at Arbitrary Levels

In this section we revoke the restriction that parallel composition is only allowed at the top level of a YAMPA specification. We have to distinguish the following cases:
1. Prefix followed by parallelly composed processes $P = (a, \lambda); (Q[[L]||R])$:

**Verbal Description:** The variable used for encoding the initial state of $P$ is different from the set(s) of variables used for encoding the parallelly composed processes. The initial state of $Q[[L]||R]$ is the concatenation of the initial states of both $Q$ and $R$ and the target state of the transition emanating from $P$'s initial state is the initial state of $Q[[L]||R]$. The initial state of $P$ is encoded by using a freshly introduced variable, $s_P$. It holds, $s_P$ is smaller than all variables used so far. We introduce a transition that leads from $s_P^{DS}$ to $s_{Q[[L]||R]}^{DS}$.

**Formal Description:**

\[ PC := Ext(PC); \quad UC := Ext_0(UC); \quad FC := PC \setminus UC \]
\[ B(Q[[L]||R]) := B(Q[[L]||R]) * (1 - s_n) * (1 - t_n) \]
\[ Enc(s_P^{DS}) := Ch(FC, <) \]
\[ B(P) := B(Q[[L]||R]) + TR(s_P^{DS}a, \lambda, s_{Q[[L]||R]}^{DS}) \]

2. Enabling followed by parallelly composed processes $P = Q >> (R[[L]||S])$:

**Verbal Description:** Both sides of the enabling operator use disjoint sets of variables for their state encodings. The initial state of $P$ is the initial state of $Q$. Each transition of $Q$ that leads to exit is redirected to the initial state of $R[[L]||S]$. The variables used for $Q$ are smaller than all variables used for $R[[L]||S]$.

**Formal Description:**

\[ Enc_Q(s_Q^{DS}) := Enc(s_Q^{DS}) \]
\[ B(Q') := B(Q) * (1 - MT(t, Enc_Q(exit))) + B(Q)_{t=Enc_Q(exit)} * MT(t, Enc_Q(s_{Q[[L]||S]}^{DS})) \]
\[ B(P) := B(Q') + B(R[[L]||S]) \]

3. Parallelly composed processes followed by enabling $P = (Q[[L]||R]) >> S$:

**Verbal Description:** Both sides of the enabling operator use disjoint sets of variables for their state encodings. $P$'s initial state is the initial state of $Q[[L]||R]$. Each transition in $Q[[L]||R]$ that leads to an $n$-tuple of exit-states is redirected to the initial state of $S$. The arity of this $n$-tuple is determined by the structure of both $Q$ and $R$. The variables used for $Q[[L]||R]$ are smaller than the variables used for $S$.

**Formal Description:**

\[ Enc_Q(s_Q^{DS}) := Enc(s_Q^{DS}) \]
\[ Enc_Q(exit, Q[[L]||R]) := Enc_Q(exit, Q) \cup Enc_Q(exit, R) \]
\[ B((Q[[L]||R])') := B(Q[[L]||R]) * (1 - MT(t, Enc_Q(exit, Q[[L]||R]))) + B(Q[[L]||R])_{t=Enc_Q(exit, Q[[L]||R])} * MT(t, Enc_Q(s_Q^{DS})) \]
\[ B(P) := B((Q[[L]||R])') + B(R) \]

4. Disabling followed by parallelly composed processes $P = Q|> (R[[L]||S])$:

**Verbal Description:** Both sides of the disabling operator use disjoint sets of variables for their state encodings. The initial state of $P$ is determined by the second call to the choice-algorithm. The variables used for $Q$ are smaller than all variables used for $R[[L]||S]$.

**Formal Description:**

\[ B((R[[L]||S])') := 0 \]
\[ B((R[[L]||S])') := 0 \]
\[ forall s_Q \]
\[ if Enc_Q(s_Q) \neq Enc_Q(exit) \text{ and } Enc_Q(s_Q) \neq Enc_Q(stop) \]
\[ B((R[[L]||S])') := B(R[[L]||S]) * (1 - MT(s, Enc_Q(s_{R[[L]||S]}^{DS}))) + B(R[[L]||S])_{s=Enc_Q(s_{R[[L]||S]}^{DS})} * MT(s, Enc_Q(s_Q)) \]
Parallely composed processes followed by disabling \( P = (Q[[L]]|R)|> S \):

**Verbal Description:** Both sides of the disabling operator use disjoint sets of variables for their state encodings. \( P \)'s initial state is the initial state of \( S \). The variables used for \( Q[[L]]|R \) are smaller than the variables used for \( S \).

**Formal Description:** This is analogous to case 4, we only have to consider the encoding of \( \text{stop} \) and \( \text{exit} \) in this case:

\[
\begin{align*}
\text{Enc}_S(\text{exit}_{Q[[L]]|R}) &:= \text{Enc}_S(\text{exit}_Q) \circ \text{Enc}_S(\text{exit}_R) \\
\text{Enc}_S(\text{stop}_{Q[[L]]|R}) &:= \text{Enc}_S(\text{stop}_Q) \circ \text{Enc}_S(\text{stop}_R)
\end{align*}
\]

6. Choice \( P = (Q_1[[L_1]]|Q_2) + (R_1[[L_2]]|R_2) \):

**Verbal Description:** Let \( Q := Q_1[[L_1]]|Q_2 \) and \( R := (R_1[[L_2]]|R_2) \). Both sides of the choice operator use disjoint sets of variables for their state encoding. The initial state of \( P \) is a newly introduced variable. All transitions from the initial states of both sub-processes of choice are copied to the initial state of the overall process.

**Formal Description:** As an abbreviation for the two subprocesses we use \( Q \) and \( R \).

\[
\begin{align*}
PC &:= \text{Ext}(PC); \quad UC := \text{Ext}_0(UC); \quad FC := PC \setminus UC \\
B(Q) &:= B(Q) \ast (1 - s_n) \ast (1 - t_n) \\
B(R) &:= B(R) \ast (1 - s_n) \ast (1 - t_n) \\
\text{Enc}_S(s^D_{P}) &:= \text{Ch}(FC, <) \\
&\text{/* copy initial transitions from } s^D_{Q} \text{ and } s^D_{R} \text{ */} \\
B(Q') &:= B(Q)\big|_{s = \text{Enc}_S(s^D_{Q})} \ast MT(s, \text{Enc}_S(s^D_{P})) \\
B(R') &:= B(R)\big|_{s = \text{Enc}_S(s^D_{P})} \ast MT(s, \text{Enc}_S(s^D_{P})) \\
B(P) &:= B(Q) + B(R) + B(Q') + B(R') \text{ /* put it all together */}
\end{align*}
\]

### 3.2.4 Example

In this section we will show how to build a symbolic representation out of a given process algebra term, using the semantic rules presented so far. Let the following process algebra term be given:

\[P := \text{rec}X:((a, \lambda); (b, \mu); X + (c, \gamma); \text{stop})\]

We will proceed as follows to get the symbolic representation \([P]\) for \( P \):

1. First, we generate \( P \)'s parse tree which is shown in figure 3.4 (a)

2. In this table the encodings of all actions occurring in \( P \) are stored.

\[
\begin{array}{|c|c|}
\hline
\text{Action} & \text{Encoding} \\
\hline
a & 01 \\
b & 10 \\
c & 11 \\
\hline
\end{array}
\]

3. First, we generate the symbolic representation of the left hand side of \( P \).
   a) \([X]\): For the process variable \( X \) we allocate one state bit, \( s_0 \) and encode \( X \) as \( s_0 = 0 \).
b) \([ (b, \mu); X] \): This is considered as a new state, so we use \(s_0 \) and encode this state as \(s_0 = 1\). Its MTBDD representation can be seen in figure 3.4 (b). (In the graphical depiction of an MTBDD, dashed lines indicate zero-edges and solid lines indicate one-edges.)

c) \([ (a, \lambda); (b, \mu); X] \): As we have to encode a new state and having already encoded two states we have to allocate a new state variable \(s_1\), and we encode the new state as \(s_1s_0 = 10\). We also have to reencode the states of a) and b):

- \([X] \): \(s_1s_0 = 00\)
- \([ (b, \mu); X] \): \(s_1s_0 = 01\).

d) Now we have finished the encoding of the left hand side of \(P\); we only store explicitly its initial state and the encoding of the only occurring process variable \(X\):

\[
(\text{Enc}(s^{DS}_{(a, \lambda); (b, \mu); X}) = 10, \text{Enc}(X) = 00).
\]

Now we have the MTBDD representation as depicted in figure 3.4 (c).

4. Now we generate the symbolic representation of \(P\)'s right hand side:

a) \([\text{stop}]\): Following our semantical rules, we have to encode the process constant stop. We use the fresh encoding \(s_1s_0 = 11\) to do so.

b) \([ (c, \gamma); \text{stop}] \): As all possible encodings using two binary variables are already used, we have to add a third variable to encode this new state: \(s_2s_1s_0 = 100\).

We have therefore to perform the following reencodings:

- \([\text{stop}] \): has now the encoding 011
- \([X] \): 000
- \([ (b, \mu); X] \): 001
- \([ (a, \lambda); (b, \mu); X] \): 010

c) We store the following additional information:

\[
(\text{Enc}(s^{DS}_{(c, \gamma); \text{stop}} = 100, \text{Enc}(\text{stop}) = 011).
\]

The right hand side of \(P\) generates the MTBDD representation as depicted in figure 3.4 (d).

5. Because of the extension of the encoding set we would have to redraw the MTBDD of \(P\)'s left hand side, that shall be omitted, we only mention that we have to add two additional variables \(s_2\) and \(t_2\), for which we have that they remain constantly 0.
3.3 Correctness of Semantics

6. We now compose the right hand side and the left hand side of $P$ according to the semantic rules for the choice operator. As no subprocess has transitions leading back to the respective initial states of the subprocesses we can choose one of the subprocesses’ initial states as the initial state of the overall process. In this case we choose the initial state of the left hand side: $\text{Enc}(s_{(a,\lambda);(b,\mu);X+(c,\gamma);\text{stop}}) = 010$. So we get the MTBDD representation as shown in figure 3.5 (a).

7. In the last step we add recursion. The encoding of $X$ is now identified with the encoding of the initial state. This yields the MTBDD depicted in figure 3.5 (b).

3.3 Correctness of Semantics

3.3.1 Outline of the Proof

In this section we show that our MTBDD-semantics is bisimulation equivalent to the standard SOS-semantics. Roughly speaking, this is shown along the following steps:

- From the MTBDD-representation $[P]$ of a given term $P$ we derive an SLTS which we denote by $\text{Tr}([P])$ and whose initial state we denote by $s^\text{DS}_P$. The SLTS $\text{Tr}([P]) = (S^\text{DS}_P, \text{Act}, \rightarrow, s^\text{DS}_P)$ is obtained from $[P]$ by the straight-forward algorithm from figure 3.6 which extracts the encoded transitions one by one from the MTBDD.

- Using induction on the term’s structure and exploiting the congruence property of Markovian bisimulation we show that for an arbitrary term $P$ the SLTS obtained by applying the SOS-rules, in the following denoted by $\text{SOS}(P)$ (with state space $S^\text{SOS}_P$ and initial state $s^\text{SOS}_P$), is bisimulation equivalent to $\text{Tr}([P])$\(^\text{5}\). (Induction on the term’s structure means that for each operator we show that its addition preserves the bisimulation equivalence relation established for the shorter term.)

\(^5\)Note that $\text{SOS}(P)$ and $\text{Tr}([P])$ are bisimulation equivalent but not necessarily isomorphic, since $\text{SOS}(P)$ may contain parallel transitions, while $\text{Tr}([P])$ cannot represent these separately.
3 Symbolic Semantics for Stochastic Process Algebras

(1) /* Initialisation: */
(2) \( S := \{ s^{DS}_P \} /* single initial state */ \\
(3) \text{Unexpl} := \{ s^{DS}_P \} /* set of unexplored states */ \\
(4) \( R := \emptyset /* empty transition relation */ \\
(5) /* find reachable states and transitions: */
(6) \text{while} \text{Unexpl} \neq \emptyset \text{ do} \\
(7) \text{choose} x \in \text{Unexpl} /* arbitrary choice */
(8) \text{New} := \emptyset /* initialise new states reachable from } x /*
(9) \( x^{\text{Trans}} := \mathcal{B}(P) \mid_{s = \text{Enc}(x)} /* transitions emanating from } x /*
(10) \text{forall} a \in \text{Act} \text{ do} \\
(11) \( x^{\text{aTrans}} := x^{\text{Trans}} \mid_{a = \text{Enc}_{\text{Act}}}(a) /* a-transitions emanating from } x /*
(12) \text{forall} y \in \text{UC} \text{ do} \\
(13) \( y := \text{Enc}^{-1}(y) /* y \text{ is the state encoded by } y */ \\
(14) \lambda := x^{\text{aTrans}}(y) /* rate of transition } x \xrightleftharpoons{a\lambda} y */
(15) \text{if} \lambda \neq 0 \text{ then} \\
(16) \( R := R \cup \{(x, a, \lambda, y)\} /* add new transition */
(17) \text{if} y \notin S \text{ then} \\
(18) \text{New} := \text{New} \cup \{y\} /* add new state */
(19) \text{endif} \\
(20) \text{endif} \\
(21) \text{endfor} \\
(22) \text{endfor} \\
(23) S := S \cup \text{New} /* update state set */
(24) \text{Unexpl} := \left( \text{Unexpl} \setminus \{x\} \right) \cup \text{New} /* update set of unexplored states */
(25) \text{endwhile}
(26) \text{return} (S, \text{Act}, R, s^{DS}_P)

Fig. 3.6. Algorithm for constructing a SLTS from the symbolic representation

3.3.2 Construction of the Transition System

In this section we show how to derive the SLTS \( T_r([P]) = (S, \text{Act}, R, s^{DS}_P) \) from \([P]\). As stated in Def. 55, \([P]\) contains the MTBDD \( \mathcal{B}(P) \) which depends on Boolean variables \( a, s \) and \( t \), the encoding of the initial state \( \text{Enc}(s^{DS}_P) \) and some additional information.

Figure 3.6 shows the algorithm for constructing the SLTS from the symbolic representation. This algorithm is given only for the purpose of explanation and does not aim to be efficient. In practice, several optimisations are possible. For instance, in line (12), instead of iterating over all vectors in \( \text{UC} \), one may traverse the MTBDD \( x^{\text{Trans}} \) in a depth-first manner in order to find exactly those Boolean vectors \( y \) which encode a valid transition. We will prove that algorithm 3.6 is correct in the sense of theorem 5:

**Theorem 5 (Correctness of Algorithm 3.6).** Every transition encoded in the MTBDD will be inserted into the explicit representation generated by algorithm 3.6. Furthermore, \((S, \text{Act}, R, s^{DS}_P)\) contains exactly those transitions that are encoded by the MTBDD from which the explicit representation was generated.

**Proof (Theorem 5).** See Appendix A.1.
3.3.3 Correctness Proof of the Semantics

For showing the correctness of our MTBDD semantics, we need some information about the equivalence class structure under parallel composition.

Equivalence Class Structure under Parallel Composition:

We now discuss how equivalence classes of process \( P = Q||L||R \) can be constructed from the equivalence classes of \( Q \) and \( R \).

**Lemma 3.** Let \( \text{Part}^Q = \{C^Q_1, \ldots, C^Q_m\} \) \((\text{Part}^R = \{C^R_1, \ldots, C^R_m\})\) be a partition of the state space of process \( Q \) (\( R \)) which corresponds to the equivalence classes of a Markovian bisimulation. The Cartesian product \( C^P_{i,j} = C^Q_i \times C^R_j \) yields a partition \( \text{Part}^P \) of the state space of process \( P = Q||L||R \) (with equivalence classes \( \{C^P_{i,j} | i = 1, \ldots, m_Q, j = 1, \ldots, m_R\} \)) which again corresponds to a Markovian bisimulation.

**Proof (Lemma 3).** See Appendix A.2. ❇

Note that some equivalence classes of the combined process \( P \) may not be reachable due to synchronisation conditions. Note further that Lemma 3 does not assume that \( \text{Part}^Q \) and \( \text{Part}^R \) correspond to the largest bisimulation relations, nor does it claim that the equivalence classes \( C^P_{i,j} \) are maximal.

Equivalence Proof

**Theorem 6.** For any process term \( P \) from the language YAMPA it holds that \( \text{Tr}([P]) \sim_M \text{SOS}(P) \).

**Proof.** See Appendix A.3. ❇

3.4 Towards a Minimal Symbolic Semantics for YAMPA

We have shown that our MTBDD semantics is correct, but so far we have not made any considerations on its minimality. It would, of course, be desirable that the MTBDD constructed by our denotational semantics encodes an SLTS which is minimal with respect to Markovian bisimulation, where minimality means that every class of bisimilar states is represented by a single macro state. Trivially, this goal could be achieved by performing bisimulation minimisation [120, 52, 84] after every construction step, thereby ensuring that all intermediate representations are minimal. BDD-based bisimulation algorithms are available [28, 75], they follow the usual iterative refinement scheme, but such a strategy is impracticable since the overhead for running the bisimulation algorithm would be prohibitive.

Ideally, we wish to perform bisimulation on-the-fly, keeping the encoded state space minimal at every step of the construction, by exploiting information about the operator at hand and the structure of the operand processes. For that purpose we investigate a set of heuristic algorithms.

For prefix \( P = (a, \lambda); Q \) the algorithm is trivial and works as follows:

1. A state whose only outgoing transition has the same label, the same rate and the same target state as the one to be added by applying prefix already exists, then the two states can be merged.
2. No equivalent state exists, in this case the new state constitutes an equivalence class of its own.

The formal description of this algorithm can be found in figure 3.7. Assume that $C_{\text{ini}}$ is the equivalence class which contains the initial state of process $Q$ and $s_P^{DS}$ is $P$’s initial state.

(1) $Part := Part^Q$
(2) if $\exists s_Q^{DS} \in S_Q^{DS} (\gamma(s_Q^{DS}, a, C_1) = \gamma(s_Q^{DS}, a, C_2) \wedge C_1 = C_2)$
(3) $s_P^{DS} := s_P^{DS}$
(4) else
(5) $Part := Part \cup s_P^{DS}$
(6) return $Part$

Fig. 3.7. Determining equivalence classes of $(a, \lambda); Q$ from the classes of $Q$

For recursion and hiding we derived some more involved algorithms, shown in figure 3.8 resp. figure 3.9.

As a simple example, we now briefly discuss the algorithm for the case of recursion. Figure 3.8 shows an algorithm which merges known equivalence classes of process $Q$ in order to obtain equivalence classes for $P = \text{rec}X : Q$. Starting from the newly formed class $C'_{\text{ini}}$ which contains both $X$ and the initial state, the algorithm looks for pairs of predecessor classes (denoted $\text{Pred}(C'_{\text{ini}})$) which can be merged. If two classes are merged, the search also considers the further predecessor classes in a chained fashion, and this aim-driven procedure makes the algorithm quite efficient.

Both algorithms coarsen the equivalence classes of the processes to which the respective operations recursion and hiding are applied. We found out that coarsening is not sufficient, as our algorithms are correct but not complete. For minimal semantics the following relations should hold:

(1) $Part := Part^Q$
(2) $C'_{\text{ini}} := C_{\text{ini}} \cup C_X /*$ the initial class and the class containing state $X$ are merged */
(3) $Part := Part \setminus \{C_{\text{ini}}, C_X\} \cup \{C'_{\text{ini}}\} /*$ the partition is updated */
(4) $Mergers := \{C'_{\text{ini}}\}$
(5) while $Mergers \neq \emptyset$
   (6) choose $C_{\text{mrg}} \in Mergers$
   (7) forall $C_i, C_j \in \text{Pred}(C_{\text{mrg}})$ do /* consider pairs of predecessor classes of $C_{\text{mrg}}$ */
   (8) if $\forall a : \gamma(C_i, a, C_i \cup C_j) = \gamma(C_j, a, C_i \cup C_j)$ then /* compare mutual rates */
   (9) if $\forall a : \gamma(C_i, a, C_k) = \gamma(C_j, a, C_k)$ then /* compare rates to third party */
   (10) $C_i' := C_i \cup C_j /*$ two classes are merged */
   (11) $Part := Part \setminus \{C_i, C_j\} \cup \{C_i'\} /*$ the partition is updated */
   (12) $Mergers := Mergers \cup \{C_i'\} /*$ a new merger is added */
   (13) endif
   (14) endfor
(15) endwhile
(16) $Mergers := Mergers \setminus \{C_{\text{mrg}}\} /*$ the processed merger is removed */
(17) return $Part$

Fig. 3.8. Determining equivalence classes of $\text{rec}X : Q$ from the classes of $Q$
3.4 Towards a Minimal Symbolic Semantics for YAMPA

/* Initialisation: */
Part\_t := Part\_Q /* records the equivalence class structure under hiding */
stable := false /* approximation of equivalence class structure is completed, if true */
/* Iteration: */
while stable = false do
stable := true
forall C\_i, C\_j \in Part\_t do
if \forall C\_k: (γ(C\_i, τ, C\_k) + γ(C\_i, b, C\_k)) = γ(C\_j, τ, C\_k) + γ(C\_j, b, C\_k)
\& ∀a \notin \{τ, b\}: γ(C\_i, a, C\_k) = γ(C\_j, a, C\_k) then
stable := false
C\_i\_f := C\_i ∪ C\_j
Part\_t := (Part\_t \setminus C\_i) \cup C\_i\_f
endif
endfor
endwhile

Fig. 3.9. Algorithm for determining the equivalence class structure under hiding

![Diagram of the algorithm](image)

Fig. 3.10. Example that demonstrates correct work of algorithm in figure 3.8

s\_1, s\_2 \in C \Rightarrow s\_1 \sim\_M s\_2 (3.1)

s\_1 \sim\_M s\_2 \Rightarrow s\_1, s\_2 \in C (3.2)

Our algorithms fulfill condition (1) but not condition (2). We could find example process terms whose state spaces were not reduced whilst applying standard bisimulation algorithms could perform substantial reductions: In the example shown in figure 3.10, the predecessor of X and the predecessor of the initial state are merged first, and in the subsequent step the two dark states are merged. This is a case where our heuristic algorithm finds the largest Markovian bisimulation, leading to a minimal state space. However, there exist situations like the one shown in figure 3.11 where the algorithm of figure 3.8 does not find the coarsest partition, since it is not possible to merge the two dark states without merging the three lightly shaded states at the same time. Altogether, one can say that the merging of two classes at a time is not sufficient, as our algorithm is correct but not complete.

For choice, as already observed in [123], the key to minimality lies in the ability to detect common behaviour within the operands Q and R. This can be achieved by identifying and comparing the strongly connected components (SCC) of Q and R. SCCs can be deter-
Fig. 3.11. Example that demonstrates incompleteness of algorithm in figure 3.8

mined symbolically in an efficient way [144]$. For parallel composition, the resulting SLTS is not minimal if the two partners contain identical behaviour which leads to symmetries in the state space (but symmetry is not a necessary precondition for non-minimality). [123] describes state space reduction for replicated processes. Although this can yield a large reduction of the state space, it is shown in [23] that the resulting SLTS is not necessarily minimal. In fact, it is minimal only if all states of the replicated process are "relatively prime", which condition is difficult to verify in practice$^7$.

---

$^6$ In addition, since only reachable behaviour should be represented, in case the initial state $s_{Q_s}^{DS}$ is unreachable after the application of the choice operator, transitions emanating from this state can be deleted.

$^7$ Furthermore, since some states of the combined process may be unreachable due to synchronisation conditions, (symbolic) reachability analysis may be necessary in order to determine the set of reachable states.
Stochastic Propositional Dynamic Logic

4.1 Introduction

In this chapter we will introduce the new logic SPDL [93, 94] (stochastic propositional
dynamic logic) which is a stochastic extension of the logic PDL (cf. [53] and chapter 2.5).

In connection with high-level modelling formalisms such as generalised stochastic Petri
nets (GSPN) or stochastic process algebras (SPA), the system’s behaviour is described as a
sequence of transitions that can be fired (GSPN) or of actions that can be executed (SPA).
The notion of state is only an auxiliary means on base of the low-level semantic model. The
user of a GSPN or SPA modelling tool should ideally not care about these low-level details.
But, if the temporal logic CSL is used for specifying requirements that are to be verified,
knowledge about the semantic model is required, as CSL is state oriented, i.e. formulae
describe properties of paths as sequences of states in which certain properties must hold.
This is not desirable, as identifying states that are relevant for a requirement is difficult and
error-prone, and it requires detailed knowledge about the semantic model, which violates
the principle of abstraction. This shift of paradigms constitutes a severe drawback with
respect to the applicability of automatic verification. The logics aCSL and aCSL+ [73, 108]
take the same abstract view on a system’s behaviour as high-level modelling formalisms
do. Performance and reliability requirements are described as sequences of actions. The
disadvantage of aCSL is, that it only has limited means to characterise satisfying paths (cf.
chapter 1.3).

The logic SPDL is related to aCSL and aCSL+, but extends these logics in various ways:

- Satisfying paths can be described by regular expressions.
- The executability of a regular expression can be made dependent on the satisfaction
  of formulae (tests) within a regular expression. This makes it possible to express pro-
  gramming language constructs like if-then-else, case or while in SPDL to characterise
  satisfying paths.

SPDL supports an approach to the verification of performance and reliability requirements
as shown in fig. 4.1. We have also defined the logic asCSL, which is also an action-oriented
stochastic logic, that is related to SPDL, we refer to our paper [14] for a thorough treatment
of this logic.

This chapter is organised as follows: In section 4.2 we describe the general idea of model
checking SPDL. In sections 4.3 to 4.4 we introduce the syntax and semantics of SDPL,
program automata, and finally in section 4.5, we introduce model checking algorithms for
SPDL. In section 4.6 we discuss in depth a small example. Section 4.7 is devoted to the proof
that the validity of SPDL formulae is invariant with respect to action-labelled Markov-AP-bisimulation, and to the time and space complexity of model checking SPDL. In section 4.8 we extend SPDL by real time intervals, i.e. time intervals of the form \([t, t']\), where \(t \neq 0\) is possible. This logic is called SPDL\(^I\). Also in section 4.8 we point out, how to automatically verify stochastic systems that also incorporate untimed, immediate transitions, this logic will be referred to as IM-SPDL.

### 4.2 General Idea

In this section we will briefly describe the general idea of model checking probabilistic SPDL formulae, i.e. formulae of the type \(\mathcal{P}_{\text{cop}}(\Phi[\rho]|_{[t,t']}\Psi)\), where \(\mathcal{P}_{\text{cop}}\) is the probabilistic path operator, familiar from CSL. \(\Phi[\rho]|_{[t,t']}\Psi\) can be explained as follows:

A path \(\sigma\) satisfies \(\Phi[\rho]|_{[t,t']}\Psi\), iff:
- within at least \(t\) and at most \(t'\) time units a state must be reached that satisfies \(\Psi\). All preceding states must satisfy \(\Phi\). The action-labels of all transitions up to the \(\Psi\) state must form a word that is derivable from the given program \(\rho\) (cf. chapter 2.5), furthermore all tests occurring in \(\rho\) must be satisfied by appropriate states of path \(\sigma\).

The idea of our approach is to reduce the model checking problem \(M, s \models \mathcal{P}_{\text{cop}}(\Phi[\rho]|_{[t,t']}\Psi)\) of SPDL to the model checking problem of CSL which consists of verifying whether \(M^\times, s^\times \models \mathcal{P}_{\text{cop}}(\mathcal{F}|_{[t,t']} \chi_{M^\times})\) for a CTMC \(M^\times\) and a state \(s^\times\) of \(M^\times\). The CTMC \(M^\times\) has to be constructed from \(M\) and an automaton \(A_\rho\) for \(\rho\), by simply speaking building the product transition system between \(M\) and \(A_\rho\). A path satisfies \(\mathcal{F}|_{[t,t']} \chi_{M^\times}\), if within the time interval \([t, t']\) a state is reached that satisfies the formula \(\chi_{M^\times}\). To reach this goal we have to take the following steps:

1. From the program \(\rho\) we derive a deterministic program automaton \(A_\rho\), which is a variant of deterministic finite automata.
2. Using the given SLTS $M$ and the program automaton $A_\rho$, we build a product SLTS or product Markov chain $M^\times$. The state space of $M^\times$ is the product of $M$ and $A_\rho$, i.e. its states are of the form $(s_i, z_j)$, where $s_i$ is a state of $M$ and $z_j$ a state of $A_\rho$. Additionally, $M^\times$ possesses one new, absorbing state:

- An error state $\text{FAIL}$.

In $M^\times$ a transition $(s_i, z_j) \xrightarrow{\lambda} (s_j, z_j)$ is introduced, iff the following two constraints are satisfied:

- $(s_i, z_j)$ must satisfy $\Phi$, this is the case iff $s_i$ satisfies $\Phi$.
- Both $s_i$ and $z_j$ must be capable to perform the same action, and if the current action is associated with a test, then $s_i$ must also satisfy this test.

If one of these two constraints is violated, we have to introduce a transition $(s_i, z_j) \xrightarrow{\lambda} \text{FAIL}$.

3. Finally, to compute the probability measure of the paths that satisfy $\phi$ we proceed as follows: All states $(s_j, z_j)$ of $M^\times$ for which $s_j$ is a $\Psi$-state and $z_j$ is an accepting state of $A$ are labelled with the special, newly introduced atomic state formula $\chi_{M^\times}$.

4. At this point, it is possible to check, whether $P_{\text{op}}(\Phi[p][t,t']\Psi)$ is functionally satisfiable:

If in $M^\times$ a path to a $\chi_{M^\times}$ state exists, then $P_{\text{op}}(\Phi[p][t,t']\Psi)$ can be satisfied at least on the functional level.

5. On $M^\times$ (which was transformed as described by step 3) we can compute the probability measure of all paths satisfying the CSL formula $P_{\text{op}}(F[t,t']\chi_{M^\times})$, which is equal to the probability measure of the paths satisfying the original formula $P_{\text{op}}(\Phi[p][t,t']\Psi)$ in the original model $M$.

Like CSL, SPDL provides a steady state operator $S_{\text{op}}$. Model Checking of steady state formulae can be done the same way as for CSL.

### 4.3 Stochastic PDL: Syntax, Semantics and Automata

#### 4.3.1 Action- and State-Labelled Continuous-Time-Markov Chains

**Definition 56 (Action- and state-labelled continuous-time-Markov chains, ASMC).** An **ASMC** $M$ is a quadruple $(S, \text{Act}, L, R)$, where

- $S$: finite set of states
- $\text{Act}$: set of action names
- $L$: state labelling function: $S \rightarrow 2^{\text{AP}}$
- $R$: state transition relation: $R \subseteq S \times (\text{Act} \times R_{>0}) \times S$

$\text{AP}$ is the set of atomic propositions. Without loss of generality we assume that for each $a \in \text{Act}$ and successor state at most one transition exists (cf. lemma 2).

**Definition 57 (Rates and probabilities).** Let $A \subseteq \text{Act}$, then:

$$R_A(s, s') := \sum_{a \in A} \{\lambda | s \xrightarrow{a\lambda} s'\}$$

$R_A(s, s')$: sum of all rates $\lambda$ leading with actions $a \in A$ from $s$ to $s'$.

$$E(s) := \sum_{s' \in S} R_{\text{Act}}(s, s')$$

$$E(s)$$
Definition 58 (Paths in $\mathcal{M}$). An infinite path $\sigma$ is a sequence of transitions of the form $s_0 \xrightarrow{a_0,t_0} s_1 \xrightarrow{a_1,t_1} s_2 \ldots$

- $s_i \in S, a_i \in \text{Act}, (s_i, a_i, \lambda, s_{i+1}) \in R$
- $t_j = \tau(\sigma, i) \in R_{\geq 0}$: real sojourn time in $s_i$ before passing to $s_{i+1}$.
- $\sigma[i]: (i+1)$th state on path $\sigma$
- $a[i]: (i+1)$th action on path $\sigma$
- $\sigma@t = \sigma[i]$: state that is reached at time instant $t$ on path $\sigma$, it holds that $i$ is the smallest index for which $t \leq \sum_{j=0}^{i} t_j$.

A finite path $\sigma$ is a finite sequence of transitions of the form: $s_0 \xrightarrow{a_0,t_0} s_1 \xrightarrow{a_1,t_1} s_2 \ldots \xrightarrow{a_{n-1},t_{n-1}} s_n$, where $R(s_i, s_{i+1}) > 0$ for all $i < n$ and $R(s_n, s'_n) = 0$ for all $s'_n \in S$. For finite paths $\sigma$, $\sigma[i]$ and $\tau(\sigma, i)$ are defined only for $i \leq n$, for $i < n$ as for infinite paths, for $i = n$ it holds $\tau(\sigma, i) = \infty$. For $t \geq \sum_{j=0}^{n-1} t_j$ let $\sigma@t = s_n$ for all other cases, $\sigma@t$ is defined as in the case of infinite paths.

The set of all paths of $\mathcal{M}$ is denoted by $\text{PATH}^{\mathcal{M}}$. The set of all paths with initial state $s$ is denoted as $\text{PATH}^{\mathcal{M}}(s)$

$$\text{PATH}^{\mathcal{M}}(s) := \{ \sigma | \sigma[0] = s \}$$

The notions of transient and steady state probabilities of definition 47 and definition 48 remain valid also in case of ASMCs.

4.3.2 Syntax of SPDL

In this section we present the syntax of the logic SPDL. SPDL extends PDL with two probabilistic operators that allow to express steady state and probabilistic transient requirements. Like in the logic CSL [17, 16], SPDL provides two types of formulae: state formulae that are interpreted over the states of an ASMC $\mathcal{M}$ and path formulae that are interpreted over paths in an ASMC.

Definition 59 (Syntax of SPDL). Let $p \in [0, 1]$, and $q \in \mathcal{AP}$ an atomic proposition, where $\mathcal{AP}$ is the set of atomic propositions and let $\varpi \in \{ \leq, <, \geq, > \}$.

The state formulae $\Phi$ of SPDL are defined as follows:

1 If $\mathcal{M}$ is clear from the context, it will be omitted
Path formulae are defined by:

\[ \phi := \Phi[\rho][0,1] \Phi \]

where \( t \in \mathbb{R}_{\geq 0} \cup \{\infty\} \). Programs \( \rho \) are described by the grammar given in definition 61.

**Definition 60 (Length of an SPDL Formula).** Let \( q \in AP, \Phi \) and \( \Psi \) be state formulae, and \( \phi \) a path formula.

The length of state formulae is defined as in the case of CSL, see definition 50. The length of path formulae is defined as follows:

\[ |\phi| = |\Phi[\rho][\Psi]| := |\Phi| + |\Psi| + 1 \]

### 4.3.3 Programs and Automata

**Definition 61 (Programs).** Let \( \text{Act} \) be a set of atomic programs, which we may also call actions, \( e \) the empty program, and \( \text{TEST} \) be a set of SPDL state (test) formulae. Together they form the alphabet \( \Sigma_\rho \) for the program \( \rho \), i.e.

\[ \Sigma_\rho := \text{TEST} \cup (\text{Act} \cup \{e\}) \]

A program \( \rho \) over an alphabet \( \Sigma_\rho \) is defined by the following grammar:

\[ \rho := e | p; \rho | \rho \cup \rho | \exists ?; \rho | \rho_1 | (\rho) \]

\[ \rho_1 := a | \rho_1 | \rho_1 \cup \rho_1 | \exists ?; \rho_1 | (\rho_1) \]

Where \( a \in \text{Act} \) and \( \exists \in \text{TEST} \).

We do not allow programs of the form \( (\Phi?; e)^* \), this can be justified as follows:

In the way we do model checking of SPDL formulae, i.e. by constructing a product Markov Chain between the system’s original SLTS and the automaton of the program defining the satisfying paths, it is not necessary to have (sub-)programs of the kind \( (\Phi?; e)^* \), as with a test no transition in the Markov chain is associated and the program can be executed also zero times, the validity or non-validity of \( \Phi \) in the current state of the Markov chain is without significance for the model checking procedure, thus, it can be omitted. To relate programs and program instances we need the following definition.

**Definition 62 (Relating programs and program instances).** \( PI(\rho) \) associates to every program \( \rho \) the set of its program instances. \( PI(\rho) \) is inductively defined as follows:

\[ PI(\emptyset) = \emptyset \]

\[ PI(e) = \{e\} \]

\[ PI(a) = \{a\} \]

\[ PI(\Phi?) = \{\Phi?\} \]

\[ PI(\Phi?; a) = \{\Phi?; a\} \]

\[ PI(\rho_1 \cup \rho_2) = PI(\rho_1) \cup PI(\rho_2) \]

\[ PI(\rho_1; \rho_2) = PI(\rho_1) \circ PI(\rho_2) \]

\[ PI(\rho^*) = \bigcup_{n \geq 0} PI(\rho^n) \]
where \( a \in \text{Act}, \Phi \in \text{TEST} \) and \( \rho, \rho_1, \rho_2 \) are programs. The operator \( \circ \) denotes the Cartesian product of program instances.

**Example 24.** Let the following program be given:

\[ a; b; c \]

Then, its program instances are derived as follows:

\[
P I(a; b; c) = P I(a) \circ P I(c) = P I(a) \circ \{c\} =
\]

\[
P I(a) \circ P I(b) \circ \{c\} = \{a\} \circ \{b\} \circ \{c\} = \{abc\}
\]

As another example let the following program be given:

\[ (a \cup b); (c \cup d) \]

Its program instances can be derived as follows:

\[
P I((a \cup b); (c \cup d)) = P I((a \cup b)) \circ P I((c \cup d)) =
\]

\[
(P I(a) \cup P I(b)) \circ (P I(c) \cup P I(d)) = \{a, b\} \circ \{c, d\} = \{ac, ad, bc, bd\}
\]

**Definition 63 (Language of a program \( \rho \)).** The set of all possible program instances of a program \( \rho \) is called its language, \( \mathcal{L}(\rho) \).

For example, let the following program \( \rho \) be given:

\[ \rho = (\Xi?; a); ((\text{true}?; b); (\text{true}?; c))^*; (\Theta?; d)^* \]

Then some instances of \( \rho \) are:

\[ \Xi?; a \ → (\Xi?; a); (\text{true}?; b); (\text{true}?; c) \ → (\Xi?; a); (\text{true}?; b); (\text{true}?; c); (\Theta?; d)... \]

As another example let \( \rho \) be the following program:

\[ \rho = (\text{true}?; a); ((\Xi?; b) \cup (\text{true}?; c)) \]

Then the language of \( \rho \) is the following set of program instances:

\[ \mathcal{L}(\rho) = \{(\text{true}?; a); (\Xi?; b), (\text{true}?; a); (\text{true}?; c)\} \]

**Definition 64 (Equivalent program instances).** Two program instances \( p_1 \) and \( p_2 \) are equivalent, \( p_1 \equiv p_2 \), iff they are either

- *syntactically equal: \( p_1 = p_2 \)

or

- *semantically equivalent: \( I_p(p_1) = I_p(p_2) \)

Two programs \( \rho_1 \) and \( \rho_2 \) are equivalent, iff for any program instance \( p \) in \( \rho_1 \) an equivalent program instance \( p' \) in \( \rho_2 \) can be found and vice versa.

\(^2 I_p \) is the interpretation function for programs as introduced in definition 43.
In the sequel we will describe, how program instances of the form \( \Xi_1?; \Xi_2?; \ldots \Xi_n?; a \) can be transformed into program instances of the form \( \bigwedge_{i=1}^{n} \Xi_i?; \).

**Lemma 4 (Program transformation).** Let a program \( \rho \) derived by the grammar from definition 61 be given, then \( L(\rho) \) is its corresponding language. For each \( p \in L(\rho) \) we can find a semantically equivalent \( p' \) in the sense of definition 64, by applying the following rules:

(T1) Sequences of test formulae with no atomic programs, i.e. elements from \( \text{Act} \), interspersed, i.e. sequences of the kind \( \Xi_1?; \Xi_2?; \ldots \Xi_n?; \) are transformed into a conjunction of the involved test formulae:

\[
\Xi_1?; \Xi_2?; \ldots \Xi_n?; \equiv \bigwedge_{i=1}^{n} \Xi_i?;
\]

(T2) Atomic programs \( a \), not preceded by a test formula are transformed into expressions of the kind \( \text{true}?; a \).

**Proof (Lemma 4).** See Appendix B.1. \( \Box \)

The imaginary alphabet \( \Sigma \) of such transformed programs is:

\[
\Sigma := \text{TEST} \times (\text{Act} \cup \{\varepsilon\})
\]
i.e. each element of the alphabet is a tuple of test and atomic program.

**Definition 65 (Length of program instances).** Let a program \( \rho \), with \( \Sigma_\rho := \text{TEST} \times (\text{Act} \cup \{\varepsilon\}) \) be given. The length of a program instance \( p \) of \( \rho \) denoted by \( |p| \), is the number of elements from \( \Sigma_\rho \) occurring in it. For \( 0 \leq i \leq |p| \) \( p[i] \) is the \( (i+1) \)st element of \( p \). \( \text{Act}(p[i]) \) is the function that returns the atomic program part of \( p[i] \). \( \text{TeF}(p[i]) \) is the function that returns the test formula part of \( p[i] \).

**Example 25.** Let the following program instance be given:

\[
p = (\text{true}?; a)(\Xi?; b)(\text{true}?; c)
\]

We have:
- \(|p| = 3 \) is the length of \( p \),
- \( p[1] = \Xi?; b \), is its 2nd element,
- \( \text{TeF}(p[1]) = \Xi \), is the test part of the 2nd element, and
- \( \text{Act}(p[1]) = b \) is the action part of the second element.

\( \Box \)

**Definition 66 (Non-deterministic program automaton NPA).** An NPA \( N \) is defined by the quintuple \((Z_N, \Sigma_N, Z_N^{\text{Start}}, E_N, \delta_N)\):

- \( Z_N \): a finite set of states
- \( \Sigma_N := \text{TEST} \cup (\text{Act} \cup \{\varepsilon\}) \): input alphabet
- \( Z_N^{\text{Start}} \): a set of initial states, \( Z_N^{\text{Start}} \subseteq Z_N \)
- \( E_N \): a set of accepting states \( E_N \subseteq Z_N \)
- \( \delta_N \): transition function: \( \delta_N : Z_N \times \Sigma_N \rightarrow 2^{Z_N} \).
Definition 67 (Language of an NPA). The language of $N$, $L(N)$ is defined as the set of all finite sequences of elements of its input alphabet $Σ_N$ such that each sequence leads from an initial state to an accepting state:

$$L(N) := \{ p \in Σ_N^n | (z_0, p[0], z_1), (z_1, p[1], z_2), ..., (z_{n-1}, p[n], z_n) \in δ_N \wedge z_0 \in Z_N^{\text{Start}} \wedge z_n \in E_N \}$$

In this definition we have used the fact, that each $n$-ary function can be interpreted as $(n + 1)$-ary relation. We have applied it to the transition function which is binary and the interpretation as transition relation is ternary.\(^3\)

Theorem 7. For each language $L(ρ)$ there exists an NPA $N_ρ$, such that $L(ρ) = L(N_ρ)$.

The proof follows the same lines as the proof for the common non-deterministic finite automata [79], and will not be presented here.

Definition 68 (Deterministic program automaton DPA). A DPA $A$ is defined by the quintuple $(Z_A, Σ_A, z_A^{\text{Start}}, E_A, δ_A)$:

- $Z_A$: a finite set of states
- $Σ_A$: TEST $\times$ (Act $∪ \{ε\}$): input alphabet
- $z_A^{\text{Start}}$: a single initial state, $z_A^{\text{Start}} \in Z_A$
- $E_A$: a set of accepting states $E_A \subseteq Z_A$
- $δ_A$: state transition function: $δ_A : Z_A × Σ_A \rightarrow Z_A$: If a state $z$ possesses more than one outgoing transition, then it must hold, that either the action parts of the labellings of all outgoing transitions are different, or if there are at least two transitions which action parts are identical, then the test formula parts of them must fulfill the property that they can’t be true at the same time.

Note the difference between the alphabets of an NPA and a DPA. In a DPA each transition labelling is of the form $Ξ; a$, i.e. an atomic program $a$ preceded by a test formula $Ξ$. The reason for this difference is treated in section 4.4. As we will prove in section 4.4, it is possible to construct for each NPA $N$ an equivalent DPA $A$.

4.3.4 Semantics of SPDL

For the semantics of path formulae we have to define the notion of words on paths. We need this, because we have to relate the sequences of atomic programs of the DPA for program $ρ$ and the paths in the ASMC $M$.

Definition 69 (Words on paths). The word $W_k^k$ of length $k$, $k \geq 0$, over a path $σ \in \text{PATH}$ is defined as follows:

$$W^0(σ) := ε$$
$$W^k(σ) := W^{k-1}(σ) \circ a[k-1]$$

where:

$$a[k-1] \in \text{Act} \wedge σ[k-1] \xrightarrow{a[k-1]} σ[k]$$

Where $W^k(σ)[i] = a[i]$ is the $i + 1$st action on path $σ$ and the operator $\circ$ denotes the concatenation of words.

\(^3\) We will use this interpretation at many places without explicitly stating it.
We are now ready to give the formal semantics of SPDL.

**Definition 70 (Semantics of SPDL).** The semantics of state formulae is defined as follows:

\( \mathcal{M}, s \models q \iff q \in L(s) \)

\( \mathcal{M}, s \models \neg \Phi \iff \mathcal{M}, s \not\models \Phi \)

\( \mathcal{M}, s \models (\Phi \lor \Psi) \iff \mathcal{M}, s \models \Phi \) or \( \mathcal{M}, s \models \Psi \)

\( \mathcal{M}, s \models S_{\text{cop}}(\Phi) \iff \pi^\mathcal{M}(s, \text{Sat}(\Phi)) \bowtie p \)

\( \mathcal{M}, s \models P_{\text{cop}}(\phi) \iff \text{Prob}^\mathcal{M}(s, \phi) \bowtie p \)

A state satisfies \( S_{\text{cop}}(\Phi) \), if the probability \( \pi^\mathcal{M}(s, \text{Sat}(\Phi)) \) to reside in equilibrium in a \( \Phi \)-state is within \( \bowtie p \).

\( \text{Prob}^\mathcal{M}(s, \phi) \) is the probability mass of all paths \( \sigma \in \text{PATH}(s) \), starting in \( s \) to satisfy \( \phi \):

\[ \text{Prob}^\mathcal{M}(s, \phi) := \text{Pr}(\sigma \in \text{PATH}^\mathcal{M}(s) | \mathcal{M}, s \models \phi) \]

**Definition 71 (Semantics of SPDL Path Formulae).** In the following semantics definition, we assume that each element of \( L(\rho) \) has been transformed according to T1 and T2 (cf. Lemma 4).

1. \( \mathcal{M}, \sigma \models \Phi[p][0..i]| \iff \exists k \geq 0((\forall 0 \leq i < k - 1 (\mathcal{M}, \sigma[i] \models \Phi)) \land \exists p \in L(\rho)((|p| = k) \land

2. \( \forall 0 \leq i < k - 1 (\text{Act}(p[i]) = \mathcal{W}(k - 1)(\sigma)[i] \land \mathcal{M}, \sigma[i] \models \text{TeF}(p[0..i])) \land

3. \( (\text{Act}(p[k - 1]) = \epsilon \land \mathcal{M}, \sigma[k - 1] \models \text{TeF}(p[k - 1]) \land \mathcal{M}, \sigma[k - 1] \models \Psi) \lor

4. \( (\text{Act}(p[k - 1]) \neq \epsilon \land \mathcal{M}, \sigma[k - 1] \models \text{TeF}(p[k - 1]) \land \mathcal{M}, \sigma[k - 1] \models \Phi \land

5. \( \mathcal{M}, \sigma[k] \models \Psi \land \text{Act}(p[k - 1]) = a[k - 1] \land \mathcal{W}(k)(\sigma) = \mathcal{W}(k - 1)(\sigma) \land a[k - 1]) \land

6. \( \sum_{i=0}^{k^{\text{fin}}-1} t_i \leq t \)

\( k^{\text{fin}} \) is defined as follows:

\[ k^{\text{fin}} = \begin{cases} k - 1, & \text{iff } \text{Act}(p[k - 1]) = \epsilon \\ k, & \text{iff } \text{Act}(p[k - 1]) \neq \epsilon \end{cases} \]

A few remarks are in order:

- Line (2): All states from 0 to \( k - 2 \) must satisfy the precondition \( \Phi \), and there must be a word of length \( k \).
- Line (3): The actions at state \( i \) on \( \sigma \) must coincide with the action parts at position \( i \) of a word that is in the language induced by the program \( \rho \), additionally all states \( \sigma[i] \) must satisfy the test formula attached to \( p[i] \).
- Line (4): If the last action of \( p \) is \( \epsilon \), then \( |\sigma| = k - 1 \), then state \( \sigma[k - 1] \) must satisfy both the test formula attached to \( p[k - 1] \) and \( \Psi \). This stems from the fact that with \( \Sigma \alpha; \epsilon \), where \( \Sigma \) is an arbitrary test formula, no transition is associated in the model \( \mathcal{M} \).
- Line (5): If \( p[k - 1] \) is any other action than \( \epsilon \), then \( \sigma[k - 1] \) must satisfy the test formula part of \( p[k - 1] \) and \( \Phi \), as in this case in \( \mathcal{M} \) a transition from \( \sigma[k - 1] \) to \( \sigma[k] \) occurs.
- Line (6): The final state of \( \sigma \) must satisfy \( \Psi \) and the action part of \( p[k - 1] \) must coincide with one of the actions offered by \( \mathcal{M} \) in this state and the word of length \( k \) induced by \( \sigma \) is the concatenation of the word of length \( k - 1 \) generated so far and the final action.
• Line (7): Here, the time spent in states 0 to \(k^{fin}\) is summed up. This sum is the time constraint the path formula has to satisfy. \(k^{fin}\) is either \(k\) or \(k - 1\), if the last action of the word \(p\) is \(\varepsilon\), we stop in state \(k - 1\), i.e. only the sojourn times in states \(\sigma[0]\) to \(\sigma[k - 2]\) have to be summed up.

If the last action is not equal to \(\varepsilon\) we have to sum up the sojourn times of states \(\sigma[0]\) to \(\sigma[k - 1]\).

4.3.5 Derived Operators

Temporal Operators

The only temporal operator presented so far is \(\rho[0,t]^{-}\). We will show, how the operators ‘U’, ‘X’ (’next’) and ‘F’ (’finally’) can be derived. We assume \(\text{Act} = \{a_0, \ldots, a_n\}\).

The U-operator can be expressed as follows by \(\rho[0,t]^{-}\):

\[
\Phi U[0,t]^{-}\Psi := \Phi[\{a_0 \cup \ldots \cup a_n\}^\ast][0,t]^{-}\Psi \\
\Phi U\Psi := \Phi[\{a_0 \cup \ldots \cup a_n\}^\ast][0,\infty]^{-}\Psi
\]

The F-operator is expressible by the following means:

\[
F[\rho][0,t]^{-}\Psi := \text{true}[\rho][0,t]^{-}\Psi \\
F[\rho]\Psi := \text{true}[\rho][0,\infty]^{-}\Psi \\
F[0,t]^{-}\Psi := \text{true}[\{a_0 \cup \ldots \cup a_n\}^\ast][0,t]^{-}\Psi \\
F\Psi := \text{true}[\{a_0 \cup \ldots \cup a_n\}^\ast][0,\infty]^{-}\Psi
\]

whereas X can be derived as follows:

\[
X[\Xi; a][0,t]^{-}\Psi := \text{true}[\Xi; a][0,t]^{-}\Psi \\
X[\Xi; a]\Psi := \text{true}[\Xi; a][0,\infty]^{-}\Psi \\
X[0,t]^{-}\Psi := \text{true}[\{a_0 \cup \ldots \cup a_n\}][0,t]^{-}\Psi \\
X\Psi := \text{true}[\{a_0 \cup \ldots \cup a_n\}][0,\infty]^{-}\Psi
\]

Modal Operators

The modal operators \(\rho\) (’necessarily’) and \(\langle \rho \rangle\) (’possibly’) can be derived using the probabilistic path operator \(P_{\rho}\) and the derived temporal operator \(F\) as follows:

\[
\langle \rho \rangle\Psi := P_{\rho}[0,\infty]^{-}\Psi \\
\rho\Psi := \neg\langle \rho \rangle\neg\Psi
\]

4.4 Automata Construction

For model checking SPDL-path formulae \(\Phi[0,t]^{-}\Psi\) it is necessary to derive a deterministic program automaton from the program \(\rho\). The derivation process from a given program \(\rho\) to its corresponding DPA \(A_\rho\) consists of four steps:

1. Using standard algorithms from classical automata theory [79] we can derive from a program \(\rho\) its corresponding NPA \(N_\rho\).
2. In this step the automaton $N'_p$ is transformed to $N_p$, such that all transition labellings are of the form $\text{TEST} \times \text{Act}$ or $\text{TEST} \times \text{\{\epsilon\}}$. In the latter case the transitions bearing such labellings must lead to absorbing accepting states. The program instances that form the language of $N_p$ are of the form as described by transformation rules $T1$ and $T2$.

3. Using standard algorithms from automata theory, $N_p$ can be transformed into an equivalent deterministic automaton $A'_p$.

4. Finally, $A'_p$ is transformed to $A_p$, such that all transitions of $A_p$ meet the requirements of definition 68. This determinisation is necessary, if $A'_p$ has transitions with labellings $\Xi?;\ a$ and $\Psi?;\ a$ that emanate from the same state. This kind of transitions is called transitions with ambiguous tests.

### 4.4.1 Building the NPA $N_p$ from $N'_p$

On construction of the automaton $N'_p$ it might happen that transitions are generated that are labelled with tests having an empty atomic program suffix, i.e. are of the form $\Xi?;\ \epsilon$. If the target state of such transitions is not an absorbing and accepting state, such transitions have to be treated in a special way. In the sequel we will use the following shorthands:

- We will write shortly $\Xi?$; for $\Xi?;\ \epsilon$ and $a$ for true$?;\ a$.
- $X$ is either of the form $a$ or $\Theta?;\ a$.
- $Y$ is either of the form $\Lambda?;\ b$ or $\Lambda?;\ b$ or $\Lambda_1?;\ b_1 \cup \ldots \cup \Lambda_n?;\ b_n$.
- Let $z_Q$ be the source state of $\Xi?$;-transitions and $Z_Y$ the (set of) target states of $Y$-transitions.

The following 'rules' can be applied to remove internal pure test transitions, i.e. transitions with a labelling that consists only of a test.

**R1** Let $z_j$ be a non-accepting state, possessing loops of the form $X$, incoming transitions of the kind $\Xi?$; and outgoing transitions $Y$. Replace the $\Xi?$;-transition from $z_Q$ to $z_j$ by $\Xi?$; $X$ and add to $z_Q$ $\Xi?$; $Y$-transitions with target states from $Z_Y$.

**R2** Let $z_j$ be a non-accepting state, with no loops, but with incoming transitions $\Xi?$; and outgoing transitions $Y$. Then, replace in $z_Q$ each $\Xi?$;-transition by $\Xi?$; $Y$-transitions with target states from $Z_Y$. The $\Xi?$;-transition can be deleted.

**R3** Let $z_j$ be an accepting state, possessing loops of the kind $X$, no outgoing transitions, but incoming transitions of the form $\Xi?$; . Replace the $\Xi?$;-transition by $\Xi?$; $X$ and add to $z_Q$ a new $\Xi?$;-transition that leads to an absorbing and accepting state. This state has possibly to be newly introduced.

**R4** Let $z_j$ be an accepting state, possessing loops of the kind $X$, incoming transitions of the kind $\Xi?$; and outgoing transitions $Y$. Replace the ingoing $\Xi?$;-transition by $\Xi?$; $X$ and add to $z_Q$ $\Xi?$; $Y$-transitions with target states from $Z_Y$. Add to $z_Q$ a new $\Xi?$;-transition to an absorbing accepting state. This state has possibly to be newly introduced.

**R5** Transition labellings of the kind $\Phi?;\ \Psi?;\ a$ and $a$ are replaced by $(\Phi \land \Psi)\ ?;\ a$ respectively true$?;\ a$.

**R6** Let $z_j$ be a non-accepting state, with no loops, but with incoming transitions $\Xi?$; and no outgoing transitions. Then transition $\Xi?$; to $z_j$ can be deleted.

The application of rules $R1$ to $R6$ might render states unreachable, such states can be eliminated on-the-fly, i.e. no additional reachability analysis has to be performed. After this transformation the imaginary input alphabet is $\text{TEST} \times (\text{Act} \cup \{\epsilon\})$.

**Example 26.** Given the program $\rho = (c;\ a \cup d;\ \Xi?;)\ \Lambda?;\ b$. Program $\rho$ consists of the following parts:
\( \rho_1 = c; a \)
\( \rho_2 = d; \Xi? \)
\( \rho_3 = \Lambda?; b \)

For \( \rho_1 \cup \rho_2 \) and \( \rho_3 \) we obtain the automata shown in figure 4.2. Initial states are marked with an incoming arrow. Accepting states are marked with a double circle. Combining

![Automata for \( \rho_1 \cup \rho_2 \), (a) and \( \rho_3 \), (b)](image)

the automata from figure 4.2 yields the nondeterministic automaton as shown on top of figure 4.3. We have applied the rule of sequential composition of two automata \( A_1 \) and \( A_2 \):

- The initial states of \( A_1; A_2 \) are those of \( A_1 \), the accepting states are those of \( A_2 \).
- Every transition of \( A_1 \) whose target state is an accepting state copied and leads after composition also to each initial state of \( A_2 \).

Transitions labelled with \( a \) and \( \Xi? \) having absorbing non-accepting states as target states are a consequence of the second rule of constructing the sequential composition of two non-deterministic automata, as described above. Determinising and application of the transformation rules for internal test transitions yields the automaton shown on bottom of figure 4.3. 

**Lemma 5 (Correctness of the Transformation Rules).** The transformation rules R1 to R6 are correct in the sense that the automata that are generated this way are equivalent with the original automata.

**Proof.** See Appendix B.2.

### 4.4.2 Determinisation of \( N_\rho \)

The automaton \( N_\rho \) constructed by the procedure as described so far might be non-deterministic. For model-checking purposes it is necessary to derive from \( A'_\rho \) its deterministic version, \( A_\rho \). Non-determinism might stem from two sources and is treated in the following manner:
4.4 Automata Construction

Test transition elimination according to rule 2

Test transition elimination according to rule 6

Determinisation

Fig. 4.3. Stepwise construction of $N_\rho$ from $N_\rho'$

- Syntactic non-determinism: In this case a single state has two or more outgoing transitions that are labelled with the same atomic program or the same pair of test and atomic program. In this case $N_\rho$ can be made deterministic by applying standard algorithms from automata theory [79]. ($N_\rho$ is transformed to $A_\rho'$.)

- Semantic non-determinism: In $A_\rho'$ ambiguous tests might occur, i.e. for the same state $z$ several outgoing transitions might exist having the same atomic program $a$ but different test formulae $\Xi_i$, $1 \leq i \leq m$. In the model $\mathcal{M}$ in which the test formulae are interpreted it is not necessarily the case that only one of the $\Xi_i$ is true while all others are false. In such cases where several test formulae are satisfied the successor state in the model that is to be model-checked is not uniquely defined, therefore we have to provide means to combat this problem. The automaton obtained by applying this procedure will be called $A_\rho$. 
Elimination of ambiguous tests from $A'_\rho$

In the sequel, we assume that from $N_\rho$, the syntactic-deterministic automaton $A'_\rho$ has been derived. The algorithm in figure 4.4 removes ambiguous transitions from $A'_\rho$ and computes the transition function of $A_\rho$. The automaton $A_\rho$ is defined as follows:

$$A_\rho = (Z, \Sigma_{A_\rho}, z^{start}, E_{A_\rho}, \delta_{A_\rho})$$

$Z$ is the power set of the states of $A'_\rho$, $z^{start}$ is the initial state of $A'_\rho$, $E_{A_\rho}$ is the set of accepting end states of $A'_\rho$.

Algorithm 4.4 deserves some words of explanation:

- Line (2): $Z$ is a set of states, it is a subset of states of $A'_\rho$.
- Line (4): $F$ is the set of all test formulae that are attached to transitions that emanate from states in $Z$.
- Line (8): In $F$ all elements are eliminated that contain a formula and its negation.
- Line (13): All elements of $Con''$ which contain only negated formulae are eliminated, as these conjunctions do not contribute to the execution of a program, as all conditions that enable a program are unsatisfiable.
- Line (15): The new transition relation is computed, by applying the same ideas that exist for computing the transition relation of a deterministic finite automaton out of the transition relation of a non-deterministic program automaton (cf. [79]).
We will now illustrate the functionality of the algorithm from figure 4.4 by means of a small example.

Example 27. Let $\rho := (\Xi?;a)^*;\Theta?;a$. Construct $A_\rho$. The construction of $N_\rho$ is straightforward, only the final result is displayed in figure 4.5: The transitions $\Xi?;a$ and $\Theta?;a$ will be replaced in $A_\rho$ by:

\[
\begin{align*}
\Xi\land \Theta\land \Xi &;a \\
\Xi\land \Theta\land \neg \Xi &;a \\
\neg \Xi\land \Theta\land \Xi &;a
\end{align*}
\]

be replaced in $A_\rho$ by: $(\Xi \land \Theta)?;a$, $(\neg \Xi \land \Theta)?;a$ and $(\Xi \land \neg \Theta)?;a$, all emanating from $z_i$. The successor states for the respective transitions are as follows:

\[
\begin{align*}
\delta_A(z_i, (\Xi \land \Theta)?;a) &= \{z_i, z_j\} \\
\delta_A(z_i, (\neg \Xi \land \Theta)?;a) &= \{z_j\} \\
\delta_A(z_i, (\Xi \land \neg \Theta)?;a) &= \{z_i\} \\
\delta_A(\{z_i, z_j\}, (\Xi \land \Theta)?;a) &= \{z_i, z_j\} \\
\delta_A(\{z_i, z_j\}, (\neg \Xi \land \Theta)?;a) &= \{z_j\} \\
\delta_A(\{z_i, z_j\}, (\Xi \land \neg \Theta)?;a) &= \{z_i\}
\end{align*}
\]

This yields the automata graph for $A_\rho$, as shown in fig. 4.6. ☑
A few remarks on the relation of deterministic and non-deterministic program automata and the meaning of equivalence in this context are in order:

### 4.4.3 On the Relation between NPA and DPA for SPDL Programs

For SPDL deterministic and non-deterministic program automata cannot be considered to be equivalent in the sense of finite automata as known from language theory, i.e. both automata accept the same language.

In our context equivalence of deterministic and non-deterministic program automata has to be interpreted as equivalence with respect to executability of programs. We will show in theorem 8 that DPAs and NPAs are equivalent in this sense.

Given a model $\mathcal{M}$ over which the tests are interpreted we will show that the programs $\rho_{DPA}$ and $\rho_{NPA}$ are equivalent, i.e. if $\rho_{DPA}$ leads to an accepting state in $A_{\rho_{DPA}}$ then $\rho_{NPA}$ also leads to an accepting state in $A_{\rho_{NPA}}$ and vice versa.

The motivation for this non-standard interpretation can be best explained by means of an example:

**Example 28.** For model checking to yield correct result it is necessary to preserve the stochastic behaviour of $\mathcal{M}$. For an artificial example, we will see that this requirement might be violated (cf. Fig. 4.7). We call tests $\Theta$ and $\Xi$ ambiguous tests, as the action part of the transition label is identical. If we assume that in state 4 of $\mathcal{M}$ the formula $\Psi$ of an imaginary path formula $\phi_{im} := \Phi[(\Theta?; a)^{+}; \Xi?; a]^{0,1}\Psi$ holds, then the lower path of $M_{A_{\rho}^{\prime}}^{\prime}$ the $\Xi$-path leads to the FAIL-state, as once in state $B$ of the automaton for the program $\rho$ of $\phi_{im}$ no transition is possible. In contrast, the upper path, the $(\Theta, \Theta, \Xi)$-path leads to state 4 of $\mathcal{M}$ and to an accepting state in the automaton for $\rho$, therefore this path is a satisfying path. But, as both paths can be taken in state $(1, A)$ we doubled the rate $\lambda$, which modifies the stochastic behaviour of $\mathcal{M}$ and would therefore lead to wrong results during model checking. In Fig. 4.8, we find the correct product Markov chain. The DPA $A_{\rho}$ was derived applying algorithm 4.4. We see that the product Markov chain in Fig. 4.8 preserves the branching and therefore the stochastic behaviour of $\mathcal{M}$. In contrast to $A_{\rho}^{\prime}$ in $A_{\rho}$ no two
transitions are activated at the same time. The tests in $A_{\rho}$ are disjoint, therefore when $M$ is in state 1 and $A_{\rho}$ is in state $A$ only the transition with labelling $(\Theta \land \Xi)?; a$ to state $AB$ can be taken, leading in the product Markov chain $M'_{A_{\rho}}$ to the unique successor state $(2, AB)$.

\[
\begin{array}{c}
A_{\rho}: \\
\begin{array}{c}
\xrightarrow{(\Theta \land \Xi)?; a} A \\
\xrightarrow{(-\Theta \land \Xi)?; a} B \\
\xrightarrow{(\Theta \land \Xi)?; a} AB
\end{array}
\end{array}
\prod
\begin{array}{c}
M: \\
\begin{array}{c}
\xrightarrow{1, \lambda} 1 \\
\xrightarrow{2, \lambda} 2 \\
\xrightarrow{3, \lambda} 3 \\
\xrightarrow{4, \lambda} 4
\end{array}
\end{array}
\begin{array}{c}
\Phi, \Theta, \Xi \rightarrow \\
\Phi, \Xi \rightarrow \\
\Phi, \Theta \rightarrow \\
\Psi
\end{array}
\]

\[
\begin{array}{c}
M'_{A_{\rho}}: \\
\begin{array}{c}
\xrightarrow{1, A} 1 \\
\xrightarrow{2, AB} 2 \\
\xrightarrow{3, A} 3 \\
\xrightarrow{4, B} 4
\end{array}
\end{array}
\begin{array}{c}
\Phi, \Theta, \Xi \rightarrow \\
\Phi, \Xi \rightarrow \\
\Phi, \Theta \rightarrow \\
\Psi
\end{array}
\]

Fig. 4.8. Correct product Markov chain for $\phi_{lm}$

Having made all these considerations, we will finally state the following theorem which was already mentioned in section 4.3:

**Theorem 8.** Given a program $\rho$, then for each NPA $N'_{\rho}$ an equivalent DPA $A_{\rho}$ can be constructed.

**Proof.** See Appendix B.3.

### 4.5 Model Checking SPDL

The model checking procedure for SPDL is based on that of CTL, where each formula $\Phi$ is divided in its subformulae $\Phi_i$, cf. definition 72. For each subformula $\Phi_i$ its corresponding satisfaction set $Sat(\Phi_i)$ is computed. The core algorithm for computing $Sat(\Phi_i)$ can be found in fig. 4.9. The computation of the satisfaction set of propositional logic formulae is trivial. The model checking procedure for computing stationary state formulae $S_{spt}(\Phi)$ is the same as for CSL, which was described in section 2.7 and more thoroughly in [15]. In the sequel we will only describe the model checking algorithm for probabilistic path formulae. We will present firstly an approach that is based on the solution of integral equations and secondly an approach that makes use of transient analysis to compute $Sat(P_{spt}(\Phi \ [\rho]^{[0,t]} \Psi))$.

**Definition 72 (Subformulae of SPDL Formulae).** Let $\Phi, \Psi$ be SPDL formulae, the set $SF$ of subformulae is recursively defined as follows:

\[
\begin{align*}
SF(q) &= \{q\} \\
SF(\neg \Phi) &= SF(\Phi) \cup \{\neg \Phi\} \\
SF(\Phi \lor \psi) &= SF(\Phi) \cup SF(\psi) \cup \{\Phi \lor \psi\} \\
SF(S_{spt}(\Phi)) &= SF(\Phi) \cup \{S_{spt}(\Phi)\} \\
SF(P_{spt}(\Phi \ [\rho]^{[0,t]} \Psi)) &= SF(\Phi) \cup SF(\Psi) \cup \{P_{spt}(\Phi \ [\rho]^{[0,t]} \Psi)\}
\end{align*}
\]
4.5.1 Model Checking Probabilistic Path Formulae by Solving Integral Equations

At first we recall from subsection 4.3.2 that probabilistic path formulae are of the form: $P_{opt}(\Phi[p][0,\ell]\Psi)$, with $\Phi$ and $\Psi$ state formulae and $\rho$ an SPDL program. From $M$ only those paths are relevant for the requirement that generate program instances on paths that are instances of $\rho$, i.e. those instances that lead in $A_\rho$ from the initial state to an accepting state.

For a state $z \in Z_{A_\rho}$ we define its activation set:

**Definition 73 (Activation set).** Given a DPA $A_\rho$ with state set $Z_{A_\rho}$. For an arbitrary state $z$ of $Z_{A_\rho}$ we define

$$L(z) := \{a \in \Sigma_{\rho} | \exists z' \in Z_{A_\rho} (\delta_{A_\rho}(z, a) = z')\}$$

i.e. $L(z)$ is the set of all elements from $\Sigma_{\rho}$ that emanate from $z$.

Furthermore, $\text{Prob}(s, \Phi[p][0,\ell]\Psi) = W(s, \Phi[p][0,\ell]\Psi, z_{\rho}^{\text{start}})$, which will be characterised as follows:

$$W(s, \Phi[p][0,\ell]\Psi, z_{\rho}) = \begin{cases} \frac{1}{t} & \text{if Case 1} \\ 0 & \text{if Case 2} \\ \int_0^t e^{-E(s)x} \cdot \sum_{a \in L(z_{\rho})} \cdot \sum_{z' \in Z_{A_\rho}} R_a(s, s') \cdot W(s', \Phi[p][0,\ell]\Psi, \delta_{A_\rho}(z_{\rho}, a)) dx & \text{if Case 3} \end{cases}$$  \hspace{1cm} (4.1)

We denote by $l \cap x$ the following difference: \{t' - x | t' \in [0, t] \land t \geq x\}.

- **Case 1:** If the current state $s$ in $M$ is a state in which $\Psi$ holds and $z_{\rho}$ in $A_\rho$ is an accepting state then the probability that formula $\phi$ is satisfied is equal to one.  
  
  - **Case 2:** If $s$ in $M$ is a state that satisfies neither $\Phi$ nor $\Psi$, then the probability to satisfy $\phi$ is equal to zero. The same holds for a state $s$ that satisfies $\neg \Phi \land \Psi$, but $A_\rho$ is not in an accepting state and no transition from $z_{\rho}$ labelled with $\Xi$ in $A_\rho$ exists, such that $s$ satisfies $\Xi$ and the target state of the automaton transition is accepting.  

---

**Fig. 4.9.** Model Checking Algorithm for SPDL
probability to satisfy $\phi$ is zero, iff $s$ is a state satisfying $\Phi \land \Psi$, but not $\Xi$ and the current $A_{\rho}$-state is not accepting and the only transition leaving $A_{\rho}$ is labelled with $\Xi$?.

Formally:

$$(M, s \models \neg \Phi \land \neg \Psi) \text{ or } (M, s \models \Phi \land \Psi \land z_{\rho} \notin E_{\rho} \land \neg \exists \Xi \in Sat(s)(\delta_{\rho}(z, \Xi?) \in \delta_{\rho}(z, \Xi?) \in E_{\rho})) \text{ or } (M, s \models (\Phi \land \Psi) \land L(z_{\rho}) = (\Xi?) \land \delta_{\rho}(z_{\rho}, \Xi?) \in E_{\rho} \land M, s \models \neg \Xi)$$

- Case 3: If $s$ in $M$ is a $\Phi$-state then the probability to satisfy $\phi$ is equal to the probability to leave $s$ in $x$ time units and reach a state $s'$. This probability is taken over all atomic programs for which $\delta_{A}$ is defined. This probability is multiplied with the probability to reach within $t \land x$, or equivalently $t' - x$, time units a successor state $s'$ in $M$. As it might be the case that $A_{\rho}$ offers several, differently labelled, outgoing transitions from its current state to some successor states the probabilities have to be summed up over all these different labellings.

Formally:

$$M, s \models \Phi \land \neg \Psi$$

When characterising the probabilities via systems of integral equations, a numerical, approximate procedure to solve them can be used. But the convergency of such methods is not satisfactory, therefore, like in [15] we propose the approach to compute these probabilities via transient analysis.

### 4.5.2 Model Checking Probabilistic Path Formulae by Transient Analysis

To be able to do model checking of probabilistic path formulae by transient analysis, it is necessary to construct a product Markov chain $M^{\times}$, from the ASMC $M$ and the deterministic program automaton $A_{\rho}$, i.e. $M := M \times A_{\rho}$. The construction process roughly proceeds as follows:

The transition labellings $a \in \text{Act}$ are omitted. Rate informations and state labellings in $M^{\times}$ are taken from $M$. Let $s$ be the current state in $M$ and $z$ the current state in $A_{\rho}$, then transitions from $s$ having labellings that do not correspond to any of the labellings of transitions emanating from $z$ are directed in $M^{\times}$ to an absorbing error state $\text{FAIL}$. Transitions in $M^{\times}$ are not directed to $\text{FAIL}$ if the current $M$-state $s$ satisfies $\Phi$ and offers a transition whose labelling corresponds to one of the labellings of the current state $z$ in the DPA or if state $s$ satisfies $\Phi$ and $\Xi$ and offers a transition with labelling $a$, and in $A_{\rho}$ $z$ possesses a transition labelled $\Xi?$; $a$. If in $M^{\times}$ an accepting state or goal state is reached, i.e. a state which components $s$ and $z$ are states, satisfying $\Psi$, respectively are an accepting state in the DPA, the procedure stops, i.e. accepting states are absorbing.

The general idea behind our method is to reduce the model checking problem for probabilistic path formulae in SPDL to the model checking problem of CSL. This means, we transform the SPDL formula $\Phi[\rho]^{[0,1]} \Psi$ into the CSL formula $F^{[0,1]}_{\chi_{M^{\times}}}$, where $\chi_{M^{\times}}$ is a characteristic formula which is attributed to those states in the product Markov chain, whose ASMC components satisfy $\Psi$ and which automaton components are accepting states of the program automaton.

The product Markov chain $M^{\times}$ is a state-labelled Markov-chain which is defined as follows:
Definition 74 (State-labelled Product Markov-chain, SPMC). Let the ASMC \( \mathcal{M} = (S, A, L, R) \) and the DPA \( A_p = (Z_p, \Sigma_p, z_p^{\text{start}}, E_p, \delta_p) \) be given. The SPMC \( \mathcal{M}^* = (S^*, R^*, L^*) \) is defined as follows:

- **state space**: \( S^* := \{ (s_i, z_p^i) | s_i \in S \land z_p^i \in Z_p \} \cup \{ \text{FAIL} \} \)
- **initial states**: \( S^*_{\text{Init}} := \{ (s_i, z_p^{\text{start}}) | s_i \in S \} \)
- **accepting states**: \( S^*_{\text{Acc}} := \{ (s_i, z_p^i) \in S^* | s_i \in \text{Sat}(\Psi) \land z_p^i \in E_p \} \)
- **labelling**:
  1. \( \forall (s_i, z_p^i) \in S^* \setminus S^*_{\text{Acc}} \), \( L^*(s_i, z_p^i) = L(s_i) \)
  2. \( \forall (s_i, z_p^i) \in S^*_{\text{Acc}} \), \( L^*(s_i, z_p^i) = \{ \chi_M^* \} \)
  3. \( L^*(\text{FAIL}) = \{ \text{FAIL} \} \)
- **transition relation**: \( R^* \subseteq (S \times Z) \times R^+ \times (S \times Z) \) as defined below

\( \chi_M^* \) is a state formula that characterises exactly those states whose automaton part is an accepting state and whose Markov chain part is a state in which the formula \( \Psi \) of the path formula \( \Phi[\rho]^{[0,1]} \Psi \) holds. Also, \( \text{FAIL} \) is a characteristic state formula that is associated with the error state.

Definition 75. For \( A, B \in 2^{S^* \times R^+ \times S^*} \) with \( B = \emptyset \) or \( |B| = 1 \), \( A \cup B \) is defined as follows:

- \( B = \emptyset \): \( A \cup B = A \)
- \( |B| = 1 \) and \( B = \{ (s, \lambda, s') \} \):
  \[
  A \cup B = \begin{cases} 
    A \cup B & \text{iff } \not \exists \gamma \in R^+ (\{ (s, \gamma, s') \} \subseteq A) \\
    (A \setminus \{ (s, \gamma, s') \}) \cup \{ (s, \gamma + \lambda, s') \} & \text{otherwise}
  \end{cases}
  \]

\( R^* \) is successively defined as follows:

1. In the SPMC no accepting state has been reached. The original state \( s \) in \( \mathcal{M} \) satisfies \( \Phi \). \( \mathcal{M} \) offers transitions with labelling \( a \), so does \( A_p \).

\[
R^* \cup \{ (s, z_p), \lambda, (s', z_p') | s \xrightarrow{a, \lambda} s' \land z_p \xrightarrow{a} z_p' \land \\
\quad s \in \text{Sat}(\Phi) \land \\
\quad (s, z_p) \not\in S^*_{\text{Acc}} \}
\]

2. In the SPMC no accepting state has been reached. The original state \( s \) in \( \mathcal{M} \) satisfies \( \Phi \). \( A_p \) offers a test transition with test \( \Xi \gamma \); and atomic program \( a \). \( \mathcal{M} \) offers transitions with labelling \( a \) and satisfies the test formula of the corresponding transition in the DPA.

\[
R^* \cup \{ (s, z_p), \lambda, (s', z_p') | s \xrightarrow{a, \lambda} s' \land z_p \xrightarrow{\Xi \gamma, a} z_p' \land \\
\quad s \in \text{Sat}(\Phi) \land \\
\quad (s, z_p) \not\in S^*_{\text{Acc}} \land \\
\quad s \in \text{Sat}(\Xi) \}
\]

3. In the SPMC no accepting state has been reached. The original state \( s \) in \( \mathcal{M} \) satisfies \( \Phi \). \( A_p \) offers in \( z \) a transition with labellings from \( \text{Act}_p \) the target state of this transition offers a transition with a labelling from \( \text{TEST} \), say \( \Theta \). \( \mathcal{M} \) satisfies \( s \) the test formula of the corresponding \( z \)-transition. The target state of \( \mathcal{M} \) satisfies \( \Psi \) and \( \Theta \).
In the SPMC no accepting state has been reached. The original state $s$ in $M$ satisfies $\Phi$. $M$ offers in $s$ a transition labelled with $a$. $A_\rho$ does not offer a transition bearing such a labelling.

$$\models R \times \{(s, z_\rho), \lambda, (s', z''_\rho)\} \leftarrow s \xrightarrow{a, \lambda} s' \wedge z_\rho \xrightarrow{\Xi, a} z''_\rho \wedge$$

$$(s, z_\rho) \in S_{\text{Sat}(\Phi)} \wedge$$

$$(s, z_\rho) \in S_{\text{Sat}(\Xi)} \wedge$$

$$(s, z_\rho) \notin S_{\text{Acc}} \wedge$$

$$(s', z''_\rho) \not\in S_{\text{Acc}}$$

5. In the SPMC no accepting state has been reached. The original state $s$ in $M$ does not satisfy $\Phi$. The source state of $M^{-\rho}$ is not accepting.

$$\models R \times \{(s, z_\rho), \lambda, \text{FAIL}\} \leftarrow s \notin S_{\text{Sat}(\Phi)} \wedge$$

$$(s, z_\rho) \notin S_{\text{Acc}}^{-\rho}$$

6. In the SPMC no accepting state has been reached. The original state $s$ in $M$ satisfies $\Phi$. Both $M$ and $A_\rho$ offer in $s$ resp. $z$ a transition labelled with $a$, where in $A_\rho$ $a$ is preceded by a test. $M$ does not satisfy the test formula in its current state $s$.

$$\models R \times \{(s, z_\rho), \lambda, \text{FAIL}\} \leftarrow s \xrightarrow{a, \lambda} s' \wedge z_\rho \xrightarrow{\Xi, a} z''_\rho \wedge$$

$$(s, z_\rho) \in S_{\text{Sat}(\Phi)} \wedge$$

$$(s, z_\rho) \notin S_{\text{Acc}}^{-\rho} \wedge$$

$$s \notin S_{\text{Sat}(\Xi)}$$

7. In the SPMC no accepting state has been reached. The original state $s$ in $M$ satisfies $\Phi$. Both $M$ and $A_\rho$ offer in $s$ resp. $z$ a transition labelled with $a$, where in $A_\rho$ $a$ is preceded by a test. The target state $z''_\rho$ is not accepting and offers a transition with labelling from $\text{TEST}$ to an accepting state $z'''_\rho$. The target state $s'$ of $M$ does not satisfy $\Theta$.

$$\models R \times \{(s, z_\rho), \lambda, \text{FAIL}\} \leftarrow (s \xrightarrow{a, \lambda} s' \wedge z_\rho \xrightarrow{\Xi, a} z''_\rho \xrightarrow{\Theta, a} z'''_\rho) \wedge$$

$$(s, z_\rho) \notin S_{\text{Acc}}^{-\rho} \wedge$$

$$s' \notin S_{\text{Acc}}^{-\rho} \wedge$$

$$z''_\rho \in \text{E}_\rho \wedge$$
We can make the following assumptions:

1. The \textit{FAIL}-state is absorbing, i.e. it has no outgoing transitions.
2. For all states \((s, z) \in S^\times_{\text{Acc}}\), the labelling is \(\chi_{M^\times}\), i.e. \(L((s, z)) = \{\chi_{M^\times}\}\).
3. All states \((s, z) \in S^\times_{\text{Acc}}\) are also absorbing.
4. Finally, all states \((s, z) \in S^\times_{\text{Acc}}\) can be replaced by a newly introduced goal state, \textit{SUCC}, to which all transitions that lead to these states can be redirected. The labelling of this state is \(\chi_{M^\times}\), i.e. \(L(\text{SUCC}) = \{\chi_{M^\times}\}\).

4.5.3 Correctness of Model Transformation

In this subsection we will show that our transformation is correct, i.e. we show that the probability mass of the CSL formula that is checked in model \(M \times\) is equal to the probability mass of the original formula \(\Phi[\rho][0,t]\Psi\) in the original model \(M\).

To summarise the idea of section 4.5.2 we have done the following to perform transient analysis to check probabilistic SPDL path formulae:

- Transforming \(M\) to \(M^\times\)
- thereby transforming \(\Phi[\rho][0,t]\Psi\) to \(F[0,t] \chi_{M^\times}\)

The following theorem states that the transformation steps are correct:

\textbf{Theorem 9 (Correctness of model transformation).} The transformation of \(M\) into \(M^\times\) is correct. I.e. the probability of satisfying \(\Phi[\rho][0,t]\Psi\) in \(M\) is equal to the probability of reaching \(\chi_{M^\times}\) within time \(t' \in [0,t]\) in \(M^\times\):

\[
Pr\{\sigma \in \text{Path}^M_{M^\times} \mid M, \sigma \models \Phi[\rho][0,t]\Psi\} = \\
Pr\{\sigma^\times \in \text{Path}^{M^\times}_{(s,z_0)} \mid \exists t \in [0,t'] : M^\times, \sigma^\times @ t \models \chi_{M^\times}\}
\]

\textbf{Proof.} See Appendix B.4. \(\Box\)

4.6 Example: System Model and Requirements

In order to illustrate our approach, specifying and checking performance requirements using the logic SPDL, we consider an example, see figure 4.10.

4.6.1 The System Model

The ASMC of a 4-place buffer with erroneous arrivals can be found in figure 4.10 represents a system that receives four data packets and processes them, this behaviour is repeated indefinitely.

In more detail, an arrival is modelled by action \(a\), each data packet can be error-free, arrival rate \(\lambda\), or erroneous, arrival rate \(\mu\). An erroneous data packet can be corrected \((c\omega, \gamma)\), or can not be corrected, \((e, \delta)\). If it can not be corrected, the buffer is emptied and all data packets have to be retransmitted, \((rt, \kappa)\). If all data packets are error-free or correctable, then the received date can be processed \((prc, \omega)\) and the system awaits new data.

\(^4\) Note that the assumptions 3 and 4 are only valid in case \(I = [0,t]\)
4.6 Example: System Model and Requirements

4.6.2 Performance Requirements

The subsequently defined probabilistic state formulae will be used throughout this section:

- empty := \( P_{>0}([a;a;a;a]^{[0,\infty]} \text{true}) \): This formula characterises state \( s_1 \), which makes a sequence of 4 arrivals possible.
- full := \( P_{>0}([prc]^{[0,\infty]} \text{true}) \): This formula characterises state \( s_5 \).
- error := \( P_{>0}([rt + e]^{[0,\infty]} \text{true}) \): This state formula characterises states \( s_6 \) – \( s_{10} \).
- \( s_{\text{ar}} \) := \( P_{>0}((-\text{full}[a]\cup[0,\infty])\text{true}) \): Is the probability to receive 4 data packets without error or with at most one non-correctable error within 5 time units greater than 0.9?

We can see, that characterising single states or sets of states can be done without knowledge about the low-level semantic model of a given system specification. Now, we will apply the above formulae to define some more complex SPDL formulae:

1. \( \Phi_1 := P_{>0.9}((-\text{full}[a];e;rt;a^*\cup a^*)^{[0,5]}(\text{full})) \): Is the probability to receive 4 data packets without error or with at most one non-correctable error within 5 time units greater than 0.9?
2. \( \Phi_2 := P_{<0.1}([a^*;s_{\text{ar}}?;a;(co \cup e)]^{[0,\infty]} \text{true}) \): Is the probability that the fourth packet contains a correctable or incorrectable error at most ten percent, given that all preceding packets were error-free?
3. \( \Phi_3 := P_{<0.05}((-\text{full}[a]\cup[0,10])\text{true}) \): Is the probability to reach state 5 within 10 time units, provided no packet contains incorrectable errors, at least 85 %?
4. \( \Phi_4 := P_{<0.01}([a^*;s_{\text{ar}}?;a;co]^{[0,7.3]}(\text{true})) \): Is the probability that the buffer is full after at most 7.3 time units and that the 4th packet contains a correctable error, given that all preceding packets were error free, at most one percent?

4.6.3 Building the Product Markov Chain

Consider the example system \( M \), from figure 4.10 and the requirement

\[ \Phi_4 := P_{<0.01}([a^*;s_{\text{ar}}?;a;co]^{[0,7.3]}(\text{true})) \]
We want to check whether $M$ satisfies $\Phi_4$, provided the system starts in state $s_1$. At first, we derive from $a^*; s_{ar}; a; co$ a non-deterministic automaton

The test $s_{ar}$ forms together with $a$ a single transition. Now, we have to transform $N_\rho$ into a deterministic automaton $A_\rho$ (cf. figure 4.12). In figure 4.12 we see that the labels

of the transitions emanating from state $AB$ are labelled with $\neg s_{ar}; a$ resp. $s_{ar}; a$. In figure 4.13 we can find the SPMC $M \times$ between $M$ and $A_\rho$. The state labelled with $\chi_M$ is an absorbing goal state in which the path formula functionally holds, i.e. it satisfies formula $\chi_M$, the state labelled with $FAIL$ is an absorbing error state, to which all transitions are redirected that lead to states that render the path formula unsatisfiable.

4.7 Bisimulation and Worst Case Complexity

In this short section we will prove that a variant of the well-known Markov bisimulation preserves the validity of SPDL formulae and give a worst case approximation of the space and time complexity of model checking probabilistic SPDL path formulae.

4.7.1 SPDL and Bisimulation

The required variant of Markov bisimulation has to account for the fact that beside rate information also action labels and state labels, i.e. atomic propositions must be respected to identify states as being equivalent or not equivalent, we call this bisimulation relation action-labelled Markov-AP-bisimulation. We will show by induction that the validity of SPDL formulae is preserved under action-labelled Markov-AP bisimulation.
Definition 76 (Action-labelled Markov-AP Bisimulation). Let $\mathcal{M} = (S, \text{Act}, L, R)$ be an action- and state-labelled CTMC. An equivalence relation $\mathcal{B}$ on $S$ is an action-labelled (a-l) Markov-AP bisimulation over $\mathcal{M}$, if for all $(s, s') \in \mathcal{B}$ it holds that:

1. $L(s) = L(s')$
2. $\forall C \in S/\mathcal{B} \ \forall a \in \text{Act}(R_a(s, C) = R_a(s', C))$

Where:

- $S/\mathcal{B} = \{C_1, ..., C_n\}$ is the partition of $S$ into equivalence classes $C_i$ induced by $\mathcal{B}$.
- Two states $s$ and $s'$ are called a-l Markov-AP-bisimilar, if there is an a-l Markov-AP-Bisimulation that contains both states.

Definition 77. Let $[s]_{\mathcal{B}}$ be the equivalence class of $s$ with respect to $\mathcal{B}$, then the ASMC $\mathcal{M}/\mathcal{B}$ is defined as follows:

$$\mathcal{M}/\mathcal{B} = (S/\mathcal{B}, \text{Act}_{\mathcal{M}/\mathcal{B}}, R_{\mathcal{B}}, L_{\mathcal{B}})$$

where

- $S/\mathcal{B}$ is $S/\mathcal{B} = \{C_1, C_2, ..., C_n\}$
- $\text{Act}_{\mathcal{M}/\mathcal{B}} = \text{Act}$
- $R_{\mathcal{B}}(s_{\mathcal{B}}, s'_{\mathcal{B}}) = R(s, s')$
- $L_{\mathcal{B}}([s]_{\mathcal{B}}) = L(s)$

Theorem 10. Let $\mathcal{B}$ be an a-l Markov-AP-Bisimulation, $s \in \mathcal{M}$, then:

1. $\forall \Phi(\mathcal{M}, s \models \Phi \iff \mathcal{M}/\mathcal{B}, [s] \models \Phi)$
2. $\forall \phi(\text{Prob}^M(s, \phi) = \text{Prob}^{\mathcal{M}/\mathcal{B}}([s], \phi))$

Where $\phi = \Phi[\rho]^0[0,1] \Psi$ and $\rho$ is an SPDL-program.

Proof. See Appendix B.5. $\boxdot$
4.7.2 Complexity Analysis

We consider both the worst case time and space complexity of probabilistic SPDL path formulae.

Worst Case Space Complexity

**Theorem 11 (Worst Case Space Complexity of SPDL Model Checking).** For an action- and state labelled Markov chain \( M \) and an SPDL formula \( \Phi = \mathcal{P}_{\text{cop}}(\phi) \), the worst case space complexity of the model checking procedure lies in:

\[
O(n + 2^n + 2^{2n} + (2^{2n} \cdot N)) = O(2^{2n} \cdot N)
\]

- \( n \) is the number of states of \( N_{\rho} \).
- \( 2^n \) is the state space size of \( A'_{\rho} \).
- \( 2^{2n} \) is the number of states of \( A_{\rho} \).
- \( 2^{2n} \cdot N \) is the size of the product Markov chain \( M \times \).

Proof. See Appendix B.6. □

Worst Case Time Complexity

**Theorem 12 (Worst Case Time Complexity of SPDL Model Checking).** For an action- and state labelled Markov chain \( M \) and an SPDL formula \( \Phi = \mathcal{P}_{\text{cop}}(\phi) \), the worst case time complexity of the model checking procedure lies in:

\[
O(n + (n \cdot m) + 2^n + 2^{2n} + |\Phi| \cdot ((2^{2n} \cdot N)^2 \cdot q \cdot t_{\text{max}} + (2^{2n} \cdot N)^{2.81}))
\]

\[
= O(|\Phi| \cdot ((2^{2n} \cdot N)^2 \cdot q \cdot t_{\text{max}} + (2^{2n} \cdot N)^{2.81}))
\]

where
- \( n \) is the number of states of \( N_{\rho} \).
- \( m \) is the number of transitions of \( N_{\rho} \).
- \( 2^n \) is the state space size of \( A'_{\rho} \).
- \( 2^{2n} \) is the number of states of \( A_{\rho} \).
- \( 2^{2n} \cdot N \) is the size of the product Markov chain \( M \times \).
- \( |\Phi| \) is the length of \( \Phi \).
- \( t_{\text{max}} \) is the maximum time bound in the path formulae occurring as subformulae of \( \Phi \).
- \( q \) is the maximum transition rate in \( M \).

Proof. See Appendix B.7. □

4.8 Extensions of the Logic SPDL

In this section we describe two extensions of the logic SPDL. In section 4.8.1 we extend SPDL with real time intervals, i.e. intervals of the kind \([t, t']\), where \( t \neq 0 \) is allowed. We will refer to this logic as SPDL\(^I\). In section 4.8.2 we point out, how to extend the model world of SPDL\(^I\) by immediate transitions. The logic that reasons about properties of such systems is referred to as IM-SPDL. This extension requires the introduction of a modified model checking algorithm for IM-SPDL.
4.8.1 Extending SPDL with Real Time Intervals

In this section we describe an extension of our basic logic SPDL that allows to model check path formulae with time intervals \( I = [t, t'] \), where \( t \neq 0 \) is possible.

Using such requirements it is possible to express properties that require that not only a maximum amount of time but also a minimum amount of times has to be passed until a satisfying state is reached.

**Definition 78 (Syntax of SPDL\(^1\)).** Let \( p \in [0,1] \), \( q \in AP \) and \( \triangledown \in \{<,>,\leq,\geq\} \). SPDL\(^1\) state formulae can be defined by the following grammar:

\[
\Phi := q \mid \neg \Phi \mid \Phi \lor \Phi \mid S_{\in}(\Phi) \mid P_{\in}(\Phi)
\]

Path formulae \( \phi \) are defined as follows:

\[
\phi := \Phi[p]_{t'}^t
\]

where \( I = [t, t'] \), \( t \in \mathbb{R}_0 \land t' \in \mathbb{R}_{>0} \cup \{\infty\} \). Programs are defined by the following grammar:

\[
\rho := e \mid \rho \land \rho \mid \Phi ? \rho \mid \rho_1 | (\rho)
\]

\[
\rho_1 := a | \rho_1 \lor \rho_1 \mid \rho_1 | (\rho_1)
\]

We have to adapt the original definition of the semantics of path formulae to the needs of \([t, t']\) time intervals. The definition of words on paths remains unchanged.

**Definition 79 (Semantics of Path Formulae).**

\[
\mathcal{M}, \sigma \models \Phi[p]_{t'}^t \Psi \iff
\]

\[
(1) \exists k (\forall 0 \leq i < k - 1 (\mathcal{M}, \sigma[i] \models \Phi)) \land
\]

\[
(2) (\exists p \in L(\rho) (|p| = k) \land
\]

\[
(3) \forall 0 \leq i < k - 1 (\text{Act}(p[i]) = W^{k-1}(\sigma)[i] \land \mathcal{M}, \sigma[i] \models \text{TeF}(p[i])) \land
\]

\[
(4) (\text{Act}(p[k-1]) = e \land \mathcal{M}, \sigma[k-1] \models \text{TeF}(p[k-1]) \land \mathcal{M}, \sigma[k-1] \models \Psi) \lor
\]

\[
(5) (\text{Act}(p[k-1]) \neq e \land \mathcal{M}, \sigma[k-1] \models \text{TeF}(p[k-1]) \land \mathcal{M}, \sigma[k-1] \models \Phi \land
\]

\[
(6) \mathcal{M}, \sigma[k] \models \Psi \land \text{Act}(p[k-1]) = a[k-1]) \land W^k(\sigma) = W^{k-1}(\sigma) \circ a[k-1]() \land
\]

\[
(7) ((t = 0 \land \sum_{i=0}^{k_{fin}-1} t_i \leq t') \lor
\]

\[
(8) (t \neq 0 \land (t \leq \sum_{i=0}^{k_{fin} - 1} t_i \leq t') \lor (\sum_{i=0}^{k_{fin} - 1} t_i \leq t \land (\sum_{i=0}^{k_{fin} - 1} t_i + \tau(\sigma, k_{fin}) \geq t) \land \mathcal{M}, \sigma[k_{fin}] \models \Phi)))
\]

\(k_{fin}\) is defined as follows:

\[
k_{fin} = \begin{cases} k - 1, & \text{iff } \text{Act}(p[k-1]) = e \\ k, & \text{iff } \text{Act}(p[k-1]) \neq e \end{cases}
\]

Except lines (7)-(8) that describe the time restrictions, the semantics is identical to that of SPDL without real time intervals.
In lines (7) - (8) we define the restrictions stemming from the time bounds that are imposed on paths, when a $\Psi$ state has to be reached. Line (7) expresses the time restrictions if $t = 0$. In this case the sojourn time in states up to $\sigma[k^\text{fin}]$ must not be greater than $t'$. Line (8) states, that either the sojourn times in states before $\sigma[k^\text{fin}]$ must lie within the interval $[t, t']$, or if this is not the case and their sojourn times are less than $t$, then the sum of the sojourn times so far and the sojourn time in $\sigma[k^\text{fin}]$ must be greater or equal to $t$.

Characterisation of Path Formulae by Integral Equations

We now have four cases, the notation is the same as in section 4.5.1.

$$W(s, \Phi[\rho]|^{t',t}\Psi, z_{\rho}) =$$

$$\begin{cases} 
1 & \text{if Case 1 and } t = 0 \\
0 & \text{if Case 2} \\
\int_0^{t'} e^{-E(s) \cdot x} \cdot \sum_{a \in A(z_{\rho})} \cdot \sum_{s' \in S} R_a(s, s') \cdot W(s, \Phi[\rho]^{I\boxtimes x}\Psi, \delta_{\rho}(z_{\rho}, a)) dx & \text{if Case 3} \\
\int_0^t e^{-E(s) \cdot x} \cdot \sum_{a \in A(z_{\rho})} \cdot \sum_{s' \in S} R_a(s, s') \cdot W(s, \Phi[\rho]^{I\boxtimes x}\Psi, \delta_{\rho}(z_{\rho}, a)) dx & \text{if Case 4 and } t > 0
\end{cases}$$

Except for the fourth case this characterisation is identical to the one in section 4.5.1. The second case has the following explanation:

- Case 4: If state $s$ satisfies $\Phi$ and $\Psi$ and the state is left, before the lower time bound $t$ is reached, then, the probability to fulfill $\phi$ is the probability to stay for more than $t > 0$ time units in state $s$ plus the probability to reach $s'$ from $s$ within $x$ time units, $x < t$, and to satisfy $\Phi[\rho]^{I\boxtimes x}\Psi$ along a path starting in $s'$.

Formally:

$$M, s \models \Phi \land \Psi$$

Model Checking by Transient Analysis

To apply transient analysis to path formulae with real time intervals we have to revoke the following assumptions from section 4.5.2:

- States $(s, z) \in S^x_{Acc}$ can not be made absorbing.
- States $(s, z) \in S^x_{Acc}$ can not be replaced by a single goal state $\text{SUCC}$.

4.8.2 Model Checking of Stochastic Systems with Immediate Transitions

In this section we describe an extension of SPDL, IM-SPDL. IM-SPDL shall be able to cope with models that have both immediate and Markovian transitions. To this end we have to adapt and extend semantics and model checking algorithms of SPDL.
Extended Stochastic Labelled Transition System

The semantic model of our logic IM-SPDL is an extended stochastic labelled transition system (ESLTS). An ESLTS possesses two types of transitions, immediate and Markovian transitions. Immediate transitions are untimed transitions, whereas Markovian transitions are associated with a delay time that is exponentially distributed.

Example 29. Fig. 4.14 depicts an example of an ESLTS.

\[ \begin{align*}
&\text{Fig. 4.14. Extended Stochastic Labelled Transition System} \\
\end{align*} \]

Definition 80. (Extended Stochastic Labelled Transition System) An extended stochastic labelled transition system (ESLTS) \( M := (S, L, R_I, R_M, s) \) is a quintuple, where:

- \( S \) is a finite set of states.
- \( L : S \mapsto 2^{\text{AP}} \) is the state labelling function that associates with every state \( s \in S \) the set of atomic propositions which hold in that state. \( \text{AP} \) is the set of atomic propositions.
- \( R_I : S \times \text{Act}_I \times \mathbb{P} \times S \) is the immediate transition relation, where \( \mathbb{P} = (0, 1] \). If \((s, a, p, s') \in R_I\), we write \( s \xrightarrow{a, p} s' \). \( \text{Act}_I \) is a finite set of immediate action labels, i.e. actions, that are associated with immediate transitions, and \( p \in \mathbb{P} \) is a probability. Furthermore, the probabilities associated with the immediate transitions leaving a particular state must sum up to either 0 or 1, formally:

\[ \forall s \in S \left( \sum_{a, p} p = 0 \lor \sum_{a, p} p = 1 \right) \]

- \( R_M : S \times \text{Act}_M \times \mathbb{R} \times S \) is the Markovian transition relation. \( \text{Act}_M \) is a finite set of Markovian action labels, i.e. actions, that are associated with Markovian transitions. If \((s, a, \lambda, s') \in R_M\), we write \( s \xrightarrow{a, \lambda} s' \).
- \( s \in S \) is the unique initial state of \( M \).

We require that \( \text{Act}_I \cap \text{Act}_M = \emptyset \).

Example 30. In Fig. 4.14 transitions \( s_1 \xrightarrow{a, \lambda} s_2 \), \( s_1 \xrightarrow{f, \nu} s_4 \), \( s_2 \xrightarrow{b, \mu} s_3 \), \( s_2 \xrightarrow{c, \gamma} s_4 \), \( s_4 \xrightarrow{d, \lambda} s_5 \), \( s_4 \xrightarrow{e, \sigma} s_2 \), and \( s_6 \xrightarrow{h, \zeta} s_4 \) are Markovian, transitions \( s_3 \xrightarrow{d, p_1} s_1 \), \( s_3 \xrightarrow{e, p_2} s_5 \), and \( s_5 \xrightarrow{f, \sigma} s_6 \) are...
immediate, and we need to have \( p_1 + p_2 = 1 \) and \( p_3 = 1 \). The state labelling with atomic
propositions is indicated in the figure by the sets shown besides each state. \( s_1 \) is the initial
state. ☒

Classification of States

Since an ESLTS may have two types of transitions, we also have two types of states, vanishing
and tangible states. They are similar to vanishing and tangible states in the reachability
graph of GSPN models [1].

**Definition 8.1. (Vanishing and Tangible States)** A state is called vanishing (instable) if it pos-
esses at least one outgoing immediate transition. A vanishing state is left as soon as it is entered,
i.e. its sojourn time is zero. A state is called tangible (stable) if it possesses no outgoing immedi-
gate transitions. If such a state has at least one outgoing Markovian transition, its sojourn time is
governed by an exponential distribution with a specific rate \( \lambda \).

**Example 31.** In Fig. 4.14 states \( s_1, s_2, s_4 \), and \( s_6 \) are tangible, and states \( s_3 \) and \( s_5 \) are vanish-
ing. ☒

**Definition 8.2. (Paths in \( M \))** An infinite path \( \sigma \) is a sequence of transitions of the form \( s_0 \overset{a_0 I_0}{\longrightarrow} s_1 \overset{a_1 I_1}{\longrightarrow} s_2 \ldots \) where:

- \( t_i = \tau(\sigma, i) \in \mathbb{R}_{\geq 0} \): real sojourn time in \( s_i \) before passing to \( s_{i+1} \).
- if \( a_i \in \text{Act}_M \), then \( (s_i, a_i, \lambda, s_{i+1}) \in R_M \) and \( t_i > 0 \) is the sojourn time in state \( s_i \).
- if \( a_i \in \text{Act}_I \), then \( s_i \in S \), \( (s_i, a_i, p, s_{i+1}) \in R_I \) and \( t_i = 0 \) is the sojourn time in state \( s_i \).
- \( \sigma[i] \): \((i + 1)\)st state on path \( \sigma \)
- \( \sigma@t \): state at time point \( t \).
- \( a[i] \): \((i + 1)\)st action on path \( \sigma \)

A finite path \( \sigma \) is a finite sequence of transitions of the form: \( s_0 \overset{a_0 I_0}{\longrightarrow} s_1 \overset{a_1 I_1}{\longrightarrow} s_2 \ldots s_{n-1} \overset{a_{n-1} I_{n-1}}{\longrightarrow} s_n \),
where \( s_n \) is an absorbing state (i.e. has no outgoing transitions). For finite paths \( \sigma \), \( \sigma[i] \) and \( \tau(\sigma, i) \)
are defined only for \( i \leq n \), for \( i < n \) as for infinite paths, for \( i = n \) it holds \( \tau(\sigma, i) = \infty \). The set of
all paths with initial state \( s \) is denoted \( \text{PATH}_M^M(s) \), i.e. we define \( \text{PATH}_M^M(s) := \{ \sigma | \sigma[0] = s \} \).

The logic IM-SPDL is an extension of the logic SPDL\(^l\). The syntax of IM-SPDL is identical to
that of SPDL\(^l\). We only have to face some changes in the semantics of IM-SPDL compared
with SPDL\(^l\).

**Semantics of IM-SPDL**

**Definition 8.3. (Words on paths)** The word \( \mathcal{W}^k \) of length \( k \geq 0 \), over a path \( \sigma \in \text{PATH} \) is
defined as follows:

\[
\mathcal{W}^0(\sigma) := \epsilon \\
\mathcal{W}^k(\sigma) := \mathcal{W}^{k-1}(\sigma) \circ a[k-1]
\]

where:

\[
a[k-1] \in \text{Act}_M \land \sigma[k-1] \overset{a[k-1], \lambda}{\longrightarrow} \sigma[k] \lor \\
a[k-1] \in \text{Act}_I \land \sigma[k-1] \overset{a[k-1], 0}{\longrightarrow} \sigma[k]
\]

Where \( \mathcal{W}^k(\sigma)[i] = \text{Act}(p[i]) \) is the \( i + 1 \)st action on path \( \sigma \).
Having this, the semantics of path formulae is defined exactly the same way as for SPDL\(^I\), we therefore refer to definition 79 in section 4.8.1.

**Model Checking IM-SPDL**

The general idea of model checking of IM-SPDL is almost the same as for SPDL\(^I\), see the remarks on the general idea of model checking IM-SPDL at the beginning of the chapter. With respect to model checking SPDL\(^I\), in IM-SPDL the following new things have to be done:

- A product ESLTS (PESLTS) has to be generated, which leads to a new procedure for generating the transition relation in the PESLTS.
- In the PESLTS the vanishing states have to be eliminated.

**Definition 84. (Product Extended Labelled Stochastic Transition System (PESLTS))** Let the ESLTS \( M = (S, \text{Act}, L, R_I, R_M, s, AP) \) and the DPA \( A_p = (Z_p, \Sigma_p, z_p^{\text{start}}, E_p, \delta_p) \) given. A PESLTS \( M^x = (S^x, R_I^x, R_M^x, L^x, s) \) is defined as follows:

  - state space: \( S^x := \{ (s_i, z_p^I) | s_i \in S \land z_p^I \in Z_p \} \cup \{ \text{FAIL} \} \)
  - initial state: \( s_{\text{start}}^x := (s_i, z_p^{\text{start}}) \)
  - accepting states: \( S_{\text{Acc}}^x := \{ (s_i, z_p^I) | s_i \in \text{Sat}(\Phi) \land z_p^I \in E_p \} \)
  - labelling:
    1. \( \forall (s_i, z_p^I) \in S^x \setminus S_{\text{Acc}}^x | L^x(s_i, z_p^I) = L(s_i) \)
    2. \( \forall (s_i, z_p^I) \in S_{\text{Acc}}^x | L^x(s_i, z_p^I) = \{ \chi_{M^x} \} \)
    3. \( L^x(\text{FAIL}) = \{ \text{FAIL} \} \)
  - Immediate transition relation: \( R_I^x \subseteq S^x \times p \times S^x \) where \( p \) is a probability.
  - Markovian transition relation: \( R_M^x \subseteq S^x \times R_{\geq 0} \times S^x \)

The relation \( R_I^x \) is defined in definition 85. The relation \( R_M^x \) is as defined in definition 75. \( \chi_{M^x} \) is a state formula that characterises exactly those states whose automaton part is an accepting state and whose Markov chain part is a state in which the formula \( \Psi \) of the path formula \( \Phi|\rho|^{[I]} \) holds.

**Definition 85.** \( R_I^x \) is successively defined as follows:

1. In the PESLTS no accepting state has been reached. The original state \( s \) in \( M \) satisfies \( \Phi \). \( M \) offers transitions with labelling \( a \), so does \( A_p \).

\[
R_I^x \cup \{ (s, z_p^I), p, (s', z_p^I) | s \xrightarrow{\rho} s' \land z_p^I \xrightarrow{a} z_p'^I \land \]
\[
\begin{align*}
\quad & \quad s \in \text{Sat}(\Phi) \land \\
\quad & \quad (s, z_p^I) \not\in S_{\text{Acc}}^x
\end{align*}
\]

2. In the PESLTS no accepting state has been reached. The original state \( s \) in \( M \) satisfies \( \Phi \). \( A_p \) offers a test transition with test \( \Xi \zeta \); and atomic program \( a \). \( M \) offers transitions with labelling \( a \) and satisfies the test formula of the corresponding transition in the DPA.

\[
R_I^x \cup \{ (s, z_p^I), p, (s', z_p^I) | s \xrightarrow{\rho} s' \land z_p^I \xrightarrow{\Xi_p \zeta_a} z_p'^I \land \]
\[
\begin{align*}
\quad & \quad s \in \text{Sat}(\Phi) \land \\
\quad & \quad (s, z_p^I) \not\in S_{\text{Acc}}^x \land \\
\quad & \quad (s, z_p^I) \not\in \text{Sat}(\Xi)\}
\end{align*}
\]
3. In the PESLTS no accepting state has been reached. The original state $s$ in $M$ satisfies $\Phi$. $A_\rho$ offers in $z$ a transition with labellings from $\text{Act}_\rho$ the target state of this transition offers a transition with a labelling from $\text{TEST}$, say $\Theta$. $M$ satisfies in $s$ the test formula of the corresponding $z$-transition. The target state of $M$ satisfies $\Psi$ and $\Theta$.

$$R_1 \cup \{(s, z_\rho), p, \langle s', z''_\rho \rangle \mid \begin{array}{l}
s \xrightarrow{a_\rho, s'} z_\rho \xrightarrow{z_\rho, \Theta^\rho, z'_\rho} z''_\rho \\
s \in \text{Sat}(\Phi) \land \\
(s, z_\rho) \notin S_\text{Acc}^x \land \\
(s', z''_\rho) \notin S_\text{Acc}^x \end{array}$$

4. In the PESLTS no accepting state has been reached. The original state $s$ in $M$ satisfies $\Phi$. $M$ offers in $s$ a transition labelled with $a$. $A_\rho$ does not offer a transition bearing such a labelling.

$$R_1 \cup \{(s, z_\rho), p, \text{FAIL} \mid \begin{array}{l}
s \xrightarrow{a_\rho, \Xi^\rho} z_\rho \xrightarrow{\Xi^\rho, z'_\rho} z''_\rho \\
s \in \text{Sat}(\Phi) \land \\
(s, z_\rho) \notin S_\text{Acc}^x \land \\
(z'_\rho, z''_\rho) \in S_\text{Acc}^x \end{array}$$

5. In the PESLTS no accepting state has been reached. The original state $s$ in $M$ does not satisfy $\Phi$. The source state of $M$ is not accepting.

$$R_1 \cup \{(s, z_\rho), p, \text{FAIL} \mid \begin{array}{l}
s \notin \text{Sat}(\Phi) \land \\
(s, z_\rho) \notin S_\text{Acc}^x \end{array}$$

6. In the PESLTS no accepting state has been reached. The original state $s$ in $M$ satisfies $\Phi$. Both $M$ and $A_\rho$ offer in $s$ resp. $z$ a transition labelled with $a$, where in $A_\rho$ $a$ is preceded by a test. $M$ does not satisfy the test formula in its current state $s$.

$$R_1 \cup \{(s, z_\rho), p, \text{FAIL} \mid \begin{array}{l}
s \xrightarrow{a_\rho, s'} z_\rho \xrightarrow{z_\rho, \Theta^\rho, z'_\rho} z''_\rho \\
s \in \text{Sat}(\Phi) \land \\
(s, z_\rho) \notin S_\text{Acc}^x \land \\
(s', z''_\rho) \notin S_\text{Acc}^x \land \\
s \notin \text{Sat}(\Xi) \end{array}$$

7. In the PESLTS no accepting state has been reached. The original state $s$ in $M$ satisfies $\Phi$. Both $M$ and $A_\rho$ offer in $s$ resp. $z$ a transition labelled with $a$, where in $A_\rho$ $a$ is preceded by a test. The target state $z''_\rho$ is not accepting and offers a transition with labelling from $\text{TEST}$ to an accepting state $z'''_{\rho'}$. The target state $s'$ of $M$ does not satisfy $\Theta$.

$$R_1 \cup \{(s, z_\rho), p, \text{FAIL} \mid \begin{array}{l}
s \xrightarrow{a_\rho, s'} z_\rho \xrightarrow{z_\rho, \Theta^\rho, z''_\rho} z''''_\rho \\
(s, z_\rho) \notin S_\text{Acc}^x \land \\
(s', z''''_\rho) \notin S_\text{Acc}^x \land \\
s' \notin \text{Sat}(\Theta) \land \\
z''''_\rho \notin E_{\rho'} \end{array}$$
4.8 Extensions of the Logic SPDL

Elimination of Vanishing States

Here, we only describe roughly how to eliminate vanishing states from a PESLTS. An efficient procedure for constructing the tangible portion of the reachability graph of a GSPN is described in [18]. This procedure is also applicable in our setting.

Let a PESLTS $M^\times$ be given. Its state space $S^\times$ can be partitioned as follows: $T$ is the set of tangible and $V$ the set of vanishing states. $S^\times = T \cup V$ and $T \cap V = \emptyset$. By an appropriate reordering of the states, the transition matrix of $M^\times$ can be written as shown in Fig. 4.15. In order to eliminate the vanishing states from $M^\times$ we must compute the transition probabilities among the tangible states. These can be represented as a matrix $\hat{R}^{(TT)}$ which can be expressed as follows:

$$\hat{R}^{(TT)} = R^{TT} + R^{(TV)} \cdot G$$

where the matrix $G$ is defined as follows:

$$G := \begin{cases} 
(\sum_{h=0}^{k_0-1} p^{VV}_h) \cdot p^{VT} \quad (a) \\
(I - p^{VV})^{-1} \cdot p^{VT} \quad (b) 
\end{cases}$$

where in case $(a)$ there are no loops among vanishing states and in case $(b)$ there are loops among vanishing states. $I$ is the identity matrix, and $k_0$ is the smallest index such that $p^{VV}_{k_0} = 0$.

**Example 32.** Let the ESLTS $M$ from Fig. 4.14 be given and let $\phi = true[\rho]^{[0,\bar{u}]} \Psi$ where

$$\rho = (a; \Phi_2?; b; \Phi_3?) \ast; ((\Phi_1?; a; \Phi_2?; c) + (a; b; \Phi_3?; e))$$

The non-deterministic program automaton $N_\rho$ is shown in fig. 4.16. The deterministic program automaton $A_\rho$ for this program is shown in fig. 4.17 The product ESLTS with states $(s_4, G)$ and $(s_5, E)$ replaced by SUCC is shown in Fig. 4.18. The transition $(s_1, A) \xrightarrow{v} FAIL$ can be justified as follows: In the automaton $A_\rho$, state $A$ does not offer action $f$, therefore we can redirect the transition with rate $\nu$ to the absorbing FAIL-state.

To do finally model checking we eliminate the vanishing state $(s_3, CD)$ and obtain the CTMC $M^*$ shown in Fig. 4.19. ☐

![Transition probability matrix of $M^\times$](image-url)
Fig. 4.16. Non-deterministic program automaton $N_\rho$

Fig. 4.17. Deterministic program automaton $A_\rho$

Fig. 4.18. Product ESLTS $M^\times$

Fig. 4.19. CTMC $M^*$
Comparing the Expressive Power of Stochastic Temporal Logics

In this chapter, the expressive power of several stochastic temporal will be compared. The logics to be compared are: CSL\(^t\), CSL, aCSL, aCSL+, SPDL, and SPDL\(^t\). We will prove that the logic SPDL is strictly more expressive than CSL\(^t\), aCSL and aCSL+. If we want to compare CSL and SPDL we will see that SPDL and CSL are not comparable, but we will see that SPDL\(^t\) is strictly more expressive than CSL.

For a quick overview of the results of this chapter we refer to the first paragraph of sections 5.2 resp. 5.3 In [108] it was only shown that all CSL\(^t\) and all aCSL formulae can be translated into aCSL+ formulae. The proof that aCSL+ is strictly more expressive than all these logics is missing.

5.1 Overview on Stochastic Temporal Logics

In this section we introduce briefly, and mainly informally, syntax and semantics of all considered stochastic temporal logics.

5.1.1 The logic aCSL

As already described in chapter 1 the main motivation for the developement of this logic stems from the usage of stochastic process algebras as modelling formalism. aCSL is interpreted over an action-labelled CTMC. aCSL possesses, like CSL, the probabilistic operators \(P_{\triangleleft}p\) and \(S_{\triangleleft}p\). The path operators \(U_{[0,t]}\) resp. \(X_{[0,t]}\) are replaced by \(A\cdot U_{[0,t]}\) resp. \(A\cdot U_{[0,t]}\).

**Definition 86 (Syntax of aCSL).** Let \(p \in [0,1]\) and \(\triangleleft \in \{<,>,\leq,\geq\}\), \(A, B \subseteq \text{Act}\), then the state formulae of aCSL are defined by the following grammar:

\[
\Phi ::= \text{true} \mid \neg \Phi \mid \Phi \lor \Phi \mid S_{\triangleleft}(\Phi) \mid P_{\triangleleft}(\Phi)
\]

Path formulae \(\phi\) are defined as follows:

\[
\phi ::= \Phi_A U_{[0,t]} \Phi \mid \Phi_A U_{[0,t]} \Phi
\]

Where it holds that \(t \in \mathbb{R}_{>0} \cup \{\infty\}\).

The logic CSL\(^t\) allows only time intervalls of the kind \([0,t]\), whereas CSL allows general time intervalls
Definition 87 (Length of an aCSL formula). Let $\Phi$ and $\Psi$ be state formulae, and $\phi$ a path formula. With exception of atomic propositions, the length of state formulae is defined as in the case of CSL, see definition 50. The length of path formulae is defined as follows:

$$|\phi| = |\Phi_A U^{[0,t]} \Psi| := |\Phi| + |\Psi| + 1$$

We introduce in an informal manner the semantics of aCSL. The semantics of the probabilistic operators is the same as for CSL', i.e. it suffices to explain the semantics of aCSL path-operators.

- $\Phi_A U^{[0,t]} \Psi$: A path $\sigma$ fulfills this formula, if finally a $\Psi$ state is reached and all preceeding states satisfy $\Phi$. The transition labels must be elements of the set $A$. Additionally, it must hold that the time before the first $\Psi$ state is reached must not exceed $t$ time units.
- $\Phi_A U^{[0,t]} B \Psi$: This formula is satisfied by a path $\sigma$ which after visiting an arbitrary number of $\Phi$ states, which are reachable by transitions labelled with actions from the set $A$, reaches with exactly one transition, being labelled with an action from $B$, a state that satisfies $\Psi$. Likewise, the time point at which a $\Psi$ state is reached must be within the bounds imposed by the interval $[0, t]$.

Formally, the semantics of the path operators is defined as follows:

Definition 88 (Formal Semantics of aCSL Path Formulae). A path $\sigma$ satisfies $\Phi_A U^{[0,t]} \Psi$, iff:

$$\mathcal{M}, \sigma \models \Phi_A U^{[0,t]} \Psi \iff \exists k \geq 0 (\mathcal{M}, \sigma[k] \models \Psi \land \sum_{i=0}^{k-1} t_i \leq t \land \forall i < k (\mathcal{M}, \sigma[i] \models \Phi \land \sigma[i] \overset{\Delta}{\rightarrow} \sigma[i + 1]))$$

$\sigma[i] \overset{\Delta}{\rightarrow} \sigma[i + 1]$ is an abbreviation for $\exists a \in A(\sigma[i] \overset{a}{\rightarrow} \sigma[i + 1])$, i.e. there is an $a \in A$ such that transition from $\sigma[i]$ to $\sigma[i + 1]$ is possible.

A path $\sigma$ satisfies $\Phi_A U^{[0,t]} B \Psi$, iff:

$$\mathcal{M}, \sigma \models \Phi_A U^{[0,t]} B \Psi \iff \exists k > 0 (\mathcal{M}, \sigma[k] \models \Psi \land \sum_{i=0}^{k-1} t_i \leq t \land \forall i < k - 1 (\mathcal{M}, \sigma[i] \models \Phi \land \sigma[i] \overset{\Delta}{\rightarrow} \sigma[i + 1]) \land$$

$$\mathcal{M}, \sigma[k - 1] \models \Phi \land \sigma[k - 1] \overset{B}{\rightarrow} \sigma[k])$$

5.1.2 The logic aCSL+

The logic aCSL faces some weaknesses on characterising action sequences that a satisfying paths must be able to execute. Paths are simply characterised by means of action sets, i.e. a path satisfies a requirement if an arbitrary sequence of these actions appears alongside. For example, it is impossible in aCSL to express that on a satisfying exactly two $a$ actions are followed by exactly one $b$ action and one $c$ action. To overcome this weakness in [107] the logic aCSL+ was introduced. In aCSL+ it is possible to characterise satisfying paths via arbitrarily complex regular expressions. aCSL+ is interpreted over an action- and state-labelled CTMC.
Definition 89 (Syntax of aCSL+). Let \( p \in [0, 1] \), \( q \in AP \) and \( \preccurlyeq \in \{<, >, \leq, \geq\} \). aCSL+ state formulae \( \Phi \) are defined by the following grammar:

\[
\Phi := q \mid \neg \Phi \mid \Phi \lor \Phi \mid S_{\preceq p} (\Phi) \mid P_{\preceq p} (\Phi)
\]

Path formulae \( \phi \) are defined as follows:

\[
\phi := \Phi [\rho] [0, t] \Psi
\]

where \( t \in \mathbb{R}_{>0} \cup \{\infty\} \). \( \rho \) describes admitted action sequences by means of regular expressions:

\[
\rho := \epsilon \mid a \mid \rho ; \rho \mid \rho \cup \rho \mid \rho^* \mid (\rho)
\]

Definition 90 (Length of an aCSL+ formula). Let \( q \in AP \), \( \Phi \) and \( \Psi \) be state formulae, and \( \phi \) a path formula.

The length of state formulae is defined as in the case of CSL, see definition 50. The length of path formulae is defined as follows:

\[
|\phi| = |\Phi [\rho] [0, t] \Psi| := |\Phi| + |\Psi| + 1
\]

The path operator \( [\rho] [0, t] \) has the following meaning:

- \( \Phi [\rho] [0, t] \Psi \): A path \( \sigma \) satisfies this formula, if after an arbitrary number of \( \Phi \) states finally a \( \Psi \) state is reached. The concatenation of the actions appearing as labels of the transitions under consideration must be a word from \( L_\rho \), the language induced by \( \rho \). Again, the time bounds, imposed by \( t \) must be respected, i.e. the \( \Psi \) state must be reached before the passage of more than \( t \) time units.

The formal semantics of the aCSL+ path operator is defined as follows:

Definition 91. A path \( \sigma \) satisfies \( \Phi [\rho] [0, t] \Psi \) iff:

\[
\mathcal{M}, \sigma \models \Phi [\rho] [0, t] \Psi \iff
\begin{align*}
(1) & \exists k \geq 0 (\mathcal{M}, \sigma[k] \models \Psi) \\
(2) & \sum_{i=0}^{k-1} t_i \leq t \\
(3) & \forall i < k (\mathcal{M}, \sigma[i] \models \Phi) \\
(4) & \chi^k (\sigma) \in L_\rho
\end{align*}
\]

\( \chi^k (\sigma) \) is the word of length \( k \) induced by path \( \sigma \), its definition is identical to the SPDL case (cf. definition 69 in chapter 4.3.2). The semantics can be explained as follows:

A path \( \sigma \) satisfies \( \Phi [\rho] [0, t] \Psi \), iff:

- Line (1) There is a state \( \sigma[k] \) on \( \sigma \) such that \( \sigma[k] \) satisfies \( \Psi \).
- Line (2): The sum of the sojourn times in all states before \( \sigma[k] \) is reached does not exceed the time bound \( t \).
- Line (3): All states \( \sigma[i] \), preceding \( \sigma[k] \) must satisfy \( \Phi \).
- Line (4): The word \( \chi^k (\sigma) \) induced by path \( \sigma \) must be element of the language of the regular expression \( \rho \).
5.1.3 Simplified Syntax and Semantics of SPDL

For the comparison of the expressive power of SPDL to that of other logics we can restrict ourselves to a simplified version of the syntax and semantics of SPDL. This simplification eases the proofs that follow in section 5.2. In the simplified version of SPDL, empty tests are not allowed at the end of words, i.e. tests are only allowed in connection with an action. It should be noted that every valid formula expressed in simplified SPDL is also a valid formula of the non-simplified logic SPDL from chapter 4. Then, the syntax of SPDL programs and the semantics of path formulae can be rewritten as follows:

**Definition 92 (Simplified Syntax of SPDL Programs).** Let \( \text{Act} \) be a set of atomic programs, which we may also call actions and \( \text{TEST} \) be a set of state formulae. Together with the empty program \( \epsilon \) they form the alphabet \( \Sigma_p \) for the program \( p \), i.e. \( \Sigma_p := \text{TEST} \cup (\text{Act} \cup \{\epsilon\}) \).

A program \( p \) over an alphabet \( \Sigma_p \) is defined by the following grammar:

\[
\rho ::= \epsilon \mid a \mid \rho \cup \rho \mid \rho^* \mid (p)
\]

Where \( a \in \text{Act} \) and \( \Xi \in \text{TEST} \).

From this simplified syntax we can derive a simplified semantics of SPDL path formulae:

**Definition 93 (Simplified Semantics of SPDL Path Formulae).**

\[
\mathcal{M}, \sigma \models \Phi[p]^0_{[0, t]} \Psi \iff
\begin{align*}
(1) \quad & \exists k ((\mathcal{M}, \sigma[k] \models \Psi) \land \\
(2) \quad & \sum_{i=0}^{k} t_i \leq t) \land \\
(3) \quad & (\forall i < k (\mathcal{M}, \sigma[i] \models \Phi)) \land \\
(4) \quad & (\exists p \in L(p)(|p| = k) \land \\
(5) \quad & \forall 0 \leq i < k (\mathcal{M}(p[i]) = \forall^k \sigma[i] \land \\
(6) \quad & \mathcal{M}, \sigma[i] \models T(p))[i])\)
\end{align*}
\]

The semantics can be explained as follows:

A path \( \sigma \) satisfies the path formula \( \Phi[p]^0_{[0, t]} \Psi \), if:

- Line (1): There is a state \( \sigma[k] \) on \( \sigma \) such that \( \sigma[k] \) satisfies \( \Psi \).
- Line (2): The sum of the sojourn times in all states before \( \sigma[k] \) is reached does not exceed the time bound \( t \).
- Line (3): All states \( \sigma[i] \) preceding \( \sigma[k] \) must satisfy \( \Phi \).
- Line (4): The language of program \( p \) must contain a program instance of length \( k \), i.e. of the length of the path \( \sigma \).
- Line (5): All states \( \sigma[i] \) on \( \sigma \) must satisfy the test formulae that belong to the \( i + 1 \)st element, \( p[i] \), of program instance \( p \).
- Line (6): The label of the transition from state \( \sigma[i] \) to \( \sigma[i + 1] \) on path \( \sigma \) must correspond to the action part of the \( i + 1 \)st element of program instance \( p \).
5.2 Comparison of Expressive Power

In this section we compare the expressive power of the logics presented in the preceding sections. We will show that SPDL is strictly more expressive than CSL\(^t\), aCSL and aCSL+.

The proofs of this claim always proceed along the same line and can be divided into two parts:

1. For arbitrary CSL\(^t\), aCSL and aCSL+ formulae we can find transformation rules, i.e. rules that transform formulae of the respective logics into SPDL formulae. We have to prove the correctness of these transformations.
2. Secondly, we have to show that for each of the logics CSL\(^t\), aCSL and aCSL+ there is an SPDL formula and a model for this formula that cannot be transformed into an equivalent formula of the respective logics.

This procedure can be justified using the following definition:

**Definition 94 (Expressive power of logics).** Two logics \( \mathcal{L} \) and \( \mathcal{L}' \) are considered to be equally expressive, iff the following conditions are met:

\[
\forall M (\forall \phi \in \mathcal{L} (\exists \psi \in \mathcal{L}' ((\forall s \in M ((M, s \models \phi \iff M, s \models \psi)))))) \land \\
\forall M (\forall \psi \in \mathcal{L}' (\exists \phi \in \mathcal{L} ((\forall s \in M ((M, s \models \phi \iff M, s \models \psi))))))
\]

If only the first conjunct holds, then the logic \( \mathcal{L}' \) is strictly more expressive than \( \mathcal{L} \).

Thus, we will show that on comparison of the above mentioned logics with SPDL only the first conjunct holds. At first we restrict CSL to time intervals of the kind \([0, t]\), i.e. considering only CSL\(^t\), we obtain the hierarchy with respect to expressive power as shown in figure 5.1. In the sequel \( A, B \) are used as subsets of \( \text{Act} \). \( A^* \) is a subset of \( \text{Act}^* \).

![Figure 5.1. Hierarchy of the expressive power of stochastic temporal logics](image)

5.2.1 CSL\(^t\) vs. SPDL

In this subsection we prove, that CSL\(^t\) is strictly less expressive than SPDL. In CSL action labels are of no importance. Before we start with the proof of this fact, we present an alternative semantics of the CSL-Next and CSL-Until operator, which is applicable in the case where \( I = [0, t] \):

**Definition 95 (Alternative Semantics of CSL\(^t\)-Next Operator).** The CSL-Next operator has the following formal semantics:
Recall, that $\sum_{i=0}^{0} t_i$ denotes the sojourn time in state $\sigma[0]$.

**Definition 96 (Alternative Semantics of CSL $^1$-Until Operator).** The CSL-$\text{Until}$ operator has the following formal semantics:

$$
\mathcal{M}, \sigma \models \Phi_{\text{CSL}} \mathop{\text{U}}_{0 \leq i \leq t} \Psi_{\text{CSL}} \iff \exists k \geq 0(\mathcal{M}, \sigma[k] \models \Psi_{\text{CSL}} \land \sum_{i=0}^{k-1} t_i \leq t \land \forall 0 \leq i < k(\mathcal{M}, \sigma[i] \models \Phi_{\text{CSL}}))
$$

This definition is equivalent with the original definition 51, which can be justified as follows:

- In case of CSL $^1$ we have $I = [0, t')$. Thus, $\exists t_2 \in I(\mathcal{M}, \sigma \otimes t_2 \models \Psi_{\text{CSL}})$ means, there is a $t_2 \geq 0$ that satisfies $\Psi_{\text{CSL}}$. The formula $\exists t_2 \in I(\mathcal{M}, \sigma \otimes t_2 \models \Psi_{\text{CSL}})$ can be replaced by $\exists k \geq 0(\mathcal{M}, \sigma[k] \models \Psi_{\text{CSL}} \land \sum_{i=0}^{k-1} t_i \leq t)$, as the condition to enter a $\Psi$ state after at least $t$ and at most $t'$ time units is satisfied for $t = 0$, if a $\Psi$ state is entered at a time point greater or equal to zero, but less then $t'$, which is exactly expressed by $\exists k \geq 0(\mathcal{M}, \sigma[k] \models \Psi_{\text{CSL}} \land \sum_{i=0}^{k-1} t_i \leq t)$.

- Similarly, we can replace $\forall t_1 \in [0, t_2)(\mathcal{M}, \sigma \otimes t_1 \models \Phi_{\text{CSL}}$ by $\forall 0 \leq i < k(\mathcal{M}, \sigma[i] \models \Phi_{\text{CSL}}))$. The first formula states that all states before state $\sigma \otimes t_2$, which is $\sigma[k]$ must satisfy $\Phi_{\text{CSL}}$, which is expressed by $\forall 0 \leq i < k(\mathcal{M}, \sigma[i] \models \Phi_{\text{CSL}})$.

**Lemma 6.** An arbitrary CSL $^1$ formula can be transformed into an equivalent SPDL formula by applying the subsequent transformation rules.

1. $\text{true}_{\text{CSL}} \sim (q_{\text{SPDL}} \lor \neg q_{\text{SPDL}})$
2. $q_{\text{CSL}} \sim q_{\text{SPDL}}$
3. $(\neg \Phi)_{\text{CSL}} \sim \neg q_{\text{SPDL}}$
4. $(\Phi \lor \Psi)_{\text{CSL}} \sim q_{\text{SPDL}} \lor q_{\text{SPDL}}$
5. $(\sigma_{\text{op}}(\Phi))_{\text{CSL}} \sim \sigma_{\text{op}}(\Phi)_{\text{SPDL}}$
6. $(P_{\text{op}}(\mathcal{X}^{0}[t])_{\text{CSL}} \sim P_{\text{op}}(true_{\text{SPDL}}[\text{Act}]^{0}[t]_{\text{SPDL}})$
7. $(P_{\text{op}}(\mathcal{Y}^{0}[t])_{\text{CSL}} \sim P_{\text{op}}(true_{\text{SPDL}}[\text{Act}^*]^{0}[t]_{\text{SPDL}})$

We prove this lemma by means of structural induction on the length $l$ of formulae.

**Proof (Lemma 6).**

1. Trivially it holds: $\forall s \in S_M (M, s \models \text{true}_{\text{CSL}})$. $q$ is an atomic state formula. Obviously for all states $s$ of the model $M$ it holds: $M, s \models q_{\text{SPDL}}$ or $M, s \not\models q_{\text{SPDL}}$, i.e. $\forall s \in S_M (M, s \models q_{\text{SPDL}} \lor \neg q_{\text{SPDL}})$. Thus, $\forall s \in S_M (M, s \models \text{true}_{\text{CSL}} \Leftrightarrow M, s \models q_{\text{SPDL}} \lor \neg q_{\text{SPDL}})$.

2. For all states $s$ of the model $M$ (cf. case 1 of this proof) it holds: $M, s \models q_{\text{CSL}}$ or $M, s \not\models q_{\text{CSL}}$. As CSL $^1$ and SPDL possess the same underlying model it is obvious: $M, s \models q_{\text{CSL}} \Leftrightarrow M, s \models q_{\text{SPDL}}$.

The first and second case serve as induction start (I.S.). As induction hypothesis (I.H.) we assume, that the claim of lemma 6) holds for formulae of length $< l$. 
4. To prove this case we apply similar auxiliary means as in case 3: I.H. and the semantics of disjunction.

\[ \mathcal{M}, s \models (\Phi \lor \Psi)_{\text{CSL}} \xrightarrow{\text{Sem}} \mathcal{M}, s \not\models \Phi_{\text{CSL}} \lor \mathcal{M}, s \not\models \Psi_{\text{SPDL}} \]

5. Here, we use the semantics of the steady-state operator and I.H.:

\[ \mathcal{M}, s \models (S_{\text{opt}}(\Phi))_{\text{CSL}} \xrightarrow{\text{Sem}} \pi(s, \text{Sat}(\Phi_{\text{CSL}})) \triangleright p \xrightarrow{\text{I.H.}} \]
\[ \pi(s, \text{Sat}(\Phi_{\text{SPDL}})) \triangleright p \xrightarrow{\text{Sem}} \mathcal{M}, s \models S_{\text{opt}}(\Phi_{\text{SPDL}}) \]

6. To show the correctness of this transformation rule, we apply I.H. and the semantics of the CSL\textsuperscript{i} next-operator and the semantics of the SPDL path operator:

\[ \mathcal{M}, \sigma \models (X^{0,t}\Phi)_{\text{CSL}} \iff \mathcal{M}, \sigma[1] \models \Phi_{\text{CSL}} \land \sum_{i=0}^{t} t_i \leq t. \]

This means, in \( \mathcal{M} \) there is a path \( \sigma \) with \( s_0 \xrightarrow{a_1 t_1} s_1 \), \( s_0 = \sigma[0] \) and \( \sigma[1] = s_1 \), \( a \in \text{Act}, \sum_{i=0}^{t} t_i \leq t \) and \( \mathcal{M}, s_1 \models \Phi_{\text{CSL}} \).

Now, we can apply the semantics of the SPDL path operator to this specific case:

\[ \mathcal{M}, \sigma \models \text{true}_{\text{SPDL}}[\text{Act}]^{0,t}\Phi_{\text{SPDL}} \iff \]
\[ \mathcal{M}, \sigma[1] \models \Phi_{\text{SPDL}} \land \sum_{i=0}^{t} t_i \leq t \land \exists p \in \mathcal{L}_{\text{Act}}(|p| = 1) \land 
\[ \mathcal{M}, \sigma[0] \models \text{true}_{\text{SPDL}} \land (W^{1}(\sigma)[0] = \text{Act}(p[0]) \land \mathcal{M}, \sigma[0] \models \text{TeF}(p[0])) \]

Using I.H. and the fact that the model is the same for both logics, then \( \Phi_{\text{SPDL}} \) can be replaced by \( \Phi_{\text{CSL}}, \sigma[0] = s_0, \sigma[1] = s_1 \). From the definition of \( \text{Act} \) it is obvious that a word \( p \) of length 1 exists that satisfies \( W^{1}(\sigma)[0] = \text{Act}(p[0]) \); furthermore, \( \text{TeF}(p[0]) = \text{true} \). Thus, we obtain the following reduction:

\[ \mathcal{M}, \sigma \models \text{true}_{\text{SPDL}}[\text{Act}]^{0,t}\Phi_{\text{SPDL}} \iff \mathcal{M}, s_1 \models \Phi_{\text{SPDL}} \land \sum_{i=0}^{t} t_i \leq t \]

This establishes the desired equivalence:

\[ \mathcal{M}, \sigma \models \text{true}_{\text{SPDL}}[\text{Act}]^{0,t}\Phi_{\text{SPDL}} \iff \mathcal{M}, \sigma \models (X^{0,t}\Phi)_{\text{CSL}} \]

This concludes the proof of the correctness of this transformation rule.

7. We prove the last case by applying the semantics of both the CSL\textsuperscript{i}-Until operator and the SPDL path operator and the I.H.
The assumption of the correctness of the induction hypothesis allows to replace in the semantics of CSL-Until formula to replace $\Phi_{\text{CSL}}(\Psi_{\text{CSL}})$ by $\Phi_{\text{SPDL}}(\Psi_{\text{SPDL}})$:

$$
\mathcal{M}, \sigma \models \Phi_{\text{CSL}}[\text{Act}^*]^{[0,t]}\Psi_{\text{CSL}} \iff \\
\exists k \geq 0(\mathcal{M}, \sigma[k] \models \Psi_{\text{SPDL}} \land \sum_{i=0}^{k-1} t_i \leq t \land \forall i < k(\mathcal{M}, \sigma[i] \models \Phi_{\text{SPDL}}))
$$

We apply the semantics of the SPDL path operator to the case that is determined by the transformation rule which has to be shown to be correct:

$$
\mathcal{M}, \sigma \models \Phi_{\text{SPDL}}[\text{Act}^*]^{[0,t]}\Psi_{\text{SPDL}} \iff \\
\exists k \geq 0(\mathcal{M}, \sigma[k] \models \Psi_{\text{SPDL}} \land \sum_{i=0}^{k-1} t_i \leq t \land \forall i < k(\mathcal{M}, \sigma[i] \models \Phi_{\text{SPDL}}) \land \\
\exists p \in \mathcal{L}_{\text{Act}}(|p| = k \land \forall i < k(W^k(\sigma)[i] = \text{Act}(p[i]) \land \mathcal{M}, \sigma[i] \models \text{TeF}(p[i])))
$$

We have to show that both semantic definitions are equivalent. I.e. we show for all conjuncts that they are equivalent. The semantics of the requirements $\mathcal{M}, \sigma[i] \models \text{TeF}(p[i])$, which is trivially fulfilled, as $\text{TeF}(p[i]) = \text{true}$ for all states of the path. Thus, the formula above can be simplified as follows:

$$
\mathcal{M}, \sigma \models \Phi_{\text{SPDL}}[\text{Act}^*]^{[0,t]}\Psi_{\text{SPDL}} \iff \\
\exists k \geq 0(\mathcal{M}, \sigma[k] \models \Psi_{\text{SPDL}} \land \sum_{i=0}^{k-1} t_i \leq t \land \forall i < k(\mathcal{M}, \sigma[i] \models \Phi_{\text{SPDL}}) \land \\
\exists p \in \mathcal{L}_{\text{Act}}(|p| = k \land \forall i < k(W^k(\sigma)[i] = \text{Act}(p[i])))
$$

Obviously, it only remains to prove the following: $\exists p \in \mathcal{L}_{\text{Act}}(|p| = k \land \forall i < k(W^k(\sigma)[i] = \text{Act}(p[i]))) = \text{true}$. $\mathcal{L}_{\text{Act}}$ contains, following the semantics of the Kleene star, words generated from elements of the alphabet Act of arbitrary finite length, especially words of length $k$. This part of the conjunction trivially holds. Using the fact that the words over path $\sigma$ are concatenations of elements from the set Act, and the elements of $\mathcal{L}_{\text{Act}}$ are concatenations of elements from Act as well it is obvious that $\mathcal{L}_{\text{Act}}$ contains an element, i.e. a word, such that the second conjunct $\forall i < k(W^k(\sigma)[i] = \text{Act}(p[i]))$ is true as well. Thus, it holds $\exists p \in \mathcal{L}_{\text{Act}}(|p| = k \land \forall i < k(W^k(\sigma)[i] = \text{Act}(p[i]))) = \text{true}$. Using this result we continue simplifying the semantic definition of the SPDL path operator applied to this specific case:

$$
\mathcal{M}, \sigma \models \Phi_{\text{SPDL}}[\text{Act}^*]^{[0,t]}\Psi_{\text{SPDL}} \iff \\
\exists k \geq 0(\mathcal{M}, \sigma[k] \models \Psi_{\text{SPDL}} \land \sum_{i=0}^{k-1} t_i \leq t \land \forall i < k(\mathcal{M}, \sigma[i] \models \Phi_{\text{SPDL}}))
$$

This corresponds to the semantics of the CSL $U$ operator if I.H. is used. This concludes the proof of this case. $\square$

To show that SPDL is in fact strictly more expressive than CSL we have to prove the following lemma:

**Lemma 7.** There exists a model $\mathcal{M}$ and an SPDL formula $\Xi_{\text{SPDL}}$ such that no equivalent CSL formula $\Xi_{\text{CSL}}$ can be found.
5.2 Comparison of Expressive Power

Let $M$ be given as shown in figure 5.2. We assume that in $s_0$ and $s_1$ the same atomic propositions hold. Let $\Xi_{SPDL}$ be defined as follows:

$$\Xi_{SPDL} := \mathcal{P}_{>0}(true \, | \,(true?; a); (true?; b)]^{\leq \infty} true)$$

In $M$ only state $s_0$ satisfies $\Xi_{SPDL}$, as the only satisfying path is $s_0 \xrightarrow{a} s_1 \xrightarrow{b} s_0$. With respect to Markov-AP-bisimulation states $s_0$ and $s_1$ are equivalent, as:

- the cumulative rates of both $s_0$ and $s_1$ are equal
- the sets of atomic propositions that hold in $s_0$ and $s_1$ are identical
- the action labels do not play any role for Markov-AP-Bisimulation

It is known that Markov-AP-bisimilar states satisfy the same set of CSL-formulae (cf. theorem 4). From the SPDL point of view both states of $M$ are not bisimilar.

For the sake of contradiction let us assume that a formula $\Xi_{CSL}$ exists, which is an equivalent transformation of $\Xi_{SPDL}$. For $\Xi_{CSL}$ to be equivalent to $\Xi_{SPDL}$ it must hold:

1. $M, s_0 \models_{CSL} \Xi_{CSL}$ and $M, s_0 \models_{SPDL} \Xi_{SPDL}$
2. $\neg (M, s_1 \models_{CSL} \Xi_{CSL})$ and $\neg (M, s_1 \models_{SPDL} \Xi_{SPDL})$

But, as $s_0$ and $s_1$ are Markov-AP-bisimilar it holds that $M, s_1 \models_{CSL} \Xi_{CSL}$, thus the condition in line (2) is violated in $M$. This contradicts the assumption that $\Xi_{CSL}$ is equivalent to $\Xi_{SPDL}$.

5.2.2 aCSL vs. SPDL

In this section we prove that the logic SPDL is strictly more expressive than aCSL. The proofs proceeds along the same lines as the one in section 5.2.1.

**Lemma 8.** Every aCSL formula can be transformed into an equivalent SPDL formula using one of the following transformation rules:

1. $true_{aCSL} \leadsto q_{SPDL} \lor \neg q_{SPDL}$
2. $(\neg \Phi)_{aCSL} \leadsto \neg \Phi_{SPDL}$
3. $(\Phi \lor \Psi)_{aCSL} \leadsto \Phi_{SPDL} \lor \Psi_{SPDL}$
4. $(S_{\neg \neg}(\Phi))_{aCSL} \leadsto S_{\neg \neg}(\Phi_{SPDL})$
5. $(P_{\neg \neg}(\Phi_A U^{[0,t]} \Psi))_{aCSL} \leadsto P_{\neg \neg}(\Phi_{SPDL}[\Lambda^*]^{[0,t]} \Psi_{SPDL})$
6. $(P_{\neg \neg}(\Phi_A U^{[0,t]} \Psi))_{aCSL} \leadsto P_{\neg \neg}(\Phi_{SPDL}[\Lambda^*; B]^{[0,t]} \Psi_{SPDL})$

Again, we prove lemma 8 with the help of structural induction on the length $l$ of the formulae.

**Proof (Lemma 8).**

The cases 1 to 4 are similar to cases 1 and 3 to 5 of the proof of lemma 6.
5. To prove the correctness of the fifth transformation rule we apply the semantics of the aCSL path operator [73] adapted for the specific case at hand and use I.H..

\[ M, \sigma \models (\Phi_A U^{[0,t]} W_{aCSL}) \iff \exists k \geq 0((M, \sigma[k] \models W_{aCSL}^k) \land \sigma[i] \xrightarrow{A} \sigma[i+1]) \land \sum_{i=0}^{k-1} t_i \leq t) \]

*I.H.* \( \exists k \geq 0(M, \sigma[k] \models W_{SPDL} \land \forall i < k(M, \sigma[i] \models W_{SPDL}) \land \sum_{i=0}^{k-1} t_i \leq t) \)

Now, we apply the semantics of the SPDL path operator to this concrete case:

\[ M, \sigma \models (\Phi_{SPDL}[A^*] U^{[0,t]} W_{SPDL}) \iff \exists k \geq 0(M, \sigma[k] \models W_{SPDL} \land \forall i < k(M, \sigma[i] \models W_{SPDL}) \land \sum_{i=0}^{k-1} t_i \leq t \land \forall i < k(W^k(\sigma)[i] = Act(p[i]) \land M, \sigma[i] \models TeF(p[i]))) \]

In this specific case each test formula is equivalent to true. This state formula is satisfied by every state, thus the equation above can be reduced to

\[ M, \sigma \models (\Phi_{SPDL}[A^*] U^{[0,t]} W_{SPDL}) \iff \exists k \geq 0(M, \sigma[k] \models W_{SPDL} \land \forall i < k(M, \sigma[i] \models W_{SPDL}) \land \sum_{i=0}^{k-1} t_i \leq t \land \forall i < k(W^k(\sigma)[i] = Act(p[i]))) \]

Using I.H. we have:

\[ M, \sigma \models (\Phi_{SPDL}[A^*] U^{[0,t]} W_{aCSL}) \iff \exists k \geq 0(M, \sigma[k] \models W_{aCSL} \land \forall i < k(M, \sigma[i] \models W_{aCSL}) \land \sum_{i=0}^{k-1} t_i \leq t \land \exists p \in L_{A^*}(\|p\| = k \land \forall i < k(W^k(\sigma)[i] = Act(p[i]))) \]

It remains to prove:

\[ \forall i < k(\sigma[i] \xrightarrow{A} \sigma[i+1]) \iff \exists p \in L_{A^*}(\|p\| = k \land \forall i < k(W^k(\sigma)[i] = Act(p[i]))) \]

We have:

\[ \sigma[i] \xrightarrow{A} \sigma[i+1] \iff \exists a \in A(\sigma[i] \xrightarrow{a} \sigma[i+1]) \]

Following the semantics of the Kleene star and the definition of \( A^* \), \( L_{A^*} \) consists of words of arbitrary but finite length composed by elements from \( A \). Thus, \( L_{A^*} \) contains all possible words of length \( k \). Therefore, \( \forall i < k(\sigma[i] \xrightarrow{A} \sigma[i+1]), \) iff there is a \( p \in L_{A^*} \), such that \( \forall i < k(W^k(\sigma)[i] = Act(p[i])) \).
5.2 Comparison of Expressive Power

To prove this case we apply the semantics of the respective path operators and the induction hypothesis. The formal semantics of the aCSL path operator is defined as follows:

$$M, \sigma \models (\Phi A U B)[0, t]_{aCSL} \iff$$

$$\exists k > 0 (M, \sigma[k] \models \Psi_{aCSL} \land \forall i < k (M, \sigma[i] \models \Phi_{aCSL} \land \sum_{i=0}^{k-1} t_i \leq t \land$$

$$\forall 0 \leq i < k - 1 (\sigma[i] \xrightarrow{A} \sigma[i+1] \land \sigma[k-1] \xrightarrow{B} \sigma[k])))$$

The semantics of the SPDL-path operator for applied to this specific case is defined as follows:

$$M, \sigma \models \Phi_{SPDL}[A^*; B][0, t]_{SPDL} \iff$$

$$\exists k > 0 (\forall i < k (M, \sigma[i] \models \Phi_{SPDL}) \land M, \sigma[k] \models \Psi_{SPDL}$$

$$\land \sum_{i=0}^{k-1} t_i \leq t \land \exists p \in L_{A^*}(|p| = k \land$$

$$\forall i < k (W^k(\sigma)[i] = Act(p[i]) \land M, \sigma[i] \models TeF(p[i])))$$

Following I.H. we obtain:

$$M, \sigma \models \Phi_{aCSL}[A^*; B][0, t]_{aCSL} \iff$$

$$\exists k > 0 (M, \sigma[k] \models \Psi_{aCSL} \land \sum_{i=0}^{k-1} t_i \leq t \land$$

$$\exists p \in L_{A^*}(|p| = k \land \forall i < k (M, \sigma[i] \models \Phi_{aCSL}) \land$$

$$\forall i < k (W^k(\sigma)[i] = Act(p[i]) \land M, \sigma[i] \models TeF(p[i])))$$

Using a similar argument as in case 5, we can conclude that iff

$$\forall 0 \leq i < k - 1 (\exists a \in A(\sigma[i] \xrightarrow{a} \sigma[i+1]) \land \exists b \in B(\sigma[k-1] \xrightarrow{b} \sigma[k]))$$

it also holds that

$$\forall i < k (W^k(\sigma)[i] = Act(p[i])$$

This concludes the proof of lemma 8.

\[ \Box \]

With the proof of the following lemma we prove the the asserted proposition on the expressive power of aCSL in comparison with SPDL is true.

**Lemma 9.** There is an SPDL formula that cannot be transformed into an equivalent aCSL formula.

**Proof.** Given the model \( M \) in figure 5.3, then

$$\Xi_{SPDL} := P_{\geq 0}(true \mid (p?; a); (q?; a))[0, \infty] \land true$$

is an SPDL formula that cannot be transformed into an equivalent aCSL formula.

Bisimulation equivalence for action-labelled Markov chains (AMC) is defined as follows:
Comparing the Expressive Power of Stochastic Temporal Logics

Two states \( s_0 \) and \( s_1 \) of an AMC \( \mathcal{M}' \) are bisimilar, if the following condition holds:

\[
R_a(s, C) = R_a(s', C)
\]

for all equivalence classes \( C \) and actions \( a \). For \( \mathcal{M} \) we easily see (cf. figure 5.3), that \( s_0 \) and \( s_1 \) are bisimilar with respect to the above definition, i.e., both states satisfy the same set of aCSL formulae [108].

Now, for the sake of contradiction we assume that there is an aCSL formula \( \Xi \) that is equivalent to \( \Xi_{SPDL} \). \( \Xi_{SPDL} \) is in \( \mathcal{M} \) only satisfied in \( s_0 \). Together with the assumption that \( \Xi_{aCSL} \) is equivalent to \( \Xi_{SPDL} \) this yields:

\[
\mathcal{M}, s_0 \models_{aCSL} \Xi_{aCSL} \text{ and } \mathcal{M}, s_1 \not\models_{aCSL} \Xi_{aCSL}
\]

But this contradicts the fact that \( s_0 \) and \( s_1 \) are bisimilar. Therefore, \( \Xi_{aCSL} \) is not equivalent to \( \Xi_{SPDL} \). \( \blacksquare \)

5.2.3 aCSL+ vs. SPDL

In this section we will prove that the logic SPDL is strictly more expressive than aCSL+.

Again, the proof proceeds along the same lines as for the logics CSL\( ^t \) and aCSL.

**Lemma 10.** Every aCSL+ formula can be transformed into an equivalent SPDL formula by applying one of the following transformation rules.

1. \( q_{aCSL+} \rightarrow q_{SPDL} \)
2. \( (\neg \Phi)_{aCSL+} \rightarrow \neg \Phi_{SPDL} \)
3. \( (\Phi \lor \Psi)_{aCSL+} \rightarrow \Phi_{SPDL} \lor \Psi_{SPDL} \)
4. \( (S_{\text{exp}}(\Phi)_{aCSL+}) \rightarrow S_{\text{exp}}(\Phi_{SPDL}) \)
5. \( (P_{\text{exp}}(\Phi[p][0, t])_{aCSL+}) \rightarrow P_{\text{exp}}(\Phi_{SPDL}[p][0, t])_{SPDL} \)

We will do the proof by using structural induction on the length \( l \) of the formula.

**Proof (Lemma 10).**

The cases 1 to 4 are similar to cases 2 to 5 of the proof of lemma 6. Using the semantics of the path operators for aCSL+ resp. SPDL and the induction hypothesis, we obtain the following chain of equivalences:

\[
\mathcal{M}, \sigma \models (\Phi[p][0, t])_{aCSL+} \iff \exists k \geq 0(\mathcal{M}, \sigma[k] \models \Psi_{aCSL+} \land \\
\sum_{i=0}^{k-1} i \leq t \land \forall k \in \mathcal{L}_{\sigma} \land \forall i < k(\mathcal{M}, \sigma[i] \models \Phi_{aCSL+}))
\]

Using I.H. we obtain:
5.2 Comparison of Expressive Power

\[ \mathcal{M}, \sigma \models (\Phi[0,\infty]\Psi)_{\text{aCSL+}} \iff \exists k \geq 0 (\mathcal{M}, \sigma[k] \models \Psi_{\text{SPDL}} \land \\
\sum_{i=0}^{k-1} t_i \leq t \land W^k(\sigma) \in \mathcal{L}_\rho \land \forall i < k (\mathcal{M}, \sigma[i] \models \Phi_{\text{SPDL}})) \]

We repeat the formal semantics of the SPDL path operator:

\[ \mathcal{M}, \sigma \models \Phi_{\text{SPDL}}[0,\infty]\Psi_{\text{SPDL}} \iff \exists k \geq 0 (\forall i < k (\mathcal{M}, \sigma[i] \models \Phi_{\text{SPDL}}) \land \mathcal{M}, \sigma[k] \models \Psi_{\text{SPDL}} \land \\
\sum_{i=0}^{k-1} t_i \leq t \land \exists p \in \mathcal{L}_\rho (|p| = k \land \\
W^k(\sigma)[i] = \text{Act}(p[i]) \land \mathcal{M}, \sigma[i] \models \text{TeF}(p[i]))) \]

Applied to the specific needs of the transformation rule under consideration this yields the following simplifications, \( \mathcal{M}, \sigma[i] \models \text{TeF}(p[i]) \) holds trivially, as \( \text{TeF}(p[i]) = \text{true} \) for all states \( \sigma[i] \) on the path. We obtain:

\[ \mathcal{M}, \sigma \models \Phi_{\text{SPDL}}[0,\infty]\Psi_{\text{SPDL}} \iff \exists k \geq 0 (\forall i < k (\mathcal{M}, \sigma[i] \models \Phi_{\text{SPDL}}) \land \mathcal{M}, \sigma[k] \models \Psi_{\text{SPDL}} \land \\
\sum_{i=0}^{k-1} t_i \leq t \land \exists p \in \mathcal{L}_\rho (|p| = k \land \\
W^k(\sigma)[i] = \text{Act}(p[i]))) \]

To establish the equivalence we have to show that:

\[ W^k(\sigma) \in \mathcal{L}_\rho \iff \exists p \in \mathcal{L}_\rho (|p| = k \land \forall i < k (W^k(\sigma)[i] = \text{Act}(p[i]))) \]

As \( \rho, \mathcal{M} \) and \( \sigma \) are the same for both logics it is clear that this equivalence must hold. This concludes our proof.

\[ \square \]

**Lemma 11.** There is at least one SPDL formula that cannot be transformed into an equivalent aCSL+ formula.

**Proof (Lemma 11).** We will show that probabilistic SPDL formulae \( \mathcal{P}_{\text{osp}}(\phi_{\text{SPDL}}) \) can not be equivalently transformed to aCSL+ formulae \( \mathcal{P}_{\text{osp}}(\phi_{\text{aCSL+}}) \). Let \( \mathcal{M} \) (cf. figure 5.4) and formula

\[ \Xi_{\text{SPDL}} = \mathcal{P}_{\text{osp}}(\text{true } [(q?; a)^*; (\neg q?; b)]^{[0,\infty]} \text{ true}) \]

![Fig. 5.4. Model of formula \( \Xi_{\text{SPDL}} \)](image)
be given, where
\[
\phi_{\text{SPDL}} = \text{true} \quad [(q; a)^*; (\neg q; b)]^{[0, \infty]} \quad \text{true}
\]

From fig. 5.4 it is obvious that, given \(s_0\) is the initial state of \(M\), only path \(\sigma = s_0 \xrightarrow{a} s_1 \xrightarrow{b} s_2 \xrightarrow{a} s_3\)

satisfies \(\phi_{\text{SPDL}}\), whereas the second path emanating from \(s_0\), \(\sigma' = s_0 \xrightarrow{b} s_4\) is not satisfying. We now have to find \(\phi_{\text{aCSL}+} := \Phi_{\text{aCSL}+}[a; a; b; b \cup b]^{[0, \infty]} \Psi_{\text{aCSL}+}\) that distinguishes \(\sigma\) from \(\sigma'\).

The ability of SPDL to distinguish between \(\sigma\) and \(\sigma'\) stems from the fact that the satisfiability of a path can be characterised by test formulae that must be satisfied on arbitrary single states on a path. aCSL+ lacks this feature. Thus, aCSL+ provides no means to distinguish \(\sigma\) from \(\sigma'\).

Following translations from \(\phi_{\text{SPDL}}\) to \(\phi_{\text{aCSL}+}\) are possible:

- \(\phi_{\text{aCSL}+} := \Phi_{\text{aCSL}+}[a; b]^{[0, \infty]} \Psi_{\text{aCSL}+}\): This simple transformation does not distinguish \(\sigma\) from \(\sigma'\).
- \(\phi_{\text{aCSL}+} := \Phi_{\text{aCSL}+}[a; a; b]^{[0, \infty]} \Psi_{\text{aCSL}+}\): This transformation distinguishes \(\sigma\) from \(\sigma'\), but it is also not equivalent to \(\phi_{\text{SPDL}}\), as \(\sigma^* = s_1 \xrightarrow{a} s_2 \xrightarrow{b} s_3\) satisfies \(\phi_{\text{SPDL}}\), whereas \(\sigma^*\) does not satisfy \(\phi_{\text{aCSL}+}\). Generally, this transformation can not be equivalent to \(\phi_{\text{SPDL}}\), as it is state dependent, where definition 94 requires a state independent transformation.
- \(\phi_{\text{aCSL}+} := \Phi_{\text{aCSL}+}[a; b \cup b]^{[0, \infty]} \Psi_{\text{aCSL}+}\) is state independent, but is also not a feasible transformation, as both \(\sigma\) and \(\sigma'\) satisfy \(\phi_{\text{aCSL}+}\).

\(\Box\)

5.2.4 aCSL vs. CSL

In this section we will briefly show that the logics aCSL and CSL are incomparable. I.e. in both logics there is at least one formula for which given a concrete model no equivalent translation into the other logic can be found.

**Lemma 12.** Not for every CSL\(^1\) formula an equivalent aCSL formula exists.

**Proof (Lemma 12).** Let for \(\mathcal{M}\) (cf. figure 5.5) the following formula \(\Xi_{\text{CSL}}\) be given:

\[
\Xi_{\text{CSL}} = \mathcal{P}_{>0}((p \land \neg q) \cup [0, \infty]q)
\]

![Fig. 5.5. Model \(\mathcal{M}\) of the formula \(\Xi_{\text{CSL}}\)](image)
Then, there is no equivalent formula $\Xi_{aCSL}$. With respect to AMC, the semantic model for aCSL, states $s_0$ and $s_1$ are bisimilar, whereas with respect to the equivalence relation for state labelled Markov chains, the semantic model for CSL, they are not. For the sake of contradiction we assume that there is an aCSL formula $\Xi_{aCSL}$ that is equivalent to $\Xi_{CSL}$. Thus, it holds that

$$M, s_0 \models_{CSL} \Xi_{CSL}$$

and

$$M, s_0 \models_{aCSL} \Xi_{aCSL}$$

according to the assumption that $\Xi_{aCSL}$ is equivalent to $\Xi_{CSL}$. As $s_0$ and $s_1$ are bisimilar it holds that $M, s_1 \models_{aCSL} \Xi_{aCSL}$. Furthermore $M, s_1 \not\models_{CSL} \Xi_{CSL}$ and thus $M, s_1 \not\models_{aCSL} \Xi_{aCSL}$. But then:

$$M, s_1 \models_{aCSL} \Xi_{aCSL}$$

$$M, s_1 \not\models_{aCSL} \Xi_{aCSL}$$

This contradicts the fact that $s_0$ and $s_1$ are bisimilar for CSL and thus initial assumption that $\Xi_{CSL}$ is equivalent to $\Xi_{aCSL}$ is wrong. $\square$

Now, we will show that there are aCSL formulae for which no equivalent CSL formulae can be found.

**Lemma 13.** Not for every aCSL formula an equivalent CSL formula exists.

**Proof (Lemma 13).** Let a model $M$ (cf. figure 5.6) be given. Let $\Xi_{aCSL} := P>0(true \cup [0,\infty)true)$.

![Fig. 5.6. Model $M$ of the formula $\Xi_{aCSL}$](image)

For CSL it holds, that $s_0$ and $s_1$ are bisimilar, whereas for aCSL both states are not. For the sake of contradiction we assume that there is a CSL formula $\Xi_{CSL}$ that is equivalent to $\Xi_{aCSL}$. It holds that

$$M, s_0 \models_{aCSL} \Xi_{aCSL}$$

and

$$M, s_0 \models_{CSL} \Xi_{CSL}$$

according to the assumption that $\Xi_{CSL}$ is equivalent to $\Xi_{aCSL}$. As $s_0$ and $s_1$ are bisimilar for CSL, it holds that $M, s_1 \models_{CSL} \Xi_{CSL}$, furthermore $M, s_1 \not\models_{aCSL} \Xi_{aCSL}$ and thus $M, s_1 \not\models_{CSL} \Xi_{CSL}$. But then:

$$M, s_1 \models_{CSL} \Xi_{CSL}$$

$$M, s_1 \not\models_{CSL} \Xi_{CSL}$$

This contradicts the fact that $s_0$ and $s_1$ are bisimilar for CSL and thus the initial assumption that $\Xi_{CSL}$ is equivalent to $\Xi_{aCSL}$ is wrong. $\square$
5.3 Extensions of Temporal Stochastic Logics

In this section we describe some extensions of the logics considered so far:

- We will extend aCSL by atomic propositions \(aCSL_{AP}\) and show that in this case CSL \(\subset aCSL_{AP}\).
- We extend SPDL by real time intervals \((SPDL^I)\), i.e. time intervals \([t, t']\), with \(t \neq t'\), to overcome the problem that CSL and SPDL are incomparable.

The hierarchy we would like to obtain by extending the logics aCSL and SPDL as described can be found in figure 5.7.

![Fig. 5.7. Hierarchy of expressive power of temporal stochastic logics](image)

5.3.1 Extension of aCSL with Atomic Propositions

If aCSL is extended with atomic propositions, we obtain the following definition of its grammar:

**Definition 97 (Syntax of \(aCSL_{AP}\)).** Let \(p \in [0, 1]\), \(q \in AP\) and \(\preceq \in \{<, >, \leq, \geq\}\), then the state formulae of \(aCSL_{AP}\) are defined as follows:

\[
\Phi := true \mid q \mid \neg \Phi \mid \Phi \lor \Phi \mid S_{\preceq p}(\Phi) \mid P_{\preceq p}(\Phi)
\]

The path formulae \(\phi\) are defined the following way:

\[
\phi := \Phi_A U_{[0, t]} \Phi \mid \Phi_A U_{[0, t]} \Phi
\]

Where it holds \(t \in \mathbb{R}_{>0} \cup \{\infty\}\).

We obtain the following rules to transform CSL formulae to \(aCSL_{AP}\) formulae:

**Lemma 14.** Any arbitrary CSL\(^I\) formula can be transformed into an equivalent \(aCSL_{AP}\) formula by applying one of the subsequently described rules.

1. \(true_{CSL} \sim true_{aCSL_{AP}}\)
2. \(q_{CSL} \sim q_{aCSL_{AP}}\)
3. \(\neg \Phi_{CSL} \sim \neg \Phi_{aCSL_{AP}}\)
4. $(\Phi \lor \Psi)_{\text{CSL}} \leadsto \Phi_{aCSL_{\text{AP}}} \lor \Psi_{aCSL_{\text{AP}}}$

5. $(S_{\text{exp}}(\Phi))_{\text{CSL}} \leadsto S_{\text{exp}}(\Phi_{aCSL_{\text{AP}}})$

6. $(P_{\text{exp}}(X[0, t] \Phi))_{\text{CSL}} \leadsto P_{\text{exp}}(true_{aCSL_{\text{AP}}} \circ U[0, t] \Phi_{aCSL_{\text{AP}}})$

7. $(P_{\text{exp}}(\Phi_{aCSL_{\text{AP}}}[0, t])_{\text{CSL}} \leadsto P_{\text{exp}}(\Phi_{aCSL_{\text{AP}}} \cup [0, t] \Psi_{aCSL_{\text{AP}}})$

The proof is a structural induction on the length $l$ of the formulae.

**Proof (Lemma 14).**

The cases 1 to 5 are similar to cases 1 to 5 of the proof of lemma 6.

6. To demonstrate the correctness of this transformation rule we apply LH. and the semantics of CSL $X$ operator and of the $aCSL_{\text{AP}}$ path operator:

$$\mathcal{M}, \sigma \models (X[0, t] \Phi_{\text{CSL}})_{\text{CSL}} \iff \mathcal{M}, \sigma[1] \models \Phi_{\text{CSL}} \land \sum_{i=0}^{0} t_i \leq t.$$  

In words, this means, in $\mathcal{M}$ exists a path $\sigma$, with $\sigma_{s_0} \xrightarrow{a_{s_0}} s_1, s_0 = \sigma[0]$ and $\sigma[1] = s_1, a \in A_1, \sum_{i=0}^{0} t_i \leq t$ and $\mathcal{M}, \sigma_1 \models \Phi_{\text{CSL}}$. The semantics of the $aCSL$ path operator applied to this specific case looks as follows:

$$\mathcal{M}, \sigma \models (true_{a} U[0, t] \Psi)_{aCSL_{\text{AP}}} \iff (\mathcal{M}, \sigma[1] \models \Psi_{aCSL_{\text{AP}}} \land \sigma[0] \models true)$$

$$\land \sum_{i=0}^{0} t_i \leq t \land \sigma[0] \xrightarrow{A} \sigma[1]).$$

As for all states, thus, especially for $\sigma[0]$ we have: $\mathcal{M}, s \models true$ the subsequent simplification is possible:

$$\mathcal{M}, \sigma \models (true_{a} U[0, t] \Psi)_{aCSL_{\text{AP}}} \iff (\mathcal{M}, \sigma[1] \models \Psi_{aCSL_{\text{AP}}}$$

$$\land \sum_{i=0}^{0} t_i \leq t \land \sigma[0] \xrightarrow{A} \sigma[1]).$$

It remains to prove that $\sigma[0] \xrightarrow{A} \sigma[1]) \equiv true$, but this follows from the fact that every possible transition labelling in $\mathcal{M}$ from the set $A$: makes $(\sigma[0] \xrightarrow{A} \sigma[1])$ possible:

$$\mathcal{M}, \sigma \models (true_{a} U[0, t] \Psi)_{aCSL_{\text{AP}}} \iff (\mathcal{M}, \sigma[1] \models \Psi_{aCSL_{\text{AP}}} \land \sum_{i=0}^{0} t_i \leq t.$$  

7. We prove the final case by applying the semantics of the $U$ operator and of the $aCSL_{\text{AP}}$ path operator and usage of LH. The CSL $U$-Until operator has the following semantics:

$$\mathcal{M}, \sigma \models \Phi_{\text{CSL}} U[0, t] \Psi_{\text{CSL}} \iff$$

$$\exists k \geq 0 (\mathcal{M}, \sigma[k] \models \Psi_{\text{CSL}} \land \sum_{i=0}^{k-1} t_i \leq t \land \forall 0 \leq i < k (\mathcal{M}, \sigma[i] \models \Phi_{\text{CSL}})).$$

The $aCSL_{\text{AP}}$ path operator has the following semantics:
All we now have to do is to show: \( \forall i < k (\sigma[i] \xrightarrow{A} \sigma[i+1]) \equiv true \). By virtue of the fact that every possible transition labelling in \( M \) is an element of \( A \), and any action from \( A \) makes \( (\sigma[i] \xrightarrow{A} \sigma[i+1]) \) possible, it holds that, \( \forall i < k (\sigma[i] \xrightarrow{A} \sigma[i+1]) \equiv true \) Thus, we obtain:

\[
M, \sigma \models (\Phi_A \cup [0,t] \Psi)_{aCSL} \iff \\
\exists k \geq 0 (M, \sigma[0] = \Psi_{aCSL} \land (\forall i < k (M, \sigma[i] = \Phi_{aCSL})) \\
\land \sigma[i] \xrightarrow{A} \sigma[i+1] \land \sum_{i=0}^{k-1} t_i \leq t)
\]

\[
\iff \exists k \geq 0 (M, \sigma[0] = \Psi_{CSL} \land (\forall i < k (M, \sigma[i] = \Phi_{CSL})) \\
\land \sigma[i] \xrightarrow{A} \sigma[i+1] \land \sum_{i=0}^{k-1} t_i \leq t)
\]

\( \Box \)

This concludes the proof of the fact that every CSL formula can be transformed into an aCSL\(_{AP}\) formula. From the proof of lemma 13 we know an example of an aCSL formula that cannot be transformed into a CSL formula. This example remains valid for aCSL\(_{AP}\), even though we have to take state labellings into account for the definition of bisimulation. These results establish the desired relation: CSL \( \subset \) aCSL\(_{AP}\)

### 5.3.2 Comparison of SPDL with Time Intervals \([t, t']\) and CSL

In this section we want to demonstrate that in CSL, with time intervals \([t, t']\), \( t \neq 0 \) possible, exist formulae that cannot be transformed into equivalent SPDL formulae given a specific model \( M \). We show that SPDL with time intervals \([t, t']\), which we will refer to with SPDL\(_I\), is strictly more expressive than CSL, i.e. CSL \( \subset \) SPDL\(_I\).

**CSL \( \not\subset \) SPDL**

To verify this claim it is sufficient to give an example of a formula that is satisfiable in CSL given a specific model but which is not satisfiable in SPDL with respect to the same model.

Let the following CSL formula be given:

\[
\Xi_{CSL} = \Phi_{CSL} \cup [t,t'] \Psi_{CSL}
\]

The model \( M \) shown in figure 5.8 possibly satisfies formula \( \Xi_{CSL} \), but in SPDL\(_I\), due to the restriction to intervals that start from 0 as lower bound, it is impossible to find a formula equivalent to \( \Xi_{CSL} \).
5.3 Extensions of Temporal Stochastic Logics

Extending SPDL: SPDL¹

For the comparison of the expressive powers of CSL and SPDL¹ it is easier to give an alternative semantics to CSL path formulae that is closer to the SPDL semantics.

Definition 98 (Alternative Semantics of CSL Until Path Formulae). A path \( \sigma \) satisfies \( \Phi \) iff the following holds:

\[
\begin{align*}
M, \sigma \models & \Phi_{\text{CSL}} \implies \\
\exists k & (M, \sigma[k] \models \Psi_{\text{CSL}}) \wedge \\
\forall i & < k ((M, \sigma[i] \models \Phi_{\text{CSL}}) \wedge \\
(1) & (t = 0 \wedge \sum_{i=0}^{k-1} t_i \leq t) \vee \\
(2) & (t \neq 0 \wedge ((t \leq \sum_{i=0}^{k-1} t_i) \vee \\
(3) & (t \leq 0 \wedge (t \leq \sum_{i=0}^{k-1} t_i \leq t') \vee \\
(4) & (t = 0 \wedge (t \leq \sum_{i=0}^{k-1} t_i) \vee \\
(5) & (t \leq 0 \wedge (t \leq \sum_{i=0}^{k-1} t_i + \tau(\sigma, k) \geq t) \wedge M, \sigma[k] \models \Phi_{\text{CSL}}))
\end{align*}
\]

For an explanation of lines (3) to (5) see definition 79.

Using similar arguments as in the case of definition 96 we can show that this semantics is equivalent to the case of definition 51.

Having these preparatory definitions we are able to specify the new transformation rules from CSL to SPDL¹.

Lemma 15. An arbitrary CSL formula can be transformed into an equivalent SPDL¹ formula by application of one of the following transformation rules:

1. \( \text{true}_{\text{CSL}} \sim q_{\text{SPDL}¹} \lor \neg q_{\text{SPDL}¹} \)
2. \( q_{\text{CSL}} \sim q_{\text{SPDL}¹} \)
3. \( (\neg \Phi)_{\text{CSL}} \sim \neg \Phi_{\text{SPDL}¹} \)
4. \( (\Phi \lor \Psi)_{\text{CSL}} \sim \Phi_{\text{SPDL}¹} \lor \Psi_{\text{SPDL}¹} \)
5. \( (S_{\text{act}}(\Phi))_{\text{CSL}} \sim S_{\text{act}}(\Phi_{\text{SPDL}¹}) \)
6. \( (P_{\text{act}}(X^I \Phi))_{\text{CSL}} \sim P_{\text{act}}(\text{true}_{\text{SPDL}¹}[\text{Act}]^{I}[\Phi_{\text{SPDL}¹}]) \)
7. \( (P_{\text{act}}(\Phi U^I \Psi))_{\text{CSL}} \sim P_{\text{act}}(\Phi_{\text{SPDL}¹}[\text{Act}^*]^{I}[\Psi_{\text{SPDL}¹}]) \)

Proof (Lemma 15). The proof is again a structural induction on the length \( l \) of the involved formulae. It is very similar to the case of lemma 6 and will be omitted. \( \square \)

From this follows that the example showing a CSL formula not transformable to an SPDL formula for SPDL¹ is not longer valid. So, the hierarchy concerning the relative expressive power of the logics compared we wanted to prove could be established.
Part III

Implementation and Case Studies
CASPA: A Symbolic Performance Evaluation and Stochastic Model Checking Tool

In this chapter we will describe the architecture of our symbolic model checking and performance evaluation tool CASPA [96, 97, 95]. CASPA implements the denotational MTBDD-semantics described in chapter 3 and the model checking algorithms as described in chapter 4. This tool enables the user both to do traditional performance analysis and stochastic model checking. Performance analysis is done by defining state measures, throughput measures and mean values. This is done in the style of tools like TIPPtool [70]. On the other hand CASPA allows to precisely and concisely define and check complex performance and reliability requirements by applying our logic SPDL.

Most notably, the tool CASPA is based on symbolic data structures:

- on MTBDDs [13, 59] for state space representation
- for efficient numerical analysis, hybrid-offset labelled MTBDDs [99, 121] are employed

This contrasts CASPA to the ETMCC model checker, which is based on sparse data structures. In contrast to PRISM, which is also a symbolic stochastic model checker, we apply a stochastic process algebra as system specification language and SPDL to define logic-based requirements. In PRISM the logics PCTL [65] and CSL are used for requirement specification.

6.1 Tool Architecture

The tool architecture consists of three major parts, as shown in figure 6.1.

- Tool driver: From the command line the specification file is passed to the tool driver. It parses the system, the performance measure specification, and the SPDL property specification, translates them into their parse graphs and passes the results to the state space manager. The functionality of the three subparsers is as follows:
  - Parser for system specification
  - Parser for performance measure definition
  - Parser for SPDL formulae

- Symbolic engine: The symbolic engine can be divided into three parts:
  - State Space manager: The state space manager generates (from the parse graphs of the system and requirement specification) the MTBDD representation of the SLTS, resp. of the performance and SPDL requirements. It can perform reachability analysis, and transforms the MTBDDs, representing the SLTS into offset-labelled hybrid MTBDDs to allow for efficient numerical analysis.
Model Checking engine: The most important task of this module is to compute for a given SPDL program of a path formula the state-labelled product Markov chain (SPMC) as described in chapter 4. The generation of the SPMC is implemented completely on the symbolic level. If necessary, the SPMC is passed to the numerical engine to compute the transient state probabilities with which the currently checked SPDL path formula is satisfied.

Numerical engine: The numerical engine computes the vector of state probabilities. Several well-known numerical algorithms for both steady-state and transient analysis are implemented. The algorithms are taken from PRISM, for their detailed description see [121]. The user can set various parameters of the algorithms, such as accuracy or maximum number of iterations.

User interface: CASPA provides a textual user interface. A typical call of the tool consists of indicating the file name that contains the system and requirement specification, the analysis method, parameters for the numerical analysis and information about the verbosity level. An example call looks as follows:

caspa -v 1 -r -T -a TRANSIENT 100 ftcs.cas

The textual user interface is also used to present the results of the tool, i.e. number of states, information about the size of the symbolic representation, computation times, results of numerical analysis, etc. Additionally, CASPA can generate output that makes it possible to visualise the state space using the tool davinci [41], or for any graphical tool that can handle the .dot [5] format.

CASPA is entirely implemented using the programming language C. The lexical analyser was realised using the tool flex, the parser was written in bison. The part of CASPA that deals with symbolic data structures was implemented using the BDD-library CUDD [135] and the extensions of CUDD for numerical analysis as provided by PRISM [121].

6.2 CASPA’s Specification Language

The language for system specification of CASPA is basically YAMPA, which is enriched with some syntactic sugar. Simple performance requirements (state measures, mean value mea-
sures, throughput measures) can be specified using context-free formal languages. Logic-based requirements are specified using SPDL.

6.2.1 The System Specification Language

CASPA A’ provides operators for prefixing, choice, parallel composition, hiding. Infinite (i.e. cyclic) behaviour is specified by means of defining equations. All actions are associated with an exponential delay, which is specified by a rate parameter. The technique used for symbolic model representation works only for finite state spaces. Therefore the grammar of the input language is such that recursion over static operators (i.e. parallel composition and hiding) is not allowed, which ensures that the underlying state space is finite.

The specification language allows the specification of parameterised processes, i.e. processes which carry one or more integer parameters. This feature is very useful for describing the behaviour of queueing, counting, or generally indexed processes. Within a parameterised process, the enabling of actions may be conditioned on the current value of the process parameters. In CASPA it is possible to define both rate and parameter constants. Parameters are always integer numbers, whereas rates are real numbers. Parallel composition is only allowed at the top-level of the system specification.

In CASPA the enabling, disabling and relabelling operators from YAMPA are not available.

Example 33 (CASPA Example Specification). We now discuss a small example (see Fig. 6.2). This specification has no special meaning, its only purpose is to introduce the language elements of CASPA. In lines (2) to (4) we find the definition of specific rate values, in line

(1) /* Rate and constant definitions */
(2) rate xi = 0.5;
(3) rate gamma = 5;
(4) rate mu = 0.3;
(5) int max = 3;
(6) /* System specification */
(7) System := (P(0) |[]| P(0)) |{b}| (hide a in Q(10))
(8) Q(10) := [n > 0] -> (a,xi); Q(n-1)
(9) [n = 0] -> (b,mu); Q(10)
(10) P(n [max]) := [n = 0] -> (b,gamma); P(n) + (c,gamma); stop
(11) [n > 0] -> (d,n*mu); P(n-1)
(12) [n < max] -> (a,0.3); P(n+1)
(13) /* Requirement specification */
(14) statemeasure XXX (P{1}(n > 0) & !P{2}(n = max)) | Q(n = 4)
(15) meanvalue YYY P{2}(n)
(16) throughputmeasure ZZZ a

Fig. 6.2. Example CASPA specification

(5) a global parameter constant is defined. Lines (7) to (12) contain the system specification. Line (7) shows both possibilities to define parallel processes: The two P(0) processes are composed in parallel without interaction, i.e. all their actions are performed independently of each other. In contrast, Q(10) is composed in parallel with the two former processes in a synchronised way, i.e. action b must be performed by one of the P-processes and the Q-process at the same time. The synchronisation semantics is the same as for TIPP [70]. In line (7) we also find the hiding operator: Action a in process Q is hidden from the environment and replaced by the special silent action tau. In line (8) we see an example of guarded choice: Action a can be performed if the value of parameter n is greater than zero.
In line (10) we see a guarded choice whose test consists of \( \ast \), which means the branch can be taken regardless of the actual parameter value. In lines (8) and (10) the maximum value of the respective parameters is given: For process \( P \) we chose a global constant \( \text{max} \), for process \( Q \) the maximum value 10 is given explicitly. As for every process parameter such a maximum value has to be defined, the finiteness of the underlying state space is guaranteed. In line (10) a choice between \((b, \text{gamma}); P(n)\) and \((c, \text{gamma}); \text{stop}\) is given. As all actions have exponential delay the choice of which action is actually taken corresponds to a race condition (as for stochastic Petri nets). In line (11) we see that rates can be arithmetic expressions, which makes it possible to define rates that are dependent on actual parameter values, similar to marking dependent rates in stochastic Petri nets. Finally, in lines (14) to (16) we find examples of requirement specifications. We see that state measures can contain Boolean expressions with the usual connectives conjunction (\&), disjunction (\|) and negation (!). The clause \((P\{1\}(n > 0) \& !P\{2\}(n = \text{max}))\) characterises states in which parameter \( n \) of process \( P\{1\} \) is greater than zero, and parameter \( n \) of process \( P\{2\} \) is smaller than the maximum value, where \( P\{i\} \) expresses that we are interested in the \( i \)-th of the two \( P \) processes. A mean value will return the expected value of the specified process parameter (in line (15) it will be the mean value of parameter \( n \) of process \( P\{2\} \)), and a throughput measure will compute the throughput of the given action (in line (16) this is action \( a \)).

The requirement specification languages will be explained in detail in section 6.3

### 6.3 Requirement Specification: Definition and Data Structures

In this section we will show how requirements in CASPA can be defined and how they are associated with appropriate data structures.

#### 6.3.1 State Measures

For state measures the main task is to identify the states resp. their binary encodings that are relevant for the measure. As many states may contribute to a particular measure, we employ BDDs for a compact representation of the state sets.

Let \( S' \subseteq S \) be the set of states that contribute to a state measure, the probability is computed by \( \sum_{s \in S'} \pi_s \), i.e. by adding up the probabilities of being in a particular state \( s \in S' \).

**Measure Definition**

Here, we will describe how state measures can be defined. The general rule for defining state measures is as follows:

\[
\text{statemeasure Name statemeasuredef}
\]

where 'Name' is an arbitrary but unique name for the state measure and \( \text{statemeasuredef} \) is defined according the following grammar:
Definition 99 (State Measure Grammar).

\[ \text{statemeasuredef} := \text{statemeasuredef} \& \text{statemeasuredef} /* \text{conjunction} */ \\
| \text{statemeasuredef} | \text{statemeasuredef} /* \text{disjunction} */ \\
| ! \text{statemeasuredef} /* \text{negation} */ \\
| (\text{statemeasuredef}) \\
| \text{process}[(\text{conditions})][\{\text{instance}\}] \]

process is a process variable. If it is a parameterised process a list of conditions has to be given. To each process variable an instance number can be associated to determine to which sequential process it belongs.

Mapping the Measure Definition to BDDs

Using the definition of the state measure, a parse graph for the state measure will be generated. Since the state measure is related to one or several process names, each node of the system’s parse graph that contains a process which is referenced in the state measure will get a pointer to the respective node in the measure’s parse graph. On generation of the MTBDD for the system, the encoding of each process that contains such a pointer is written to the correspondig measure’s sub-BDD. The overall MTBDD for the state measure is obtained by applying to the sub-BDDs the logical operators that appear in the state measure.

6.3.2 Mean Values

As in the case of state measures here again we have to identify the states that contribute to the measure. Again, many state might be affected, therefore we choose BDDs as appropriate data structure to represent this set of states. Let \( i \) be the actual value of a parameter \( n \), then the corresponding mean value is computed as follows:

\[
\sum_{i} i \cdot P(n = i)
\]

where \( P(n = i) \) is a state measure, i.e. the sum of the probabilities of all states in which \( n \) has value \( i \).

Measure Definition

Mean values are only meaningful for parameterised processes. In CASPA, they can be defined as follows:

\[ \text{meanvalue Name process (parameter) [[instance]]} \]

where ‘Name’ is an arbitrary but unique name of the measure, process is the name of a parameterised process and parameter is the name of the parameter of interest. To each process variable an instance number can be associated to determine to which sequential process it belongs.
Mapping the Measure Definition to BDDs

For mean values we have to generate for each possible parameter value a BDD that encodes the states in which the parameter has exactly that value. Since processes can have several parameters (and since processes are composed in parallel with other processes), there may be many states in which the parameter of interest has the same value (whereas the values of the remaining parameters, or the states of the other processes, may change). After the generation of the system’s MTBDD representation, the measure BDDs are added up, thereby weighing each BDD with the associated parameter value. The result is an MTBDD in which every state encoding is related to its respective parameter value.

6.3.3 Throughput Measures

For throughput measures the situation is slightly different, throughput measures are not related to specific processes. Therefore no extra BDD for them needs to be generated. The throughput of an action is computed using the following formula:

$$\sum_{s \in S_a} \pi_s \cdot \lambda_{s,a}$$

where $S_a \subseteq S$, is the set of states which have an outgoing transition labelled with $a$, $\pi_s$ is the probability to be in state $s$ and $\lambda_{s,a}$ is the rate with which action $a$ is actually performed in state $s$.

Measure Definition

Throughput measures can be defined as follows:

```
throughputmeasure Name action
```

where ‘Name’ is an arbitrary but unique name of the measure, and `action` is the name of the action whose throughput shall be computed.

Mapping the Measure Definition to BDDs

The system’s MTBDD representation is restricted to the action label whose throughput is to be determined, and the target states are abstracted away. The result is then an MTBDD consisting of the states in which the relevant action is enabled, weighed with the respective transition rates.

6.3.4 SPDL Formulae

To define propositional logic and steady state formulae we can proceed as described in section 6.3.1. For probabilistic state formulae $P_{\omega p}(\phi)$, with $\phi := \Phi[\rho] \leq \Psi$ we proceed as follows: We separately define the pre- and postcondition formulae $\Phi$ resp. $\Psi$, the test formulae of $\rho$, and $\rho$. 
Definition of Logic Formulae

Generally, probabilistic path formulae can be specified as follows:

\[ \text{spdl } \text{Name } \text{precond } \text{program } \text{postcond } \text{time_bound} \]

where ‘Name’ is an arbitrary but unique string constant precond and postcond can be SPDL state formulae. time_bound is a real number that indicates the maximum time to execute the given program program.

Definition of SPDL Programs

In general, an SPDL program in CASPA is defined as follows:

\[ \text{program} := \text{programdef} \]

where programdef is defined as follows:

**Definition 100 (Grammar of SPDL Programs).**

\[
\begin{align*}
\text{programdef} := \ a | \ & \text{programdef ; programdef} | \ & \text{programdef + programdef} | \\
& \text{programdef}^* | \ & \text{name}?, \text{programdef} | \ (\text{programdef})
\end{align*}
\]

testname is the name of an existing test formula, which are SPDL state formulae.

6.4 Performance Requirements and Process Instances

In a system which is composed in parallel from several sequential processes, it is possible to associate with the respective sequential processes so called instance numbers, such that on defining state measures or mean values it is possible to determine which of the processes shall contribute to the measure. As CASPA provides the possibility for textual output of the parse graph of the system specification the user is enabled to check which instance number is associated to which process. The line

\[ \text{parameter-node Queue}(n \ 8)(4/1) \]

of the textual representation of the parse graph (cf. fig. 6.4) of the system whose specification can be found in fig. 6.3 can be interpreted as follows:

The parameterised process with name ‘Queue’ possesses a parameter \(n\), which has maximum value 8, there are four references to the node in the parse graph representing Queue and the node belongs to the sequential subprocess with instance number 1.

6.5 The State Space Manager

Generally, CASPA implements the semantics as it was described in chapter 3 and in [92]. Here, we only discuss the translation of parameterised processes, i.e. we describe our approach of how to represent parameterised processes symbolically. The parse graph structure describes the transitions depending on the parameter values, thereby also describing the possible changes of the parameter values. In CASPA the definition of the transitions is
Fig. 6.3. Specification of a system with two parallel queues

```plaintext
process P (1/1) has no measures
{ PARALLEL
  parameterised node Queue (0) (2/1)
  parameter-node Queue (n 8) (4/1) has measures:
  statemeasure: Queue {1} ((n = max)), meanvalue: Queue (n) {1},
  [(n > 0)]
  { PRAEFIX
    (serve, 3.000000);
    parameterised node Queue ((n - 1)) (2/1)
  } process guarded
  [(n < max)]
  { PRAEFIX
    (arrival, 2.000000);
    parameterised node Queue ((n + 1)) (2/1)
  } process guarded
} [ ]
parameterised node Queue (0) (2/2)
parameter-node Queue (n 8) (4/2) has measures:
meanvalue: Queue (n) {2},
[(n > 0)]
{ PRAEFIX
  (serve, 3.000000);
  parameterised node Queue ((n - 1)) (2/2)
} process guarded
[(n < max)]
{ PRAEFIX
  (arrival, 2.000000);
  parameterised node Queue ((n + 1)) (2/2)
} process guarded
}
```

Fig. 6.4. Textual parse graph
6.5 The State Space Manager 147

(1) /* Rate and constant definitions */
(2) ...
(3) */ System specification */
(4) Process := Queue(0)
(5) Queue(n [3]) := [n >= 1] -> (serve, mu); Queue(n-1)
(6) [n < 3] -> (arrival, lambda); Queue(n+1)
(7) [*] -> (fail, gamma); Repair
(8) Repair := (repair, rho); Queue(0)
(9) /* Measure specification */
(10) statemeasure Queuenotfull Queue(n < 3)
(11) meanvalue Fill Queue(n)
(12) throughputmeasure Service serve

Fig. 6.5. Example specification

separated from the change of parameters. Let \( X \) be a parameterised process, then there is in \( X \)'s parse graph exactly one node, called \( \text{PARAM} \) node, which describes the possible transitions. The condition list of a guarded choice is stored in this node. Furthermore, the parse graph may contain several nodes that store the possible changes of the parameter values, called \( \text{PARAMDEF} \) nodes. In order to generate from this information the actual MTBDD representation of a parameterised process, it is necessary that the generation algorithm keeps track of the current parameter value, which information is taken from the \( \text{PARAMDEF} \) nodes. The \( \text{PARAM} \) node serves to determine which transitions are possible in view of the current parameter values. For every satisfied condition, the successor process is determined, and the overall representation of a parameterised process is then a choice over all possible successor processes.

Example 34 (Parameterised Process- and Measure MTBDDs). We will clarify the concepts of generating an MTBDD representation for parameterised processes and relating encodings and measures by means of the example shown in Fig. 6.5. The parse graph of the specification can be found in Fig. 6.6. In the \( \text{PARAMDEF} \) node (2) the parameter value is initialised to zero. In the \( \text{PARAM} \) node (3), when it is visited for the first time, the conditions of the first and the third field are fulfilled, therefore we can generate the MTBDD for their respective successor nodes. To generate the successor node we use the information about the actual parameter value and the change of parameter values of the \( \text{PARAMDEF} \) nodes. In the initial case the successor processes are \((\text{arrival}, \lambda); \text{Queue}(1)\) and \((\text{fail}, \gamma); \text{Repair}\). For \(\text{Queue}(1)\) and \(\text{Repair}\) we then compute again the successor processes, and so on. For \(\text{Queue}(1)\) all three conditions are fulfilled and we have three successor processes, namely \((\text{arrival}, \lambda); \text{Queue}(2), (\text{serve}, \mu); \text{Queue}(0)\) and \((\text{fail}, \gamma); \text{Repair}\). For the latter two, the successor nodes are already known, whereas for \(\text{Queue}(2)\) the successor processes still have to be computed. The overall MTBDD representation is obtained as a choice between the MTBDD representations of all successor processes which were found. For the state measure \(\text{Queuenotfull}\) of Fig. 6.5 the states for \(\text{Queue}(0), \text{Queue}(1)\) and \(\text{Queue}(2)\) are relevant. Therefore, on generation of the respective MTBDDs the encodings of these states are copied to the measure BDD. For the mean value measure \(\text{Fill}\) an MTBDD is constructed where the encoding of each reachable state leads to the corresponding value of parameter \(n\). Assuming that states are encoded as shown in Fig. 6.7 (left), the resulting MTBDD for this mean value measure looks as shown in Fig. 6.7 (right). ■
6.6 Symbolic Product Markov Chain Generation for SPDL Path Formulae

In fig. 6.8 we find a simplified version of the symbolic SPMC generation algorithm for SPDL path formulae $\Phi[\rho]^{[0,t]}\Psi$. Here, $M$ and $A_\rho$ refer to the symbolic representation of the model and the deterministic program automaton.

The algorithm can be explained as follows:

- Line (3): $PMC$ is the MTBDD that contains only transitions that are labelled with actions that appear both in $M$ and $A_\rho$.
- Line (4): As the product Markov chain $PMC$ from line (3) may contain unreachable states, a symbolic reachability analysis has to be performed.
- Line (5): Transitions, labelled with actions that do not appear in the automaton do not contribute to the satisfaction of the path formula. Therefore, they are redirected to the $FAIL$-state, i.e. $FAILPMC$ contains those transitions that lead to the $FAIL$-state.
6.6 Symbolic Product Markov Chain Generation for SPDL Path Formulae

(1) \( \text{GenSPMC}(\mathcal{M}, A_\rho) \) \{
(2) \( \text{SUCCPMC} := 0 \)
(3) \( \text{PMC} := \text{MAutActions}(\mathcal{M}, A_\rho) \)
(4) \( \text{PMC}' := \text{Reach}(\text{PMC}) \)
(5) \( \text{FAILPMC} := \text{notAutActions}(\mathcal{M}, A_\rho) \)
(6) \( \text{PMC} := \text{CheckPrecond}(\text{PMC}') \)
(7) \( \text{FAILPMC} := \text{Apply}(\text{FailedPrecond}(\text{PMC}'), \text{FAILPMC}, +) \)
(8) \( \text{while} (\text{TESTS} \neq \emptyset) \) \{ /* TESTS is the set of test formulae of \( \rho \) */
(9) \( \text{currtest} := \text{Choose}(\text{TESTS}) \) /*Choose an arbitrary test formula from TESTS */
(10) \( \text{PMC} := \text{CheckTest}(\text{currtest}, \text{PMC}) \)
(11) \( \text{FAILPMC} := \text{Apply}(\text{FailedTest}(\text{PMC}), \text{FAILPMC}, +) \)
(12) \( \text{PMC}'' := \text{CheckPostcond}(\text{PMC}') \)
(13) \( \text{SUCCPMC} := \text{Apply}(\text{PMC}'', \text{SUCCPMC}, +) \)
(14) \( \text{PMC} := \text{Apply}(\text{FailedPostcond}(\text{PMC}'), \text{SUCCPMC}, +) \)
(15) \( \text{SPMC} := \text{Apply}(\text{PMC}, \text{FAILPMC}, +) \)
(16) \( \text{TESTS} := \text{TESTS} \setminus \text{currtest} \)
(17) \}
(18) \text{return} \text{SPMC} \}

Fig. 6.8. Algorithm for symbolic SPMC generation

- Line (6): Here, all transitions, emanating from states that satisfy the precondition \( \Phi \) are singled out.
- Line (7): \textit{FailedPrecond} contains all transitions from states that do not satisfy \( \Phi \). These transitions are redirected to the \textit{FAIL}-state. The new \textit{FAIL}-MTBDD is then the sum of the old one and \textit{FailedPrecond}(PMC').
- Line (10): PMC' is the MTBDD that contains all transitions from states that satisfy the current test \textit{currtest}.
- Line (11): \textit{FailedTest}(PMC) contains all transitions from states that do not satisfy the current test formula to the \textit{FAIL}-state; again, the new \textit{FAIL}-MTBDD is then the sum of the old one and \textit{FailedPrecond}(PMC').
- Line (12): Here, it is checked, which states of PMC' satisfy both \( \Psi \) in \( \mathcal{M} \) and are accepting states of \( A_\rho \). Such transitions are redirected to the \textit{SUCC}-state.
- Line (13): \textit{SUCCPMC} contains all transitions from states that lead to \textit{SUCC}.
- Lines (14) and (15): Here, the final SPMC SPMC is computed; SPMC is the sum of all transitions that neither lead to \textit{FAIL} nor to \textit{SUCC} (\textit{FailedPostcond}) and those transitions that lead either to \textit{FAIL} or to \textit{SUCC}.

All pseudo-code routines from algorithm 6.8 can be implemented on a purely symbolic level. For example \textit{MAutActions}(\( \mathcal{M}, A_\rho \)) has the implementation, as shown in fig. 6.9, which consists simply of a call of the \textit{apply} algorithm, and product (\( \ast \)) as operator.

(1) \( \text{MAutActions}(\mathcal{M}, A_\rho) \) \{
(2) \( \text{CMP} := \text{Apply}(\mathcal{M}, A_\rho, \ast) \)
(3) \( \text{return} \text{CMP} \)
(4) \}

Fig. 6.9. Algorithm for symbolic computation of transitions in \( \mathcal{M} \) and \( A_\rho \) with common transition labels
It must be noted, that algorithm 6.8 only draws a rough picture of the SPMC generation. For example, the pseudo-routine CheckTest includes MTBDD-based operations that restrict the intermediate SPMC PMC to those transitions that are relevant for the current test currtest.

6.7 Numerical Analysis

The numerical engine relies on the hybrid offset-labelled MTBDD approach of PRISM [121, 99].

6.7.1 Methods of Numerical Analysis

To compute stationary state probabilities we have to solve the linear equation system

\[ \mathbf{Q}^T \pi^T = 0 \]

we have to respect the side condition \( \sum_{s \in S} \pi_s = 1 \), where \( \pi_s \) is the steady state probability to reside in state \( s \). \( \mathbf{Q} \) is the generator matrix of the transition system, \( \pi \) is a probability vector of dimension \( |S| \).

Solving this equation system directly, e.g. using Gaussian elimination is not recommended in the symbolic context. MTBDDs representing the matrices may become very large, as much of the regularity that contributes to compact representations is lost during computation. Therefore, iterative procedures are used.

By splitting the generator matrix \( \mathbf{Q} \) appropriately, the equation system can be transformed into a fixed point equation:

\[ \pi^T = \mathbf{M} \pi^T \]

where \( \mathbf{M} \) is the iteration matrix. The iteration procedure resulting from this can be written as follows:

\[ x^{(i+1)} = \mathbf{M} x^{(i)} \]

Iterative methods have the advantage that the iteration matrix \( \mathbf{M} \), which is derived from \( \mathbf{Q} \), does not change during computation and that computation can be stopped, if a certain accuracy \( \epsilon \) of the result has been reached. CASPA supports the following methods for computing steady state probabilities:

- Jacobi, Pseudo-Gauss-Seidel, Power,
- Pseudo-SOR (successive over relaxation), JOR (Jacobi over relaxation),
- Backwards-Pseudo-Gauss-Seidel, and Backwards-Pseudo-SOR

The implementation of these methods is taken from PRISM [121]. For details we refer to the literature [137, 67, 121].

It can be noted that for the Pseudo-Gauss-Seidel and the Pseudo-SOR methods so called-backward variants exist, which starts with the last vector element \( x^{(i+1)}(n) \).
6.7.2 Hybrid Matrix-Vector Multiplication

The purely MTBDD based realisation or implementation of the iterative methods, where not only the matrices but also the probability vectors are stored as MTBDDs faces many problems. The major problem is that the probability vector changes during computation, this change leads often to a growth of the MTBDD representation, due to loss of regularity. This stems from the fact, that in the beginning either only very few state have an initial probability other than zero, or all states are equally probable. This situation changes during computation such that many different state probabilities have to be stored. This destroys the compactness of the MTBDD representation of the probability vector.

In [57, 58] it was shown that the performance of purely MTBDD based numerical analysis is rather poor in comparison to sparse matrix based approaches. Numerical algorithms based on sparse data structures have the advantage of being efficient with respect to time, but their memory requirements are higher than in the case of symbolic data structures.

In [121] a hybrid approach is proposed which combines the advantages of both the symbolic and the sparse approach. The hybrid approach stores the probability vector explicitly in an array, which means that its memory requirements are constant during the computation, the iteration matrix is represented compactly by an MTBDD, for which the memory requirements are also constant, as the matrix does not change during computation.

Matrix-vector multiplication on these different types of data structures is based on the formula

\[
\begin{align*}
\text{for all } m: x^{(i+1)}(n) & := x^{(i+1)}(n) + e(n, m) \cdot x^{(i)}(m)
\end{align*}
\]

where \( e(n, m) \) is the entry of the iteration matrix on position \((n, m)\). To work correctly, \( x^{(i+1)} \) has to be initialised with value 0 before each iteration step.

As each matrix entry is processed exactly once, one traversal of the MTBDD per iteration step suffices. This approach makes it necessary to rely on methods that use two vectors, as matrix entries cannot be read row-wise in this case.

6.7.3 Offset-labelled MTBDDs

A prerequisite for efficient hybrid matrix-vector multiplication is an efficient method to relate to the MTBDD terminal vertices, that represent the matrix entries their respective indices \((n, m)\), where \( n \) is the row index and \( m \) the column index of the entry. One simple solution of this problem is to keep track whether an \textit{hi} or an \textit{lo} branch is taken in the MTBDD and then add up the appropriate powers of 2 when descending in the MTBDD from top to bottom. This approach is not suitable, as it is not taken into account that many states may not be reachable. The iteration vector is therefore larger than necessary, which puts unacceptable limits on the size of systems that can be analysed.

To avoid this problem, each vertex in the MTBDD is marked with an offset, that indicates how many rows resp. columns are reachable from this vertex. Vertices, that represent binary variables from the encoding of a source state of a transition are marked with a row offset, if the resp. binary variable encodes a target state a column offset is generated. Thus, the probability vector only has to store the probabilities for reachable states. Then an offset is only added when leaving the vertex via the \textit{hi} edge. This approach requires the relaxation of some of the reducedness criteria for MTBDDs: Redundant vertices now will only be skipped, if its successor is the terminal vertex 0 as the offsets of internal vertices are necessary to compute the offset of higher levels. This restriction has the advantage that
the traversal of the MTBDD can be sped up, as it must not be tested whether there are skipped levels. Further, isomorphic subtrees may not be superposed, as different paths may have different offsets. In [121] it is shown that these relaxations do not lead to a significant growth of the MTBDD sizes.

### 6.7.4 Hybrid MTBDDs

The approach described so far improves the efficiency of symbolic numerical analysis, but further optimisations are possible. The basic idea is to mix MTBDDs and sparse data structures to store the iteration matrix.

Subgraphs, having a common root vertex, represent the same submatrix of the entire iteration matrix. Using this observation one can gain some speed-up in the numerical analysis, if instead of storing single matrix entries in the terminal nodes such submatrices are stored, using sparse data structures. In this case the traversal of the MTBDD can be stopped earlier and the computation of the probabilities can make use of the submatrices as a whole. In [121] some considerations concerning the distance from the MTBDD’s root node (level), at which single nodes shall be replaced by matrices, are made. Experiments showed that the (heuristically) optimal level is two thirds of the overall distance from the root to the leaves.

### 6.7.5 Pseudo Gauss-Seidel

Using the standard interleaved ordering for source and target variables, we can not directly apply efficient methods like Gauss-Seidel or SOR as here a row-wise access to the entries of the transition matrix is required (cf. fig 6.10 (a)), which cannot be realised given an interleaved variable ordering. On the other hand a non-interleaved variable ordering is also not feasible as here, the memory requirements would be prohibitive. For the Jacobi method no elementwise access is required, as shown in fig 6.10 (b). If the matrix is decomposed into submatrices a pseudo-Gauss-Seidel method can be implemented on base of MTBDDs. Here, matrix-vector multiplication is done in a block-wise fashion, such that the iteration vector can be partially updated. Only one iteration vector and an auxiliary vector of the submatrices dimension is required (cf. fig 6.10 (c)). The submatrices are obtained by transforming the MTBDDs to hybrid MTBDDs, see section 6.7.4.

![Fig. 6.10. Graphical representation of (a) Gauss-Seidel, (b) Jacobi, (c) Pseudo-Gauss-Seidel methods](image)

### 6.7.6 Integration in CASPA

The approach of hybrid offset labelled MTBDDs can be easily integrated into CASPA. The MTBDD of the rate matrix can be derived from the MTBDD encoding the SLTS by abstracting from action encodings. On this MTBDD the numerical analysis is performed (including transformation of the ‘normal’ MTBDD to hybrid offset labelled MTBDD). The result is a
vector of stationary or transient state probabilities. This vector can then be used to compute the performance requirements.
In this chapter we will demonstrate the applicability of the CASPA tool by means of a number of case studies. Most of these case studies are widespread in performance and reliability evaluation literature and can be viewed as benchmarks to assess the performance of the tool under consideration.

All results were computed on a computer with Pentium IV 3.0 GHz Processor, 1 GB RAM and running the operating system SuSe Linux 9.0.

7.1 Introduction

The following remarks apply to all the case studies done in this thesis.

7.1.1 Model Statistics and Construction Times

- Model construction times include the time for the derivation of an SLTS (represented as an MTBDD) from the process algebraic system description, and the time for the computation of the reachable portion of the state space using a BDD based algorithm [130].
- In the tables that show the time required for generating the state space of the current system the number of iterations refers to the number of iterations required to compute the reachable part of the state space.
- The tables of the model statistics give information about the CTMC generated using CASPA: The number of reachable states, number of transitions, the number of nodes of the MTBDD. Here, we give both the final number of nodes and the peak number of nodes that were allocated during generation of the final MTBDD representation. Finally, we also find the number of leaf nodes of the MTBDD.

7.1.2 Steady State Computation

- The computation of steady state probabilities is considered to be non-convergent, if the steady state probability computation algorithm does not terminate within at most 100,000 iterations. Non-convergence is indicated by n.c. at the appropriate places in the respective tables.
- If the convergence of a method for the system under investigation is significantly worse than that of another method applied to the same system, we do not apply the worse method to the larger scaled system.
• The algorithms for computing stationary state probability vectors are abbreviated as follows:
  – Pseudo-Gauss-Seidel: PGS
  – BGS: Backwards-Pseudo-Gauss-Seidel: BPGS
  – Pseudo-Successive-Over-Relaxation: PSOR
  – Backwards-Pseudo-Successive-Over-Relaxation: BPSOR
  – Jacobi-Over-Relaxation: JOR

7.1.3 Model Checking Probabilistic SPDL State Formulae

• The time required to generate from a given SPDL program its deterministic program automaton is negligible.
• The size of a deterministic program automaton is usually small compared to the sizes of the models under investigation.
• Here, the number of iterations indicates the number of times the loop in lines (8) to (17) of the product Markov chain generation algorithm from fig 6.8 had to be executed. This number is equal to the number of tests plus one that occur in the SPDL program of the path formula that is to be verified.
• For transient analysis the Fox-Glynn-algorithm was employed.

7.2 M/M/1/N-Queue

7.2.1 Introduction

As a first example we study a simple queue with Markovian arrival with rate $\lambda$ and service with rate $\mu$. The queue possesses a buffer of limited capacity $N$ and a single server. The underlying CTMC for a queue with buffer capacity $N = 3$ is shown in figure 7.1. For $N$ we can assume that always $N = 2^k - 1, k \in \mathbb{N}$, we also assume the standard state enumeration scheme, where states are enumerated from 0 to $2^k - 1$. Thus, the rate matrix is very regular having entries $\neq 0$ in the case $R(i, i + 1) = \lambda$ and $R(i + 1, i) = \mu$. This regularity allows a very compact encoding of the CTMC as MTBDD.
7.2 M/M/1/N-Queue

<table>
<thead>
<tr>
<th>N:</th>
<th>Model:</th>
<th>MTBDD:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>States:</td>
<td>Transitions:</td>
</tr>
<tr>
<td></td>
<td>Peak</td>
<td>Final</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>30</td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>62</td>
</tr>
<tr>
<td>63</td>
<td>64</td>
<td>126</td>
</tr>
<tr>
<td>127</td>
<td>128</td>
<td>254</td>
</tr>
<tr>
<td>255</td>
<td>256</td>
<td>510</td>
</tr>
<tr>
<td>511</td>
<td>512</td>
<td>1,022</td>
</tr>
<tr>
<td>1,023, 1,024</td>
<td>2,046</td>
<td>8,252</td>
</tr>
<tr>
<td>2,047, 2,048</td>
<td>4,094</td>
<td>16,451</td>
</tr>
<tr>
<td>4,095, 4,096</td>
<td>8,190</td>
<td>32,830</td>
</tr>
<tr>
<td>8,191, 8,192</td>
<td>16,382</td>
<td>65,604</td>
</tr>
</tbody>
</table>

Table 7.1. Model Statistics for M/M/1/N queue

7.2.2 Model Statistics

In table 7.1 the size of the MTBDDs for varying N can be found. We observe that on duplication of the capacity N of the queue’s buffer the MTBDD size grows by a constant factor of only 10 additional nodes needed to encode the enlarged state space. Regardless of this, we have to admit that the peak MTBDD size is very large, as we have to store intermediate results. The reason for this large intermediate MTBDDs can be found in the fact that the system consists of a single sequential component.

7.2.3 Model Construction Times

In table 7.2 we find the times to construct the models.

<table>
<thead>
<tr>
<th>N:</th>
<th>Construction:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (sec.): Iterations:</td>
</tr>
<tr>
<td>15</td>
<td>&lt; 10^{-6}</td>
</tr>
<tr>
<td>31</td>
<td>&lt; 10^{-6}</td>
</tr>
<tr>
<td>63</td>
<td>&lt; 10^{-6}</td>
</tr>
<tr>
<td>127</td>
<td>0.01</td>
</tr>
<tr>
<td>255</td>
<td>0.03</td>
</tr>
<tr>
<td>511</td>
<td>0.06</td>
</tr>
<tr>
<td>1,023</td>
<td>0.14</td>
</tr>
<tr>
<td>2,047</td>
<td>0.31</td>
</tr>
<tr>
<td>4,095</td>
<td>0.69</td>
</tr>
<tr>
<td>8,191</td>
<td>1.38</td>
</tr>
</tbody>
</table>

Table 7.2. Model Construction Times for M/M/1/N queue

7.2.4 Steady State Probability Computation

Table 7.3 gives the number of iterations needed to compute the steady state probabilities. We use rate values \( \lambda = 3 \) and \( \mu = 2 \). The Jacobi method does not converge, as we have
<table>
<thead>
<tr>
<th>N:</th>
<th>Jacobi</th>
<th>Power</th>
<th>PGS</th>
<th>BPGS</th>
<th>PSOR</th>
<th>BPSOR</th>
<th>JOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>n.c.</td>
<td>357</td>
<td>1,836</td>
<td>1,938</td>
<td>355</td>
<td>354</td>
<td>362</td>
</tr>
<tr>
<td>31</td>
<td>n.c.</td>
<td>729</td>
<td>55,220</td>
<td>57,842</td>
<td>765</td>
<td>765</td>
<td>776</td>
</tr>
<tr>
<td>63</td>
<td>n.c.</td>
<td>1,336</td>
<td>n.c.</td>
<td>n.c.</td>
<td>1,416</td>
<td>1,415</td>
<td>1,446</td>
</tr>
<tr>
<td>127</td>
<td>n.c.</td>
<td>2,542</td>
<td>n.c.</td>
<td>n.c.</td>
<td>2,747</td>
<td>2,747</td>
<td>2,773</td>
</tr>
<tr>
<td>255</td>
<td>n.c.</td>
<td>5,026</td>
<td>n.c.</td>
<td>n.c.</td>
<td>5,490</td>
<td>5,490</td>
<td>5,508</td>
</tr>
<tr>
<td>511</td>
<td>n.c.</td>
<td>10,078</td>
<td>n.c.</td>
<td>n.c.</td>
<td>11,015</td>
<td>11,015</td>
<td>11,070</td>
</tr>
<tr>
<td>1,023</td>
<td>n.c.</td>
<td>2,0267</td>
<td>n.c.</td>
<td>n.c.</td>
<td>22,252</td>
<td>22,287</td>
<td></td>
</tr>
<tr>
<td>2,047</td>
<td>n.c.</td>
<td>38,408</td>
<td>n.c.</td>
<td>n.c.</td>
<td>n.c.</td>
<td>42,278</td>
<td></td>
</tr>
<tr>
<td>4,095</td>
<td>n.c.</td>
<td>56,706</td>
<td>n.c.</td>
<td>n.c.</td>
<td>n.c.</td>
<td>62,410</td>
<td></td>
</tr>
<tr>
<td>8,191</td>
<td>n.c.</td>
<td>88,558</td>
<td>n.c.</td>
<td>n.c.</td>
<td>n.c.</td>
<td>97,463</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.3. Steady State Computation for $M/M/1/N$ queue: Number of Iterations

The initialisation of the probability vector uses the assumption that all states are equally probable. In Table 7.4 we see that the numerical analysis performs quite poorly. The generation of the largest queueing system takes less about 1.5 seconds the computation of the steady state probabilities with the best method (Power) takes about 8.5 minutes.  

7.3 Buffer with Erroneous Arrivals

Let us consider a slight variant of the example system from chapter 4.6. Here, we assume that only the packet that contains an incorrectable error must be retransmitted and not all preceding packets as in chapter 4.6. This system is scalable with respect to the number of packets that must be received before their processing can be started. The rest of the functionality is the same as described in chapter 4.6.

7.3.1 Model Statistics

In table 7.5 the size of the MTBDDs for varying $N$ can be found.

7.3.2 Model Checking of Probabilistic SPDL State Formulae

In this section we want to check the following formulae:

1. $\Phi_1 := \mathcal{P}_{\text{csp}}((\neg \text{full})[a^*]^{0..t} \text{(full)}):$ Is the probability to receive $N$ data packets without error within $t$ time units greater or less than $p$?

2. $\Phi_2 := \mathcal{P}_{\text{csp}}((\neg \text{full})[a; \text{TEST1?}; e; rt; a^* \cup a^*]^{0..t} \text{(full)}):$ Is the probability to receive $N$ data packets without error or with at most one non-correctable error within $t$ time units greater or less than $p$, given that this non-correctable error appears in the first data packet? The test formula TEST1 is defined those states, in which it holds that 1 packet has arrived.

---

1 The first line of each column indicates the time per iteration, whereas the second line gives the total time of the computation.
7.3 Buffer with Erroneous Arrivals

<table>
<thead>
<tr>
<th>N:</th>
<th>Model:</th>
<th>States:</th>
<th>Transitions:</th>
<th>Nodes:</th>
<th>Leaves:</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>n.c.</td>
<td>15,001</td>
<td>25,000</td>
<td>1,149</td>
<td>432</td>
</tr>
<tr>
<td>15</td>
<td>n.c.</td>
<td>45,001</td>
<td>75,000</td>
<td>1,198</td>
<td>537</td>
</tr>
<tr>
<td>30</td>
<td>n.c.</td>
<td>90,001</td>
<td>150,000</td>
<td>1,415</td>
<td>566</td>
</tr>
<tr>
<td>50</td>
<td>n.c.</td>
<td>150,001</td>
<td>250,000</td>
<td>1,332</td>
<td>608</td>
</tr>
</tbody>
</table>

Table 7.4. Steady State Computation Times for $M/M/1/N$ queue

Table 7.5. Model Statistics for Buffer with Erroneous Arrivals

3. $\Phi_3 := P_{\text{opt}}(\text{true}[a^*;\text{TEST2};a;\text{co}]^{[0,i]}\text{full})$: Is the probability that the buffer is full after at most $t$ time units and that the $N$th packet contains a correctable error, given that all preceding packets were error free, within the probability bounds given by $\bowtie p$? The test formula $\text{TEST2}$ describes those states, in which it holds that $N - 1$ packets have arrived.

Results and Discussion

In tables 7.6 to 7.8 we can find the sizes of the SLTS of the product Markov chains and the sizes of the MTBDDs representing the respective product Markov chain.

In tables 7.6 and 7.8 we see that the size of the product Markov chain is smaller than that of the original model, which stems from the restrictions that are imposed by the programs for satisfying paths.
In Table 7.7 we see that the size of the product Markov chain is greater than that of the original model. This is clear since an erroneous packet requires retransmission of all packets arrived so far and these retransmission leads to new states in the product Markov chain. In Tables 7.9 to 7.11 we find the construction times of the product Markov chains of the respective formulae.

We can see that the construction times of the product Markov chain are very low. This is not surprising, as we can process a large number of states and transitions at once (cf. algorithm 6.8). The time required for constructing the SPMC mainly depends on the time required for reachability analysis, as we can see from algorithm 6.8. The number of times the main loop of algorithm 6.8 (lines (8) - (17)) has to be executed, is negligible in comparison. In Table 7.12 we find the times required for transient analysis for varying time bounds. Transient analysis was only done for $N = 5000$. We can observe that for varying time bounds the time per iteration does not change significantly. For the most complex...
formula $\Phi_2$ we can observe that both the number of iterations and the time per iteration is larger than for $\Phi_1$ and $\Phi_3$, which can be explained by the fact that in case of $\Phi_2$ the size of the PMC is much larger than in the other cases.

<table>
<thead>
<tr>
<th>Time Bound:</th>
<th>Formula:</th>
<th>$\Phi_1$</th>
<th>$\Phi_2$</th>
<th>$\Phi_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Iterations:</td>
<td>253</td>
<td>253</td>
<td>289</td>
</tr>
<tr>
<td></td>
<td>Time per Iteration:</td>
<td>0.001304</td>
<td>0.00198</td>
<td>0.016955</td>
</tr>
<tr>
<td>50</td>
<td>Iterations:</td>
<td>335</td>
<td>416</td>
<td>523</td>
</tr>
<tr>
<td></td>
<td>Time per Iteration:</td>
<td>0.001224</td>
<td>0.001154</td>
<td>0.015296</td>
</tr>
<tr>
<td>100</td>
<td>Iterations:</td>
<td>416</td>
<td>416</td>
<td>655</td>
</tr>
<tr>
<td></td>
<td>Time per Iteration:</td>
<td>0.001154</td>
<td>0.00216</td>
<td>0.014608</td>
</tr>
<tr>
<td>150</td>
<td>Iterations:</td>
<td>498</td>
<td>498</td>
<td>655</td>
</tr>
<tr>
<td></td>
<td>Time per Iteration:</td>
<td>0.001104</td>
<td>0.00208</td>
<td>0.013588</td>
</tr>
<tr>
<td>200</td>
<td>Iterations:</td>
<td>540</td>
<td>540</td>
<td>655</td>
</tr>
<tr>
<td></td>
<td>Time per Iteration:</td>
<td>0.001056</td>
<td>0.00216</td>
<td>0.013588</td>
</tr>
</tbody>
</table>

Table 7.12. Results for Transient Analysis with $N = 5000$
7.4 Tandem Queueing System with Blocking

7.4.1 Introduction

The tandem queueing network was introduced in [137]. This system consists of two stations where the first is an \( M/\text{Cox}_2/1/N \) station which is coupled in a blocking manner with a second \(-/M/1/N\) queue. Both buffers have the same limited capacity of \( N \). The arrival at station 1 is a Poisson stream with rate \( \lambda \). The service at station 1 obeys a Coxian distribution with two exponentially distributed phases with rate \( \mu_1 \) resp. \( \mu_2 \). After the first phase, phase 2 is entered with probability \( p \), thus, with probability \( 1 - p \) phase 2 is omitted. After being served in station 1, station 2 is entered, this station has service rate \( \kappa \). As both buffers have limited capacity it is possible that station 2 is blocked, i.e. the server is busy and its associated buffer is full. We can find a diagram of the tandem queueing system in figure 7.3.

\[ \begin{align*} 
\lambda & \quad | \quad \text{Buffer} \quad \text{Buffer} \\
& \quad \quad \quad \quad \quad \mu_1 \quad p \\
& \quad \quad \quad \quad \quad \mu_2 \\
& \quad \quad \quad \quad \quad \kappa \\
\end{align*} \]

Fig. 7.3. Tandem Queueing Network

7.4.2 Model Statistics

Table 7.13 gives information about the CTMC, representing the model, we have generated using CASPA. We can see that a quadruplication of the capacity of the queue results in a state space that grows by a factor of 16. The size of the MTBDD grows only linearly. Each quadruplication of the state space only requires roughly 104 additional MTBDD nodes to represent the underlying state space. If we compare the maximum number of MTBDD nodes for the queueing system from section 7.2 for \( N = 8,191 \), which has roughly the same state space size as the tandem queueing system with \( N = 63 \), we can see that the \( M/M/1 \)-queue has a maximum number of 65,604 nodes, whereas in the case of the tandem queueing network we have a maximum of only 3,328 nodes. The reason for this huge difference lies in the fact that the tandem system is composed of two parallel processes, opposed to the \( M/M/1 \)-queue which consists of a single process, such that intermediate results can be stored locally in the subprocess whose state space is actually generated. These local intermediate state spaces are much smaller than the overall system, thus less nodes are required.

7.4.3 Model Construction Time

In table 7.14 we find the times to construct the models. We can see there that the number of iterations and thus the time to compute the reachable portion of the system’s state space rapidly increases, although the system does not contain unreachable states. A possible explanation of this somewhat surprising fact may be that the MTBDDs that are generated during reachability analysis are quite irregular such that they cannot be represented compactly and their traversal consumes much time.
7.4 Tandem Queueing System with Blocking

<table>
<thead>
<tr>
<th>N:</th>
<th>Model:</th>
<th>MTBDD:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>States:</td>
<td>Transitions:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>28</td>
<td>78</td>
</tr>
<tr>
<td>15</td>
<td>496</td>
<td>1,650</td>
</tr>
<tr>
<td>63</td>
<td>8,128</td>
<td>28,098</td>
</tr>
<tr>
<td>255</td>
<td>130,816</td>
<td>456,450</td>
</tr>
<tr>
<td>1,023</td>
<td>2,096,128</td>
<td>7,330,820</td>
</tr>
</tbody>
</table>

Table 7.13. Model Statistics for Tandem Queueing Network

<table>
<thead>
<tr>
<th>N:</th>
<th>Construction:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (sec.): Iterations:</td>
</tr>
<tr>
<td>3</td>
<td>&lt; 10^{-6}</td>
</tr>
<tr>
<td>5</td>
<td>&lt; 10^{-6}</td>
</tr>
<tr>
<td>10</td>
<td>&lt; 10^{-6}</td>
</tr>
<tr>
<td>15</td>
<td>0.01</td>
</tr>
<tr>
<td>63</td>
<td>0.09</td>
</tr>
<tr>
<td>255</td>
<td>1.94</td>
</tr>
<tr>
<td>511</td>
<td>9.34</td>
</tr>
<tr>
<td>1023</td>
<td>37.56</td>
</tr>
</tbody>
</table>

Table 7.14. Model Construction Times for Tandem Queueing Network

7.4.4 Comparison with PRISM

In table 7.15 we can find the numbers concerning number of iterations for generating the reachable portion of the overall state space of the model, generation time, including reachability analysis, and the final number of nodes for the tandem queueing system. Here, we see that the generation times in CASPA are always smaller than in the case of PRISM, whereas the MTBDD sizes are always smaller in PRISM. This stems mostly from the fact that in PRISM, in contrast to CASPA the action labels of the specification are not stored in the MTBDD. In CASPA we have to store these information to be able to compute throughput measures and more important to be able to do SPDL model checking. Another reason for different MTBDD sizes lies in the different ways the state space is generated and then encoded as an MTBDD in CASPA and PRISM.

<table>
<thead>
<tr>
<th>N:</th>
<th>States:</th>
<th>Iterations:</th>
<th>Generation Time:</th>
<th>Nodes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>28</td>
<td>11</td>
<td>0.0024</td>
<td>63</td>
</tr>
<tr>
<td>5</td>
<td>66</td>
<td>17</td>
<td>0.0046</td>
<td>100</td>
</tr>
<tr>
<td>10</td>
<td>231</td>
<td>32</td>
<td>0.0064</td>
<td>147</td>
</tr>
<tr>
<td>15</td>
<td>496</td>
<td>47</td>
<td>0.013</td>
<td>123</td>
</tr>
<tr>
<td>63</td>
<td>8,128</td>
<td>191</td>
<td>0.119</td>
<td>183</td>
</tr>
<tr>
<td>255</td>
<td>130,816</td>
<td>767</td>
<td>2.332</td>
<td>243</td>
</tr>
<tr>
<td>511</td>
<td>523,776</td>
<td>1,535</td>
<td>9.68</td>
<td>273</td>
</tr>
<tr>
<td>1,023</td>
<td>2,096,128</td>
<td>2,071</td>
<td>40.629</td>
<td>303</td>
</tr>
</tbody>
</table>

Table 7.15. Model Statistics for PRISM
Table 7.16 gives the number of iterations to compute the steady state probabilities. In table 7.17 we can find the times needed to compute the steady state probabilities. Again, the first line in each column indicates the time per iteration whereas the second line shows the total time of steady state probability computation. From tables 7.16 and 7.17 we can see that for the cases \( N = 255 \) and \( N = 1023 \) the Jacobi method is the only method that converged in less than 100,000 computation steps.

### 7.4.6 Model Checking of Probabilistic SPDL State Formulae

For the tandem queueing network we want to check the following simple SPDL formula:

- \( \Phi := P_{sp} (\neg \text{full}[Act^*]^{(0,4)} \text{full}) \)

\( Act^* \) is an abbreviation and states that the SPDL program generates a language that contains all finite words that are derivable from an arbitrary concatenation of actions that occur in the specification of the tandem system. \( \text{full} \) is a state formula that states that the two queues of the system are full. In fact, \( \Phi \) can be interpreted as a simple CSL until-formula: \( P_{sp}(\neg \text{full} \cup^{(0,4)} \text{full}) \)
Results and Discussion

In table 7.18, we find the sizes of the product Markov chain for various values of N. We can see that the number of states of the SPMC is always larger than the original models. This is not surprising as we do not impose any restrictions on the satisfying paths, as long as we finally reach a state in which the queues are fully occupied. Finally, in table 7.19, we can find the construction times for the product Markov chain.

<table>
<thead>
<tr>
<th>N:</th>
<th>Model:</th>
<th>MTBDD:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>States:</td>
<td>Transitions:</td>
</tr>
<tr>
<td>10</td>
<td>589</td>
<td>1,920</td>
</tr>
<tr>
<td>15</td>
<td>1,334</td>
<td>4,455</td>
</tr>
<tr>
<td>63</td>
<td>23,750</td>
<td>82,215</td>
</tr>
<tr>
<td>255</td>
<td>389,894</td>
<td>1,360,940</td>
</tr>
<tr>
<td>511</td>
<td>1,566,214</td>
<td>5,474,340</td>
</tr>
<tr>
<td>1,023</td>
<td>6,278,150</td>
<td>21,958,700</td>
</tr>
</tbody>
</table>

Table 7.18. Model Statistics for Product Markov Chain for Formula $\Phi$

<table>
<thead>
<tr>
<th>N:</th>
<th>Construction:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (sec.)</td>
</tr>
<tr>
<td>10</td>
<td>0.02</td>
</tr>
<tr>
<td>15</td>
<td>0.03</td>
</tr>
<tr>
<td>63</td>
<td>0.44</td>
</tr>
<tr>
<td>255</td>
<td>8.53</td>
</tr>
<tr>
<td>511</td>
<td>39.34</td>
</tr>
<tr>
<td>1,023</td>
<td>187.3</td>
</tr>
</tbody>
</table>

Table 7.19. Product Markov Chain Construction Time for Formula $\Phi$

7.5 Polling System

7.5.1 Introduction

This case study analyses a cyclic server-polling system with N stations. The model is taken from [80] therefore it was modelled as GSPN. For $N = 2$ we see in figure 7.4 its principal construction. For more than two stations it can be extended in the natural way. Place $idle_i$ expresses that station $i$ is idle. Place $busy_i$ means that station $i$ generated a service request. The server polls the $N$ stations in a cyclic way. After polling station $i$, station $i$ is served. If station $i$ is idle it is skipped, modelled by transition $skip_i$, which is untimed. All rates are exponentially distributed. There are three different rates: generating request $\lambda_i$, polling $\gamma_i$ and processing the request $\mu_i$. As we assume polling systems to be symmetric, all $\lambda_i$ are identical, the same holds for $\gamma_i$ and $\mu_i$. The value of $\lambda_i$ depends on the number of stations: $\lambda_i = \mu_i / N$. 
7.5.2 Model Statistics

Table 7.20 gives information about the CTMC, representing the model, we have generated using CASPA. Interesting here, is the fact that the peak number of MTBDD nodes needed to represent a system of more than 30 million states ($N = 20$) is with 20,252 nodes less than the half as in the case of the tandem queueing network where for the largest system a peak number of 47,909 nodes can be observed. The reason herefore lies again in the compositional structure of the polling system specification, which is composed of many small parallel processes.

<table>
<thead>
<tr>
<th>N</th>
<th>Model</th>
<th>MTBDD:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>States</td>
<td>Transitions</td>
</tr>
<tr>
<td>5</td>
<td>240</td>
<td>800</td>
</tr>
<tr>
<td>10</td>
<td>15,360</td>
<td>89,600</td>
</tr>
<tr>
<td>15</td>
<td>737,280</td>
<td>6,144,000</td>
</tr>
<tr>
<td>20</td>
<td>31,457,280</td>
<td>702,546,000</td>
</tr>
</tbody>
</table>

Table 7.20. Model Statistics for Polling

7.5.3 Model Construction Times

In table 7.21 we find the times to construct the models.

7.5.4 Comparison with PRISM

In table 7.22 we can find the numbers concerning number of iterations for generating the reachable portion of the overall state space of the model, generation time, including reach-
ability analysis, and the final number of nodes for the polling system. Here again, we see that the generation times in CASPA are always smaller than in the case of PRISM, whereas the MTBDD sizes are always smaller in PRISM. Again, we can give the same reasons as in the case of the tandem queueing network to explain this fact.

Table 7.21. Model Construction Times for Polling

<table>
<thead>
<tr>
<th>N</th>
<th>Construction:</th>
<th>Time (sec.)</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td>&lt; $10^{-6}$</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>0.01</td>
<td>21</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>0.05</td>
<td>31</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>0.17</td>
<td>41</td>
</tr>
</tbody>
</table>

Table 7.22. Model Statistics for PRISM

<table>
<thead>
<tr>
<th>N</th>
<th>States</th>
<th>Iterations</th>
<th>Generation Time</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>240</td>
<td>11</td>
<td>0.036</td>
<td>271</td>
</tr>
<tr>
<td>10</td>
<td>15,360</td>
<td>21</td>
<td>0.075</td>
<td>921</td>
</tr>
<tr>
<td>15</td>
<td>737,280</td>
<td>31</td>
<td>0.09</td>
<td>1,942</td>
</tr>
<tr>
<td>20</td>
<td>31,457,280</td>
<td>41</td>
<td>0.64</td>
<td>3,346</td>
</tr>
</tbody>
</table>

7.5.5 Steady State Probability Computation

Table 7.23 gives the number of iterations required to compute the steady state probabilities. In table 7.24 we find the times for computing the steady state probabilities for the various system sizes.

Table 7.23. Steady State Probability Computation for Polling: Number of Iterations

<table>
<thead>
<tr>
<th>N</th>
<th>Number of Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Jacobi</td>
</tr>
<tr>
<td>5</td>
<td>73</td>
</tr>
<tr>
<td>10</td>
<td>406</td>
</tr>
<tr>
<td>15</td>
<td>442</td>
</tr>
<tr>
<td>20</td>
<td>920</td>
</tr>
</tbody>
</table>

7.6 Kanban System

7.6.1 Introduction

For this case study we computed steady-state probabilities for the well-known Kanban example (originally described in [38]). We model a Kanban system with four cells, a single
type of Kanban cards, and the possibility that some workpiece may need to be reworked. We use \( N \) to denote the number of tokens in the system.

### 7.6.2 Model Statistics

Table 7.25 gives information about the CTMC, representing the model, we have generated using CASPA. The Kanban system is the largest system we have investigated in this thesis. Despite a maximum state space of more than 5 billion states, its generation consumes only a few seconds, for the largest model with 5,519,907,575 reachable states it is possible to generate, and compute the reachable state space within 17.22 seconds. The maximum number of MTBDD nodes is in this case not greater than 414,719.

| \( N \) | Model: | Transition: | MTBDD: | | | |
|---|---|---|---|---|---|
| | States: | | Nodes: | Leaves: | |
| 3 | 58,400 | 446,400 | Peak 5.738 | Final 2.241 | 14 |
| 4 | 454,475 | 3,979,850 | 14,058 | 3,990 | 14 |
| 5 | 2,546,432 | 4.44e+07 | 25,514 | 5,392 | 14 |
| 6 | 11,261,376 | 1.13e+08 | 47,395 | 8,086 | 14 |
| 7 | 41,644,800 | 4.50e+08 | 76,230 | 10,389 | 14 |
| 8 | 133,865,325 | 1.50e+09 | 116,783 | 13,998 | 14 |
| 9 | 384,392,800 | 4.74e+09 | 168,694 | 17,762 | 14 |
| 10 | 1,005,927,208 | 1.20e+10 | 248,461 | 23,231 | 14 |
| 11 | 2,435,541,472 | 2.98e+10 | 323,115 | 27,411 | 14 |
| 12 | 5,519,907,575 | 6.88e+10 | 414,719 | 32,324 | 14 |

Table 7.25. Model Statistics for Kanban

### 7.6.3 Model Construction Times

In table 7.26 we find the times to construct the models.
### 7.6.4 Comparison with PRISM

In table 7.27 we can find the numbers concerning number of iterations for generating the reachable portion of the overall state space of the model, generation time, including reachability analysis, and the final number of nodes for the Kanban system. We can see that CASPA consumes less time for generation of the reachable state space of the Kanban system. For small $N$ it is also the case that the final number of nodes required to represent the state space of the system is smaller for CASPA, but from $N = 6$ on CASPA requires a greater number of nodes than PRISM. Here, we presumably find the reason for this observation in the different ways the state spaces are generated and thus the different ways states are encoded in both tools.

<table>
<thead>
<tr>
<th>$N$</th>
<th>States:</th>
<th>Iterations:</th>
<th>Generation Time:</th>
<th>Nodes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>58,400</td>
<td>43</td>
<td>0.06</td>
<td>2,474</td>
</tr>
<tr>
<td>4</td>
<td>454,475</td>
<td>57</td>
<td>0.285</td>
<td>4,900</td>
</tr>
<tr>
<td>5</td>
<td>2,546,432</td>
<td>71</td>
<td>0.918</td>
<td>6,308</td>
</tr>
<tr>
<td>6</td>
<td>11,261,376</td>
<td>85</td>
<td>1.828</td>
<td>7,876</td>
</tr>
<tr>
<td>7</td>
<td>41,644,800</td>
<td>99</td>
<td>3.578</td>
<td>9,521</td>
</tr>
<tr>
<td>8</td>
<td>133,865,325</td>
<td>113</td>
<td>6.05</td>
<td>14,702</td>
</tr>
<tr>
<td>9</td>
<td>384,392,800</td>
<td>127</td>
<td>9.304</td>
<td>17,196</td>
</tr>
<tr>
<td>10</td>
<td>1,005,927,208</td>
<td>141</td>
<td>13.972</td>
<td>19,877</td>
</tr>
<tr>
<td>11</td>
<td>2,435,541,472</td>
<td>155</td>
<td>20.119</td>
<td>22,666</td>
</tr>
<tr>
<td>12</td>
<td>5,519,907,575</td>
<td>169</td>
<td>27.978</td>
<td>25,710</td>
</tr>
</tbody>
</table>

**Table 7.27. Model Statistics for PRISM**

### 7.6.5 Steady State Probability Computation

Table 7.28 gives the number of iterations to compute the steady state probabilities. In table 7.29 we find the times per iteration and the total computation time of steady state probabilities.
### Table 7.28. Steady State Probability Computation for Kanban: Number of Iterations

<table>
<thead>
<tr>
<th>N</th>
<th>Jacobi</th>
<th>Power</th>
<th>PGS</th>
<th>BPGS</th>
<th>PSOR</th>
<th>BPSOR</th>
<th>JOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>n.c.</td>
<td>946</td>
<td>213</td>
<td>237</td>
<td>241</td>
<td>265</td>
<td>300</td>
</tr>
<tr>
<td>4</td>
<td>n.c.</td>
<td>1,312</td>
<td>319</td>
<td>359</td>
<td>363</td>
<td>403</td>
<td>466</td>
</tr>
<tr>
<td>5</td>
<td>n.c.</td>
<td>1,648</td>
<td>457</td>
<td>507</td>
<td>520</td>
<td>569</td>
<td>663</td>
</tr>
<tr>
<td>6</td>
<td>n.c.</td>
<td>2,023</td>
<td>625</td>
<td>683</td>
<td>709</td>
<td>767</td>
<td>891</td>
</tr>
<tr>
<td>7</td>
<td>n.c.</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

### Table 7.29. Steady State Probability Computation Times for Kanban

<table>
<thead>
<tr>
<th>N</th>
<th>Time (sec):</th>
<th>Jacobi</th>
<th>Power</th>
<th>PGS</th>
<th>BPGS</th>
<th>PSOR</th>
<th>BPSOR</th>
<th>JOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td>0.006406</td>
<td>0.00608</td>
<td>0.006793</td>
<td>0.006805</td>
<td>0.006755</td>
<td>0.006900</td>
<td>2.08</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>0.054047</td>
<td>0.055298</td>
<td>0.061086</td>
<td>0.058237</td>
<td>0.057519</td>
<td>0.056974</td>
<td>26.62</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>0.317949</td>
<td>0.360744</td>
<td>0.371105</td>
<td>0.433788</td>
<td>0.384851</td>
<td>0.451373</td>
<td>299.63</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>1.489244</td>
<td>1.74976</td>
<td>1.817950</td>
<td>1.743032</td>
<td>1.852295</td>
<td>1.693356</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>-</td>
<td>75.600</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

#### 7.6.6 Model Checking Probabilistic SPDL State Formulae

We will check the following three probabilistic state formulae:

- \( \Phi_1 := P_{csp}(true \land [in_1; (in_1^*; tback_1; in_1^*; tredo_1; in_1^*); tok_1]^{[0,t]} \land true) \): Is the requirement that within \( t \) time units exactly three reworks are required in station 1 is satisfied with a probability that lies within \( p \) percent.

- \( \Phi_2 := P_{csp}(true \land [in_1; (in_1^*; tback_1; in_1^*; tredo_1; in_1^*); tok_1; tsync_{123}; tok_2]^{[0,t]} \land true) \): Can the requirement that a job needs three reworks in the first and zero reworks in the second station be satisfied with a probability of \( \approx p \), given that we have to observe the time bound \( t \)?

- \( \Phi_3 := P_{csp}(true \land [in_1; tok_1; tsync_{123}; in_1; tok_1; tsync_{123}; tok_2; tok_3; tsync_{234}; tok_4; tout_4]^{[0,t]} \land true) \): Is the probability that a single job needs at most \( t \) time units to go through all 4 stations within \( p \) percent?

For the formulae \( \Phi_1 \) to \( \Phi_3 \) we can observe that the state space of the SPMC is dramatically smaller than that of the original system, which stems from the fact that for all three formulae only very specific paths in the system are of interest. We can observe that for \( \Phi_3 \) the size of the SPMC is independent of the number of Kanban cards, which is not surprising, as we consider a specific card that goes through the system. In table 7.30 we can find the generation time for the SPMC for \( \Phi_1 \) to \( \Phi_3 \) with varying \( N \). In table 7.31 we can find the
7.7 Flexible Manufacturing System

### 7.7.1 Introduction

This case study is based on the flexible manufacturing system (FMS) of [39]. $N$ denotes the number of tokens in the system.

### 7.7.2 Model Statistics

Table 7.32 gives information about the CTMC, representing the model, we have generated using CASPA.

### 7.7.3 Model Construction Times

In table 7.33 we find the times to construct the models.

---

| Table 7.30. Model Statistics for Kanban for $\Phi_1$ to $\Phi_3$ |
| --- | --- | --- | --- |
| $N$: | $\Phi_1$: States: Generation (sec): | $\Phi_2$: States: Generation Time (sec): | $\Phi_3$: States: Generation (sec): |
| 4 | 35 | 0.19 | 42 | 0.12 | 13 | 0.08 |
| 5 | 44 | 0.39 | 53 | 0.19 | 13 | 0.09 |
| 6 | 53 | 0.91 | 64 | 0.93 | 13 | 0.10 |
| 7 | 62 | 1.71 | 75 | 1.93 | 13 | 0.13 |
| 8 | 71 | 2.93 | 86 | 2.95 | 13 | 0.15 |
| 9 | 80 | 4.86 | 97 | 4.76 | 13 | 0.16 |
| 10 | 89 | 7.76 | 108 | 7.71 | 13 | 0.20 |
| 11 | 98 | 12.88 | 119 | 11.46 | 13 | 0.21 |
| 12 | 107 | 16.03 | 130 | 17.89 | 13 | 0.24 |

| Table 7.31. Results for Transient Analysis with $N = 10$ |
| --- | --- | --- | --- |
| Time Bound: Iterations: Time per Iteration: | Iterations: Time per Iteration: | Iterations: Time per Iteration: |
| 50 | 192 | 0.000052 | 558 | 0.000054 | 182 | $< 10^{-6}$ |
| 100 | 192 | 0.000052 | 1011 | 0.000040 | 192 | $< 10^{-6}$ |
| 150 | 192 | 0.000052 | 1450 | 0.000041 | 192 | $< 10^{-6}$ |
| 200 | 192 | 0.000052 | 1882 | 0.000043 | 192 | $< 10^{-6}$ |

results for transient analysis of the SPMCs obtained for formulae $\Phi_1$ to $\Phi_3$, we have restricted ourselves to the case where $N = 10$. In case of formulae $\Phi_1$ and $\Phi_3$ the Fox-Glynn algorithm detected a steady state for time bounds $> 50$. 

---
### 7.7.4 Comparison with PRISM

In table 7.34 we can find the PRISM numbers concerning number of iterations for generating the reachable portion of the overall state space of the model, generation time, including reachability analysis, and the final number of nodes for the flexible manufacturing system. Here we can see that CASPA always yields better results than PRISM, with exception \( N = 10 \), where the generation takes in CASPA longer than in PRISM. The MTBDD sizes of CASPA are roughly a half of that of PRISM, this is surely due to the different state encoding schemes in the respective tools.

### 7.7.5 Steady State Probability Computation

Table 7.35 gives the number of iterations needed to compute the steady state probabilities in CASPA. In table 7.36 we find the times required to compute the steady state probabilities of the differently scaled FMS models.
### Table 7.34. Model Statistics for PRISM

<table>
<thead>
<tr>
<th>N</th>
<th>States</th>
<th>Iterations</th>
<th>Generation Time</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>810</td>
<td>17</td>
<td>0.041</td>
<td>4,371</td>
</tr>
<tr>
<td>3</td>
<td>6,520</td>
<td>25</td>
<td>0.082</td>
<td>10,896</td>
</tr>
<tr>
<td>4</td>
<td>35,910</td>
<td>33</td>
<td>0.334</td>
<td>28,470</td>
</tr>
<tr>
<td>5</td>
<td>152,712</td>
<td>41</td>
<td>0.625</td>
<td>50,877</td>
</tr>
<tr>
<td>6</td>
<td>537,768</td>
<td>48</td>
<td>1.4</td>
<td>78,128</td>
</tr>
<tr>
<td>7</td>
<td>1,639,440</td>
<td>57</td>
<td>2.571</td>
<td>120,384</td>
</tr>
<tr>
<td>8</td>
<td>4,459,455</td>
<td>65</td>
<td>9.85</td>
<td>215,520</td>
</tr>
<tr>
<td>9</td>
<td>11,058,190</td>
<td>73</td>
<td>17.479</td>
<td>309,067</td>
</tr>
<tr>
<td>10</td>
<td>25,397,658</td>
<td>81</td>
<td>28.784</td>
<td>406,018</td>
</tr>
</tbody>
</table>

### Table 7.35. Steady State Probability Computation Times for FMS: Number of Iterations

<table>
<thead>
<tr>
<th>N</th>
<th>Time (sec):</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Jacobi</td>
</tr>
<tr>
<td>3</td>
<td>0.000740</td>
</tr>
<tr>
<td></td>
<td>0.004552</td>
</tr>
<tr>
<td></td>
<td>0.019514</td>
</tr>
<tr>
<td>4</td>
<td>0.068241</td>
</tr>
<tr>
<td></td>
<td>0.284936</td>
</tr>
<tr>
<td></td>
<td>0.562107</td>
</tr>
<tr>
<td>5</td>
<td>2.821828</td>
</tr>
<tr>
<td>6</td>
<td>7.696084</td>
</tr>
</tbody>
</table>

### Table 7.36. Steady State Probability Computation Times for FMS

### 7.8 Mainframe with Software Errors

#### 7.8.1 Introduction

This case study is based on [71]. The mainframe processes two external types of requests that generate system load:
1. The mainframe has to serve database queries that are generated by some users.
2. The mainframe has to provide resources for programmers that use the computer to compile and test their programs.

A third, internal load type are software failures. Hardware failures are assumed to be very rare in comparison to software failures thus, we do not consider them in our model. The three different load types have different priorities, with which requests that contribute to the respective load type have to be processed, failures have highest priority, programmer load has the lowest.

The system has two queues, one for user jobs, i.e. database requests and a second queue for programmer jobs. We assume that the capacity of the queue for user jobs is four times larger than that for programmer jobs. We use \( N \) to denote the capacity of the programmers jobs’ queue. A diagrammatic overview of the system’s architecture can be found in figure 7.5

![Diagram of mainframe multiprocessor](image)

**Fig. 7.5. Mainframe Multiprocessor**

### 7.8.2 Model Statistics

Table 7.37 gives information about the CTMC, representing the model, we have generated using CASPA. We can see here that the number of MTBDD nodes needed to represent the system’s state space is huge, even for small systems. In table 7.38 we see the number of nodes required, if no reachability analysis is performed. This increase is due to the restriction of the state space to its reachable portion. Normally, reachability analysis destroys the regularity of the state space which is besides compositionality the key to a compact representation of state spaces using MTBDDs.

### 7.8.3 Model Construction Times

In table 7.39 we find the times to construct the models.

### 7.8.4 Steady State Probability Computation

Table 7.40 gives the number of iterations needed to compute the steady state probabilities. In table 7.41 we can find the times needed to compute the steady state probabilities of the mainframe model. We can see that already for very small models the time per iteration is
### Table 7.37. Model Statistics for Mainframe

<table>
<thead>
<tr>
<th>N:</th>
<th>MTBDD:</th>
<th>States:</th>
<th>Transitions:</th>
<th>Nodes:</th>
<th>Leaves:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Peak</td>
<td>Final</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2,496</td>
<td>11,472</td>
<td>8,087</td>
<td>6,653</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>4,080</td>
<td>18,945</td>
<td>13,870</td>
<td>11,852</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>6,048</td>
<td>28,266</td>
<td>18,446</td>
<td>16,033</td>
<td>18</td>
</tr>
<tr>
<td>6</td>
<td>8,400</td>
<td>39,435</td>
<td>25,881</td>
<td>23,221</td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td>11,136</td>
<td>52,452</td>
<td>32,340</td>
<td>28,887</td>
<td>18</td>
</tr>
<tr>
<td>8</td>
<td>14,256</td>
<td>67,317</td>
<td>42,409</td>
<td>38,664</td>
<td>18</td>
</tr>
<tr>
<td>10</td>
<td>21,648</td>
<td>102,591</td>
<td>62,399</td>
<td>57,176</td>
<td>18</td>
</tr>
<tr>
<td>12</td>
<td>30,576</td>
<td>145,257</td>
<td>91,296</td>
<td>84,703</td>
<td>18</td>
</tr>
<tr>
<td>16</td>
<td>53,040</td>
<td>252,765</td>
<td>142,231</td>
<td>133,683</td>
<td>18</td>
</tr>
<tr>
<td>32</td>
<td>204,336</td>
<td>978,477</td>
<td>509,229</td>
<td>484,934</td>
<td>18</td>
</tr>
<tr>
<td>64</td>
<td>801,840</td>
<td>3,849,160</td>
<td>1,903,767</td>
<td>1,823,069</td>
<td>18</td>
</tr>
</tbody>
</table>

### Table 7.38. MTBDD Sizes for Mainframe with Unreachable States

<table>
<thead>
<tr>
<th>N:</th>
<th>MTBDD:</th>
<th>Nodes:</th>
<th>Leaves:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Peak</td>
<td>Final</td>
</tr>
<tr>
<td>3</td>
<td>3,589</td>
<td>1,737</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>4,584</td>
<td>2,122</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>4,695</td>
<td>2,383</td>
<td>18</td>
</tr>
<tr>
<td>6</td>
<td>4,879</td>
<td>2,432</td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td>6,824</td>
<td>3,207</td>
<td>18</td>
</tr>
<tr>
<td>8</td>
<td>6,217</td>
<td>2,980</td>
<td>18</td>
</tr>
<tr>
<td>10</td>
<td>6,942</td>
<td>3,749</td>
<td>18</td>
</tr>
<tr>
<td>12</td>
<td>7,307</td>
<td>4,038</td>
<td>18</td>
</tr>
<tr>
<td>16</td>
<td>8,456</td>
<td>4,367</td>
<td>18</td>
</tr>
<tr>
<td>32</td>
<td>10,555</td>
<td>6,639</td>
<td>18</td>
</tr>
<tr>
<td>64</td>
<td>15,788</td>
<td>11,606</td>
<td>18</td>
</tr>
</tbody>
</table>

### Table 7.39. Model Construction Times for Mainframe

<table>
<thead>
<tr>
<th>N:</th>
<th>Construction:</th>
<th>Time (sec.)</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.09</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.15</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.21</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0.42</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0.57</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.75</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1.16</td>
<td>61</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1.74</td>
<td>71</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>3.16</td>
<td>91</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>14.12</td>
<td>171</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>63.23</td>
<td>331</td>
<td></td>
</tr>
</tbody>
</table>
Table 7.40. Steady State Probability Computation for Mainframe: Number of Iterations

<table>
<thead>
<tr>
<th>N</th>
<th>Number of Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Jacobi</td>
</tr>
<tr>
<td>3</td>
<td>n.c.</td>
</tr>
<tr>
<td>4</td>
<td>n.c.</td>
</tr>
<tr>
<td>5</td>
<td>n.c.</td>
</tr>
<tr>
<td>6</td>
<td>n.c.</td>
</tr>
<tr>
<td>7</td>
<td>n.c.</td>
</tr>
</tbody>
</table>

Table 7.41. Steady State Probability Computation Times for Mainframe

much larger than for all other models considered so far. For PGS, the time per iteration for $N = 7$ is almost 80 seconds leading to an overall computation time of about 21 hours, recall that for the Kanban system the analysis of a system with more than 41 millions of states took about the same time. Thus, we can conclude that the hybrid approach is not very well suitable for this model.

Table 7.41. Steady State Probability Computation Times for Mainframe

7.9 Fault Tolerant Multi Computer System (FTMCS)

7.9.1 Introduction

This example is based on a case study described originally in [125] and used again in [43]. Due to the different modelling formalisms (stochastic activity networks versus stochastic process algebra), some re-modelling effort was required. The original model consists of $N$ computers each of which has the following components:

- 3 memory modules, of which 2 must be operational
- 3 CPU units, of which 2 must be operational
- 2 I/O ports, of which 1 must be operational
7.9 Fault Tolerant Multi Computer System (FTMCS)

- 2 error-handling chips, which are not redundant.

Each CPU and I/O-port consists of 6 non-redundant chips. Each memory module possesses 41 RAM chips, of which at most 2 may fail, and 2 interface chips that all must be operational. A computer fails, if one of its components fails. The overall system is operational if at least one computer is operational. A diagramatic overview can be found in Fig. 7.6.

![Diagram of FTMCS](image)

**Fig. 7.6.** Configuration of a single computer for the fault tolerant multi computer system

### 7.9.2 Model Statistics

We have generated the CTMC of the FTMCS for different configurations: C1 is the configuration of the original system (i.e. consisting of two computers with three memory modules each) which has about 750,000 reachable states. C2 consists of two computers with only one memory module each. C3 is identical to C2 but possesses no redundant I/O port. C4 has the same configuration as C2, but consists of 3 computers instead of 2. C5 has 3 computers and 3 memory modules.

Table 7.42 gives information about the CTMC, representing the model, we have generated using CASPA.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Model:</th>
<th>MTBDD:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>States:</td>
<td>Transitions:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Peak</td>
</tr>
<tr>
<td>C1</td>
<td>753,664</td>
<td>9,109,500</td>
</tr>
<tr>
<td>C2</td>
<td>2,132</td>
<td>10,456</td>
</tr>
<tr>
<td>C3</td>
<td>889</td>
<td>3,536</td>
</tr>
<tr>
<td>C4</td>
<td>123,760</td>
<td>917,712</td>
</tr>
<tr>
<td>C5</td>
<td>381,681,664</td>
<td>6,763,320,000</td>
</tr>
</tbody>
</table>

**Table 7.42.** Model Statistics for FTMCS

### 7.9.3 Model Construction Times

In table 7.43 we find the times to construct the models.
7.9.4 Transient Probability Computation

We compute the survival probability in dependence of the mission time. For a mission time \( t \) of 1000 time units we find the results of numerical computation in table 7.44. The survival probability, dependent on the mission time, is shown in Fig. 7.7.

Our results are not directly comparable to the ones reported in [43] (where multi-valued decision diagrams and matrix diagrams are employed as the underlying data structures): Firstly, we consider slightly different system configurations, and secondly, we do not exploit lumpability (which is due to replicated components). However, it is interesting that [43] reports a computation time of 15.5 sec per iteration for a 463,000 state model, while we measured only 0.12 sec per iteration for the 750,000 state model (the machine speeds are almost identical, and ours has only one third of the memory).

7.9.5 Model Checking of Probabilistic SPDL State Formulae

In this section we want to check the following probabilistic state formula:

- \( \Phi_1 := P_{\text{top}}(\text{true}[(\text{mem\_fail}; \text{mem\_fail}^*; \text{computer\_fail}^*; \text{system\_fail})^0]_{[0,t]} \text{true}) \): Does the probability that computer failures (action computer\_failure) and subsequently a system failure (action system\_failure) is only due to memory failures (action: mem\_fail) lie within the bounds as given by \( \bowtie p \), given that the maximum time to reach a state in which the system fails is less or equal to \( t \)?

Results and Discussion

In table 7.45 we find we can find the sizes of the SLTS of the product Markov chains and the sizes of the MTBDDs representing the respective product Markov chain.
In Table 7.45 we see that the size of the product Markov chain is smaller than that of the original model, which stems from the restrictions that are imposed by the programs for satisfying paths.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Model:</th>
<th>MTBDD:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>States</td>
<td>Transitions</td>
</tr>
<tr>
<td>C1</td>
<td>4,098</td>
<td>34,815</td>
</tr>
<tr>
<td>C2</td>
<td>33</td>
<td>129</td>
</tr>
<tr>
<td>C3</td>
<td>11</td>
<td>30</td>
</tr>
<tr>
<td>C4</td>
<td>134</td>
<td>580</td>
</tr>
</tbody>
</table>

Table 7.45. Model Statistics for Product Markov Chain for Formula $\Phi_1$

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Construction:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (sec.)</td>
</tr>
<tr>
<td>C1</td>
<td>0.06</td>
</tr>
<tr>
<td>C2</td>
<td>0.02</td>
</tr>
<tr>
<td>C3</td>
<td>0.02</td>
</tr>
<tr>
<td>C4</td>
<td>0.06</td>
</tr>
</tbody>
</table>

Table 7.46. Product Markov Chain Construction Time for Formula $\Phi_1$
7.10 Handover Procedure of a Cellular Mobile Radio Network

7.10.1 Introduction

This case study is described in [14]. This very simple model of a mobile radio network consists of several cells, a single user who moves from cell to cell, and a switching centre (SWC). The number of cells is the scaling factor of the model. Other users (the "population") are not modelled explicitly. Their effect is modelled by the load level of the SWC and the state of the individual cells which may be either full or not full. The user can have several status, either being idle, he might request a switch, or he waits for a handover, in case of non-successful switching request he might retry switching or the user is active within a cell. The switching centre can have different load factors: low and high load or blocked, in which case a switching request of a user will be refused. We use $N$ to denote the number of cells. For $1 \leq i < N$, the user can move from cell $i$ to $i + 1$, if the user is in cell $N$ he can only move to cell 1.

![Diagram](image)

Fig. 7.8. Wireless communication network consisting of $N$ cells

7.10.2 Model Statistics

Table 7.47 gives information about the CTMC, representing the model, we have generated using CASPA.

<table>
<thead>
<tr>
<th>$N$</th>
<th>Model:</th>
<th>MTBDD:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>States:</td>
<td>Transitions:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4,800</td>
<td>59,128</td>
</tr>
<tr>
<td>10</td>
<td>307,200</td>
<td>4,042,500</td>
</tr>
<tr>
<td>15</td>
<td>14,745,600</td>
<td>267,805,000</td>
</tr>
<tr>
<td>20</td>
<td>629,145,600</td>
<td>14,572,800,000</td>
</tr>
</tbody>
</table>

Table 7.47. Model Statistics for Wireless Radio Communication Network
7.10.3 Model Construction Times

In table 7.48 we find the times to construct the models. These times include the time for the derivation of a CTMC (represented as an MTBDD) from the process algebraic system description, and the time for the computation of the reachable portion of the state space using a BDD based algorithm [130].

<table>
<thead>
<tr>
<th>N</th>
<th>Construction:</th>
<th>Time (sec.)</th>
<th>Iterations:</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.01</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.16</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>4.19</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>221.11</td>
<td>53</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.48. Model Construction Times for Wireless Radio Communication Network

7.10.4 Steady State Probability Computation

Table 7.49) gives the times taken to compute the steady state probabilities. In table 7.50 we find the times it took to compute steady state probabilities. Here, it can be noted that only

<table>
<thead>
<tr>
<th>N</th>
<th>Number of Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>18,042</td>
</tr>
<tr>
<td>10</td>
<td>21,686</td>
</tr>
<tr>
<td>15</td>
<td>n.c.</td>
</tr>
</tbody>
</table>

Table 7.49. Steady State Computation for Wireless Radio Communication Network: Number of Iterations

<table>
<thead>
<tr>
<th>N</th>
<th>Time (sec):</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.001586</td>
</tr>
<tr>
<td>10</td>
<td>0.143153</td>
</tr>
<tr>
<td>15</td>
<td>n.c.</td>
</tr>
</tbody>
</table>

Table 7.50. Steady State Computation Times for Wireless Radio Communication Network

Jacobi and JOR converge for the smallest system under investigation. All other methods do not converge after 100,000 computation steps.
Conclusions

8.1 Summary

The contributions of this thesis can be divided in two main parts:

1. Semantics of stochastic process algebras
2. Verification of stochastic process algebra models

We have picked stochastic process algebras as modelling formalism because it supports important requirements for modelling and specification languages, such as compositionality, hierarchy, and abstraction.

One problem with state based analysis methods is the state space explosion problem. This problem stems from the fact, that usually the high-level specification is mapped to its low-level semantic state-based model, which in the case of stochastic process algebras is a continuous time Markov chain (CTMC). To combat this problem clever approaches to generate and store these CTMCs are necessary. One such approach is to store the CTMCs as MTBDDs. This approach is also advocated in this thesis.

Traditionally, before a CTMC is stored as an MTBDD its explicit representation is generated. In the case of the parallel operator this naive approach can lead to state space explosion, as the memory requirements grows exponentially in the number of parallelly composed components.

We propose a compositional approach, that avoids this problem by exploiting compositionality of process algebras and by devising a semantics that defines a semantic rule for every process algebraic operator that maps its transitional behaviour directly to MTBDDs without the detour of generating an explicit CTMC representation of the process.

In the second part of the thesis we have addressed the problem of checking automatically that the modelled system satisfies the desired performance and reliability requirements.

In traditional performance analysis, measures are mostly state based, i.e. writing down measures requires knowledge about the structure of the Markov chain that represents the system model. This contradicts in some ways the high-level modelling approach of process algebras, where system behaviour is modelled more abstractly by action sequences. To avoid this problem we have devised a stochastic logic, which is based on the logic PDL, that can express complex performance and reliability requirements. This logic, SPDL, is capable of expressing requirements in both an action- and state-oriented way. This combination of both kinds of expressing performance and reliability requirements can be desirable for the modeller, we may think for example of control structures in programming...
languages such as if-then or while as being state-oriented. An example of a measure, for which such control-structures are useful might be:

If the system has a certain load factor, is the probability that within \( t \) time units \( n \) more jobs arrive, such that the system is then overloaded, greater than \( x \) percent?

In the tool CASPA we have implemented both the denotational MTBDD semantics of YAMPA and the model checking algorithms for SPDL.

8.2 Future Work

Here, we give some pointers to possible future research in the area covered by this thesis.

8.2.1 Extension of SPDL

The path operator may be extended in such a way that time bounds are no longer necessarily fixed values but can also be random values drawn from an arbitrary distribution. In [98] an approach for CSL is described, where for intervals of the form \([0, t]\) the upper bound is such a random value. One can extend this approach to intervals of the form \([t, t + T]\) or \([T, T + T']\) where \( t \neq 0 \) and the upper bound is a random value and the lower bound a fixed value or where both the upper and the lower bounds are random values.

8.2.2 Definition of Templates

It would be extremely useful to have a library of templates for often used types of properties of the rich class of liveness and safety properties. Similar ideas are realised in [49, 34], where typical patterns of behaviour like Absence, Response, Precedence are given. A simple example of such a pattern is a path starting with some specific initial action, terminating with an other specific action, and in between some arbitrary programs can be executed. The goal is, to enable the user to choose from a library a given template and adapt this to the special needs of the current system under investigation.

8.2.3 Compositional Symbolic Bisimulation

We have shown that the validity of SPDL formulae is invariant with respect to action-labelled Markov-AP-bisimulation. It is therefore desirable to do model checking on such a reduced model. On the other hand it is known that the size of BDDs that are minimal with respect to bisimulation are often larger than their non-reduced counterparts.

In this thesis we have proposed for a number of process algebraic operators heuristic algorithms that aim to keep the state space and the size of BDDs minimal at every construction step. Future work shall make the set of algorithms complete.

An other direction, inspired by the work of [9] might be as follows:

In [9] an exact procedure for incremental state space generation and reduction on a symbolic basis is described. The high-level specification method used are interacting state machines. This method aims to keep both the state space and the BDD size minimal. Reduction can be done with respect to a number of equivalence relations, for example for bisimulation.
The work to be done is to investigate the application of this approach to compositional stochastic modelling. Additionally we have to develop algorithms that compute the reachable part and the equivalence relation of the state graph on-the-fly [102, 27].

In [8] a method for generating equivalences depending on the property to be checked is presented. This method works for CTL specifications. We can investigate whether it is beneficial to apply it also to the case of SPDL properties to be checked.

For all these works we have to carefully consider the trade-off between additional computational effort to generate the reduced models and the speed-up in checking properties and the gain in the size of systems that can be modelled and analysed.

8.2.4 Influence of Variable Ordering on MTBDD Sizes

It can be noted that restricting the state space of a model to its reachable part results in an increase of the MTBDD sizes, this could be seen in extreme in chapter 7.8, but applies to a lesser extend to all models investigated in this thesis. This presumably stems from the fact that such a restriction destroys regularity, which in combination with the chosen interleaved variable order is an important factor for compact representation.

It is worthy to investigate the influence of choosing a different way to order the variables on the MTBDD sizes. Here, we have to carefully consider the trade-off between additionally computational effort to compute and evaluate different variable orders and the gain in memory space if a better order can be found.
References


Part IV

Appendices
Proofs from Chapter 3

A.1 Proof of Theorem 5

**Theorem 5:** Every transition encoded in the MTBDD will be inserted into the explicit representation generated by algorithm 3.6. Furthermore, \((S, \text{Act}, R, s^D_p)\) contains exactly those transitions that are encoded by the MTBDD from which the explicit representation was generated.

**Proof (Theorem 5).** The first claim holds, as the exploration algorithm starts at the initial state, thus every reachable state will be finally inserted in the set Unexpl. Every state in Unexpl eventually will be explored, as the set of states is finite such that finally no new state is added to Unexpl. Secondly, choose can be implemented such that every state in Unexpl is picked.

To prove the second claim we will show that the original MTBDD is equivalent to the one that can be built from the explicit SLTS generated by algorithm 3.6. The proof is an induction on the number of transitions encoded by the MTBDD under consideration:

**Induction start:** We start with zero transitions encoded in the original MTBDD. In this case the MTBDD may consist only of the 0 terminal node, representing an inactive process.

Here, the algorithm returns \((S, \text{Act}, R, s^D_p)\) with \(S = s^D_p\), \(R = \emptyset\) and Act is a global set, it is irrelevant whether it is empty or not. The MTBDD generated from this quadruple consists only of the 0 terminal node, such that both MTBDDs are obviously isomorphic.

As **induction hypothesis** (I.H.) we assume that the claim is true for the MTBDD encoding \(n\) transitions. The **induction step** for \(n + 1\) transition holds, as if the current transition is not in \(R\) it will be added and \(R\) is extended (line (16)), if it is already in \(R\), \(R\) will not be altered by the set union operation in line (16). Additionally, in line (18) the target state of the current transition is added to New, if it was not already investigated.

We will not prove formally the termination of the algorithm:

The loops **forall** \(a \in \text{Act} \) do and **forall** \(y \in \text{UC}\) iterate over finite sets, i.e. both loops will eventually terminate. I.e. the set Unexpl will be recomputed, as the set of states is finite, eventually no new states will be found, i.e. Unexpl will be reduced by one element and no new states are added, so Unexpl will finally be empty such that the condition of the outermost loop while \(\text{Unexpl} \neq \emptyset\) will be violated and the algorithm terminates. \(\Box\)

A.2 Proof of Lemma 3

**Lemma 2:** Let \(\text{Part}_Q = \{C^Q_1, \ldots, C^Q_{m_Q}\}\) \((\text{Part}_R = \{C^R_1, \ldots, C^R_{m_R}\})\) be a partition of the state space of process \(Q\) \((R)\) which corresponds to the equivalence classes of a Markovian bisimulation.
The Cartesian product $C_{i,j}^P = C_i^Q \times C_j^R$ yields a partition $\text{Part}^P$ of the state space of process $P = Q|\{L\}|R$ (with equivalence classes $\{C_{i,j}^P | i = 1, \ldots, m_Q, j = 1, \ldots, m_R \}$) which again corresponds to a Markovian bisimulation.

Proof (Lemma 3). Lemma 3 can be shown as follows: Consider states $s_1 \in C_{i,j}^P$ and $s_2 \in C_{i,j}^P$ (these states are actually tuples $s_i = (s_i^Q, s_i^R)$). We show that $s_1$ and $s_2$ are Markovian bisimilar. For every non-synchronising action $a \not\in L$ and every target class $C_{i,j}^P$, we have

$$\gamma(s_1, a, C_{i,j}^P) = \gamma(s_2, a, C_{i,j}^P) = \gamma(s_2, a, C_{i,j}^P)$$

(For non-synchronising action $a \not\in L$ and target class $C_{i,j}^P$, there is a symmetric argument.)

For every synchronising action $a \in L$ and every target class $C_{i,j}^P$, we have (we use the notation $s_i' = (s_i^Q, s_i^R)$)

$$\gamma(s_1, a, C_{i,j}^P) = \sum_{s_1' \in C_{i,j}^P} \text{rate}(s_1, a, s_1')$$

$$= \sum_{s_1'^Q \in C_i^Q, s_1'^R \in C_j^R} \text{rate}(s_1'^Q, a, s_1'^R) \cdot \text{rate}(s_1^R, a, s_1'^R)$$

$$= \sum_{s_1'^Q \in C_i^Q} \text{rate}(s_1'^Q, a, s_1'^Q) \cdot \sum_{s_1'^R \in C_j^R} \text{rate}(s_1^R, a, s_1'^R)$$

$$= \gamma(s_1^Q, a, C_{i,j}^P) \cdot \gamma(s_1^R, a, C_{i,j}^P)$$

$$= \gamma(s_2^Q, a, C_{i,j}^P) \cdot \gamma(s_2^R, a, C_{i,j}^P)$$

$$= \ldots \text{similarly backwards} \ldots$$

$$= \gamma(s_2, a, C_{i,j}^P)$$

This concludes the proof of Lemma 3. \qed

A.3 Proof of Theorem 6

Theorem 6: For any process term $P$ from the language YAMPA it holds that $\text{Tr}([P]) \sim_M \text{SOS}(P)$.

Proof (Theorem 6). We distinguish the following cases:

1. $P = \text{stop}$: This case constitutes the start of the induction. For the stop process, our MTBDD semantics generates a state encoding and the 0-MTBDD. The SLTS $\text{Tr}([\text{stop}])$ derived from this symbolic representation according to the algorithm in figure 3.6 consists of a single state with no outgoing transitions, which is isomorphic (and therefore Markovian bisimulation equivalent) with the SLTS $\text{SOS}(\text{stop})$. The case $P = X$ is similar.
2. \(P = (a, \lambda); Q\): By construction, in \(\text{Tr}([[(a, \lambda); Q]])\) there is only a single transition emanating from the initial state, namely \(s_p^P \xrightarrow{a, \lambda} s_Q^P\). Similarly, in \(\text{SOS}((a, \lambda); Q)\) there is only a single transition emanating from the initial state, namely \(s_p^{\text{SOS}} \quad s_Q^{\text{SOS}}\). Since by the induction hypothesis we have \(s_Q^{\text{SOS}} \sim_M s_Q^{\text{SOS}}\), it follows that \(s_p^{\text{SOS}} \sim_M s_p^{\text{SOS}}\) and thus \(\text{Tr}([P]) \sim_M \text{SOS}(P)\).

3. \(P = Q + R\): We need to show that \(s_p^{\text{SOS}} \sim_M s_p^{\text{SOS}}\) which means that we have to show that for all actions \(a \in \text{Act}\) and for all equivalence classes \(C \subseteq S_p^{\text{SOS}} \cup S_p^{\text{SOS}}\) we have

\[
\gamma(s_p^{\text{DS}}, a, C) = \gamma(s_p^{\text{SOS}}, a, C)
\]

(Note that the bisimulation relation \(\mathcal{B}\) is now defined on the union of the state spaces of \(\text{Tr}([Q + R])\) and \(\text{SOS}(Q + R)\), i.e. each equivalence class \(C\) is a subset of that union.)

\[
\begin{align*}
\gamma(s_p^{\text{DS}}, a, C) &= \gamma(s_p^{\text{DS}}, a^Q, C) + \gamma(s_p^{\text{DS}}, a^R, C) \\
&= \gamma(s_Q^{\text{DS}}, a^Q, C) + \gamma(s_R^{\text{DS}}, a^R, C) \\
&= \gamma(s_Q^{\text{SOS}}, a^Q, C) + \gamma(s_R^{\text{SOS}}, a^R, C) \\
&= \gamma(s_p^{\text{SOS}}, a, C)
\end{align*}
\]

Hereby \(\gamma(\text{DS}, a^Q, C)\) cumulates exactly those \(a\)-transitions which are due to an \(a\)-transition in subprocess \(Q\) and similarly for \(R\). We have used the induction hypothesis, namely that \(s_Q^{\text{DS}} \sim_M s_Q^{\text{SOS}}\) and that \(s_R^{\text{DS}} \sim_M s_R^{\text{SOS}}\).

4. \(P = Q||[L]; R\): We prove \(s_p^{\text{DS}} \sim_M s_p^{\text{SOS}}\) by showing that for every action \(a\) and for every equivalence class \(C_{P,l}^p\) (note that the equivalence relation is now defined on the union of the state spaces \(S = S_p^{\text{DS}} \cup S_p^{\text{SOS}}\) where \(S_p^{\text{DS}} = S_Q^{\text{DS}} \times S_R^{\text{DS}}\) and \(S_p^{\text{SOS}} = S_Q^{\text{SOS}} \times S_R^{\text{SOS}}\)) we have \(\gamma(s_p^{\text{DS}}, a, C_{P,l}^p) = \gamma(s_p^{\text{SOS}}, a, C_{P,l}^p)\). We need to distinguish two cases:

a) \(a \notin L\), i.e. \(a\) is a non-synchronising action.

\[
\begin{align*}
\gamma(s_p^{\text{DS}}, a, C_{P,l}^p) &= \gamma(s_p^{\text{DS}}, a^Q, C_{P,l}^p) + \gamma(s_p^{\text{DS}}, a^R, C_{P,l}^p) \\
&= \gamma(s_Q^{\text{DS}}, a^Q, C_{P,l}^p) + \gamma(s_R^{\text{DS}}, a^R, C_{P,l}^p) \\
&= \gamma(s_Q^{\text{SOS}}, a^Q, C_{P,l}^p) + \gamma(s_R^{\text{SOS}}, a^R, C_{P,l}^p) \\
&= \gamma(s_p^{\text{SOS}}, a, C_{P,l}^p)
\end{align*}
\]

Here we have used the notation \(\gamma(., a^Q, .) \quad \gamma(., a^R, .)\) to denote those transitions which are due to an \(a\)-transition of process \(Q\) (\(R\)). Note that (assuming that \(s_p^{\text{DS}} \in C_{P,l}^p\)) in this non-synchronising case, target class \(C_{P,l}^p\) is either \(C_{P,l}^p\) or \(C_{P,l}^p\) since only one partner makes a move, and thus one of the summed cumulative rates in the above equation is always equal to zero.
b) \( a \in L \), i.e. \( a \) is a synchronising action.

\[
\gamma(s^p_{D_S}, a, C_{i,j}) = \text{(def. of } \gamma) \sum_{c \in C_{i,j}^p} \text{rate}(s^p_{D_S}, a, c) \\
\text{(by constr.)} = \sum_{c \in C_{i,j}^p} \text{rate}(s^p_{D_S}, a, c^Q) \cdot \text{rate}(s^p_{D_S}, a, c^R) \\
\text{(ind. hyp.)} = \sum_{c \in C_{i,j}^Q} \text{rate}(s^{SOS}_{Q}, a, c^Q) \cdot \text{rate}(s^{SOS}_{R}, a, c^R) \\
\text{(SOS sem.)} = \sum_{c \in C_{i,j}^p} \text{rate}(s^{SOS}_{p}, a, c) \\
\text{(def. of } \gamma) = \gamma(s^{SOS}_{Q}, a, C_{i,j})
\]

*: reordering and Lemma 3

5. \( P = \text{rec}X : Q \): Here we can distinguish two cases:

a) \( Q \) does not contain a free (i.e. unbound) occurrence of process variable \( X \). Following the MTBDD-semantics we get \( Tr([P]) = Tr([Q]) \), and following the SOS-semantics we get \( SOS(P) = SOS(Q) \). From the induction hypothesis it now follows easily that \( Tr([P]) \sim_M SOS(P) \).

b) \( Q \) contains one or more free occurrences of process variable \( X \). According to the induction hypothesis, \( s^X_Q \sim_M s^{SOS}_Q \). Both \( Tr([Q]) \) and \( SOS(Q) \) contain a state which corresponds to \( X \) and in both SLTSs that state does not have any outgoing transitions, i.e. we have \( s^X_Q \sim_M s^{SOS}_X \). Transition system \( Tr([P]) \) differs from \( Tr([Q]) \) only in that transitions leading to \( s^X_Q \) are redirected to the initial state \( s^X_D \) (i.e. \( s^X_D \)). Therefore it suffices to show that for any pair of states \( s^Q_D \in S^Q_D \) and \( s^{SOS}_Q \in S^{SOS}_Q \) it holds that if \( s^X_Q \sim_M s^{SOS}_Q \) before applying the recursion operator then also \( s^D_Q \sim_M s^{SOS}_Q \) after applying recursion. We need to show that

\[ \forall a : \forall C : \gamma_{after}(s^Q_D, a, C) = \gamma_{after}(s^{SOS}_Q, a, C) \]  \((\gamma_{after} \) denotes the cumulative rates after applying the recursion operator, whereas \( \gamma_{before} \) denotes the cumulative rates before applying the recursion operator) For \( C \not\ni s^Q_D \land C \not\ni X \) we have for all actions \( a \)

\[ \gamma_{after}(s^Q_Q, a, C) = \gamma_{before}(s^Q_Q, a, C) \]

For \( C \ni X \) (i.e. \( C = \{ X \} \)) we have for all actions \( a \)

\[ \gamma_{after}(s^Q_Q, a, C) = \gamma_{after}(s^{SOS}_Q, a, C) = 0 \]
Finally, for $C \ni s^D_Q$ we have for all actions $a$

$$\gamma^\text{after}(s^D_Q, a, C) = \gamma^\text{before}(s^D_Q, a, C) + \gamma^\text{before}(s^D_Q, a, X)$$

$$= \gamma^\text{before}(s^\text{SOS}_Q, a, C) + \gamma^\text{before}(s^\text{SOS}_Q, a, X)$$

$$= \gamma^\text{after}(s^\text{SOS}_Q, a, C)$$

6. $P = \text{hide } b$ in $Q$: We can distinguish two cases:
   a) $b \notin A(Q)$, where $A(Q)$ is the set of actions that occur in $Q$.
   According to the MTBDD-semantics this case yields $\lceil \text{hide } b \rceil_Q = [Q]$ and therefore we have

$$\text{Tr}(\lceil \text{hide } b \rceil_Q) \sim_M \text{Tr}(\lceil Q \rceil)$$

Applying the SOS-semantics to this case we have

$$\text{SOS}(\text{hide } b) \sim_M \text{SOS}(Q)$$

Using the induction hypothesis $\text{SOS}(Q) \sim_M \text{Tr}(\lceil Q \rceil)$ and transitivity of $\sim_M$ we can conclude that

$$\text{Tr}(\lceil \text{hide } b \rceil_Q) \sim_M \text{SOS}(\text{hide } b)$$

b) $b \in A(Q)$: We need to show that for all actions $a$ and all equivalence classes $C$ it holds that $\gamma(s^D_p, a, C) = \gamma(s^\text{SOS}_p, a, C)$. There are three subcases:
   i. $a \neq b \land a \neq \tau$: By construction (since $a$-transitions are not affected by the hiding of action $b$) $\gamma(s^D_p, a, C) = \gamma(s^D_Q, a, C)$ and by the SOS-semantics $\gamma(s^\text{SOS}_p, a, C) = \gamma(s^\text{SOS}_Q, a, C)$ and these two values are identical by the induction hypothesis.
   ii. $a = b$: By construction (since action $b$ is no longer present in $P$) $\gamma(s^D_p, b, C) = 0$ and by the SOS-semantics $\gamma(s^\text{SOS}_p, b, C) = 0$.
   iii. $a = \tau$:

$$\gamma(s^D_p, \tau, C) = \gamma(s^D_Q, \tau, C) = \gamma^\text{after}(s^\text{SOS}_p, \tau, C)$$

7. $P = Q[a/b]$: The proof of this case follows the same lines as for hiding, we will omit the details.

8. $P = \text{exit}$: Using our non-standard interpretation of exit as a process constant with no out-going behaviour. For exit the MTBDD semantics generates a state encoding and the 0-MTBDD. The SLTS $\text{Tr}(\lceil \text{exit} \rceil)$ derived from this MTBDD consists of a single state with no outgoing behaviour. This SLTS is isomorphic to the SLTS $\text{SOS}(\text{exit})$ derived by applying our adopted SOS-rules to exit.

9. $P := Q >> R$: By I.H. we know $\text{Tr}(\lceil Q \rceil) \sim_M \text{SOS}(Q)$ and $\text{Tr}(\lceil R \rceil) \sim_M \text{SOS}(R)$.
   We have to show that the cumulative rate of all transitions leading before sequential composition to the state encoding successful termination of $Q$ is equal to the cumulative rate leading after sequential composition to the initial state of $R$ is equal in both semantics, where $s^D_Q$ is an arbitrary state of $\lceil Q \rceil$ and $s^D_R$ is $\lceil R \rceil$ initial state.
202 A Proofs from Chapter 3

\[
\gamma(u^{DS}_Q, a, s^{DS}_R) \quad \text{(by construction)} = \gamma(u^{DS}_Q, a, \text{exit})
\]

(by ind. hypothesis)

\[
\gamma(s^{SOS}_Q, a, \text{exit}) \quad \text{(by SOS semantics)} = \gamma(u^{SOS}_Q, a, s^{SOS}_R)
\]

10. \( P := Q[> R: \)

\[
\gamma(s^{DS}_P, a, C) \quad \text{(by construction)} = \gamma(s^{DS}_P, a^Q, C) + \gamma(s^{DS}_P, a^R, C)
\]

(by construction)

\[
\gamma(s^{DS}_Q, a^Q, C) + \gamma(s^{DS}_R, a^R, C)
\]

(by ind. hypothesis)

\[
\gamma(s^{SOS}_Q, a^Q, C) + \gamma(s^{SOS}_R, a^R, C)
\]

(by SOS semantics)

\[
\gamma(s^{SOS}_P, a, C)
\]

This proof is equal to the proof for choice (cf. [91]). This stems from the fact that disabling can be interpreted as iterated choice.

11. \( P = (a, \lambda); Q[| L] | R: \) As an abbreviation we use \( S \) instead of \( Q[| L] | R. \)

By construction, in \( Tr([[(a, \lambda); S]]) \) there is only a single transition emanating from the initial state, namely \( s^{DS}_P \overset{a, \lambda}{\longrightarrow} s^{DS}_S \). Similarly, in \( SOS((a, \lambda); S) \) there is only a single transition emanating from the initial state, namely \( s^{SOS}_P \overset{a, \lambda}{\longrightarrow} s^{SOS}_S \). Since by the induction hypothesis we have \( s^{DS}_S \sim_M s^{SOS}_S \), it follows that \( s^{DS}_P \sim_M s^{SOS}_P \) and thus \( Tr([P]) \sim_M SOS(P) \).

The proofs of the remaining cases for parallel composition at arbitrary levels will be omitted as they are similar to the cases already shown. \( \Box \)
B.1 Proof of Lemma 4

**Lemma 3.** Let a program $\rho$ derived by the grammar from definition 61 be given, then $L(\rho)$ is its corresponding language. For each $p \in L(\rho)$ we can find a semantically equivalent $p'$ in the sense of definition 64, by applying the following rules:

(T1) Sequences of test formulae with no atomic programs, i.e. elements from $\text{Act}$, interspersed, i.e. sequences of the kind $\Xi_1; \Xi_2; \ldots \Xi_n;$, are transformed into a conjunction of the involved test formulae:

$$\Xi_1; \Xi_2; \ldots \Xi_n; \equiv \bigwedge_{i=1}^{n} \Xi_i;$$

(T2) Atomic programs $a$, not preceded by a test formula are transformed into expressions of the kind $\text{true}; a$.

We will only prove the correctness of $T_1$, the case of $T_2$ is trivial.

**Proof (Lemma 4).** The transformation of $T_1$ is correct, with respect to the semantics of PDL (cf. definition 43):

We show the correctness for $i = 2$, the general case is an easy induction on the number of conjuncts.

$$I_p((\Xi \land \Theta); \rho) = I_p((\Xi \land \Theta);) \circ I_p(\rho) = \{(u, u) | u \in I_f((\Xi \land \Theta)) \} \circ I_p(\rho) = \{(u, u) | u \in (I_f(\Xi) \cap I_f(\Theta)) \} \circ I_p(\rho) = \{(u, u) | u \in (\text{Sat}(\Xi) \cap \text{Sat}(\Theta)) \} \circ I_p(\rho)$$

We have to show that the semantic definition for $\Xi; \Theta; \rho$ is identical to the one above:

$$I_p(\Xi; \Theta; \rho) = I_p(\Xi; \Theta) \circ I_p(\rho) = \{(u, v) | \exists w (u, w) \in I_p(\Xi; \Theta) \land (w, v) \in I_p(\Theta) \} \circ I_p(\rho)$$

For the remaining derivations we need the following equivalences:

$(u, w) \in I_p(\Xi) \iff u = w \land u \in \text{Sat}(\Xi)$

$(w, v) \in I_p(\Theta) \iff v = w \land v \in \text{Sat}(\Theta)$
Thus, it holds: $u = w \land w = v \rightarrow u = v$.
Furthermore, we have:

$$(u,u) \in I_p(\Xi) \iff u \in \text{Sat}(\Xi)$$

Using the equivalences above, we can deduce:

$${(u,u) \mid \exists w((u,w) \in I_p(\Xi) \land (w,v) \in I_p(\Theta))} \circ I_p(\rho)
= {(u,u) \mid u \in \text{Sat}(\Xi) \land u \in \text{Sat}(\Theta)} \circ I_p(\rho)
= {(u,u) \mid u \in (\text{Sat}(\Xi) \cap \text{Sat}(\Theta))} \circ I_p(\rho)$$

\[ \Box \]

### B.2 Proof of Lemma 5

**Lemma 4.** The transformation rules R1 to R6 are correct in the sense that the automata that are generated this way are equivalent with the original automata.

**Proof (Lemma 5).**

- **Correctness of rule R1:** An automaton having the form as described in rule R1 is derived from a program of the form $\Xi?; X^*; Y$. This yields the following syntactical derivations:

  $\Xi?; X^*; Y \equiv \Xi?; (e \cup X; X^*); Y$  \text{Semantics of Kleene star}

  $\Xi?; (e \cup X; X^*); Y \equiv (\Xi?; e \cup \Xi?; X; X^*); Y$  \text{Distributivity of ’;}'

  $(\Xi?; e \cup \Xi?; X; X^*); Y \equiv (\Xi? \cup e; X; X^*); Y$  $a; e \equiv a$

  $(\Xi? \cup e; X; X^*); Y \equiv (\Xi?; Y \cup e; X; X^*; Y)$  \text{Distributivity of ’;}'

- **Correctness of rule R2:** An automaton having the form as described in rule 2 is derived from a program of the form $\Xi?; e; Y$:

  $\Xi?; e; Y \equiv \Xi?; Y$

  If $Y$ is of the form $\Lambda?; b$ or $b$, we are ready. If $Y$ is of the form $\Lambda_1?; b_1 \cup ... \cup \Lambda_n?; b_n$, we proceed as follows:

  $\Xi?; (\Lambda_1?; b_1 \cup ... \cup \Lambda_n?; b_n) \equiv \Xi?; \Lambda_1?; b_1 \cup ... \cup \Xi?; \Lambda_n?; b_n$

  which is correct due to distributivity laws of sequential composition and choice in the algebra of regular expressions.

- **Correctness of rule R3:** An automaton having the form as described in rule 3 is derived from a program of the form $\Xi?; X^*$. We have the following syntactic conversions:

  $\Xi?; X^* \equiv \Xi?; (e \cup X; X^*)$  \text{Semantics of Kleene star}

  $\Xi?; (e \cup X; X^*) \equiv \Xi?; e \cup \Xi?; X; X^*$  \text{Distributivity of ’;}'

  $\Xi?; e \cup \Xi?; X; X^* \equiv \Xi?; \cup e; X; X^*$  $a; e \equiv a$
• **Correctness of rule R4:** An automaton having the form as described in rule 4 is derived from a program of the form $\Xi?; X^* \cup \Xi?; X^*; Y$.

$$
\Xi?; X^* \cup \Xi?; X^*; Y \equiv \Xi?; (\epsilon \cup X; X^*) \cup \Xi?; (\epsilon \cup X; X^*); Y
$$

$$
\Xi?; (\epsilon \cup X; X^*) \cup \Xi?; X; X^* \equiv \Xi?; X; X^* \equiv \Xi?; X; X^*; Y \equiv \Xi?; \Xi?; X; X^* \equiv \Xi?; Y \equiv \Xi?; X; X^* \equiv \Xi?; Y \equiv \Xi?; Y
$$

• **Correctness of rule R5:** See the proof of lemma 4.

• **Correctness of rule R6:** A state $z_j$ as described in rule R6 results from the sequential composition of two programs $\rho_1; \rho_2$, if $z_j$ is an absorbing end state of the automaton for $\rho_1$. As the end states of the automaton that results from the sequential composition are those of $\rho_2$, and $z_j$ is not longer an accepting state in the new automaton, transitions to this kind of state can be deleted.

\[ \blacksquare \]

### B.3 Proof of Theorem 8

**Theorem 8.** Given a program $\rho$, then for each NPA $N^\rho_p$ an equivalent DPA $A_\rho$ can be constructed.

**Proof (Theorem 8).** The proof of theorem 8 requires the following steps:

1. We have to show that $N^\rho_p \equiv N_\rho$
2. We have to show that $N_\rho \equiv A^\rho_p$
3. Finally, we have to show that $A^\rho_p \equiv A_\rho$

Steps 1 and 2 are already established, it suffices to prove the last item:

It suffices to prove this claim for two transitions $\Phi?; a$ and $\Psi?; a$ that are emanating from a single source state. For more than two ambiguous transitions, we can proceed by induction.

1. Given an automon $A^\rho_p$ the transitions we want to avoid can be translated into a program fragment of the form:

   $$
   \rho_{amb} := \Phi?; a \cup \Psi?; a
   $$

   We will give the formal semantics of $\rho_{amb}$:

   $$
   I_\rho(\Phi?; a \cup \Psi?; a) =
   I_\rho(\Phi?; a) \cup I_\rho(\Psi?; a)
   $$

   $$(I_\rho(\Phi?) \circ I_\rho(a)) \cup (I_\rho(\Psi?) \circ I_\rho(a)) =
   (((u, u) \cup u' \in I_\rho(\Phi)) \circ I_\rho(a)) \cup (((u', u') \cup u' \in I_\rho(\Psi)) \circ I_\rho(a)) =
   ((u, u) \cup u' \in I_\rho(\Phi) \cup u' \in I_\rho(\Psi)) \circ I_\rho(a) =
   \{(v, v) | v \in I_\rho(\Phi) \cup I_\rho(\Psi)) \circ I_\rho(a) =
   \{(v, v) | v \in I_\rho(\Phi \lor \Psi) \circ I_\rho(a)
   $$

2. From the discussion above we know, that $A^\rho_p$ has undesired behaviour, i.e. we must transform $A^\rho_p$ into the desired $A_\rho$ in which program fragments like $\rho_{amb}$ can not appear.
To this end, we rewrite $\rho_{\text{notamb}}$ in a way that makes it impossible that more than one successor state is possible. This yields:

$$\rho_{\text{notamb}} := (\Phi \land \Psi)?; a \cup (\neg \Phi \land \Psi)?; a \cup (\Phi \land \neg \Psi)?; a$$

Applying to $\rho_{\text{notamb}}$ the semantic definitions of PDL we obtain:

$$\mathcal{I}_p((\Phi \land \Psi)?; a \cup (\neg \Phi \land \Psi)?; a \cup (\Phi \land \neg \Psi)?; a) = \mathcal{I}_p((\Phi \land \Psi)?; a) \cup \mathcal{I}_p((\neg \Phi \land \Psi)?; a) \cup \mathcal{I}_p((\Phi \land \neg \Psi)?; a) = ... =$$

$$[((w, w) | w \in (\mathcal{I}_f((\Phi \land \Psi)) \cup \mathcal{I}_f((\neg \Phi \land \Psi)) \cup \mathcal{I}_f((\Phi \land \neg \Psi)))) \circ \mathcal{I}_p(a) =$$

$$[((w, w) | w \in (\mathcal{I}_f((\Phi \land \Psi) \lor (\neg \Phi \land \Psi) \lor (\Phi \land \neg \Psi)))) \circ \mathcal{I}_p(a)$$

Using syntactic transformations, we can show:

$$\mathcal{I}_f(\Phi \lor \Psi) \equiv \mathcal{I}_f((\Phi \land \Psi) \lor (\neg \Phi \land \Psi) \lor (\Phi \land \neg \Psi))$$

$$(\Phi \land \Psi) \lor (\neg \Phi \land \Psi) = (\Psi \land \Phi) \lor (\Psi \land \neg \Phi) = \Psi \land (\Phi \lor \neg \Phi) = \Psi$$

Now applying this result to the third disjunct:

$$\Psi \lor (\Phi \land \neg \Psi) = (\Psi \lor \Phi) \land (\Psi \lor \neg \Psi) = \Psi \lor \Phi$$

Thus, we could prove the claim that both automata are in fact equivalent. \(\square\)

### B.4 Proof of Theorem 9

**Theorem 9.** The transformation of $\mathcal{M}$ into $\mathcal{M}^x$ is correct. I.e. the probability of satisfying $\Phi[\rho][0,t] \Psi$ in $\mathcal{M}$ is equal to the probability of reaching $\chi_{\mathcal{M}^x}$ within time $t \in [0, t']$ in $\mathcal{M}^x$:

$$\Pr\{\sigma \in \text{Path}^x_{\mathcal{M}} | \mathcal{M}, \sigma \models \Phi[\rho][0,t] \Psi\} =$$

$$\Pr\{\sigma^x \in \text{Path}^x_{\mathcal{M}^x} | \exists t' \in [0, t'] : \mathcal{M}^x, \sigma^x @ t' \models \chi_{\mathcal{M}^x}\}$$

Before we can prove theorem 9 we need the following definitions:

**Definition 101 (Indicator function).** The function $\text{Ind}(\mathcal{M}, s, \phi)$ indicates, whether an arbitrary SPDL state formula $\phi$ is satisfied in a given state $s$ of a fixed model $\mathcal{M}$:

$$\text{Ind}(\mathcal{M}, s, \phi) = \begin{cases} 1 & \text{iff } \mathcal{M}, s \models \phi \\ 0 & \text{else} \end{cases}$$

Recall, that $L(z)$ is the activation set for automata states from definition 73.

**Definition 102 (End condition of a program).** Let $\rho$ be a program and $A_{\rho}$ its corresponding deterministic program automaton. The end condition of a program $\rho$ are those suffixes of form $\Phi(z); e$, where $\Phi = \text{true}$ is possible.

$$\text{Fin}_z(A) = \begin{cases} \text{true} & \text{iff } z \in E \\ \text{false} & \text{iff } z \notin E \land \forall a \in L(z) : (\delta(z, a) \notin E) \\ \Phi_1 \lor ... \lor \Phi_n & \text{iff } z \notin E \land \forall i(\Phi_i); e \in L(z) \land (\delta(z, \Phi_i); e) \in E) \end{cases}$$
Proof (Theorem 9).
We will prove theorem 9 by induction on the length of paths.

**Induction start:** $|\sigma| = |\sigma^\times| = 1$: Using the standard semantics of CSL (cf. [15]) we obtain:

$$
Pr\{\sigma^\times \in \text{Path}_{(s,z_0)}^M \mid \exists t \in I(M^\times, \sigma^\times @ t \models \chi_{M^\times})\} =
\int_0^t \sum_{(s',z') \in S^\times} \mathbf{R}((s,z_0), (s',z')) \cdot e^{-E((s,z_0)) \cdot x} \cdot \text{Ind}(M^\times, (s',z'), \chi_{M^\times}) dx
$$

As the length of the path is one, $\text{Ind}(M^\times, (s',z'), \chi_{M^\times})$ is either 1 or 0, i.e. $\chi_{M^\times}$ either holds in $(s',z')$ or does not.

For the original formula, the probability measure can be characterised as follows:

$$
Pr\{\sigma \in \text{Path}_{(s,z_0)}^M \mid M, \sigma \models \Phi[\rho]^{|0|}\Psi\} =
\int_0^t \sum_{\{\Phi[\rho]\models \Phi[\rho]\Lambda L(z) \land M, s \models \Phi\}} \sum_{s' \in S} \mathbf{R}_a(s,s') \cdot e^{-E(s) \cdot x} \cdot \text{Ind}(M, s', \Psi \land \text{Fin}_z(A)) dx
$$

Therefore we will now show that:

$$
\int_0^t \sum_{\{\Phi[\rho]\models \Phi[\rho]\Lambda L(z) \land M, s \models \Phi\}} \sum_{s' \in S} \mathbf{R}_a(s,s') \cdot e^{-E(s) \cdot x} \cdot \text{Ind}(M, s', \Psi \land \text{Fin}_z(A)) dx =
\int_0^t \sum_{s \in S} \sum_{\{\Phi[\rho]\models \Phi[\rho]\Lambda L(z) \land M, s \models \Phi\}} \mathbf{R}_a(s,s') \cdot e^{-E(s) \cdot x} \cdot \text{Ind}(M, s', \Psi \land \text{Fin}_z(A)) dx =
\int_0^t \sum_{(s',z') \in S^\times} \mathbf{R}((s,z_0), (s',z')) \cdot e^{-E((s,z_0)) \cdot x} \cdot \text{Ind}(M^\times, (s',z'), \chi_{M^\times}) dx
$$

The last equation holds, since by construction of $M^\times$ we can conclude that:

$$
\sum_{\{\Phi[\rho]\models \Phi[\rho]\Lambda L(z) \land M, s \models \Phi\}} \mathbf{R}(s,s') = \mathbf{R}((s,z_0), (s',z'))
$$

Therefore and by construction it holds that the two outer sums are equal. By construction of $M^\times$ from $M$ we conclude:

$$
\mathbf{E}(s) = \mathbf{E}((s,z_0))
$$

$\text{Ind}(M^\times, (s',z'), \chi_{M^\times}) = \text{Ind}(M, s', \Psi \land \text{Fin}_z(A))$ by construction, as those states are labelled with $\chi_{M^\times}$ in which $\text{Fin}_z(A)$ and $\Psi$ hold and in $A$ an accepting state has been reached.

**Induction step:** We assume that for paths of length $n$ the assumption holds, now we consider paths $\sigma^\times$ resp. $\sigma'$ of length $n + 1$:

Let $\sigma^\times$ resp. $\sigma'$ be paths of length $n$, where $\sigma^\times'$ is suffix of $\sigma^\times$ and $\sigma'$ is suffix of $\sigma$, then

$$
Pr\{\sigma^\times \in \text{Path}_{(s,z_0)}^M \mid M^\times, \sigma^\times @ t \models \chi_{M^\times}\} =
\int_0^t \sum_{(s',Z') \in S^\times} \mathbf{R}((s,Z_0), (s',Z')) \cdot e^{-E((s,Z_0)) \cdot x} dx
$$

$$
Pr\{\sigma^\times' \in \text{Path}_{(s,z_0)}^M \mid M^\times, \sigma^\times' @ (t - x) \models \chi_{M^\times}\}
$$
Analogously:

$$\Pr\{\sigma \in \text{Path}_{s \in S}^M \mid M, \sigma \models \Phi[\rho][0,t] \Psi\} =$$

$$\int_0^t \left( \sum_{\{s',t: \Phi[\rho][s \in L(\emptyset \wedge M) \wedge \Phi[\rho][s] \}} \Pr_0(s, s') \cdot e^{-E(s) \cdot x} \right) \cdot \Pr\{\sigma' \in \text{Path}_{s \in S}^M \mid M, \sigma' \models \Phi[\rho][0,1-x] \Psi\}$$

where $\rho'$ is the suffix of $\rho$. Using I.H. and the induction start we conclude that the theorem holds, i.e.

$$\Pr\{\sigma^x \in \text{Path}_{(s, z_0)}^M \mid M^x, \sigma^x @ t \models \chi_M^x \} = \Pr\{\sigma \in \text{Path}_{s \in S}^M \mid M, \sigma \models \Phi[\rho][0,t] \Psi\}$$

\[\blacksquare\]

### B.5 Proof of Theorem 10

**Theorem 10.** Let $B$ be an $a$-I Markov-AP-Bisimulation, $s \in M$, then:

1. $\forall \Phi(M, s \models \Phi \iff M \models B, [s] \models \Phi)$
2. $\forall \Phi(\Pr^M(s, \phi) = \Pr^{M/\overline{B}}([s], \phi))$

Let $\phi = \Phi[\rho][0,t] \Psi$ and $\rho$ be an SPDL-program. For the proof of theorem 10 we need the following auxiliary definition:

**Definition 103.** We define $P(t, s, z, n, \phi)$ as the path $\sigma$ of minimal length $n$, that satisfies path formula $\phi := \Phi[\rho][t] \Psi$ within $t$ time units, given the program automaton is initially in state $z$ and the Markov chain is in state $s$.

It holds that $\{\sigma \in \text{PATH}(s) \mid [s] \models \phi\} = \bigcup_{n \geq 0} P(t, s, z, n, \phi)$ and therefore

$$\Pr\{\sigma \in \text{PATH}(s) \mid [s] \models \phi\} = \sum_{n=0}^{\infty} \Pr\{\sigma \in P(t, s, z, n, \phi)\}$$

**Proof (Theorem 10).** We start with formulae of length one, i.e. with atomic formulae. Let the states $s$ and $t$ be in $B$:

1. Let $\Phi \in \text{AP}$, i.e. $\Phi = q$: Using the prerequisites it holds: $(s, t) \in B$. Following the definition of $B$ we can conclude: $M, s \models q \iff M, t \models q$.

This case serves as induction start. As induction hypothesis we assume that the proposition holds for formulae of length $< k$.

2. Let $\Phi = \neg \Psi$. Following I.H. it is true that $M, s \models \Psi \iff M, t \models \Psi$, we are able to prove:

$$M, s \models \neg \Psi \iff M, s \not\models \Psi \iff M, t \not\models \Psi \iff M, t \models \neg \Psi$$

3. Let $\Phi = \Psi \lor \Xi$: Following I.H. gilt $M, s \models \Psi \iff M, t \models \Psi$ and $M, s \models \Xi \iff M, t \models \Xi$. Thus, we have as well:

$$M, s \models \Psi \lor \Xi \iff M, s \models \Psi \text{ or } M, s \models \Xi \iff M, t \models \Psi \lor \Xi$$
4. Let $\Phi = \mathcal{P}_{\text{eq}}(\Delta[p])$. For the proof of this case we have to demonstrate that the probability measures of the paths satisfying path formula $\phi$ and emanating from $s$ and $u$ are identical. This requires an induction over the length $n$ of the paths.
   a) Let the length $n = 0$: The probability measure of a path satisfying $\phi$ is either zero or one.
      i. Let the measure be one: This is the case, iff $M, s \models \Psi \land z_\rho \in E$ or $M, s \models \Psi \land M, s \models \Xi \land \rho = \Xi ? \land \delta(z_\rho, \Xi ?) \in E$. Using the outer I.H. these assumptions are also valid for state $u$.
      ii. Let the measure be zero: This is the case iff $M, s \not\models \Psi \lor M, s \not\models \Psi \land M, s \not\models \Xi \land \rho = \Xi ? \land z_\rho \not\in E$. Also in this case we can apply the outer induction hypothesis to exchange $s$ and $u$.
   b) Let $n \neq 0$: We assume that the claim holds for paths of length $< n$. Using definition 103, it suffices to show that:

   $$ Pr(\sigma \in P(t, s, z, n, \phi)) = Pr(\sigma \in P(t, u, z, n, \phi)) $$

   Let $n \neq 0$: We assume the assumption holds for paths of length $< n$.

   $$ Pr(\sigma \in P(t, s, z, n, \phi)) = \\
   \int_0^T e^{E(s) \cdot x} \sum_{a \in L(z) s' \in S} R_d(s, s') \cdot Pr(\sigma \in P(t - x, s', \delta(z, a), n - 1, \phi)) dx = \\
   \int_0^T e^{E(s) \cdot x} \sum_{a \in L(z) C \in M_B / s' \in C} \sum_{s' \in S} R_d(s, s') \cdot Pr(\sigma \in P(t - x, s', \delta(z, a), n - 1, \phi)) dx = \\
   \int_0^T e^{E(u) \cdot x} \sum_{a \in L(z) C \in M_B / s' \in C} \sum_{s' \in S} R_d(u, s') \cdot Pr(\sigma \in P(t - x, s', \delta(z, a), n - 1, \phi)) dx = \\
   \int_0^T e^{E(u) \cdot x} \sum_{a \in L(z) s' \in S} \sum_{s' \in S} R_d(u, s') \cdot Pr(\sigma \in P(t - x, s', \delta(z, a), n - 1, \phi)) dx = \\
   Pr(\sigma \in P(t, u, z, n, \phi)) $$

   $\equiv$ this is true due to the prerequisite that $(s, u) \in B$ and the resulting fact that if $s$ and $u$ are in $B$ they have the same cumulative rates for every target equivalence class $C$.

5. Let $\Phi = S_{\text{eq}}(\Psi)$. It remains to show that $\pi(s, \text{Sat}(\Psi)) = \pi(t, \text{Sat}(\Psi))$:

   $$ \pi(s, \text{Sat}(\Psi)) = \\
   \sum_{s' \in \text{Sat}(\Psi)} \pi(s, \{s'\}) = \sum_{c \in M_B} \sum_{s' \in C} \pi(s, \{s'\}) = \\
   \sum_{c \in M_B} \pi([s], C) = \sum_{c \in M_B} \pi([t], C) = \\
   \sum_{c \in M_B} \sum_{s' \in C} \pi(t, \{s'\}) = \sum_{s' \in \text{Sat}(\Psi)} \pi(t, \{s'\}) = \\
   \pi(t, \text{Sat}(\Psi)) $$

$\blacksquare$
B.6 Proof of Theorem 11

**Theorem 11.** For an action- and state labelled Markov chain $\mathcal{M}$ and an SPDL formula $\Phi = \mathcal{P}_{\text{sp}}(\phi)$, the worst case space complexity of the model checking procedure lies in:

$$O(n + 2^n + 2^m + (2^{2^n} \cdot N)) = O(2^{2^n} \cdot N)$$

$n$ is the number of states of $N^\rho$, $2^n$ is the state space size of $A^\rho$, $2^m$ is the number of states of $A^\rho$, and finally $2^{2^n} \cdot N$ is the size of the product Markov chain $\mathcal{M}^\times$, where $N$ is the number of states of the model $\mathcal{M}$.

**Proof (Theorem 11).** The following steps of the model checking procedure of probabilistic SPDL path formulae contribute to the space complexity:

- Derivation of an NPA $N^\rho$ from a given program $\rho$, which has space complexity of $O(n)$. According to [3] from a program $\rho$, an NPA can be derived, whose state space size is bounded by $n = 2 \cdot |\rho|$.
- Determinisation of NPA $N^\rho$ to obtain $A^\rho$, the determinisation of an NPA requires space in $O(2^n)$.
- Elimination of ambiguous tests from $A^\rho$, which yields an additional space requirement of $O(2^{2^n})$.
- Construction of product Markov chain from the model $\mathcal{M}$ and $A^\rho$, requires $O(2^{2^n} \cdot N)$ space.

The derivation of all these factors can be found in any standard textbooks on automata theory, like [3, 79].

B.7 Proof of Theorem 12

**Theorem 12.** For an action- and state labelled Markov chain $\mathcal{M}$ and an SPDL formula $\Phi = \mathcal{P}_{\text{sp}}(\phi)$, the worst case time complexity of the model checking procedure lies in:

$$O(n + (n \cdot m) + 2^n + 2^{2^n} + |\Phi| \cdot ((2^{2^n} \cdot N)^2 \cdot q \cdot t_{\text{max}} + (2^{2^n} \cdot N)^{2.81}))$$

$$= O(|\Phi| \cdot ((2^{2^n} \cdot N)^2 \cdot q \cdot t_{\text{max}} + (2^{2^n} \cdot N)^{2.81}))$$

where $n$ is the number of states of $N^\rho$, $m$ is the number of transitions of $N^\rho$, $2^n$ is the state space size of $A^\rho$, $2^{2^n}$ is the number of states of $A^\rho$ and finally $2^{2^n} \cdot N$ is the size of the product Markov chain $\mathcal{M}^\times$, with $N$ the number of states of the model $\mathcal{M}$, $m$ is the number of transitions of $N^\rho$, $\Phi = \mathcal{P}_{\text{sp}}(\phi)$, $|\Phi|$ is the length of $\Phi$, $t_{\text{max}}$ is the maximum time bound in the path formulae occurring as subformulae of $\Phi$, and $q$ is the maximum transition rate in $\mathcal{M}$.

**Proof (Theorem 12).** To determine the worst case time complexity of the model checking procedure for probabilistic path formulae we have to take the following steps into account:

- The derivation of an NPA $N^\rho$ from a given program $\rho$, requires $O(n)$ time.
- The derivation of $N^\rho$ from $N^\rho$ by eliminating transitions labelled with $\Xi$ is linear in the number of transitions and states and thus requires $O(n \cdot m)$ time.
- Determinisation of NPA $N^\rho$ to obtain $A^\rho$ requires $O(2^n)$ time.
- The elimination of ambiguous tests from $A^\rho$, requires $O(2^{2^n})$ time, which leads to automaton $A^\rho$.
• Construction of product Markov chain from the model $\mathcal{M}$ and $A_\rho$ is linear in the number of states of $\mathcal{M}$ and $A_\rho$, thus it needs $O(2^{|\mathcal{M}|} \cdot N)$ time.

• According to [15], model Checking of CSL formula $\mathcal{P}_{scp}(F[0,t] X M_{\times})$ on transformed model $M_{\times}$ requires $O(|\Phi|((2^{2n} \cdot N)^2 \cdot q \cdot t_{max}) + (2^{2n} \cdot N)^{2.81})$ time.
Part V

German Part
Kurzfassung

Im Bereich der Leistungs- und Zuverlässigkeitsanalyse haben sich abstrakte Spezifikations-
Sprachen wie stochastische Prozessalgebren (SPA) oder verallgemeinerte stochastische Pe-
trinetze (engl. generalised stochastic Petri nets, GSPN) als sehr hilfreich erwiesen. Sollen
Spezifikationen, die mit solchen Sprachen erzeugt wurden, analysiert werden, können
zwei Hauptprobleme identifiziert werden:

1. Zustandsraumexplosion: Dieses Problem entsteht bei der Erzeugung und Abspeiche-
runge der semantischen Modelle komplexer Systeme, auf deren Basis die Analyse der
Spezifikation stattfinden soll. Die semantischen Modelle sind eine Art zeitkontinuier-
licher Markovkette oder eines stochastischen beschrifteten Transitionssystems (engl.
stochastic labelled transitions systems, SLTS).

2. Die Spezifikation und automatische Überprüfung komplexer Systeme

Das Zustandsraumexplosionsproblem tritt am häufigsten bei der Analyse nebenläufiger
Systeme auf. Dies ist die Art von Systemen, für deren Analyse wir in dieser Arbeit Metho-
den entwickeln wollen. Werden SPA als Spezifikationssprache eingesetzt, so ist das Ge-
samtsystem aus vielen kleineren parallelen Teilprozessen zusammengesetzt. Der Gesamt-
zustandsraum der Spezifikation ist dabei das Cartesische Produkt der Zustandsräume der
beteiligten Prozesse. Der dadurch entstehende hohe Speicherbedarf verhindert häufig die
Analyse komplexer Systeme. Um dieses Problem zu beseitigen, sind verschiedene Maß-
nahmen zu ergreifen. Erstens müssen Datenstrukturen, die die kompakte Speicherung
großer Zustandsräume ermöglichen eingesetzt werden. Binäre Entscheidungsdiagramme
mit mehrfachen Endknoten (engl. multi-terminal binary decision diagrams, MTBDD), ha-
ben sich bei der Speicherung von großen SLTSs als vorteilhaft erwiesen. Zweitens muß
während der Erzeugung der SLTSs aus der gegebenen SPA-Spezifikation die Tatsache be-
rücksichtigt werden, daß solche Spezifikationen oft aus mehreren kleineren Subprozeß-
spezifikationen zusammengesetzt sind. Die traditionelle Semantik von Prozessalgebren
unterstützt diese Tatsache nur sehr eingeschränkt. Ein semantischer Ansatz, der den Ge-
samtzustandsraum auf diese Weise generiert, wird auch als monolithisch bezeichnet. Ein
solcher Ansatz führt zu exponentiellem Speicherbedarf für das abgeleitete SLTS. Um dies
zu vermeiden, entwickeln wir in dieser Arbeit eine neue Semantik, die die Struktur ei-
er gegebenen SPA-Spezifikation bei der Erzeugung des zugehörigen SLTS ausnutzt. Für
jeden möglichen Teilprozeß einer gegebenen Spezifikation wird die MTBDD-basierte Dar-
stellung seines zugehörigen SLTS gewonnen. Gemäß der syntaktischen Struktur des Ge-
samtprozesses wird dann aus den Teil-MTBDDs das MTBDD des SLTS des Gesamtpro-
zesses gewonnen. Es kann gezeigt werden, daß ein semantischer Ansatz, der die kom-
positionelle Natur von Prozessalgebren ausnutzt, zur Darstellung des Zustandsraum nur
einen linearen Speicherbedarf benötigt, wohingegen die traditionelle SPA Semantik einen
exponentiellen Speicherbedarf bedeutet.
Im zweiten Teil dieser Arbeit behandeln wir das Problem der automatischen Überprüfung von Leistungs- und Zuverlässigkeitsmaßen. Hier können wir drei Teilprobleme identifizieren:

1. Komplexität der Maße
2. Abstraktion
3. Effiziente numerische Analyse

Im allgemeinen können mit den Mitteln der traditionellen Leistungs- und Zuverlässigkeitsbewertung nur recht einfache Maße beschrieben werden, z.B.

- Mittlere Anzahl von Aufträgen in einem System,
- die Wahrscheinlichkeit, daß das System eine bestimmte Zeit betriebsbereit ist,
- mittlere Reparaturdauern, mittlere Lebenszeit.

Um diesem Umstand abzuhelfen, sind in den letzten Jahren Anstrengungen unternommen worden, die aus dem Bereich der funktionalen Verifikation von Systemeigenschaften bekannten Methoden des sog. Model Checking auf den Bereich der Leistungs- und Zuverlässigkeitsbewertung zu übertragen (Model Checking stochastischer Systeme). Dieses Verfahren soll in dieser Arbeit weiterentwickelt werden, was am besten im Zusammenhang mit dem Problem der Abstraktion erklärt werden kann. Das Problem der Abstraktion bezieht sich auf die Tatsache, daß sowohl bei traditioneller Leistungsbewertung als auch beim Model Checking stochastischer Systeme die definierbaren Maße stark auf die Zustände des SLTS abzustimmen waren. D.h. um sinnvolle Maße definieren zu können, waren Kenntnisse über die Struktur des Zustandsraumes notwendig. Dies widerspricht der Verwendung von abstrakten Modellierungsmethoden im Allgemeinen und der Verwendung von SPAs als Modellierungssprachen im Besonderen. Im Allgemeinen soll die Verwendung von SPAs als Modellierungssprachen den Kenntnissen der konkreten semantischen Modelle, auf die die Spezifikation abgebildet wird, unnötig machen. Im Besonderen bedeutet die Verwendung von SPAs als Modellierungssprache aber auch noch, daß das Systemverhalten als Folge von Aktionen spezifiziert werden kann, die Kenntnisse des der Spezifikation zugrundeliegenden Zustandsraumes bedeutet also nicht nur eine Verletzung des Informatikprinzips der Abstraktion, sondern auch eine Verschiebung der Sichtweise von aktions- zu zustandsorientiert. Um diese beiden Probleme, Verletzung des Abstraktionsprinzips und Verschiebung der Sichtweisen zu verhindern, haben wir in dieser Arbeit eine neue Logik entwickelt. Diese Logik ist eine stochastische Erweiterung der Logik PDL (propositional dynamic logic), genannt SPDL. Mit dieser Logik ist es möglich, komplexe Leistungs- und Zuverlässigkeitsmaße auszudrücken und automatisch überprüfen zu lassen. SPDL erlaubt es, diese Maße in Form von erweiterten regulären Ausdrücken zu spezifizieren, d.h. die aktionsorientierte und abstrakte Sichtweise auf ein Modell kann erhalten bleiben. Die genannten Erweiterungen erlauben es dem Modellierer bekannte programmiersprachliche Konstrukte wie z.B. if-then oder while einzusetzen. Auf diese Weise ist es möglich, auch sehr komplexe Systemeigenschaften zu spezifizieren und zu verifizieren.

Um Leistungs- und Zuverlässigkeitsmaße berechnen zu können, müssen die Datenstrukturen, die für die Darstellung des SLTS eingesetzt werden, effiziente numerische Berechnungen erlauben. Zu diesem Zweck setzen wir die in [121] beschriebenen MTBDD-basierten numerischen Algorithmen ein.

Im Werkzeug CASPA haben wir sowohl die neue Semantik als auch die Model Checking-Algorithmen für die Logik SPDL implementiert und demonstrieren deren Effizienz anhand zahlreicher Fallstudien.
## Inhaltsverzeichnis

### Teil I  Einführung und Grundlagen

1  Einführung ................................................................. 1  
   1.1  Motivation ............................................................. 1  
       1.1.1  Wahl einer geeigneten Spezifikationssprache .................. 3  
       1.1.2  Modellierung komplexer Systeme ............................. 4  
       1.1.3  Spezifikation qualitativer und quantitativer Maße ............ 6  
   1.2  Beiträge der Arbeit ................................................ 9  
       1.2.1  Kompositionelle symbolische Semantik ........................ 10  
       1.2.2  Automatische Verifikation stochastischer Systeme .......... 11  
   1.3  Verwandte Arbeiten ................................................. 12  
       1.3.1  Symbolische Semantiken für Prozessalgebren ................ 13  
       1.3.2  Zeitlogiken .................................................. 13  
   1.4  Aufbau der Arbeit .................................................. 14  

2  Grundlagen ............................................................... 17  
   2.1  Zeit-kontinuierliche Markovketten ................................ 17  
       2.1.1  Transiente Analyse .......................................... 19  
       2.1.2  Die Uniformisierungsmethode ................................ 19  
       2.1.3  Stationäre Analyse .......................................... 21  
   2.2  Boolesche und Pseudo-Boolesche Funktionen ........................ 22  
       2.2.1  Boolesche Algebra ........................................... 22  
       2.2.2  Boolesche Formeln ........................................... 24  
       2.2.3  Schaltformeln ................................................ 24  
   2.3  Binäre Entscheidungsdiagramme ................................... 25  
       2.3.1  Motivation .................................................... 25  
       2.3.2  Darstellung Boolescher Formeln .............................. 25  
       2.3.3  Darstellung Boolescher Formeln als Graphen - BDDs ........ 27  
       2.3.4  Grundlegende BDD Algorithmen ................................ 33  
       2.3.5  BDD Erweiterungen zur Darstellung Pseudo-Boolescher Funktionen 39  
   2.4  Prozessalgebren ...................................................... 43
# Inhalt (pagetop)


<table>
<thead>
<tr>
<th>Kapitel</th>
<th>Unterkapitel</th>
<th>Seitenzahl</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4.1</td>
<td>Syntax, Semantik und Bisimulation für LOTOS</td>
<td>44</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Syntax, Semantik und Bisimulation für YAMP A</td>
<td>49</td>
</tr>
<tr>
<td>2.5</td>
<td>Die Logik PDL</td>
<td>52</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Syntax von PDL</td>
<td>52</td>
</tr>
<tr>
<td>2.5.2</td>
<td>Semantik von PDL</td>
<td>53</td>
</tr>
<tr>
<td>2.6</td>
<td>Die Logik CTL</td>
<td>55</td>
</tr>
<tr>
<td>2.6.1</td>
<td>Einführung</td>
<td>55</td>
</tr>
<tr>
<td>2.6.2</td>
<td>Syntax und Semantik von CTL</td>
<td>55</td>
</tr>
<tr>
<td>2.7</td>
<td>Die Logik CSL</td>
<td>56</td>
</tr>
<tr>
<td>2.7.1</td>
<td>Zustandsbeschriftete zeitkontinuierliche Markovketten</td>
<td>56</td>
</tr>
<tr>
<td>2.7.2</td>
<td>Syntax und Semantik von CSL</td>
<td>58</td>
</tr>
<tr>
<td>2.7.3</td>
<td>Model Checking von CSL</td>
<td>60</td>
</tr>
<tr>
<td>2.7.4</td>
<td>Markov-AP-Bisimulation</td>
<td>62</td>
</tr>
</tbody>
</table>

## Teil II - Semantik und Verifikation von Stochastischen Prozessalgebren

<table>
<thead>
<tr>
<th>Kapitel</th>
<th>Unterkapitel</th>
<th>Seitenzahl</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Symbolische Semantik stochastischer Prozessalgebren</td>
<td>65</td>
</tr>
<tr>
<td>3.1</td>
<td>Einführung</td>
<td>65</td>
</tr>
<tr>
<td>3.2</td>
<td>Symbolische denotationelle Semantik für YAMP A</td>
<td>66</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Allgemeine Idee</td>
<td>67</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Semantische Regeln</td>
<td>68</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Parallele Komposition auf beliebigen Ebenen</td>
<td>73</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Beispiel</td>
<td>75</td>
</tr>
<tr>
<td>3.3</td>
<td>Korrektheit der Semantik</td>
<td>77</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Beweisidee</td>
<td>77</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Konstruktion des Transitionssystems</td>
<td>78</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Korrektheitsbeweis der Semantik</td>
<td>79</td>
</tr>
<tr>
<td>3.4</td>
<td>Minimale symbolische Semantik für YAMP A</td>
<td>79</td>
</tr>
</tbody>
</table>

## Kapitel 4 - Stochastic Propositional Dynamic Logic

<table>
<thead>
<tr>
<th>Unterkapitel</th>
<th>Seitenzahl</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Einführung</td>
<td>83</td>
</tr>
<tr>
<td>4.2 Allgemeine Idee</td>
<td>84</td>
</tr>
<tr>
<td>4.3 Stochastisches PDL: Syntax, Semantik und Automaten</td>
<td>85</td>
</tr>
<tr>
<td>4.3.1 Aktions- und zustandsbeschriftete zeitkontinuierliche Markovketten</td>
<td>85</td>
</tr>
<tr>
<td>4.3.2 Syntax von SPDL</td>
<td>86</td>
</tr>
<tr>
<td>4.3.3 Programme und Automaten</td>
<td>87</td>
</tr>
<tr>
<td>4.3.4 Semantik von SPDL</td>
<td>90</td>
</tr>
<tr>
<td>4.3.5 Abgeleitete Operatoren</td>
<td>92</td>
</tr>
<tr>
<td>4.4 Automatenkonstruktion</td>
<td>92</td>
</tr>
<tr>
<td>4.4.1 Erzeugung des NPA $N_p$ aus $N'_p$</td>
<td>93</td>
</tr>
<tr>
<td>4.4.2 Determinisierung von $N_p$</td>
<td>94</td>
</tr>
<tr>
<td>4.4.3 Die Beziehung zwischen $NPA$ und $DPA$ für SPDL Programme</td>
<td>98</td>
</tr>
<tr>
<td>4.5 Model Checking von SPDL</td>
<td>99</td>
</tr>
<tr>
<td>4.5.1 Model Checking von Probabilistischen Pfadformeln durch Lösen von Integralgleichungssystemen</td>
<td>100</td>
</tr>
<tr>
<td>4.5.2 Model Checking von Probabilistischen Pfadformeln durch Transiente Analyse</td>
<td>101</td>
</tr>
</tbody>
</table>
Inhaltsverzeichnis

4.5.3 Korrektheit der Modelltransformation .......................... 104
4.6 Beispiel: Systemmodell und Anforderungen ...................... 104
4.6.1 Das Systemmodell ....................................... 104
4.6.2 Leistungsanforderungen .................................... 105
4.6.3 Konstruktion der Produktmarkovkette ....................... 105
4.7 Bisimulation und Worst-Case-Komplexität ....................... 106
4.7.1 SPDL und Bisimulation .................................... 106
4.7.2 Komplexitätsanalyse ...................................... 108
4.8 Erweiterungen der Logik SPDL .................................. 108
4.8.1 Erweiterung von SPDL um echte Zeitintervalle ............... 109
4.8.2 Model Checking stochastischer Systeme mit zeitlosen Transitionen ........................................... 110

5 Vergleich der Ausdrucksstärke stochastischer Zeitlogiken ..... 117
5.1 Überblick über stochastische Zeitlogiken ....................... 117
5.1.1 Die Logik aCSL ........................................... 117
5.1.2 Die Logik aCSL+ .......................................... 118
5.1.3 Vereinfachte Syntax und Semantik von SPDL .................. 120
5.2 Vergleich der Ausdrucksstärke ................................... 121
5.2.1 CSL¹ vs. SPDL ........................................... 121
5.2.2 aCSL vs. SPDL ............................................ 125
5.2.3 aCSL+ vs. SPDL .......................................... 128
5.2.4 aCSL vs. CSL ............................................. 130
5.3 Erweiterungen stochastischer Zeitlogiken ....................... 132
5.3.1 Erweiterung von aCSL um atomare Aussagen ................ 132
5.3.2 Vergleich von SPDL mit Zeitintervallen [t, t'] mit CSL .......... 134

Teil III Implementierung und Fallstudien

6 CASPA: Ein Werkzeug für symbolische Leistungsbewertung und stochastisches Model Checking .................................. 139
6.1 Werkzeugarchitektur ........................................... 139
6.2 CASPAs Spezifikationssprache .................................. 140
   6.2.1 Die Systemspezifikationssprache .......................... 141
6.3 Anforderungsspezifikation: Definition und Datenstrukturen .......... 142
   6.3.1 Zustandsmaße .......................................... 142
   6.3.2 Mittelwertmaße ......................................... 143
   6.3.3 Durchsatz .............................................. 144
   6.3.4 SPDL-Formeln .......................................... 144
6.4 Leistungsmaße und Prozessinstanzen .............................. 145
6.5 Die Zustandsraumverwaltung .................................... 145
6.6 Symbolische Produktmarkovkettenenerzeugung für probabilistische SPDL-Pfadformeln ........................................... 148
6.7 Numerische Analyse ........................................... 150
   6.7.1 Methoden der numerischen Analyse ....................... 150
   6.7.2 Hybride Matrix-Vektor-Multiplikation ..................... 151
   6.7.3 Offset-beschriftete MTBDDs .............................. 151
   6.7.4 Hybride MTBDDs ........................................ 152
   6.7.5 Pseudo-Gauss-Seidel ..................................... 152
## Inhaltsverzeichnis

### 7 Fallstudien

<table>
<thead>
<tr>
<th>Fallstudie</th>
<th>Seite</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Einführung</td>
<td>155</td>
</tr>
<tr>
<td>7.1.1 Modellstatistik und Erzeugungszeiten</td>
<td>155</td>
</tr>
<tr>
<td>7.1.2 Berechnung stationärer Zustandswahrscheinlichkeiten</td>
<td>155</td>
</tr>
<tr>
<td>7.1.3 Modell Checking von probabilistischen SPDL Zustandsformeln</td>
<td>156</td>
</tr>
<tr>
<td>7.2 M/M/1/N-Warteschlange</td>
<td>156</td>
</tr>
<tr>
<td>7.2.1 Einführung</td>
<td>156</td>
</tr>
<tr>
<td>7.2.2 Modellstatistik</td>
<td>157</td>
</tr>
<tr>
<td>7.2.3 Modell Checking von probabilistischen SPDL Zustandsformeln</td>
<td>157</td>
</tr>
<tr>
<td>7.2.4 Berechnung stationärer Zustandswahrscheinlichkeiten</td>
<td>157</td>
</tr>
<tr>
<td>7.3 Puffer mit fehlerhaften Ankünfnit</td>
<td>158</td>
</tr>
<tr>
<td>7.3.1 Modellstatistik</td>
<td>158</td>
</tr>
<tr>
<td>7.3.2 Model Checking von probabilistischen SPDL Zustandsformeln</td>
<td>158</td>
</tr>
<tr>
<td>7.4 Tandem-Warteschlange</td>
<td>162</td>
</tr>
<tr>
<td>7.4.1 Einführung</td>
<td>162</td>
</tr>
<tr>
<td>7.4.2 Modellstatistik</td>
<td>162</td>
</tr>
<tr>
<td>7.4.3 Modell Checking von probabilistischen SPDL Zustandsformeln</td>
<td>162</td>
</tr>
<tr>
<td>7.4.4 Vergleich mit PRISM</td>
<td>163</td>
</tr>
<tr>
<td>7.4.5 Berechnung stationärer Zustandswahrscheinlichkeiten</td>
<td>164</td>
</tr>
<tr>
<td>7.4.6 Model Checking von probabilistischen SPDL Zustandsformeln</td>
<td>164</td>
</tr>
<tr>
<td>7.5 Polling</td>
<td>165</td>
</tr>
<tr>
<td>7.5.1 Einführung</td>
<td>165</td>
</tr>
<tr>
<td>7.5.2 Modellstatistik</td>
<td>166</td>
</tr>
<tr>
<td>7.5.3 Modell Checking von probabilistischen SPDL Zustandsformeln</td>
<td>166</td>
</tr>
<tr>
<td>7.5.4 Vergleich mit PRISM</td>
<td>166</td>
</tr>
<tr>
<td>7.5.5 Berechnung stationärer Zustandswahrscheinlichkeiten</td>
<td>167</td>
</tr>
<tr>
<td>7.6 Kanban-System</td>
<td>167</td>
</tr>
<tr>
<td>7.6.1 Einführung</td>
<td>167</td>
</tr>
<tr>
<td>7.6.2 Modellstatistik</td>
<td>168</td>
</tr>
<tr>
<td>7.6.3 Modell Checking von probabilistischen SPDL Zustandsformeln</td>
<td>168</td>
</tr>
<tr>
<td>7.6.4 Vergleich mit PRISM</td>
<td>169</td>
</tr>
<tr>
<td>7.6.5 Berechnung stationärer Zustandswahrscheinlichkeiten</td>
<td>169</td>
</tr>
<tr>
<td>7.6.6 Model Checking von probabilistischen SPDL Zustandsformeln</td>
<td>170</td>
</tr>
<tr>
<td>7.7 Flexibles Fertigungssystem</td>
<td>171</td>
</tr>
<tr>
<td>7.7.1 Einführung</td>
<td>171</td>
</tr>
<tr>
<td>7.7.2 Modellstatistik</td>
<td>171</td>
</tr>
<tr>
<td>7.7.3 Modell Checking von probabilistischen SPDL Zustandsformeln</td>
<td>171</td>
</tr>
<tr>
<td>7.7.4 Vergleich mit PRISM</td>
<td>172</td>
</tr>
<tr>
<td>7.7.5 Berechnung stationärer Zustandswahrscheinlichkeiten</td>
<td>172</td>
</tr>
<tr>
<td>7.8 Mainframe mit Softwarefehlern</td>
<td>173</td>
</tr>
<tr>
<td>7.8.1 Einführung</td>
<td>173</td>
</tr>
<tr>
<td>7.8.2 Modellstatistik</td>
<td>174</td>
</tr>
<tr>
<td>7.8.3 Modell Checking von probabilistischen SPDL Zustandsformeln</td>
<td>174</td>
</tr>
<tr>
<td>7.8.4 Berechnung stationärer Zustandswahrscheinlichkeiten</td>
<td>174</td>
</tr>
<tr>
<td>7.9 Fehlertolerantes Mehrrechnersystem</td>
<td>176</td>
</tr>
<tr>
<td>7.9.1 Einführung</td>
<td>176</td>
</tr>
<tr>
<td>7.9.2 Modellstatistik</td>
<td>177</td>
</tr>
<tr>
<td>7.9.3 Modellerzeugungszeiten</td>
<td>177</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----</td>
</tr>
<tr>
<td>7.9.4 Berechnung transierter Zustandswahrscheinlichkeiten</td>
<td>178</td>
</tr>
<tr>
<td>7.9.5 Model Checking von probabilistischen SPDL Zustandsformeln</td>
<td>178</td>
</tr>
<tr>
<td>7.10 Handover-Prozedur für ein zelluläres Mobilfunknetz</td>
<td>180</td>
</tr>
<tr>
<td>7.10.1 Einführung</td>
<td>180</td>
</tr>
<tr>
<td>7.10.2 Modellstatistik</td>
<td>180</td>
</tr>
<tr>
<td>7.10.3 Modellerzeugungszeiten</td>
<td>181</td>
</tr>
<tr>
<td>7.10.4 Berechnung stationärer Zustandswahrscheinlichkeiten</td>
<td>181</td>
</tr>
<tr>
<td>8 Schlußfolgerungen</td>
<td>183</td>
</tr>
<tr>
<td>8.1 Zusammenfassung</td>
<td>183</td>
</tr>
<tr>
<td>8.2 Zukünftige Forschung</td>
<td>185</td>
</tr>
<tr>
<td>8.2.1 Erweiterungen von SPDL</td>
<td>185</td>
</tr>
<tr>
<td>8.2.2 Definition von Mustern (Templates)</td>
<td>185</td>
</tr>
<tr>
<td>8.2.3 Kompositionelle symbolische Bisimulation</td>
<td>185</td>
</tr>
<tr>
<td>8.2.4 Einfluß der Variablenordnung auf BDD-Größen</td>
<td>186</td>
</tr>
<tr>
<td>Bibliographie</td>
<td></td>
</tr>
<tr>
<td>Teil IV Anhänge</td>
<td></td>
</tr>
<tr>
<td>A Beweise aus Kapitel 3</td>
<td>197</td>
</tr>
<tr>
<td>A.1 Beweis von Theorem 5</td>
<td>197</td>
</tr>
<tr>
<td>A.2 Beweis von Lemma 2</td>
<td>197</td>
</tr>
<tr>
<td>A.1 Beweis von Theorem 6</td>
<td>198</td>
</tr>
<tr>
<td>B Beweise aus Kapitel 4</td>
<td>203</td>
</tr>
<tr>
<td>B.1 Beweis von Lemma 3</td>
<td>203</td>
</tr>
<tr>
<td>B.2 Beweis von Lemma 4</td>
<td>204</td>
</tr>
<tr>
<td>B.3 Beweis von Theorem 8</td>
<td>205</td>
</tr>
<tr>
<td>B.4 Beweis von Theorem 9</td>
<td>206</td>
</tr>
<tr>
<td>B.5 Beweis von Theorem 10</td>
<td>208</td>
</tr>
<tr>
<td>B.6 Beweis von Theorem 11</td>
<td>210</td>
</tr>
<tr>
<td>B.7 Beweis von Theorem 12</td>
<td>210</td>
</tr>
<tr>
<td>Teil V Deutscher Teil</td>
<td></td>
</tr>
<tr>
<td>Kurzfassung</td>
<td>215</td>
</tr>
<tr>
<td>Inhaltsverzeichnis</td>
<td>217</td>
</tr>
<tr>
<td>Einleitung</td>
<td>223</td>
</tr>
<tr>
<td>Schlußfolgerungen</td>
<td>239</td>
</tr>
</tbody>
</table>
Einleitung

C.1 Motivation

Es ist allgemein bekannt, daß verteilte und nebenläufige Hard- und Softwaresysteme Teil unseres täglichen Lebens geworden sind. Wegen unserer hohen Abhängigkeit von diesen Systemen wird es zunehmend wichtig, daß wir uns auf deren korrektes Funktionieren verlassen können. Wir können drei grundlegende Erwartungen unterscheiden, die wir bezüglich der Funktionalität eines Systems haben können:


sind Computernetzwerke wie das Internet oder Streamingsysteme, wo eine Nichtbeachtung von Zeitgrenzen keinen wirklichen Schaden sondern nur eine beeinträchtigte Dienstgüte zur Folge hat. In dieser Arbeit werden wir uns ausschließlich mit weichen Echtzeitsystemen befassen. Typische Fragen, die in diesem Zusammenhang auftreten sind:

- Wie lange muß ein Kunde, ein Auftrag, ein Prozeß im Mittel warten, bis eine Anfrage vollständig bearbeitet wurde?
- Was ist die mittlere Anzahl von Kunden, Jobs, etc. in einem System, was ist deren mittlere Aufenthaltszeit?
- Wie hoch ist die Auslastung eines Servers, von Kanälen etc.?


Der zweite und dritte Aspekt kann zusammenfassend als quantitative Anforderungen bezeichnet werden.

Um zu bestimmen, ob ein System sowohl qualitative als auch quantitative Anforderungen erfüllt kann auf verschiedene Weisen vorgegangen werden:


- Formale Verifikation: Im Bereich der formalen Verifikation ist es mit Hilfe mathematischer Methoden möglich zu beweisen, daß eine gegebene Systemspezifikation korrekt ist. Um formale Verifikation durchführen zu können, werden die folgenden Dinge benögt:
  - eine Spezifikationssprache um das zu analysierende System zu modellieren (Systemspezifikation),
C.1 Motivation

– eine formale Sprache, um die gewünschten Eigenschaften zu beschreiben (Eigenschaftsspezifikation),
– eine Verifikationsmethode um zu überprüfen, ob die Systemspezifikation die Eigenschaftsspezifikation erfüllt.

Die existierenden Methoden der formalen Verifikation können in zwei Klassen aufgeteilt werden:

– beweisbasierte Verifikation: Hier wird das System $\Gamma$ als Menge logischer Formeln $\{\Gamma_1, ..., \Gamma_n\}$ beschrieben, die gewünschten Systemeigenschaften $\Psi_i$ werden ebenfalls in dieser Logik formuliert. Mit Hilfe eines logischen Kalküls muß nun bewiesen werden, daß das durch $\Psi_i$ beschriebene Verhalten in $\Gamma$ möglich ist, formal, wir müssen beweisen, daß $\Psi_i$ eine logische Konsequenz von $\Gamma$ ist.


In dieser Arbeit beschränken wir uns ausschließlich auf den modellbasierten Verifikationsansatz. Die Nachteile der formalen Verifikation mit Bezug auf das Modell sind die gleichen wie für den simulationsbasierten Ansatz. Der Vorteil der formalen Verifikation besteht darin, daß wir uns auf die Ergebnisse verlassen können, d.h. wenn die Überprüfung ergibt, daß das System bzgl. einer bestimmten Anforderung fehlerfrei ist, dann können wir sicher davon ausgehen, daß dieser Fehler zumindest im Modell tatsächlich nicht auftritt.

Sowohl Testen als auch Simulation können angewandt werden um die Leistung und Zuverlässigkeit zu bewerten, aber es gibt auch hier formale, mathematische Ansätze:


- Numerische Leistungs- und Zuverlässigkeitsbewertung: In diesem Fall wird die Analyse auf Grundlage von Markov- oder Semi-Markovketten durchgeführt. Auf Grundlage dieser werden dann numerische Algorithmen zur Lösung von linearen und Differentialgleichungssystemen angewandt, um die interessierenden Maße zu berechnen.

In dieser Arbeit vertreten wir einen Ansatz, der modellbasierte formale Verifikation und numerische Leistungs- und Zuverlässigkeitsbewertung miteinander verbindet. Es gibt vielfältige Gründe so zu verfahren:

- Im Hinblick auf sinkende Markteinführungs- und Lebenszyklen ist es nicht realistisch anzunehmen, daß zwei verschiedene Modelle, eines für qualitative, ein anderes für quantitative Systemanalyse entwickelt werden können.

- Die Verwendung zweier Modelle für qualitative und quantitative Analyse birgt das Problem, beide Modelle untereinander konsistent zu erhalten.
Die Verwendung zweier getrennter Formalismen für qualitative und quantitative Verifikation erfordert Fachwissen in beiden Gebieten.

Ein Ansatz, der die vereinheitlichte qualitative und quantitative Systemanalyse verfolgt, wirft die folgenden Probleme auf:

1. Wahl einer geeigneten Spezifikationssprache
2. Modellierung komplexer Systeme
3. Spezifikation und Verifikation komplexer Systemeigenschaften

C.1.1 Wahl einer geeigneten Spezifikationssprache

Da es unmöglich ist, komplexe Systeme auf Grundlage des wenig abstrakten Modells auf dem die Analyse vorgenommen wird, zu beschreiben, muß eine geeignete Modellierungssprache gewählt werden. Diese Sprache muß Mittel zur Verfügung zu stellen, die sowohl qualitative als auch quantitative Systemanalyse zulassen.


In dieser Arbeit sollen nicht die Vor- und Nachteile der einzelnen Modellierungsformalismen diskutiert werden, wir wollen stattdessen kurz begründen, weshalb wir glauben, daß stochastische Prozessalgebren besonders für uns geeignet erscheinen:

- Strukturierter Entwurf: Kompositionsoperatoren ermöglichen es, aus kleinen Subsystemen einfach große und komplexe Systeme zu erzeugen.
- Mathematischer Kalkül: Wie in der gewöhnlichen Algebra, besitzen Prozessalgebren einen mathematischen Kalkül, der es erlaubt Terme zu manipulieren, zu vereinfachen, usw. Es können beispielsweise Teilterme einer gegebenen Spezifikation durch einfacherere ersetzt werden, so daß der gesamte Term immer noch äquivalent zu dem Originalterm ist.


Im Folgenden werden wir die Konsequenzen diskutieren, die die Wahl von Prozessalgebren als Modellierungsformalismus für die restlichen zwei von uns identifizierten Probleme zeitigt.
C.1.2 Modellierung komplexer Systeme

Die Verwendung von Prozessalgebren (sowohl zeitloser als auch stochastischer) als Modellierungssprache bedeutet, daß die Analyse qualitativer und quantitativer Systemeigenschaften auf Basis eines beschriebenen Transitionssystems (engl. labelled transition system, LTS) stattfindet. LTSs können als das semantische Standardmodell von Prozessalgebren betrachtet werden. Von einer gegebenen prozessalgebraischen Spezifikation kann durch die Anwendung semantischer Regeln das zugehörige LTS automatisch und eindeutig abgeleitet werden. Im Falle stochastischer Prozessalgebren ist das semantische Modell eine beschriftete zeitkontinuierliche Markovkette, die gewöhnlich als stochatisches beschriftetes Transitionssystem (engl. stochastic LTS, SLTS) bezeichnet wird. Ein SLTS ist ein LTS, bei dem die Transitionen sowohl mit Aktionsnamen als auch mit Raten beschriftet sind. Die Raten beschreiben, mit welcher Geschwindigkeit ein Zustand verlassen wird. SLTSs können sowohl für qualitative wie auch für quantitative Analyse eingesetzt werden.

Einerseits sind (S)LTSs sehr einfach und flexibel, andererseits werden die semantischen Modelle schnell sehr groß, was in der Tat ein Problem darstellt:

1. Sollen die Verifikationsergebnisse verläßlich sein, so muß das Modellverhalten nahe an dem des realen Systems sein. Solche Modelle müssen sehr detailliert sein, somit sind auch die zugehörigen (S)LTSs sehr groß.

Dieses zentrale Problem der modell- oder zustandsbasierten Analyse wurde bekannt als das 
Zustandsraumexplosionsproblem: Das Zustandsraumexplosionsproblem ist gut erforscht und es gibt einige Ansätze es zu bekämpfen.

Der von Prozessalgebren zur Verfügung gestellte mathematische Kalkül kann angewandt werden, um dieses Problem zu bekämpfen. Die Transformationen, die ein solcher Kalkül erlaubt, können angewandt werden, um Subsysteme durch solche zu ersetzen, die ein kleineres semantisches Modell besitzen. Der Nachteil dieses Ansatzes ist, daß er formalismusspezifisch ist und daß es sehr rechenaufwendig sein kann, kleinere Systeme aus größeren abzuleiten.

Es gibt andere Ansätze, die diese Nachteile verhindern können. Diese Ansätze können auf der Basis des semantischen Modells eingesetzt werden. Die existierenden Methoden können in zwei Klassen aufgeteilt werden: Ansätze, die den gesamten Zustandsraum erzeugen und solche, die nur Teile davon erzeugen:

1. Die Klasse der den gesamten Zustandsraum erzeugenden Methoden kann weiter unterteilt werden in Ansätze, die Massenspeicher oder verteilte Hardware einsetzen [68, 86, 106] und solche, die den Zustandsraum auf eine kompakte und effiziente Weise darstellen [33, 74, 130].
2. Bei den Methoden in dieser Klasse werden nur Teile des erreichbaren Zustandsraumes erzeugt.

Es sollte angemerkt werden, daß Halbordnungsverfahren die Analyse des gesamten Zustandsraumes erlauben.

Im Bereich der quantitativen Analyse taucht noch ein weiteres Problem auf: Die gewählte Technik der Zustandsraumdarstellung muß auch effiziente numerische Analyse zulassen [37, 35, 121].


C.1.3 Spezifikation qualitativer und quantitativer Maße

Es können Techniken zur Spezifikation funktionaler und quantitativer Maße unterschieden werden.

**Funktionale Eigenschaften**: Neben der Möglichkeit einfache Eigenschaften, wie die Abwesenheit von Verklemmungen zu überprüfen, gibt es mächtige Methoden komplexe Systemeigenschaften zu verifizieren, indem für die Eigenschaftsspezifikation eine formale Sprache und für deren Überprüfung effiziente Algorithmen eingesetzt werden. Die hier vorgestellte Technik ist das sogenannte Model Checking. Model Checking ist eine automatische Verifikationstechnik, die die Gültigkeit von Systemeigenschaften bezüglich eines gegebenen Modells überprüft. Model Checking kann auf alle Systemspezifikationen angewandt werden, deren semantisches Modell ein endliches LTS ist. Im Prinzip funktioniert Model Checking wie in Abbildung C.1 gezeigt. Der Model Checker benötigt zwei Eingaben:

1. Ein Systemmodell, welches das mögliche Verhalten beschreibt
2. Eine Beschreibung der Anforderungen, die von dem System erfüllt werden müssen.

Nachdem der Model Checker gelaufen ist, erhalten wir als Ergebnis, daß die Anforderung entweder erfüllt wurde oder nicht. Im Falle eines negativen Ergebnisses erhalten wir ein Gegenbeispiel, das zusätzlich Informationen enthält, wie und wo die Eigenschaft verletzt wurde, was im Korrekturprozeß eine große Hilfe sein kann. Model Checking ist ein iterativer Prozeß, im Falle der Eigenschaftsverletzung kann das Modell solange korrigiert werden, bis die Eigenschaft endlich erfüllt ist. Wenn das Modell keine Fehler enthält, kann das Modell verfeinert werden und der Model Checking Prozeß kann von Neuem beginnen, usw.

Das Model Checking ist im Prinzip eine erschöpfende Durchsuchung des Zustandsraums, d.h. für jeden Zustand wird überprüft, ob er die Eigenschaft erfüllt oder nicht. Das
Modell muß ein endliches LTS sein, das entweder direkt vom Benutzer vorgegeben wird, oder das aus einer abstrakten Systembeschreibungssprache, wie Prozessalgebren oder Petrinetzen durch Anwendung der semantischen Regeln abgeleitet wurde.

Die zweite Eingabe, d.h. die Spezifikation der gewünschten Systemeigenschaften erfolgt in der formalen Sprache einer Zeitlogik. Mit Hilfe von Zeitlogiken können zwei, im Bereich der Verifikation verteilter Systeme besonders wichtige Klassen von Eigenschaften ausgedrückt werden:

- Sicherheitseigenschaften umfassen Aussagen der Art:
  Jetzt und in der Zukunft wird immer Eigenschaft \( p \) gelten, oder äquivalent dazu, die Negation \( \neg p \) wird nie gelten.
- Lebendigkeitseigenschaften beinhalten Aussagen der Art:
  Zu irgendeinem zukünftigen Zeitpunkt wird Eigenschaft \( p \) gelten.

Zeitlogiken bestehen aus den folgenden drei Hauptbestandteilen:

- Aussagenlogik, d.h. atomaren Aussagen, Disjunktion, Konjunktion und Negation, um Zustandseigenschaften zu beschreiben.
- Pfadquantoren werden genutzt, daß eine bestimmte Eigenschaft auf allen Pfaden (universeller Pfadquantor) oder nur auf mindestens einem Pfad (existentieller Pfadquantor) gelten soll.
Es gibt eine Vielzahl von Zeitlogiken von denen LTL [4] (linear temporal logic) and
CTL [40] (computational tree logic) die zwei bekanntesten Vertreter sind. Für diese Lo-
giken existieren viele Werkzeuge, von denen SMV [105] für CTL und SPIN [78] für LTL
die wichtigsten sind. Es ist wert anzumerken, daß funktionales Model Checking bei vielen
großen Konzernen bereits zum Standardverfahren beim Entwurfsprozeß gehört.

**Quantitative Anforderungen:** Für die numerische Leistungs- und Zuverlässigkeitsbewer-
tung können die folgenden Ansätze zur Verifikation von Anforderungen identifiziert wer-
den:

- Spezifikation einfacher Anforderungen auf Grundlage von Markovketten
- Ableitung eines Markov-Rewardmodells aus einer gegebenen Hochsprachenspezifikation
  wie stochastischen Prozessalgebren oder stochastischen Petrinetzen.
- Erweiterung und Anpassung des funktionalen Model Checking für das Model Checking
  stochastischer Systeme

Alle diese Methoden haben gemeinsam, daß die Anforderungsspezifikation hauptsächlich
zustandsorientiert (vgl. Abbildung C.2) ist. Aus der Tatsache, daß mit Hilfe von Prozessalge-
bern das Systemverhalten einfach durch Aktionsfolgen beschrieben werden kann und
nicht nur durch Zustandsfolgen, wird kein Vorteil gezogen. TIPP ist ein Vertreter der er-

![Diagramm von Markov-(Reward)-Modell-basierte Anforderungsberechnung](image)

**Abb. C.2.** Markov-(Reward)-Modell-basierte Anforderungsberechnung

sten Klasse, wie Leistungsmaße beschrieben werden können:

- In TIPP können drei verschiedene Anforderungstypen beschrieben werden:
  - Zustandsmaße: Solche Maße definieren die Wahrscheinlichkeit, daß sich das System
    in einer bestimmten Zustandsmenge befindet. Ein Beispiel für ein solches Maß ist
    die Wahrscheinlichkeit, daß sich in einer Warteschlange \( n \) Aufträge befinden.
  - Mittelwerte: Mittels dieser Maße können die mittleren Werte bestimmter Parameter
    einer Spezifikation berechnet werden. Zum Beispiel kann die mittlere Anzahl von
    Aufträgen in einer Warteschlange berechnet werden.
C.2 Beiträge der Arbeit

In Abschnitt C.1 haben wir die drei Hauptprobleme identifiziert, die im Bereich der formalen Verifikation verteilter Systeme auftreten können. Diese Arbeit trägt zur Lösung des zweiten und dritten Problems in der folgenden Weise bei:

1. Wir schlagen eine neue Semantik für stochastische Prozessalgebren vor, die die kompositionelle Natur von Prozessalgebren ausnutzt. Die Speicheranforderungen für die

Durchsatz: Mittels dieser Maße kann die mittlere Häufigkeit berechnet werden, wie oft eine Aktion pro Zeiteinheit ausgeführt wird. Zum Beispiel kann die mittlere Anzahl von in einer Zeiteinheit bedienten Aufträge berechnet werden.

Die Prozessalgebra EMPA, wie beschrieben im Softwarewerkzeug TwoTowers [21], erlaubt die Spezifikation von transienten und stationären Maßen wie Auslastung und Durchsatz indem Zustands- und Transitionsrewards, d.h. Impulsrewards definiert werden. Die Rewardmaße werden definiert indem Aktionen, die für das Maß von Bedeutung sind, identifiziert werden und ihnen entsprechende Rewards zugeordnet werden. Die betreffenden Rewards können wie folgt berechnet werden: Jeder in der Spezifikation vorhandenen Aktion wird ein Zustands- oder Transitionsreward zugeordnet. Wann immer eine Transition, die eine relevante Aktionenbeschriftung trägt, aktiviert ist, bekommt der Quellzustand den entsprechenden Reward. Im Falle von Transitionsrewards, bekommt die aktivierte Transition selbst den Reward.


Alle diese Ansätze haben Vor- und Nachteile. Die Ansätze realisiert in TIPP und TwoTowers sind einfach anzuwenden, aber die Komplexität der beschreibbaren Anforderungen ist relativ gering.

Abspeicherung des Zustandsraumes sind hierbei linear, wohingegen die traditionelle Herangehensweise einen exponentiellen Speicheraufwand erforderlich machte. Durch diese Semantik wird die Spezifikation direkt auf eine kompakte MTBDD-basierte Darstellung des zugrundeliegenden Modells abgebildet.

2. Wir erweitern die existierenden Möglichkeiten Leistungs- und Zuverlässigkeitsmaße mit stochastischen Zeitlogiken zu spezifizieren, indem wir eine neue Logik und die dazu zugehörigen Verifikationsalgorithmen einführen. Diese neue Logik erlaubt die Spezifikation und Verifikation komplexer qualitativer und quantitativer Anforderungen auf der verhaltensorientierten Ebene.

C.2.1 Kompositionelle symbolische Semantik

In Abbildung C.4 ist ein Überblick über unseren neuen semantischen Ansatz abgebildet. Prozessalgebren sind eine kompositionelle Modellierungssprache, d.h. es ist möglich große Systeme aus kleineren zu konstruieren, indem eine große Anzahl von Kompositionsoperatoren eingesetzt wird. Beispiele sind sequentielle Komposition, Parallelschaltung, Auswahl, usw.


Unsere neue Semantik verhindert diese Probleme:

- Unsere Semantik ist symbolisch, d.h. die MTBDD-basierte Darstellung des Zustandsraumes kann direkt aus der Systembeschreibung gewonnen werden.

Abb. C.3. Verhaltensorientierte Beschreibung von Anforderungen
C.2 Beiträge der Arbeit


C.2.2 Automatische Verifikation stochastischer Systeme

Wir wollen die Methoden der automatischen Verifikation funktionaler Anforderungen auch im stochastischen Bereich einsetzen, dabei die Möglichkeit bekommen, das gewünschte Systemverhalten auf der gleichen abstrakten aktionsbzw. verhaltensorientierten Ebene wie die Systemspezifikation zu beschreiben. Zu diesem Zwecke führen wir eine neue stochastische Zeitlogik ein, die wir SPDL (stochastic propositional dynamic logic) nennen. Wir führen ebenso Algorithmen ein, die es ermöglichen, komplexe qualitative und quantitative Anforderungen automatisch zu verifizieren. SPDL ist eine stochastische Erweiterung der Logik PDL (propositional dynamic logic) [53].

Es sollte angemerkt werden, daß die Anwendbarkeit von SPDL nicht auf stochastische Prozessalgebren beschränkt ist. Tatsächlich ist die Logik und ihre Verifikationsalgorithmen auf jeden Formalismus anwendbar, dessen semantisches Modell ein SLTS ist und dessen Verhalten durch Folgen von Aktionen beschreibbar ist. Stochastische Petrinetze sind ein solcher Formalismus, Aktionsfolgen können hier als Transitionsfolgen interpretiert werden.

SPDL kann schwache Echtzeitforderungen ausdrücken. SPDL stellt Operatoren bereit, mit denen die Wahrscheinlichkeit, transient oder stationär, mit der eine Anforderung erfüllt sein soll quantifiziert werden kann. Beispiele solcher probabilistischer Anforderungen sind:

Abb. C.4. Direkte Erzeugung symbolischer Zustandsraumdarstellungen aus der Prozessalgebraspezifikation
• Transiente Anforderung:
Nach dem Erhalt eines Datenpakets soll der Empfänger mit einer Wahrscheinlichkeit von 0.8 innerhalb von 10 Zeiteinheiten eine Empfangsbestätigung verschicken.
• Stationäre Anforderung:
In einem stationären Gleichgewichtszustand soll das System mit einer Wahrscheinlichkeit von 0.99 korrekt funktionieren.

In Abbildung C.5 kann der Ansatz zur Verifikation stochastischer Systeme, der hier verfolgt werden soll gefunden werden. Wir können sehen, daß unser Ansatz die anderen An-

sätze erweitert, indem hier die Spezifikation von Anforderungen auf der verhaltensorientierten Ebene möglich wird. Dies ermöglicht die Spezifikation komplexer Leistungsmaße ohne Kenntnis des zugrundeliegenden SLTS.

Die Verwendung von Zeitlogiken, speziell von SPDL hat die folgenden Vorteile:
• Eigenschaften können spezifiziert werden, indem die gleiche abstrakte, aktionsorientierte Sichtweise, wie bei der Systemspezifikation beibehalten wird.
• Die formale Semantik von SPDL garantiert eine eindeutige Interpretation der zu verifizierenden Eigenschaften.
• Der Nutzer benötigt keine Kenntnis des zugrundeliegenden semantischen Modells.
• SPDL bietet Sprachmittel an, die es möglich machen einfach und genau Systemläufe zu spezifizieren, die es ermöglichen sollen, gewisse Eigenschaften zu erfüllen. Dies

Abb. C.5. Automatische Verifikation von Leistungs- und Zuverlässigkeitsmaßen
kann erreicht werden, durch Verwendung von erweiterten regulären Ausdrücken zur Spezifikation der erfüllenden Pfade.

- Es ist möglich SPDL auf verschiedenen Ebenen und in verschiedenen Phasen des Systementwurfs einzusetzen.

C.3 Verwandte Arbeiten

In diesem Abschnitt wollen wir verwandte Arbeiten auf dem Gebiet der symbolischen Semantik und der automatischen Verifikation stochastischer Systeme vorstellen.

C.3.1 Symbolische Semantiken für Prozessalgebren

Für rein funktionale Prozessalgebren sind uns drei vorherige Ansätze bekannt, direkt aus den Prozeßbeschreibungen eine BDD basierte Darstellung des zugrundeliegenden LTS abzuleiten.

In [50] beschreiben die Autoren für die Prozessalgebra CCS [109] Prozeduren wie BDD-basierte LTS-Darstellungen gewonnen werden können. Es wird nur eine Teilmenge von CCS betrachtet und die LTS der Operanden müssen bereits als BDD vorliegen.


C.3.2 Zeitlogiken

Es existieren mehrere Erweiterungen der Logik CTL für den stochastischen Fall. Zuerst sollte die Logik PCTL [65] erwähnt werden, eine probabilistische Logik die über zeitdiskreten Markovketten interpretiert wird. In PCTL werden die Pfadquantoren von CTL durch probabilistische Operatoren $P_{\bowtie \triangleleft p}(\phi)$ ersetzt, die ausdrücken, daß die Wahrscheinlichkeit, daß ein Pfad die Formel $\phi$ erfüllt in den durch $\bowtie p$ gesetzten Grenzen liegt, wobei $\bowtie \in \{<,>,\leq,\geq\}$.

Einleitung

CSL ist zustandsorientiert, d.h. zur Spezifikation der Maße muß die zugrundeliegende Markovkette bekannt sein und die Zustände der Markovkette können durch einfache Zustandsformeln (atomare Formeln) angereichert werden, um Maße definieren zu können.


C.4 Aufbau der Arbeit

Diese Arbeit ist wie folgt aufgebaut:

In Kapitel 2 werden wir die für die Arbeit notwendigen Grundlagen einführen.


In Kapitel 5 vergleichen wir die Ausdrucksstärke einiger stochastischer Logiken. Wir beweisen, daß die Logik SPDL ausdrucksstärker ist als die Logiken CSL, aCSL und aCSL+.

In Kapitel 6 stellen wir das Softwarewerkzeug CASPA vor, das ein stochastischer symbolischer Model Checker ist. Hier werden die in Kapitel 3 und 4 vorgestellten Konzepte in die Praxis umgesetzt.

In Kapitel 7 demonstrieren wir die Anwendbarkeit unserer Konzepte durch einige Fallstudien. Viele dieser Fallstudien sind in der Literatur weitverbreitet und dienen als
Benchmark für die Leistungsfähigkeit der eingesetzten Analysemethoden. Unter den untersuchten Fallstudien sind: Kanban-System [38] und das flexible manufacturing system (FMS) [39].

In Kapitel 8 beschließen wir diese Arbeit und geben einige Hinweise auf zukünftige Forschung.
D

Schlußfolgerungen

D.1 Zusammenfassung

Die Beiträge dieser Arbeit können in zwei Hauptteile gegliedert werden:

1. Semantik für stochastische Prozessalgebren
2. Verifikation aus stochastischen Prozessalgebraspezifikationen abgeleiteter Modelle

Wir haben stochastische Prozessalgebren als Modellierungsformalismus gewählt, weil diese wichtige Forderungen, die man an Modellierungs- und Spezifikationssprachen stellt, unterstützen. Dazu zählen Kompositionalität, Hierarchie und Abstraktion.


Früher wurde, bevor ein SLTS als MTBDD gespeichert wird, dessen explizite Darstellung gewonnen. Im Falle von parallel geschalteten Prozessen erfordert dieser naive Ansatz exponentiell viel Speicherplatz.

Wir schlagen einen semantischen Ansatz vor, der dieses Problem dadurch umgeht, indem das Konzept der Kompositionalität von Prozessalgebren ausgenutzt wird. Zu diesem Zweck haben wir für jeden Operator der Prozessalgebra eine semantische Regel definiert, mit deren Hilfe die MTBDD-Darstellung des Transitionsverhaltens der an diesen Operatoren gebundenen Teilprozesse ohne Umweg über eine explizite Darstellung, direkt auf ein MTBDD abgebildet wird.

Im zweiten Teil der Arbeit haben wir uns der Frage zugewandt, wie Leistungs- und Zuverlässigkeitsanforderungen, die das modellierte System erfüllen soll, automatisch verifiziert werden können.

In der herkömmlichen Leistungsbewertung sind die zu berechnenden Maße häufig zustandsbasiert, d.h. um sinnvolle Maße definieren zu können, ist es notwendig, Kenntnisse über die Struktur des Zustandsraumes zu besitzen, der das Systemmodell darstellt. Dies widerspricht in gewisser Weise dem Modellierungsansatz, wie er durch die Verwendung von Prozessalgebren vertreten wird, da hier das Systemverhalten abstrakt durch Folgen von Aktionen charakterisiert wird. Um dieses Problem zu vermeiden, haben wir eine stochastische Logik entwickelt, die auf der modalen Logik PDL basiert. Diese Logik, SPDL,
kann komplexe Leistungs- und Zuverlässigkeitsmaße sowohl aktions- als auch zustandssoriert beschreiben. Diese Verbindung von zwei Arten, wie Leistungs- und Zuverlässigkeitsmaße ausgedrückt werden können, ermöglicht es dem Modellierer in SPDL aus Programmiersprachen bekannte Konstrukte wie *if*-then oder *while* einzusetzen. Ein Beispiel, in dem die Verwendung solcher Strukturen von Vorteil ist lautet wie folgt:

Wenn das System eine bestimmte Last hat, ist die Wahrscheinlichkeit, daß dann innerhalb von \( t \) Zeiteinheiten \( n \) weitere Aufträge ankommen, und das System dann überlastet ist, größer als \( x \) Prozent?

Im Werkzeug CASPA haben wir sowohl die neue symbolische Semantik als auch die Logik SPDL implementiert und deren Effizienz anhand zahlreicher Fallstudien demonstriert.

### D.2 Zukünftige Forschung

Wir wollen hier einige Hinweise auf mögliche zukünftige Forschung geben.

#### D.2.1 Erweiterungen von SPDL

Der Pfadoperator von SPDL könnte so erweitert werden, daß die Zeitgrenzen nicht mehr länger notwendigerweise feste Werte, sondern auch Zufallsverteilungen, die aus beliebigen Verteilungen gewonnen werden, sein können. In [98] wird für CSL ein Ansatz beschrieben, wo für Intervalle \([0, t]\) die Obergrenze ein solcher Zufallswert ist. Man kann den Ansatz nun auf Intervalle der Form \([t, t + T]\), oder \([T, T + T']\) ausweiten, wobei \( t \neq 0 \), und die obere Grenze ein Zufallswert und die untere Grenze entweder ein fester Wert oder ebenfalls ein Zufallswert ist.

#### D.2.2 Definition von Entwurfsmustern


#### D.2.3 Kompositionelle Symbolische Bisimulation

Wir haben bewiesen, daß die Gültigkeit von SPDL Formeln unter action-labelled Markov-AP-Bisimulation erhalten bleibt. Es ist daher wünschenswert Model Checking auf einem solchen reduzierten Modell durchführen zu können. Auf der anderen Seite ist bekannt, daß die Größe von BDDs, die ein bisimulations-minimiertes Transitionssystem darstellen, größer ist als die BDDs, die nicht-reduzierte Transitionssysteme darstellen.


Bei all diesen Arbeiten muß eine sorgfältige Abwägung zwischen dem zusätzlichen Aufwand zur Berechnung der reduzierten Zustandsräume, dem Geschwindigkeitsgewinn bei der Überprüfung von Eigenschaften und der Größe der dann analysierbaren Systeme getroffen werden.

### D.2.4 Einfluß der Variablenordnung auf MTBDD-Größen

Wird der Zustandsraum eines Modells auf dessen erreichbaren Teil eingeschränkt, so ist auffällig, daß hieraus ein Zuwachs der MTBDD-Größen resultiert, dies konnte im Extremfall in Kapitel 7.8 beobachtet werden, trifft aber, weniger extrem, auf alle untersuchten Modelle zu. Die Ursache dieses Verhaltens ist darin zu suchen, daß durch die Erreichbarkeitsanalyse Regelmäßigkeiten in der Zustandsraumstruktur zerstört werden, die in Verbindung mit den gewählten Variablenordnungen zu einer kompakten Darstellung führen.

Es ist wert untersucht zu werden, wie groß der Einfluß einer anderen Variablenordnung auf die MTBDD-Größen ist. Auch hier muß wieder sorgfältig abgewogen werden, ob der erhoffte Gewinn, kleinere MTBDDs, falls eine geeignete Ordnung gefunden wurde, den Aufwand zur Berechnung anderer Ordnungen lohnt.
Lebenslauf
Georg Wolfgang Matthias Kuntz

Geburtsdatum: 14.05.1974
Geburtsort: Erlangen

seit Dezember 2005: Wissenschaftlicher Mitarbeiter am Lehrstuhl “Ontwerp en Analyse van Communicatiesystemen”, Universiteit Twente, Enschede, Niederlande  