High-Level Modelling and Efficient Analysis of Randomized Protocols

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden,
Fakultät für Informatik

eingereicht von Dipl.-Inf. Frank Ciesinski

Dresden, im Oktober 2010

1st revised version
Für meine Eltern

Renate und Bernhard Ciesinski

(to my parents)
Abstract

Model checking is a fully automatic verification technique that allows for proving that a given model satisfies - under all circumstances - a given property. In this thesis we develop and evaluate techniques that concern quantitative model checking of probabilistic nondeterministic systems against linear temporal properties, i.e., calculating the (minimum or maximum) probability that a linear temporal property is satisfied by the model.

In this context we discuss the design and implementation of a probabilistic modelling language for probabilistic nondeterministic systems. This language named PROBMela features parallel interleaved processes, shared and local variables, nondeterministic and probabilistic choice, loops, asynchronous and synchronous communication via channels, lossy channels, randomized assignments and atomic regions. For this language we give operational semantics in terms of Markov decision processes (MDPs).

The main obstacle in model checking is the state explosion problem which means that the size of the system under consideration (i.e., the model) usually grows exponentially with the number and size of their compositional parts like processes, variables and level of reactive behaviour. In this thesis we present work concerning reduction techniques to cope with the state explosion problem. The quantitative analysis of MDPs relies on the graph-theoretic analysis of particular sub-structures, the so-called end components. We discuss some improvements of the numerical solving stage of the quantitative analysis, which exploit particular properties of end components to construct a reduced version of the MDP that is considerably easier to solve, thus weakening the impact of the state explosion.

We also discuss how the concept of fairness, a concept that is very important in the context of high-level modelling of parallel systems, can seamlessly be integrated into the quantitative analysis procedure.

Furthermore we turn our attention to practical aspects of partial order reduction reduction method for probabilistic systems. The ample-set method involves five conditions, the so-called ample set conditions, that fix precise circumstances for a property-preserving reduction of the outgoing edges of the MDP's states, thus yielding an (property-wise) equivalent reduced MDP. Without further changes these conditions are, because of their partially global nature, not ready-to-use for a practical implementation. We discuss sufficient, weaker local criteria for these five conditions and report on their potential in a practical implementation.

All the aspects, the language, the numerical reduction, the partial order reduction and the fairness approach mentioned above, have been implemented in the explicit, virtual-machine based quantitative linear-time model checker LiQuor for experiments.
Acknowledgements

The "amount of thank" that I owe to Christel Baier can neither fit on this nor on any other piece of paper. I deeply admire her for being so friendly every day and for letting me participate in her work. I thank her for encouraging me countless times, for her patience, her guidance and for teaching me all that I know about theoretical computer science, logics and mathematics in all the discussions, her beautiful lectures, papers and books. She is an outstanding and truly unique person and I thank her for her time.

I also thank my long-time office-mate Marcus Größer who is as funny as he thinks but not always a delight. This is, of course, his feature and not a bug. On countless occasions he was not too tired to rush to the whiteboard and amaze me with his razor-sharp ability to create, out of thin air, beautiful counter-examples, theorems and proofs, thus uncovering the sometimes painful mathematical truth. I also feel the strange desire to thank him for introducing me (also on countless occasions) to the works of John Denver.

I thank Joachim Klein for his work on the PROBMela compiler, maybe the only piece of code I know that is really 100 percent bug-free. Besides possessing many other talents Joachim is an outstanding code-wizard.

Last but not least I want to thank the other woman in my life, Manuela Berg, for her endless and pure love.

---

1 See "acknowledgements" in [Gro08]
# Table of Contents

**Chapter 1 – Introduction**

**Chapter 2 – Preliminaries on notations, models and methods**

- Basic mathematical notations and Markov models . . . . . . . . . 11
- Quantitative analysis of Markov decision processes . . . . . . . . 19
- Probabilities for reaching goal sets in Markovian models . . . . . . 19
- Extremal probabilities for \( \omega \)-regular properties . . . . . . . . 24
- Graph-based preprocessing . . . . . . . . . . . . . . . . . . . . 31

**Chapter 3 – PROBMela specification language**

- The syntax of the core language . . . . . . . . . . . . . . . . . . . . . . . . . 37
- Operational semantics for the core language . . . . . . . . . . . . . . . . . . 42
- Probabilistic control graph semantics for PROBMela programs . . . . . . 43
- MDP semantics for Probabilistic control graphs . . . . . . . . . . . . . . . . 49
- Advanced language features . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
- PASM bytecode and LiQuor’s virtual machine . . . . . . . . . . . . . . . . . . 57

**Chapter 4 – Efficient probabilistic reachability analysis**

- Value iteration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
- Priorized updates . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
- Structured solving approaches . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
- Experimental results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80

**Chapter 5 – Quantitative analysis under fairness constraints**

- Fairness notions for probabilistic systems . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
- Fair schedulers and realizability of fairness conditions . . . . . . . . . . . . . . . . . . . . . 89
- Fair satisfaction of \( \omega \)-regular LT-properties . . . . . . . . . . . . . . . . . . . . . . . . . 95
- Checking realizability on-the-fly . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
- Handling of unrealizable fairness conditions . . . . . . . . . . . . . . . . . . . . . . . . . . . 100
# TABLE OF CONTENTS

## CHAPTER 6 – Partial order reduction of Markov decision processes

6.1. The ample-set method for MDPs .............................................. 103  
6.2. Efficient calculation of ample sets ...................................... 108  
6.3. Experimental results ......................................................... 125  
6.4. Partial order reduction and fairness ..................................... 126  

## CHAPTER 7 – Conclusions .......................................................... 133
1

Introduction

Randomized algorithms. The concept of probabilistic or randomized algorithms and protocols is well established in computer science since the 1970s. A vast amount of sequential randomized algorithms, i.e., algorithms that partly rely on randomized computation steps, covers many fields in computer science from complexity theory to state-of-the-art practical applications. E.g., consider concepts like random sampling to examine and statistically analyse large amounts of data much faster than deterministic methods, randomized fingerprinting that provides the basis for ultra-fast, ultra-efficient matrix multiplication or efficient randomized primality tests that are indispensable for today’s state-of-the-art mechanisms in security of digital communication on which today’s business processes and bank transactions deeply rely. In distributed systems such as adhoc networks that are formed by embedded systems, e.g., telecommunication- or handheld devices, randomized decision steps are a simple but powerful technique for coordination protocols to break the symmetry of the system, e.g., to achieve mutual exclusion of shared resource usage or to elect a dedicated leader node, which may be necessary for the primary application of the devices. In some cases of application it can be proven that a deterministic algorithm does not exist while a randomized algorithm solves the problem surprisingly easy.

Probabilism elsewhere. Besides the application of randomized steps in algorithms that contribute for realizing concrete applications, probabilistic means are also utilized in a wide range of modelling tasks, be it to model randomized systems for model-based analysis or to provide a powerful abstraction mechanism to non-probabilistic systems. For instance, stochastic distributions can be used to model the frequency of failures in hardware components of resilient systems without the necessity of specifying a complex deterministic “failure scheme”, which is in many cases too hard or impossible. Stochastic assumptions about the system environment are used in performance evaluation to specify, e.g., a workload or average durations of execution times of jobs. When deriving these stochastic assumptions from empirical data collected with real-world systems like past failures of production machines in such cases a simple but expressive abstraction of the environment can be applied to the model. Another field of successful applications of stochastic methods is the modelling of chemical reaction pathways or feed-back systems that involve concentrations of chemical
agents, e.g., in biological systems. In robotics, imprecise knowledge about the environment is often formalized by means of stochastic concepts, e.g., with so called belief states that use stochastic assumptions to reflect uncertain knowledge such as the actual location of an autonomous agent.

**Markov decision processes.** Markov decision processes (MDPs) can be seen as a general-purpose stochastic model with a wide range of real-world problems that involve stochastic and dynamic decisions \[\text{[Bel57, Bel58, Put94, Ber87]}\]. Their history goes back to the 1950s and their field of successful application ranges from economics to robotics, planning, artificial intelligence, verification of complex systems, etc. Since MDPs combine discrete transition probabilities with nondeterminism it is a most appropriate basis for providing an interleaving semantics of randomized distributed or concurrent (parallel) systems. The role of the nondeterminism in this context is versatile. Besides its obvious role for modelling process interleaving it can be crucial for several other purposes:

- Modelling the interface to a completely unpredictable system environment (e.g., a human user).
- Nondeterminism is the "key" feature of almost all abstraction techniques, where the potential concrete behaviour is modelled by (abstract) nondeterministic choices.
- Underspecification which represents "implementation freedom" and which is possibly refined in subsequent modelling steps.

It is apparent that the occasions for applying nondeterministic steps in a model may overlap with those for using probabilistic behaviour because, as mentioned above, probabilistic behaviour may also be used for abstraction. The combination of probabilistic and nondeterministic steps is therefore of particular interest. In addition to that MDPs allow for specifying (state- or transition-based) costs/rewards that in connection with probabilistic transitions can be used to reason about expected values.

Though MDPs are not equipped with any explicit formalism regarding timing behaviour they can be used to express discrete-time behaviour since their transitions can be interpreted as discrete-time steps. Many algorithms proposed in the literature for the quantitative analysis of probabilistic models with dense time domain fundamentally rely on a reduction to discrete-time MDPs, e.g.:

- probabilistic timed automata that link to MDPs via region graphs \[\text{[KNSS02]}\], zones and Difference Bound Matrices \[\text{[KNSW07]}\] or the digital clock approach \[\text{[KNPS06]}\] and
Chapter 1. Introduction

- discretization techniques for continuous-time MDPs (i.e., MDPs with exponential distributions rather than discrete distributions for the successor states) and variants thereof, see, e.g., [HvdD84, Neu09, ZN10].

Model checking methods for MDPs (for temporal properties). MDPs in general are used in the literature for studying a wide range of optimization problems solved via dynamic programming, linear programming, iterative methods or, e.g., reinforcement learning. In the context of MDPs the verification against temporal properties usually means qualitative or quantitative analysis either by deductive methods or by employing probabilistic variants of the model checking approach (see [QS82, CES86, EL85, LP85]), a fully automatic verification method. The emphasis of this thesis is on the fully algorithmical approach with model checking techniques against temporal logics. In contrast to simulation or testing, model checking explores the possibilities of system behaviour in its entirety. Consequently it may reveal errors in system behaviour that may be critical, but also very "rare". It thus can be seen as orthogonal to simulation, which is unlikely to detect such very rare errors. In the context of model checking the term "qualitative analysis" refers to the question whether or not - under consideration of the most unfavourable resolving of the nondeterministic behaviour - the MDP satisfies the temporal property almost surely, i.e., with probability 1 (or dually, almost surely not, i.e., with probability 0). The term 'quantitative analysis' implies the goal of calculating the actual probability for this.

Linear time properties, a popular subclass of temporal properties for these kinds of analysis were considered in the literature first with the focus on qualitative analysis [HSP83] in the form of the question for almost sure termination of probabilistic distributed systems. In [PZ86] tableaux-based methods for specifications in several variants of linear time logics were considered. Later, the automata-theoretic approach for model checking of linear time properties was examined in [VW86, CY88, CY90, CY95, JA97, BK98].

A popular quantitative logic, which relies on branching-time semantics rather than on linear-time semantics, is Probabilistic Computation Tree Logic (PCTL). Verification of non-concurrent probabilistic sequential programs against PCTL were considered in [HJ89, HJ94]. The logic PCTL for concurrent probabilistic programs modelled by MDPs was considered in [BdA95].

Fairness. In probabilistic nondeterministic systems the notion of fairness is inherently important for excluding unrealistic extreme behaviour. Consider, for instance, the unrealistic execution of two concurrent processes where only one of them gets ever executed by the execution environment. In the context of their analysis for termination of concurrent probabilistic systems Hart, Sharir, Pnueli [HSP83] introduced also fairness. Shortly after that Vardi [Var85] introduced the more general concept of fair schedulers for models that are
similar to MDPs in order to allow for a qualitative analysis against a given linear temporal formula. In context of PCTL the concept of fairness has been discussed in [BK98], where also algorithms for the treatment of PCTL* (and by this also linear-time properties) under fairness assumptions are given. There, a characterization of particular sets of states associated with fair behaviour is given. These sets are strongly related to end components that are used in [CY95, dA97].

Different notions for the fairness of randomized schedulers of an MDP together with algorithms for extremal reachability probabilities and long-run averages have been presented in [dA99], where lower probability bounds for the chosen actions are required.

In [PZ93, APZ03] the concepts of \(\alpha\)-fairness or \(\gamma\)-fairness have been introduced as proof techniques to verify that a given linear-time property holds almost surely (called P-validity in [APZ03]) for an MDP-like model equipped with action-related fairness constraints. The variants of \(\alpha\)- and \(\gamma\)-fairness serve for reducing the qualitative analysis to fair verification by means of “non-probabilistic” methods.

**State space explosion and methods.** The original concept of model checking, namely exploring all possible system behaviours, contributes also to its most challenging problem, namely the state explosion problem. Consider a scenario involving \(n\) concurrent processes that (for the sake of a simpler estimation) each have \(l\) program states and possess \(v\) binary variables. Then, the upper bound for the number of possible different states for one single process amounts to \(2^v \cdot l\) and the upper bound of the complete state space size is \(2^\nu n \cdot l^n\).

Though this theoretical upper bound is usually not met by models with practical relevance their size is nevertheless typically still exponentially in \(n, l\) and \(v\). Since the ultimate goal of a model checking technique is to examine this state space in its entirety, all approaches to model checking can be classified according to the way they deal with the state explosion problem which means how they attempt to reduce the amount of calculation time and memory in comparison to the amount that are required for a plain, unreduced analysis.

In explicit model checking the state space is constructed explicitly, i.e., states and transitions of the system are traversed (and stored when necessary) with the goal of analyzing its structure according to the needs of the undergoing model checking procedure. For explicit model checking, partial order reduction methods proved to be very successful in the non-probabilistic setting [HP94, GW91, Val91] for reducing the state space of a given model significantly while at the same time preserving their linear-time or branching-time properties. For the probabilistic case this approach was extended in [BGC04, dN04, BdG06].

Other methods rely on implicit representations of the state space. In the symbolic approach the transition relation is represented implicitly by a characteristic multidimensional Boolean function with valuations in \([0, 1]\). Because (MT)BDDs share parts of their structure between different functional parts
they tend to have very compact representations for large systems and thus work as a reduction technique. This approach is very suitable for branching time logics since the set-based calculations that are typically carried out in model checking branching-time logics can be expressed as operators of Boolean functions. Thus the whole model checking process takes part in the ‘world’ of a Boolean function. SAT-based model checking takes existing SAT-solvers and pursues the idea of symbolic model checking with first-order logic instead of Boolean functions.

The abstraction-refinement method [GS97] reduces the state space drastically through predicate abstraction and successive refinement of the “important” parts of the abstracted system. A very successful way to determine which aspects of the system are to be refined is through counterexamples, which are witnesses for the system violating the considered property (CEGAR, counter-example guided abstraction-refinement, [CGJ+00]). For probabilistic systems heuristics [dJJL02] were proposed to guide these refinement steps. A probabilistic variant of CEGAR was considered in [HWZ07].

Another popular reduction technique which aims at constructing a reduced equivalent system by exploiting information about symmetry is symmetry reduction. This technique reduces systems of interchangeable, components such as those which commonly arise in randomised distributed algorithms or probabilistic communication protocols by generating a (typically much smaller) quotient system. For probabilistic models an approach was proposed in [DM05, KNP06].

The assume-guarantee approach, which, amongst for others, has been considered for probabilistic systems in [dAHJ01, DC08, KNPQ10], is a compositional approach in which a system comprising multiple concurrent interacting components is verified by analysing each component in isolation rather than verifying the much larger model of the whole system, which is typically a product of the components.

Other reduction approaches like compositional abstraction techniques [d’A03a], reduction of the numerical problem prior to solving [CBGK08] or bisimulation minimizations [KKZJ07], here in the context of model checking for Discrete Time Markov Chains.

Statistical model checking, see, e.g., [LD10], is another approach to solve the model checking problem. Here, the system is simulated for finitely many runs and hypothesis testing is used to infer whether the samples provide statistical evidence for the satisfaction or violation of the specification.

Tools. Practical implementations (tools) that demonstrate the applicability and relevance of proposed model checking techniques vary from numerous prototype implementations to sophisticated model checkers that in some cases reach the level of reliable, mature and easy-to-use software. All implementations that are available today can be classified in terms of their main concept that is used to tackle the state explosion problem. The first model check-
ers with significant impact on the industrial market were the explicit model checker SPIN [Hol97] for the logic LTL and the BDD-based symbolic model checkers SMV (see [McM92] and NUSMV [CCGR00] for CTL. While SPIN mainly uses partial order reduction approach to reduce the state space, the CTL model checkers rely on the reduction abilities of BDDs. For the analysis of probabilistic systems there is also a range of tools available. The tool PRISM [KNP02] uses a hybrid approach of symbolic representations with Multiterminal Binary Decision Diagrams (MTBDDs) and explicit array-wise data organization to overcome the state space problem and the fact that symbolic representations are often compact but also slow on the other hand. The tool CASPA [Kun06b, Kun06a] uses stochastic process algebras as its input language and verifies stochastic performance properties formulated in SPDL. It employs specialized variants of MTBDDs. RAPTURE [JdL02] uses automated abstraction and refinement strategies to reduce the state space for calculating extremal reachability probabilities. In addition to that a reduction technique that is called "essential state reduction" is applied that combines groups of states that are connected with probability 1 into equivalence classes. The tool DiVinE [BBCRar] is designed for the execution in parallel working environments like multicore-computers and networks of homogeneous computers. It utilizes partial order reduction to reduce the state space and uses standard approaches to solve the associated numerical problem. The tool MRMC (see, e.g., [KZH"09]) uses bisimulation minimization to reduce the state space of Markov Chain models.

**Modelling formalisms.** A large amount of a tool’s potential is rooted in its specification formalism. A good modelling formalism finds a balanced compromise between being powerful enough to express behaviour about properties that are interesting to the analysis and providing a reasonable grade of abstraction to allow for an efficient analysis on the other hand. This can be achieved by employing high-level modelling language for randomized systems with operational interleaving semantics, i.e., stochastic models with nondeterminism. The family of ‘traditional’ mathematical formalisms and theories for formal reasoning about concurrency in general is normally referred to as process algebras or process calculi. This family was founded with CSP [Hoa85] and CCS [Mil80, Mil89] and is still evolving from there. Process calculi provide as a basis formalism the parallel interleaved execution of communicating processes that may make use of nondeterministic choice (called 'internal' choices).

In the literature, a variety of probabilistic variants of process algebras have been proposed with an operational semantics based on some probabilistic model. Prominent examples are variants of CCS with a Markov chain- or MDP-semantics, see, e.g., [vGSST90, dAG01] or process algebras with stochastic delays based on continuous-time Markovian models.
We also mention here LOTOS (see, e.g., [BB87]) and the $\pi$-calculus (see, e.g., [Mil91]). The process algebra MoDeST [BdHK06] contains classical features of process algebras with real-time constraints, stochastic delays and probabilistic choice operators. It relies on an operational semantics based on stochastic timed automata. MoDeST is a very rich compositional formalism that subsumes a wider spectrum of functionality, partly from other process calculi, in one language. The language has, amongst others, semantics of plain labeled transition systems, Markov Chains and Markov decision processes and probabilistic timed automata. It supports nondeterministic and probabilistic delaying of actions and exception handling.

Reactive modules [AH99] are a formal framework for synchronous and asynchronous systems and supports component based modelling and an abstraction operator that allows for switching between different levels of abstraction and switching between synchronous and asynchronous generation for a specification. Reactive modules very naturally reflect the stepwise transition relation of transition based models. A probabilistic variant of reactive modules (without the abstraction operator) is used in the model checker PRISM. The language pGCL [Mor99] extends Dijkstra’s GCL [Dij76]. While in GCL one is interested in correctness and termination of the examined program pGCL reasons about the (almost sure) termination and correctness of randomized algorithms. The popular language Promela that is used by the model checker SPIN was inspired by GCL and contains buffered channels and atomic regions, i.e., the ability to execute multiple commands as one single step without the interleaving of other processes.

Main contribution of this thesis. This thesis contains main contributions concerning all relevant sub-topics of probabilistic analysis mentioned above. In the following this contribution is briefly outlined. We designed a guard-oriented operational process language that is strongly inspired by GCL and Promela for the use in the context of quantitative analysis of linear-time properties. We called this language PROBMela. We provide operational semantics for systems of asynchronous parallel interleaved processes that are specified using PROBMela. The language has the following features:

- guarded commands
- buffered asynchronous channels that may be perfect or lossy
- handshaking between processes (synchronous actions)
- randomized assignments
- different numerical datatypes
- probabilistic and nondeterministic choice
Chapter 1. Introduction

• atomic regions, i.e., program sections composed out of multiple commands that are executed as one single step.

For the analysis stage the aim was to develop, apply and analyse reduction techniques that cope with the state explosion problem. The ample set method for probabilistic systems [BGC04] is able to reduce the MDP in an on-the-fly fashion while preserving \( \omega \)-regular properties. We report on the implementation and the performance of this method.

**Figure 1.1:** Overview on quantitative automata-based model checking

**Organisation of the thesis.** In chapters 3-6 the thesis addresses four different but strongly related topics in the field quantitative analysis of probabilistic systems, whereas chapter 2 recalls some preliminaries from the literature about Markov models and their quantitative analysis against linear time properties. Please refer to figure 1.1 for a guidance on the different keywords.

Chapter 3 describes central aspects of the design and implementation of the modelling language PROBMela, an imperative, process based, probabilistic modelling language. We give the syntax and operational program graph- and Markov decision process semantics for the core language. Moreover, this language was implemented in the model checker LiQuor and serves as the foundation for the implementation and evaluation of the techniques addressed
in the remaining chapters. This language is suitable for elegant modelling randomized nondeterministic communication protocols on a level that is relevant in practical settings. We provide operational semantics defined in a two-fold manner. First we give probabilistic control graph semantics for PROBMela programs, then we provide a Markov decision semantics for PCGs. The language PASM, which defines the low-level operational behaviour of probabilistic programs graphs in terms of virtual-machine operations in LiQuor, is strongly related to probabilistic control graphs, which makes the implementation of a non-trivial high-level modelling language this approach elegant instead of complicated.

In Chapter 4 we explain how the efficiency of the numerical solving stage that is an integral part of the quantitative analysis of Markov decision processes can be increased significantly by a potent reduction technique. For this we make two main observations. First, we propose the usage of value iteration for quantitative explicit model checking, which outperforms classical approaches as the Simplex algorithm. Secondly we observe a key property of end components that is directly connected to the numerical optimization problem. With this it is possible to build the linear optimization problem on the basis of a reduced quotient system rather than on the unreduced one. We conclude the chapter with some additional structural improvements of this approach.

In Chapter 5 we propose an algorithmical solution for quantitative model checking of Markov decision processes under strong and weak fairness constraints. We show how to handle realizability in context of a practical implementation and how to employ fairness on-the-fly. We also show that the overall complexity of the model checking process is not affected by using fairness constraints. In Chapter 6 we report on implementation details of the ample set method for partial order reduction of probabilistic systems. We discuss some weaker criteria of the original ample set conditions \cite{BGC04} and report briefly on experimental results. Chapter 7 concludes the thesis with comments on the results.

Several concepts and results of this thesis have been presented on conferences and workshops: \cite{BGC04, BCG04, BCG05, CB07, CB07, BGC09b, BGC09a, BGC09a}.
Preliminaries on notations, models and methods

Throughout the thesis, we suppose that the reader is familiar with temporal logics and model checking [CES86, Eme90, CGP99, BK08], automata over infinite words [GTW02], measure and probability theory [Fel91, Pan01], and discrete-time Markovian models (Markov chains and Markov decision processes) [KS60, Put94]. For the sake of self-containment this chapter provides a brief summary of the main concepts of quantitative analysis that are relevant for the thesis and fixes the notations that will be used most throughout the thesis. The summary not intended to be complete. For additional details we refer to the above mentioned literature.

2.1. Basic mathematical notations and Markov models

Let \( \mathbb{N} \geq 1 = \{1, 2, 3, \ldots\} \) and \( \mathbb{N} = \mathbb{N} \geq 1 \cup \{0\} \).

Definition 2.1.1. [Finite and infinite words] Given a nonempty set \( \Sigma \), the set \( \Sigma^+ \) of finite nonempty words over \( \Sigma \) is defined as

\[
\Sigma^+ = \{ w = \sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_n : n \in \mathbb{N}, \sigma_i \in \Sigma, 0 \leq i \leq n \}.
\]

The set \( \Sigma \) is called the alphabet, while the \( \sigma_i \)’s are called symbols. Let \( \Sigma^* = \Sigma^+ \cup \{\epsilon\} \), where \( \epsilon \) is the empty word. The set of infinite words is defined as

\[
\Sigma^\omega = \{ w = \sigma_0, \sigma_1, \sigma_2, \ldots : \sigma_i \in \Sigma, i \in \mathbb{N} \}.
\]

Finally the set of finite and infinite words is denoted as \( \Sigma^\infty = \Sigma^\omega \cup \Sigma^* \).

Given a finite or infinite word \( w = \sigma_0, \sigma_1, \ldots \in \Sigma^\infty \) we denote by \( |w| \) the length of \( w \), where \( |w| = \infty \) if \( w \) is an infinite word. We denote the \( i \)-th symbol (where \( i < |w| \)) as \( w^i \), the \( i \)-th prefix \( \sigma_0, \sigma_1, \ldots, \sigma_i \) of \( w \) is denoted as \( w^i \), and the \( i \)-th suffix \( \sigma_i, \sigma_{i+1}, \ldots \) of \( w \) is denoted as \( w^i \).

Definition 2.1.2. [Stochastic Distribution] For a countable set \( S \) a substochastic distribution is a function

\[
\mu : S \rightarrow [0, 1], \text{ such that } \sum_{s \in S} \mu(s) \leq 1.
\]
We denote with $\text{support}(\mu)$ the support of $\mu$ as the set $s \in S$ for which $\mu(s) > 0$. If $\sum_{s \in S} \mu(s) = 1$, then we call $\mu$ a stochastic distribution, or briefly distribution. For distributions $\mu$ where $\mu(s) = 1$ (and henceforth $\mu(s') = 0$ for all other $s' \in S$) we write $\mu_s^1$. This is often called a Dirac distribution. The set of all stochastic distributions over a countable set $S$ is called $\text{Distr}(S)$.

**Definition 2.1.3. [Markov chain]** A Markov chain is a tuple

$$\mathcal{M} = (S, \rightarrow, s_0, \text{AP}, L),$$

where

- $S$ is a countable nonempty set of states,
- $\rightarrow: S \rightarrow \text{Distr}(S)$ is the transition probability function,
- $s_0 \in S$ is the initial state of $\mathcal{M}$,
- $\text{AP}$ is a set of labels, and
- $L: S \rightarrow 2^{\text{AP}}$ is the labeling function.

In the literature a Markov chain is often called DTMC (discrete time Markov chain) as a stepwise behaviour is modelled, that can be interpreted as a sequence of (discrete) time passages. For better readability we write $s \rightarrow \mu$ rather than $\rightarrow (s) = \mu$ and $s \rightarrow s'$ if $s \rightarrow \mu_{s'}^1$. To directly obtain the probability $\rightarrow (s)(s')$ we write $P(s \rightarrow s')$.

**Definition 2.1.4. [Notation for paths in a Markov chain]** A path (of a Markov chain) is a (finite or infinite) sequence $\pi = s_0, s_1, \ldots \in (S^* \cup S^\omega)$ of states. We denote the first state of $\pi$ by first($\pi$), the $i$th state by $\pi_i$ and denote a path that starts in state $s$ as $\pi^s$. In case of $\pi$ being finite (i.e., $\pi = s_0, ..., s_{n-1}$) the length of a path is $|\pi| = n$ and the last state is obtained by last($\pi$) = $s_{n-1}$. In case of being infinite, $|\pi| = \infty$ and last($\pi$) = $\bot$ (undefined). The suffix starting with the $i$th state is denoted by $\pi^{\uparrow i}$. The set of states that are contained infinitely often in $\pi$ is denoted $\text{inf}(\pi)$.

We denote the set of finite paths of a Markov chain $\mathcal{M}$ as $\text{Paths}_{\text{fin}}^{\mathcal{M}}$ and the set of infinite paths as $\text{Paths}_{\text{inf}}^{\mathcal{M}}$.

**Definition 2.1.5. [Probability of finite paths (Markov chain)]** Given Markov chain $\mathcal{M} = (S, \rightarrow, s_0, \text{AP}, L)$ and a finite path $\pi \in \text{Paths}_{\text{fin}}^{\mathcal{M}}$, the probability of $\pi$ is defined by
Chapter 2. Preliminaries on notations, models and methods

2.1. Basic mathematical notations and Markov models

\[ P(\pi) = \prod_{i=1}^{|\pi|-1} P(\pi^{i-1} \rightarrow \pi^i). \]

Definition 2.1.6. [Markov Decision Process] A Markov Decision Process (see [Put94, Bel58, How60], MDP for short) is a tuple

\[ \mathcal{M} = (S, \text{Act}, \rightarrow, s_0, \text{AP}, L), \]

where

- \( S \) is a finite nonempty set of states,
- \( \text{Act} \) is a finite nonempty set of actions,
- \( \rightarrow : S \times \text{Act} \rightarrow \text{Distr}(S) \) is the partial transition function which may be undefined for some states and actions,
- \( s_0 \in S \) is the initial state in \( \mathcal{M} \),
- \( \text{AP} \) is the set of labels, and
- \( L : S \rightarrow 2^{\text{AP}} \) is the labeling function.

Typically, the transition function is not total such that for a state \( s \) there exist actions for which \( \rightarrow \) is not defined. For a state \( s \) the set of actions \( \text{Act}(s) \subseteq \text{Act} \),

\[ \text{Act}(s) = \{ \alpha | (s, \alpha) \text{ is defined} \} \]

is denoted as the set of enabled actions of state \( s \). The set \( \{ s | \text{Act}(s) = \emptyset \} \) contains all terminal states, i.e., the states that do not have any outgoing transition. We write \( s \xrightarrow{\alpha} \mu \) instead of \( (s, \alpha) = \mu \). The notation \( s \xrightarrow{\alpha} \mu \) means that \( s \xrightarrow{\alpha} \mu \) for some \( \alpha \) and for Dirac distributions we write \( s \xrightarrow{\alpha} s' \) instead of \( s \xrightarrow{\alpha} \mu_{s'}^1 \). A probability function \( P \) is defined by

\[ P(s \xrightarrow{\alpha} s') = \begin{cases} \mu(s') & \text{if } s \xrightarrow{\alpha} \mu \\ 0 & \text{otherwise.} \end{cases} \]

Remark: In the literature these definitions occur in quite many variations. We only remark here, that the transitions \( \rightarrow \) are often defined as a relation \( \rightarrow \subseteq S \times \text{Act} \times \text{Distr}(S) \). Then one obtains the option of dealing with multiple distributions \( \mu_i \) for a fixed action \( \alpha \) and
state \( s \), which allows for executing the same action \( \alpha \) in multiple ways in a state \( s \). For the purposes of this thesis, however, it is sufficient to deal with MDPs where for each state \( s \) and action \( \alpha \) there is at most one transition \( s \xrightarrow{\alpha} \mu \), i.e., an action \( \alpha \) is associated with at most one distribution.

Throughout the thesis we use Markov chains and Markov decision processes with a single starting state \( s_0 \in S \). In the literature often definitions can be found that use initial distributions as well as multiple starting states instead of a single initial state. This is, however, not a serious restriction and only suits well with the concepts presented in this thesis, especially the operational semantics of the specification language PROBMela and its implementation in the model checker LiQuor. The sets \( S \) and Act of states and actions used in this thesis are supposed to be finite. Infinite state systems or systems with an infinite out-degree are not a target of investigation in this thesis, but may be mentioned at some point, where a comparison, reference or connection might be of interest.

We denote the set of all distributions that are associated with an enabled action of \( s \) as

\[
\text{Steps}(s) = \{ \mu \mid s \rightarrow \mu \}.
\]

Furthermore let the sets of successors and predecessors of a state \( s \) and an action \( \alpha \) be defined as

\[
\text{Post}(s, \alpha) = \left\{ s' \in S \mid P(s \xrightarrow{\alpha} s') > 0 \right\}
\]

and

\[
\text{Pre}(s', \alpha) = \left\{ s \in S \mid P(s \xrightarrow{\alpha} s') > 0 \right\}
\]

If we are not concerned about particular actions, we just write \( \text{Post}(s) \), i.e., \( \text{Post}(s) = \bigcup_{\alpha \in \text{Act}(s)} \text{Post}(s, \alpha) \). For an action set \( A \subseteq \text{Act} \) we write

\[
\text{Post}(s, A) = \left\{ s' \in S \mid \exists \alpha \in A : P(s \xrightarrow{\alpha} s') > 0 \right\}.
\]

Notations for paths in MDPs are defined similar to paths in Markov chains, but the information about the (taken) actions is included.
Definition 2.1.7. [Notation for paths in an MDP] An infinite path (of an MDP) is an alternating infinite sequence $\pi = s_0, a_0, s_1, a_1, \ldots \in (S \times Act)^\omega$ of states and actions. A finite path is a finite sequence of states and actions $\pi \in (S \times Act)^* \times S$. We denote the first state of $\pi$ by $\pi(0) = s_0$ and the $i$th state by $\pi(i)$. A path that starts in $s$ may be written as $\pi_s$. We require that $a_i \in Act(s_i)$ and write (for better readability) $\pi = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots$. In case of a finite sequence $\pi = s_0, a_0, s_1, a_1, \ldots, s_n$ the length of a path is $|\pi| = n$ and the last state is denoted by $last(\pi) = s_n$. To refer to the suffix of $\pi$ starting with the $i$th state, we write $\pi|_i$. Similar to the definition in the context of Markov chains but under consideration of actions Act as well, we denote the set of states and actions that are contained infinitely often in a path $\pi$ as $inf(\pi)$, i.e., $inf(\pi) \subseteq S \cup Act$. Given a path $\pi = s_0, a_0, s_1, a_1, \ldots$, the pair $(T, A)$, where $T = inf(\pi)$ and $A : T \rightarrow 2^{Act}$ such that for $t \in T$ it holds that

$$A(t) = \{ \alpha \mid (s_i = t) \wedge (a_{i+1} = \alpha) \text{ for infinitely many } i \}$$

is denoted by $lim(\pi)$.

As for Markov chains the set of all finite paths of an MDP $\mathcal{M}$ is denoted by $Paths_{fin}^\mathcal{M}$ and the set of infinite paths by $Paths_{inf}^\mathcal{M}$. Finite and infinite paths that start in a state $s$ are denoted as $Paths_{fin}(s)$ and $Paths_{inf}(s)$ respectively. Let $Paths = Paths_{inf} \cup Paths_{fin}$. The set of paths that start in state $s$ will be denoted by $Paths(s) = Paths_{inf}(s) \cup Paths_{fin}(s)$.

Definition 2.1.8. [Trace of a path and LT properties] Given a labeled MDP $\mathcal{M} = \langle S, Act, \rightarrow, s_0, AP, L \rangle$ and a (finite or infinite) path $\pi = s_0, a_0, s_1, a_1, \ldots$, then the trace of $\pi$ is defined as the sequence

$$trace(\pi) = L(\pi^0), L(\pi^1), L(\pi^2), \ldots .$$

We call $E \subseteq (2^{AP})^\omega$ a linear-time property (LT property for short). An LT property $E$ is called $\omega$-regular, if there exists an $\omega$-regular expression $a$, such that $L(a) = E$. We denote a path satisfying an LT-property by $\pi \models E$ if and only if $trace(\pi) \in E$.

Definition 2.1.9. [Stutter actions] Given an MDP $\mathcal{M} = \langle S, Act, \rightarrow, s_0, AP, L \rangle$, an action $a \in Act$ is called a stutter action, if for all states $s, t \in S$ we have that $P(s \xrightarrow{a} t) > 0$ implies $L(s) = L(t)$.

Definition 2.1.10. [Stutter equivalence of paths] Given a labeled MDP $\mathcal{M}$ with labels AP, two paths $\pi_1$ and $\pi_2$ are called stutter equivalent (denoted $\pi_1 \equiv_{st} \pi_2$) if and only if there exist numbers $n_0, n_1, n_2, \ldots \in \mathbb{N}$ and $m_0, m_1, m_2, \ldots \in \mathbb{N}$ and a sequence $A_0, A_1, A_2, \ldots$, with $A_i \in 2^{AP}$ such that

$$trace(\pi_1) = A_0^{n_0}, A_1^{n_1}, A_2^{n_2}, \ldots \text{ and } trace(\pi_2) = A_0^{m_0}, A_1^{m_1}, A_2^{m_2}, \ldots .$$
We call an LT property \( E \) stutter invariant, if for all paths \( \pi, \pi' \):
\[
\pi \equiv_{st} \pi' \wedge |\pi| = E \Rightarrow |\pi'| = E.
\]

**Definition 2.1.11. [Reachable states in an MDP]** Given an MDP \( \mathcal{M} = (S, \text{Act}, \rightarrow, s_0, \text{AP}, L) \) the set of all states that can be reached through \( \rightarrow \) starting in \( s \) is denoted by
\[
\Diamond_{\mathcal{M}}(s) = \{ t \in S | \exists j \in \mathbb{N}, \pi \in \text{Paths}(s) : \pi^i \in \text{Post}(\pi^{i-1}) \text{ for } 1 \leq i \leq j \text{ and } \pi^j = t \},
\]
or simply \( \Diamond(s) \).

**Definition 2.1.12. [Probability of finite paths (MDP)]** Given Markov decision process \( \mathcal{M} = (S, \text{Act}, \rightarrow, s_0, \text{AP}, L) \) the probability of a path \( \pi \in \text{Paths}_{\text{fin}}(\mathcal{M}) \) is given by
\[
P(\pi) \overset{\text{def}}{=} \prod_{i=1}^{||\pi||} P(\pi^{i-1} \xrightarrow{\alpha_{i-1}} \pi^i).
\]

For a fixed state \( s \), an MDP may offer multiple actions to perform (i.e., \( |\text{Act}(s)| > 1 \)). This freedom of choice, the nondeterministic behaviour, represents from the modelling point of view an essential tool to model undetermined, underspecified behaviour, that can be resolved arbitrarily. However, in order to reason about probabilities of paths, this nondeterministic behaviour needs to be resolved by decision makers, which are referred to as schedulers or policies (see below).

**Definition 2.1.13. [Scheduler]** Given an MDP \( \mathcal{M} = (S, \text{Act}, \rightarrow, s_0, \text{AP}, L) \), a randomized, history-dependent scheduler (or policy) is a function
\[
U : \text{Paths}_{\text{fin}} \rightarrow \text{Distr}(\text{Act})
\]
that assigns to each finite path a distribution on the set of actions such that \( \text{support}(U(\pi)) \subseteq \text{Act}(\text{last}(\pi)) \) (see for instance [Put94]).

In the literature different classes of schedulers are distinguished. A scheduler is called memoryless, if and only if for all \( \pi \in \text{Paths}_{\text{fin}} \) it holds that \( U(\pi) = U(\text{last}(\pi)) \), i.e., the scheduler makes its decisions just on the basis of the last state. A finite memory scheduler extends this by a concept of an internal finite automaton that sets the scheduler in a specific "mode", depending on the action that was taken for \( \text{last}(\pi) \). I.e., the scheduler chooses the action depending on \( \text{last}(\pi) \) and the current mode and performs a mode-transition to a successor mode. A scheduler that yields for each path \( \pi \) only Dirac distributions is called deterministic. In this case, we write \( U(s) = \alpha \) instead of \( U(s)(\alpha) = 1 \). We denote the different sets of schedulers as follows:
Chapter 2. Preliminaries on notations, models and methods

2.1. Basic mathematical notations and Markov models

History-dependent and randomized: $\text{Sched}_{HR}$

History-dependent and deterministic: $\text{Sched}_{HD}$

Finite-memory and randomized: $\text{Sched}_{FR}$

Finite-memory and deterministic: $\text{Sched}_{FD}$

Memoryless and randomized: $\text{Sched}_{MR}$

Memoryless and deterministic: $\text{Sched}_{MD}$

Furthermore the set of all schedulers (see figure 2.1) is denoted by

$$\text{Sched} = \text{Sched}_{HR}.$$

A scheduler can by "applied" to an MDP to obtain a Markov chain:

**Definition 2.1.14. [Scheduled Markov chain]** For an MDP $\mathcal{M} = (S, \text{Act}, \rightarrow, s_0, \text{AP}, L)$ a scheduler $\mathcal{U}$ induces a Markov chain

$$\mathcal{M}_\mathcal{U} = (\text{Paths}_{\text{fin}}^\mathcal{M}, \rightarrow_\mathcal{U}, s_0)$$

where

$$P(\pi \rightarrow_\mathcal{U} \pi') = U(\pi)(\alpha) \cdot P(\text{last}(\pi) \xrightarrow{\alpha} s'),$$

with $\pi, \pi' \in \text{Paths}_{\text{fin}}^\mathcal{M}$ and $\pi' = \pi \xrightarrow{\alpha} s'$, i.e., $\pi'$ is the path resulting from concatenating path $\pi$ with action $\alpha$ and state $s'$. The paths in $\mathcal{M}_\mathcal{U}$ are called $\mathcal{U}$-paths.

**Remark:** Since the set $\text{Paths}_{\text{fin}}$ is infinite, the resulting Markov chain from Definition 2.1.14 is infinite, too. However, for the class of memoryless schedulers the resulting Markov chain can always be replaced by an equivalent finite one, because the choice of action depends only on the last state of the path. In terms of corresponding Markov chains that are induced for an MDP $\mathcal{M}$ the classes of schedulers for $\mathcal{M}$ form a hierarchy that is sketched in figure 2.1.

![Figure 2.1: Hierarchy of schedulers](image-url)
The set of (finite, infinite and in general) paths that is obtained from an MDP \( \mathcal{M} \) by applying a scheduler \( \mathcal{U} \) in the way described above is denoted as \( (\text{Paths}_{\text{fin}}^{\mathcal{M}_\mathcal{U}}, \text{Paths}_{\text{inf}}^{\mathcal{M}_\mathcal{U}}, \text{Paths}^{\mathcal{M}_\mathcal{U}}) \). If the context (i.e., \( \mathcal{M} \)) is obvious, we omit the superscript \( \mathcal{M} \).

For the sake of completeness, the standard concepts of reasoning about probabilities of path events in Markov chains are briefly recalled. In particular it is defined formally what is meant by stating, that an LT-property is satisfied with a certain probability. After this the corresponding problem in the context of Markov decision processes is addressed.

**Definition 2.1.15. [Basic cylinder]** Given a Markov chain \( \mathcal{M} = (S, \rightarrow, s_0, AP, L) \), and a path \( \pi \in \text{Paths}_{\text{fin}}^{\mathcal{M}} \) the basic cylinder of \( \pi \) is defined as
\[
\Delta(\pi) = \{ \rho \in \text{Paths}_{\text{inf}}^{\mathcal{M}} | \rho[|\pi|] = \pi \}.
\]

**Definition 2.1.16. [Probability space of a Markov chain]** Given a discrete Markov chain \( \mathcal{M} = (S, \rightarrow, s_0, AP, L) \), a probability space \( \Psi = (\Delta^M, \Pr^M) \), is defined such that
\[
\Delta^M \text{ is the } \sigma \text{-algebra generated by the set of basic cylinders over } \mathcal{M}.
\]
\[
\Pr^M \text{ is the uniquely determined probability measure which satisfies the following:}
\]
\[
\Pr^M(\Delta(s_0, s_1, \ldots, s_n)) = P(s_0 \rightarrow s_1) \cdot P(s_1 \rightarrow s_2) \cdot \ldots \cdot P(s_{n-1} \rightarrow s_n)
\]

As mentioned before, a scheduler \( \mathcal{U} \in \text{Sched} \) induces a (possibly infinite) Markov chain \( \mathcal{M}_\mathcal{U} = (\text{Paths}_{\text{fin}}^{\mathcal{M}}, \rightarrow_{\mathcal{U}}, s_0) \) when applied to a Markov Decision process \( \mathcal{M} \). Note that the states in \( \mathcal{M}_\mathcal{U} \) are now paths in \( \mathcal{M} \), the paths that \( \mathcal{U} \) observes when resolving nondeterministic behaviour. Let \( \Delta^{\mathcal{M}_\mathcal{U}} \) be the \( \sigma \)-algebra generated by the empty set and the set of basic cylinders over \( \mathcal{M}_\mathcal{U} \). Then \( \Pr^{\mathcal{M}_\mathcal{U}} \) denotes the probability measure of \( \Pr^{\mathcal{M}_\mathcal{U}} \), i.e.,
\[
\Pr^{\mathcal{M}_\mathcal{U}}(\Delta^{\mathcal{M}_\mathcal{U}}(\pi^0, \pi^1, \ldots, \pi^n)) = P(\pi^0 \rightarrow_{\mathcal{U}} \pi^1) \cdot P(\pi^1 \rightarrow_{\mathcal{U}} \pi^2) \cdot \ldots \cdot P(\pi^{n-1} \rightarrow_{\mathcal{U}} \pi^n).
\]

In this thesis, the measure function \( \Pr \) will be used for yielding the probability measure for paths of Markov chains respectively scheduler-associated paths in Markov decision processes. As defined above, \( \Pr^{\mathcal{M}} \) and \( \Pr^{\mathcal{M}_\mathcal{U}} \) are functions over measurable sets of paths that
correspond to a Markov Chain or a Markov decision process $\mathcal{M}$ and a scheduler $\mathcal{U}$. For a measurable set of paths $P \subseteq \text{Paths}$ and $s \in S$ we define $\Pr_{s}^{\mathcal{M}\mathcal{U}}(P) = \Pr_{s}^{\mathcal{M}\mathcal{U}}\{\pi \in P : \pi^0 = s\}$ and seek to find

$$\sup_{\mathcal{U} \in \text{Sched}} \Pr_{s}^{\mathcal{M}\mathcal{U}}(P), \quad \text{and} \quad \inf_{\mathcal{U} \in \text{Sched}} \Pr_{s}^{\mathcal{M}\mathcal{U}}(P).$$

**Definition 2.1.17. [Stutter-equivalent MDPs]** Given two MDPs $\mathcal{M}_1$ and $\mathcal{M}_2$ with the same set of atomic propositions $\text{AP}$, then $\mathcal{M}_1$ and $\mathcal{M}_2$ are called *stutter-equivalent*, denoted $\mathcal{M}_1 \equiv_{\text{st}} \mathcal{M}_2$, if for each scheduler $\mathcal{U}_1$ of $\mathcal{M}_1$ there exists a scheduler $\mathcal{U}_2$ of $\mathcal{M}_2$ such that $\Pr_{\mathcal{M}_1\mathcal{U}_1}(E) = \Pr_{\mathcal{M}_2\mathcal{U}_2}(E)$ for all stutter-invariant measurable LT-properties $E \subseteq (2^{\text{AP}})^{\omega}$, and vice versa.

2.2. Quantitative analysis of Markov decision processes

For providing a short overview on all central concepts addressed in this thesis, we briefly summarize the main steps of the automata-based approach for a quantitative analysis of MDPs against LT-properties. This section is organized as follows: First we recall a central theorem for calculating the probability of reaching a goal set in a Markov chain and then we describe the general approach for calculating the worst case and best case (i.e., the minimal and maximal) probabilities for reaching a goal set in an MDP (section 2.2.1), which is equivalent to quantitative model checking of reachability properties. In practice, the problem involves solving a linear program, i.e., optimizing a linear objective function with respect to a linear inequality system [Put94, CY95, dA97]. Based on this, in section 2.2.2, model checking for $\omega$-regular LT properties using the automata-theoretic approach is outlined, which is a central aspect of the works [VW86, CY95, dA97]. Here, the concept of accepting (maximal) end components as one of the central elements in model checking Markov decision processes against $\omega$-regular LT properties is recalled. Given an MDP $\mathcal{M}$ and an $\omega$-automaton $\mathcal{A}$, end components of $\mathcal{M} \otimes \mathcal{A}$, certain subgraphs that are reached eventually with probability 1 in an MDP, are identified and checked for acceptance with respect to an $\omega$-regular LT property, that is given in terms of an automaton component $\mathcal{A}$. With this, the problem of calculating extremal probabilities for satisfying an $\omega$-regular LT property is reduced to the problem of reaching a goal set in the MDP.

2.2.1. Probabilities for reaching goal sets in Markovian models

Before discussing the calculation of extremal probabilities for reaching a goal set in a given MDP, the context for Markov chains is outlined (see, e.g., [KS60]). The core aspects of
these concepts reappear in PCTL model checking of Markov chains [HJ94] and can be seen as a basis for the MDP case.

Given a DTMC $\mathcal{M} = (S, \rightarrow, s_0, \mathcal{AP}, L)$, we fix an arbitrary "goal set" $B \subseteq S$ and ask, what the probability of reaching $B$ might be. For a state $s \in S$ and with the LTL-like notation
\[ \Diamond B = \{ \pi \in \text{Paths}^{\mathcal{M}}_{\inf} | \exists i \in \mathbb{N} : \pi^i \in B \}, \]
the probability for eventually reaching $B \subseteq S$ is defined by
\[
\Pr_s^{\mathcal{M}}(\Diamond B) = 1
\]
if $s \in B$, and furthermore it follows, that if $s \notin B$, it holds that
\[
\Pr_s^{\mathcal{M}}(\Diamond B) = \Pr(\{ \pi \in \text{Paths}^{\mathcal{M}}_{\inf}(s) | \exists i \text{ s.t. } \pi^i \in B \}).
\]
Then, $\Pr_s^{\mathcal{M}}(\Diamond B) = 1$ if $s \in B$, and furthermore it follows, that if $s \notin B$, it holds that
\[
\Pr_s^{\mathcal{M}}(\Diamond B) = \Pr(\{ \pi | \exists i > 0 \text{ s.t. } \pi^i \in B \wedge \pi^{i+2} = s, t \})
\]
\[
= \sum_{t \in S} \Pr(\{ \pi | \exists i > 0 \text{ s.t. } \pi^i \in B \wedge \pi^{i+2} = s, t \})
\]
\[
= \sum_{t \in S} \sum_{i=0}^{\infty} \sum_{s_i \in B} \Pr(\Delta(s, t, s_2, ..., s_i))
\]
\[
= \sum_{t \in S} \sum_{i=0}^{\infty} \sum_{s_i \in B} \Pr(\Delta(s, t, s_2, ..., s_i))
\]
\[
= \sum_{t \in S} \sum_{i=0}^{\infty} \Pr(\Delta(t, s_2, ..., s_i))
\]
\[
= \sum_{t \in S} \Pr(s \rightarrow t) \cdot \Pr_s^{\mathcal{M}}(\Diamond B)
\]
This leads to a linear equation system that can also be used as a starting point in practical approaches for determining probabilities for reachability analysis in $\mathcal{M}$ (see, e.g., [KS60]).

**Theorem 2.2.1** (Probability of reaching a goal set). Given a labeled Markov chain $\mathcal{M} = (S, \rightarrow, s_0, \mathcal{AP}, L)$, a goal set $B \subseteq S$, let $(x_s)_{s \in S}$, $0 \leq x_s \leq 1$ be the variables of the following linear equation system:

- If $s \in B$ then $x_s = 1$,
• if \( s \notin B \), then
  \[
x_s = \sum_{t \in S} P(s \rightarrow t) \cdot x_t.
  \]

With \((\bar{x}_s \in S)\) being the least solution of this equation system, it holds that

\[
\Pr^M_s(\Diamond B) = \bar{x}_s.
\]

In order to obtain an equation system that has a unique solution, it is sufficient to set \( x_s = 0 \) if \( B \cap \Diamond \mathcal{M}(s) = \emptyset \), i.e., \( x_s = 0 \) for all corresponding states \( s \) from which the goal set \( B \) is unreachable.

This notion of the numerical problem, which yields the solution for the reachability problem in Markov chains can be extended to the reachability problem in MDPs (see, e.g., [Der70, Put94]). Since MDPs additionally allow for nondeterministic choices between enabled actions \( \text{Act}(s) = \{ \alpha \mid \rightarrow (s, \alpha) \text{ is defined} \} \) in a state \( s \), that are resolved by schedulers, the question of finding the probability for reaching a goal set amounts to finding the supremum and infimum of these probabilities taken over all possible schedulers. Formally, for a scheduler \( U \in \text{Sched} \) that is applied to an MDP \( M \) we write \( \Pr^M_{sU}(\Diamond B) \) and focus on finding the supremum and infimum

\[
\sup_{U} \Pr^M_{sU}(\Diamond B) \quad \text{and} \quad \inf_{U} \Pr^M_{sU}(\Diamond B).
\]

For this supremum it holds that

\[
\sup_{U \in \text{Sched}} \Pr^M_{sU}(\Diamond B) = \max_{U \in \text{Sched}} \Pr^M_{sU}(\Diamond B) = \max_{U \in \text{Sched} \cup \text{MD}} \Pr^M_{sU}(\Diamond B). \quad (2.1)
\]

Similarly, the following can be observed for the infimum:

\[
\inf_{U \in \text{Sched}} \Pr^M_{sU}(\Diamond B) = \min_{U \in \text{Sched}} \Pr^M_{sU}(\Diamond B) = \min_{U \in \text{Sched} \cup \text{MD}} \Pr^M_{sU}(\Diamond B). \quad (2.2)
\]

We write

\[
\Pr^{\text{max}}_s(\Diamond B) \overset{\text{def}}{=} \max_{U \in \text{Sched} \cup \text{MD}} \Pr^M_{sU}(\Diamond B)
\]

and

\[
\Pr^{\text{min}}_s(\Diamond B) \overset{\text{def}}{=} \min_{U \in \text{Sched} \cup \text{MD}} \Pr^M_{sU}(\Diamond B).
\]

Calculating the values of \( \Pr^{\text{max}}_s(\Diamond B) \) and \( \Pr^{\text{min}}_s(\Diamond B) \) leads to the following systems of inequations:

**Theorem 2.2.2** (System of inequations for maximal reachability probabilities). Let \( M \) be an MDP with states \( S \), \( s \in S \) and \( B \subseteq S \) and \((x_s)_{s \in S}, 0 \leq x_s \leq 1\) be the variables of the following system of inequations:
2.2. Quantitative analysis of Markov decision processes

Chapter 2. Preliminaries on notations, models and methods

- If $s \in B$, then $x_s = 1$.
- If $s \notin B$, then
  \[
  x_s = \max \left\{ \sum_{t \in S} P(s \xrightarrow{\alpha} t) \cdot x_t \mid \alpha \in \text{Act}(s) \right\}.
  \]

Let $(\bar{x}_s)_{s \in S}$ be the least solution of these inequations. Then it holds, that

\[
Pr_{s}^{\text{max}}(\Diamond B) = \bar{x}_s.
\]

Again, this system of inequations can be transformed into one with a unique solution if variables $x_s$ are set to 0 for all $s$ such that $B \cap \Diamond \mathcal{M}(s) = \emptyset$.

To compute the values $Pr_{s}^{\text{max}}(\Diamond B)$ in practice, the inequation system in Theorem 2.2.2 can be rephrased as the following linear program

- If $s \in B$, then $x_s = 1$.
- Optional: if $B \cap \Diamond \mathcal{M}(s) = \emptyset$ then $x_s = 0$
- If $s \notin B$, then $0 \leq x_s \leq 1$ and for all actions $\alpha \in \text{Act}(s)$:
  \[
  x_s \geq \sum_{t \in S} P(s \xrightarrow{\alpha} t) \cdot x_t
  \]

With the objective to minimize $\sum_{s \in S} x_s$ the vector $(x_s)_{s \in S}$ with $x_s = Pr_{s}^{\text{max}}(\Diamond B)$ yields a solution for this linear program. The value for $Pr_{s}^{\text{max}}(\Diamond B)$ is the solution value calculated for variable $x_s$. Let $S_\gamma = S \setminus B$. Then one can rewrite the second item as

\[
(1 - P(s \xrightarrow{\alpha} s)) \cdot x_s - \sum_{t \in S_\gamma \setminus \{s\}} P(s \xrightarrow{\alpha} t) \cdot x_t \geq P(s \xrightarrow{\alpha} B)
\]

where $P(s \xrightarrow{\alpha} B) = \sum_{t \in B} P(s \xrightarrow{\alpha} t)$. Thus, the second item in the above theorem can be reformulated as a linear inequality $A \cdot x \geq b$ where $x$ is the vector $(x_s)_{s \in S}$, and $A$ is a matrix with a row for each pair $(s, \alpha)$ with $s \in S_\gamma$ and $\alpha \in \text{Act}(s)$, two extra rows for each state $s \in S_\gamma$ to represent the inequality $0 \leq x_s \leq 1$ and a column for each state $s \in S_\gamma$.

Similar to the case $Pr_{s}^{\text{max}}(\Diamond B)$ the case $Pr_{s}^{\text{min}}(\Diamond B)$ leads to an equation system, but an additional side constraint has to be made concerning states $s$, that are not connected to the set $B$ in the graph of the MDP:

**Theorem 2.2.3** (System of inequations for minimal reachability probabilities). Let $\mathcal{M}$ be an MDP with states $S$, $s \in S$ and $B \subseteq S$ and $(x_s)_{s \in S}$, $0 \leq x_s \leq 1$ be the variables of the following system of inequations:
Chapter 2. Preliminaries on notations, models and methods

2.2 Quantitative analysis of Markov decision processes

• If \( B \cap \diamond^M(s) = \emptyset \), then \( x_s = 0 \).

• If \( s \notin B \) and \( B \cap \diamond^M(s) \neq \emptyset \), then

\[
    x_s = \min \left\{ \sum_{t \in S} P(s \xrightarrow{\alpha} t) \cdot x_t \mid \alpha \in \text{Act}(s) \right\}.
\]

Let \( (\bar{x}_s)_{s \in S} \) be the greatest solution of these inequations. Then it holds, that

\[
    \Pr_s^{\min}(\diamond B) = \bar{x}_s.
\]

Similar to the max-case it is possible to force the existence of a unique solution by setting \( x_s = 1 \) for all \( s \in B \).

With the objective to maximize \( \sum_{s \in S} x_s \) the vector \( (x_s)_{s \in S} \) with \( x_{\Pr_s^{\min}(\diamond B)} = \bar{x}_s \) yields the solution of the following linear program:

• If \( B \cap \diamond^M(s) = \emptyset \), then \( x_s = 0 \).

• Optional: if \( s \in B \), then \( x_s = 1 \).

• If \( s \notin B \) and \( B \cap \diamond^M(s) \neq \emptyset \), then

\[
    0 \leq x_s \leq 1 \quad \text{and for all actions } \alpha \in \text{Act}(s):
\]

\[
    x_s \leq \sum_{t \in S} P(s \xrightarrow{\alpha} t) \cdot x_t.
\]

Identifying again the states \( s \) such that the value \( x_s = \Pr_s^{\min}(\diamond B) \) is not fixed to 1 or 0 by the first two items of the linear program as \( S_2 = S \setminus B \), one can rewrite the third item as

\[
    (1 - P(s \xrightarrow{\alpha} s)) \cdot x_s - \sum_{t \in S \setminus \{s\}} P(s \xrightarrow{\alpha} t) \cdot x_t \leq P(s \xrightarrow{\alpha} B)
\]

where \( P(s \xrightarrow{\alpha} B) = \sum_{t \in B} P(s \xrightarrow{\alpha} t) \). Again, the third item in the above theorem can be rewritten as a linear inequality \( A \cdot x \leq b \) where \( x \) is the vector \( (x_s)_{s \in S_2} \) and \( A \) is a matrix with a row for each pair \( (s, \alpha) \) with \( s \in S_2 \) and \( \alpha \in \text{Act}(s) \), two extra rows for each state \( s \in S_2 \) to represent the inequality \( 0 \leq x_s \leq 1 \) and a column for each state \( s \in S_2 \).

Approximate or precise solutions of the linear programs from Theorems 2.2.2 and 2.2.3 can be computed by standard algorithms to solve linear programs, e.g., the simplex algorithm or other methods.
2.2.2. Extremal probabilities for \( \omega \)-regular properties

In this section, quantitative model checking of MDPs against \( \omega \)-regular properties will be targeted [VW86, CY95, dA97]. In contrast to calculating extremal probabilities of (only) reaching a goal set \( B \subseteq S \) in an MDP, this requires a more general approach, where eventually the process of calculating extremal probabilities for an MDP satisfying an \( \omega \)-regular property is reduced to the reachability case described in the section above. More precisely, recall that a path \( \pi \in \text{Paths}^\mathcal{M} \) of a given MDP \( \mathcal{M} \) satisfies an \( \omega \)-regular property \( E \subseteq (2^{\text{AP}})^\omega \), written as \( \pi \models E \), if and only if \( \text{trace}(\pi) \in E. \) Thus, we seek to compute

\[
\sup_{U \in \text{Sched}} \Pr\{ \{ \pi \in \text{Paths}^\mathcal{M}_U(s) \mid \pi \models E \}\}
\]

and

\[
\inf_{U \in \text{Sched}} \Pr\{ \{ \pi \in \text{Paths}^\mathcal{M}_U(s) \mid \pi \models E \}\}.
\]

The method, that will be described in this section to derive these values, is known as the automata-theoretic approach [VW86, CY95, dA97]. Essentially, the automata-theoretic approach involves the identification of particular subgraphs in the cross product \( \mathcal{M} \otimes A \) of an \( \omega \)-automaton \( A \) and an MDP \( \mathcal{M} \), namely the accepting maximal end components. The union of states in these end components then form a goal set \( B \subseteq S \), for which solving the reachability problems \( \Pr_{s_0}^{\max}(\Diamond B) \) and \( \Pr_{s_0}^{\min}(\Diamond B) \) as outlined in the previous section 2.2.1 yield the desired values. For an elaborate background on \( \omega \)-automata and their properties the reader might refer, e.g., to [GTW02].

**Definition 2.2.4.** [\( \omega \)-automaton] A (nondeterministic) \( \omega \)-automaton is a tuple

\[
\mathcal{A} = (Q, \Sigma, \delta, Q_0, \text{Acc}),
\]

where

- \( Q \) is a finite nonempty set of automaton states,
- \( \Sigma \) is a finite alphabet,
- \( \delta : Q \times \Sigma \to 2^Q \) is the automaton transition function, and
- \( Q_0 \subseteq Q \) is the set of initial states, and
- \( \text{Acc} \) is the acceptance condition of \( \mathcal{A} \) which is one from the following forms:

  - **Büchi**: \( \text{Acc} \subseteq Q \), a set \( T \subseteq Q \) is called Büchi-accepting, denoted \( T \succeq_{\text{Büchi/Acc}}, \) if and only if \( T \cap \text{Acc} \neq \emptyset. \)
Chapter 2. Preliminaries on notations, models and methods

2.2. Quantitative analysis of Markov decision processes

– **Rabin:** \( \text{Acc} = \{ (L_1, U_1), ..., (L_n, U_n) \} \), \( L_i, U_i \subseteq Q \) for all \( 1 \leq i \leq n \). A set \( T \subseteq Q \) is called Rabin-accepting, denoted \( T \vdash_{\text{Rabin Acc}} \), if and only if
\[
\exists i : T \cap L_i \neq \emptyset \text{ and } T \cap U_i = \emptyset.
\]

– **Streett:** \( \text{Acc} = \{ (L_1, U_1), ..., (L_n, U_n) \} \), \( L_i, U_i \subseteq Q \) for all \( 1 \leq i \leq n \). A set \( T \subseteq Q \) is called Streett-accepting, denoted \( T \vdash_{\text{Streett Acc}} \), if and only if
\[
\forall i : T \cap L_i \neq \emptyset \Rightarrow T \cap U_i \neq \emptyset.
\]

Depending on the type of acceptance condition, we have a Büchi, Rabin, or Streett automaton. An \( \omega \)-automaton is called deterministic, if \( |Q_0| = 1 \) and \( |\delta(q, \sigma)| \leq 1 \) for all \( q \in Q \) and \( \sigma \in \Sigma \). We require that any \( \omega \)-automaton is complete, by which we mean that for all \( q \in Q \) and \( \sigma \in \Sigma \) it holds that \( |\delta(q, \sigma)| \geq 1 \).

Similar to paths in MDPs we define **runs** of an automaton and under which conditions a run is **accepting**.

**Definition 2.2.5. [Notations for runs and the language of an automaton]** A run of an \( \omega \)-automaton \( A = (Q, \Sigma, \delta, Q_0, \text{Acc}) \) is an infinite sequence \( q_0, q_1, ... \in Q^\omega \) such that for all \( i \geq 0 \) we have \( q_{i+1} \in \delta(q_i, \sigma) \) for some \( \sigma \in \Sigma \). For a word \( w \in \Sigma^\omega \) the corresponding run \( \rho_w \) is the sequence \( \rho_w = q_0, q_1, ... \in Q^\omega \), where \( (\rho_w, w, \rho_{w+1}) \in \delta \). The set of automaton states that is contained infinitely often in \( \rho_w \) is denoted \( \text{inf}(\rho_w) \) and run \( \rho_w \) is called accepting, if \( \text{inf}(\rho_w) \vdash_{\text{Acc}} \). We denote the language of \( A \) as \( L(A) = \{ w \in \Sigma^\omega \mid \text{inf}(\rho_w) \vdash_{\text{Acc}} \} \).
2.2. Quantitative analysis of Markov decision processes

Chapter 2. Preliminaries on notations, models and methods

with an acceptance condition $\text{Acc} = \{(L_1, U_1), \ldots, (L_n, U_n)\}$. Furthermore, we write $\mathcal{A}_{\text{Streett}}$ if this acceptance condition is interpreted according to Streett semantics and $\mathcal{A}_{\text{Rabin}}$ when Rabin semantics are applied. Then it holds for deterministic automata, that $\mathcal{L}(\mathcal{A}_{\text{Rabin}}) = \Sigma^\omega \setminus \mathcal{L}(\mathcal{A}_{\text{Streett}})$ (and vice versa).

For a consistent notation for Büchi, Rabin and Streett acceptance conditions it is possible to denote a Büchi acceptance condition $\text{Acc}$ as being of type Rabin with $\text{Acc}' = \{\emptyset\}$ or being of type Streett with $\text{Acc}'' = \{\emptyset\}$.

Complexity: The comparably easy complementation step for deterministic Rabin and Streett automata does not spoof the general hierarchy of complexity, since the transformation of NBA to DRA or DSA itself involves a $2^{\log |Q|}$ step in the worst case ([Löd99], see also [Löd98]).

In the following we recall basic concepts of automaton-based model checking:

Definition 2.2.6. [Cross product of an MDP and a deterministic $\omega$-automaton] Given an MDP $\mathcal{M} = (S, \text{Act}, \rightarrow, s_0, \text{AP}, L)$ and a deterministic $\omega$-automaton $\mathcal{A} = (Q, 2^{\text{AP}}, \delta, q_0, F)$, the cross product $\mathcal{M} \otimes \mathcal{A}$ is defined as the MDP

$\mathcal{M} \otimes \mathcal{A} = (S \times Q, \text{Act}, \rightarrow_{\mathcal{M} \otimes \mathcal{A}}, s_0^{\mathcal{M} \otimes \mathcal{A}}, L)$,

where $s_0^{\mathcal{M} \otimes \mathcal{A}} = \langle s_0, \delta(q_0, L(s_0)) \rangle$, and

$\rightarrow_{\mathcal{M} \otimes \mathcal{A}}: S \times Q \times \text{Act} \rightarrow \text{Distr}(S \times Q)$

such that $\langle s, q \rangle \rightarrow_{\mathcal{M} \otimes \mathcal{A}} \nu$, with $\nu((s', q')) = P(s \rightarrow s')$ and $\{q'\} = \delta(q, L(s'))$. In unambiguous situations we simply write $\rightarrow$ instead of $\rightarrow_{\mathcal{M} \otimes \mathcal{A}}$.

The cross product $\mathcal{M} \otimes \mathcal{A}$ is itself an MDP over states in $S \times Q$, and since $\mathcal{A}$ is deterministic, for any path $\pi$ in $\mathcal{M}$ there is a unique path $\hat{\pi}$ in $\mathcal{M} \otimes \mathcal{A}$ that resembles $\pi$ in its MDP-state components. On the other hand, through the automaton component $\rho_{\pi} \in Q^\omega$ of $\hat{\pi}$, which is actually a run in $\mathcal{A}$, the path $\pi$ is "tested" against the automaton’s acceptance condition. Hence, the fundamental connection of paths in $\mathcal{M}$ and an LT-property $E$ is as follows:

Let $E$ be an $\omega$-regular LT-property and $\mathcal{A}_E$ with alphabet $\Sigma = 2^{\text{AP}}$ its representation as a deterministic $\omega$-automaton. Then, for a path $\pi \in S^\omega$ in $\mathcal{M}$ it holds that:

$\pi \models E$ iff $\text{trace}(\pi) \in E$

iff $\text{trace}(\pi) \in \mathcal{L}(\mathcal{A}_E)$

iff $\rho_{\pi}$ is accepting in $\mathcal{A}_E$

iff $\inf(\rho_{\pi}) \models \text{Acc}$.

Figure 2.2 shows an example, where labelings are displayed using brackets (see states $s_2$ of $\mathcal{M}$ and $\langle s_2, q_1 \rangle$ of $\mathcal{M} \otimes \mathcal{A}$). The Rabin acceptance condition $\text{Acc} = \{\{q_1\}, \emptyset\}$ is displayed.
with a boxed state \( q_1 \). The state \( \langle s_2, q_1 \rangle \) is also drawn as "accepting", though in \( \mathcal{M} \otimes \mathcal{A} \) the acceptance condition is not preserved. It is just useful to identify the accepting states of the automaton component of \( \mathcal{M} \otimes \mathcal{A} \).

In the following paragraphs, the key concept of quantitative model checking of MDPs against deterministic \( \omega \)-automata is recalled. Particular subgraphs of MDPs, called end components \([CY95, dA97]\) suit to serve as structural components of infinite behaviour in MDPs, similar to bottom strongly connected components (BSCCs) in finite Markov chains. BSCCs in finite Markov chains, i.e., strongly connected components that cannot be left once they are entered, are eventually reached almost surely. More precisely, the set of paths that eventually reach a BSCC have measure 1. Similar to this, end components are reached with probability 1 in any MDP. Thus, end components in a crossproduct \( \mathcal{M} \otimes \mathcal{A} \) can serve as a structural basis for checking against an acceptance condition.

**Definition 2.2.7. [Subgraph, end components]** Given MDP \( \mathcal{M} = (S, \text{Act}, \rightarrow, s_0, \text{AP}, L) \) a subgraph of \( \mathcal{M} \) is defined as a tuple \( \mathcal{E} = (T, \text{Act}_E) \), where the following conditions hold:

1. \( \emptyset \neq T \subseteq S \),
2. \( \text{Act}_E : T \rightarrow 2^\text{Act} \) such that for all \( t \in T \) we have \( \emptyset \neq \text{Act}_E(t) \subseteq \text{Act}(t) \), and
3. If \( t \in T, \alpha \in \text{Act}_E, \) and \( \mathbf{P}(t \xrightarrow{\alpha} u) > 0 \) for \( u \in S \), then \( u \in T \).

An end component is a subgraph, (4.) where the underlying digraph \( G_E = (T, E) \) with

\[
(s, s') \in E \iff \exists \alpha \in \text{Act}_E(s) \text{ such that } \mathbf{P}(s \xrightarrow{\alpha} s') > 0
\]

is strongly connected. An end component is denoted maximal if \( \mathcal{E} \) cannot be extended by any \( s \in S \) or an action \( \alpha \in \text{Act} \) without violating any of conditions 2.2.7(1)-(4). I.e., \( \mathcal{E} \) is maximal if for each end component \( \mathcal{E}' = (T', \text{Act}_{E'}) \) with \( T \subseteq T' \) and \( \text{Act}_E(t) \subseteq \text{Act}_{E'}(t) \),
for all \( t \in T \) we have \( \mathcal{E} = \mathcal{E}' \). We denote the set of all maximal end components as \( \mathcal{MEC} \).

Some algorithms used in this thesis deal with identifying end components, often under additional constraints. For this it may well be the case that we have a subgraph \( \mathcal{E} = (T, \text{Act}_E) \) for which it is yet to be determined whether it actually is an end component, or not. Such a subgraph is called a candidate end component, or just candidate. Also, some notations about end components that are contained in some (larger) subgraph are necessary. For this we define for a given subgraph \( \mathcal{E} = (T, \text{Act}_E) \) and an end component \( \mathcal{E}' = (T', \text{Act}_{E'}) \), that \( \mathcal{E}' \) is a sub-end component of \( \mathcal{E} \) if and only if \( T' \subseteq T \) and \( \text{Act}_{E'} \subseteq \text{Act}_E \).

As stated above, the concept of end components is the central feature for quantitative model checking of Markov decision processes. The following two theorems and two lemmas are most important in this context. The first theorem states that given an MDP, there is no scheduler that manages to avoid all end components forever. Almost surely, any scheduler is eventually forced into an end component and is forced to remain there.

**Theorem 2.2.8** (Central theorem about end components [JA97]). In an MDP every path will finally "end up" in an end component with probability 1, i.e., for all MDPs \( \mathcal{M} \), schedulers \( \mathcal{U} \) and states \( s \in S \) it holds that

\[
\Pr_s^{\mathcal{M}_\mathcal{U}}(\{ \pi \in \text{Paths}(s) \mid \lim(\pi) \text{ forms an end component } \}) = 1.
\]

The following lemma states that for any end component \( \mathcal{E} \) there exists a scheduler that visits each state in \( \mathcal{E} \) with probability 1 when starting at some state in \( \mathcal{E} \).

**Lemma 2.2.9.** For any end component \( \mathcal{E} = (T, \text{Act}_E) \) there exists a scheduler \( \mathcal{U} \) with finite memory such that

\[
\Pr_s^{\mathcal{M}_\mathcal{U}}(\{ \pi \in \text{Paths} : \lim(\pi) = T \}) = 1
\]

for any state \( s \in T \).

**Proof.** In each state \( s \in T \) the scheduler \( \mathcal{U} \) chooses the next action deterministically from the available actions, that lead to a successor state \( t \in T \). To achieve \( \lim(\pi) = T \) the scheduler chooses between the actions in a round robin fashion using its finite-memory "modes".

The following theorem states that for any scheduler \( \mathcal{U} \) that reaches a particular end component \( \mathcal{E} \) with a certain probability, there always exists a scheduler \( \mathcal{U}' \) that reaches this end component with the same probability but then manages to stay inside this end component forever.
Chapter 2. Preliminaries on notations, models and methods  

2.2. Quantitative analysis of Markov decision processes

**Theorem 2.2.10** (Stability of end components). Let $E = (T, \text{Act}_E)$ be an end component of an MDP $M = (S, \text{Act}, \rightarrow, s_0, AP, L)$. For every scheduler $U$ there exists a scheduler $U'$ with finite memory such that for all $s \in S$ it holds that

$$\Pr_s^{\text{max}}(\diamond T) = \Pr_s^{M_U}(\diamond T) = \Pr_s^{M_{U'}}(\diamond T) = \Pr_s^{M_{U'}}\left(\{\pi \in \text{Paths} | \lim(\pi) = T\}\right).$$

**Proof.** As long as $T$ is not reached, $U'$ behaves exactly like $U$. When $\pi$ enters $T$ the scheduler behaves as described in Lemma 2.2.9. 

From this Theorem and Lemma 2.2.9 a second lemma follows immediately that forms the basis of many argumentations in quantitative model checking of MDPs. It states that there exists a scheduler that, once it reaches an end component $E$, it manages to reach some sub-end component of $E$ and furthermore stay inside this sub-end component.

**Lemma 2.2.11.** For any end component $E = (T, \text{Act}_E)$ and sub-end component $E' = (T', \text{Act}_{E'})$ of $E$, there exists a finite-memory scheduler $U$ such that

$$\Pr_s^{M_U}(\{\pi \in \text{Paths} : \lim(\pi) = E'\}) = 1$$

for any state $s \in T$.

In the following the two concepts of end components and deterministic $\omega$-automata are combined. Acceptance of end components of the cross product $M \otimes A$ will be defined such that the maximum probability of satisfying an $\omega$-regular property in $M$ equals the maximum probability to reach an accepting end component in $M \otimes A$, where $A$ represents the property (see Theorem 2.2.14). The case of minimal probabilities is covered by a reduction to the case of maximal probabilities (see Theorem 2.2.13).

**Definition 2.2.12. [Accepting end components]** Given an MDP $M = (S, \text{Act}, \rightarrow, s_0, AP, L)$ and a deterministic $\omega$-automaton $A = (Q, \Sigma, \delta, Q_0, \text{Acc})$, an end component $E = (T, \text{Act}_E)$ of $M \otimes A$ is called accepting if and only if $T|_Q \rightarrow \text{Acc}$, where

$$T|_Q = \{q \in Q | \exists s \in S : (s, q) \in T\};$$

that is, the projection of the end component’s automaton component.

We denote as AEC the union of the state component of all accepting end components in a cross product MDP $M \otimes A$, i.e.,

$$\text{AEC} = \{(s, q) | (s, q) \in T \text{ for at least one accepting end component } E = (T, \text{Act}_E)\}.$$ 

The following example, revisited from figure 2.2, illustrates this.
The example MDP fragment in figure 2.3 contains the (only) accepting state \( (s_2, q_1) \) and it is assumed a Rabin automaton with acceptance condition \( Acc = \{(\{q_1\},\emptyset)\} \) (see figure 2.2). It contains the following end components (instead of the actual functions \( Act_{\varepsilon_i} \) for \( 1 \leq i \leq 3 \) the actions are displayed for each end component):

\[
\begin{align*}
E_1 &= (\{s_0, s_1, s_2\}, \{\alpha, \beta, \gamma\}) \\
E_2 &= (\{s_0, s_3\}, \{\delta, \epsilon\}) \\
E_3 &= (\{s_0, s_1, s_2, s_3\}, \{\alpha, \beta, \gamma, \delta, \epsilon\}).
\end{align*}
\]

We then have \( AEC = \{s_0, s_1, s_2\} \).

**Theorem 2.2.13** (Correlation between \( Pr_{s}^{\text{max}} \) and \( Pr_{s}^{\text{min}} \)). Given an MDP \( M = (S, Act, \rightarrow, s_0, AP, L) \) and an \( \omega \)-regular property \( E \), it holds that

\[
Pr_{s}^{\text{min}}(E) = 1 - Pr_{s}^{\text{max}}(\neg E).
\]

**Theorem 2.2.14** (Extremal probabilities for satisfying \( \omega \)-regular properties). The extremal probabilities \( Pr_{s}^{\text{max}}(E) \) and \( Pr_{s}^{\text{min}}(E) \) of an MDP \( M \) satisfying an \( \omega \)-regular property \( E \) when starting in state \( s \) are obtained by:

\[
Pr_{s}^{\text{max}}(E) = Pr_{s}^{\text{max}}(\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\n
2.3. Graph-based preprocessing

That the inequation systems for the MDP reachability problem (see section 2.2.1), that are somewhat naturally derived from the linear equation system for Markov chains, may indeed have multiple solutions, from which the vectors \( \bar{x}_s = \min \bar{x}_s / \max \bar{x}_s \) are just one (the least resp. the greatest). E.g., there are always the trivial solutions \( \bar{x}_s = 1 \) and \( \bar{x}_s = 0 \) for all \( s \in S \) in the systems of inequations of Theorems 2.2.2 and 2.2.3. In this context it is convenient for practical purposes to characterise and calculate subsets of \( S \) for which it holds, that \( \Pr_{min}^{\min} / \max (\Diamond B) \in \{0, 1\} \) and rephrase the linear inequations as follows:

- If \( \Pr_{s}^{\min / \max} (\Diamond B) = 1 \), then \( x_s = 1 \).
- If \( \Pr_{s}^{\min / \max} (\Diamond B) = 0 \), then \( x_s = 0 \).
- Otherwise,

\[
x_s = \min / \max \left\{ \sum_{t \in S} P(s \xrightarrow{\alpha} t) \cdot x_t \mid \alpha \in \text{Act}(s) \right\}.
\]

It can be shown that these inequation systems do have the unique solutions \( (\bar{x}_s)_{s \in S}, x_s = \Pr_{s}^{\min / \max} (\Diamond B) \). In [HJ94] sets \( R \) and \( Q \) are calculated in that fashion to ensure a unique solution in the linear equation system that emerges from the problem of PCTL model checking against Markov chains. In [Par02] algorithms PROB0A, PROB1E, and PROB0E are used to calculate these sets in the context of PCTL model checking against MDPs. In [Char] algorithms for these sets are used in the context of stochastic \( \omega \)-regular games. The precalculation of these sets prior solving the numerical problem has two main advantages:

Due to the fact that these sets subsume the states \( s \) that will hold the values \( x_s = 0 \) or \( 1 \) in two potentially large equivalence classes, it shrinks the size of the problem, that is passed to the numerical solving module. It may even be the case that the solving method under consideration depends on the existence of a unique solution. The second advantage is that these sets can be calculated with pure graph based algorithms, which is often faster than leaving this problem (implicitly) to the numerical solver.

We denote the four different sets \( \{s \mid \Pr_{s}^{\min / \max} (\Diamond B) \in \{0, 1\}\} \) as follows:

- PROB0MAX = \( \{s \in S \mid \Pr_{s}^{\max} (\Diamond B) = 0\} \),
- PROB0MIN = \( \{s \in S \mid \Pr_{s}^{\min} (\Diamond B) = 0\} \),
- PROB1MIN = \( \{s \in S \mid \Pr_{s}^{\min} (\Diamond B) = 1\} \), and
- PROB1MAX = \( \{s \in S \mid \Pr_{s}^{\max} (\Diamond B) = 1\} \).
To provide characterizations in terms of graph based arguments for them, we define to associated sets

**Definition 2.3.1. [Subgraphs \( \mathcal{M}\backslash B \) and \( \mathcal{M}\Diamond B \)]** Given an MDP \( \mathcal{M} = (S, \text{Act}, \rightarrow, s_0, \text{AP}, \text{L}) \) and a goal set \( B \subseteq S \), let \( \mathcal{M}\backslash B \) be the largest sub-MDP of \( \mathcal{M} \) such that it does not contain any state in \( B \).

The subgraph \( \mathcal{M}\Diamond B \) is defined as the largest subgraph, such that

\[ \forall s \in \mathcal{M}\Diamond B : B \text{ is reachable from } s. \]

Let \( \text{MEC}_{\mathcal{M}\backslash B} \) be the union of all state components of maximal end components of \( \mathcal{M}\backslash B \).

Then, the sets \( \text{PROB0MIN}, \text{PROB1MIN}, \) and \( \text{PROB0MAX} \) can be characterised as follows:

- \( s \notin \text{PROB0MIN} \) if and only if
  - \( \forall U \in \text{Sched} : \Pr_{s}^{M} (\Diamond B) > 0 \)
  - \( \neg \exists U \in \text{Sched} : \Pr_{s}^{M} (\Diamond B) = 0 \)
  - \( \forall U \in \text{Sched} : \Pr_{s}^{M} (\Diamond B) > 0 \)
  - \( s \notin \mathcal{M}\backslash B \)

- \( s \notin \text{PROB1MIN} \) if and only if
  - \( \forall U \in \text{Sched} : \Pr_{s}^{M} (\Diamond B) = 1 \)
  - \( \text{MEC}_{\mathcal{M}\backslash B} \) is not reachable from \( s \) in \( \mathcal{M} \) without visiting states in \( B \)
  - \( s \notin \exists (\neg B)U(M\backslash B) \)

- \( s \in \text{PROB0MAX} \) if and only if
  - \( \forall U \in \text{Sched} : \Pr_{s}^{M} (\Diamond B) = 0 \)
  - \( B \) is not reachable from \( s \) in \( \mathcal{M} \)
  - \( s \notin \mathcal{M}\Diamond B \)

The set \( \text{PROB1MAX} \) cannot be characterized by terms of subgraphs \( \mathcal{M}\backslash B \) respectively the set \( \text{MEC}_{\mathcal{M}\backslash B} \). Nevertheless, calculating \( \text{PROB1MAX} \) can be done in a double fixed point calculation by means of purely graph based operations (see procedure below):

- \( s \notin \text{PROB1MAX} \) if and only if
  - \( \forall U \in \text{Sched} : \Pr_{s}^{M} (\Diamond B) < 1 \)
  - \( \exists U \in \text{Sched} : \Pr_{s}^{M} (\Diamond B) = 1 \)
  - \( \exists U \in \text{Sched} : \Pr_{s}^{M} (\Diamond B) < 1 \)
  - \( s \in \mathcal{M}\Diamond B \)
  - \( s \) is contained in the set calculated by the following procedure:
With this algorithm we conclude the preliminary explanations.
PROBMela specification
language

Building automated tools to address the analysis of reactive probabilistic systems requires a simple, but expressive input language with a formal semantics based on a probabilistic operational model that can serve as starting point for automated algorithmic verification analysis (e.g., simulation or model checking against temporal logical specifications). In this chapter we describe PROBMela, a probabilistic guarded process language inspired by Promela [Hol97]. This language is suitable for specifying the stepwise behaviour of communicating interleaved processes and allows for modelling various forms of probabilistic behaviour. The formal semantics of a PROBMela program is an MDP (see page 13).

In Promela and PROBMela the processes of a parallel system are described by means of a guarded command language [Dij76] [AO91] with nondeterministic choice and classical features of imperative programming languages, such as variable assignments, conditional and repetitive commands. Both languages support communication over shared variables and message passing via channels. Communication over channels can either be asynchronous using a buffer for the sent messages that is organized as a queue or follow the synchronous handshaking principle. Channels of the former type are called FIFO channels, while channels used for message passing via handshaking are called synchronous channels. Handshaking means, that two processes simultaneously perform a step each while exchanging a variable value over a synchronous channel. The additional probabilistic features of PROBMela are random assignments, a probabilistic choice operator and lossy FIFO channels. With these probabilistic features an interleaving semantics of PROBMela programs can be provided that formalizes the stepwise behaviour by means of an MDP. Unlike Promela and other probabilistic variants of guarded command languages PROBMela supports two variants of guarded commands. Guarded commands $g \Rightarrow P$, as they are used in Promela, have a two-step semantics which treats only the evaluation of the guard $g$ as an atomic step but allows for arbitrary interleaving before executing $P$. The execution of the first command in $P$ can then be “delayed” according to the interleaving semantics that allows other processes to perform some events after $g$ has been evaluated to true. The interrupt between the evaluation of a guard and the execution of the corresponding command $P$ can have an undesired effect, since other processes might affect the truth value of the guard $g$. 
e.g., by modifying the values of shared variables that appear in $g$. To avoid such interrupts, Promela offers the concepts of atomic regions. PROBMela offers one-step guarded commands $g \rightarrow P$ for this, which rely on the test-and-set principle and treat the successful evaluation of the guard $g$ and the execution of the first statement of $P$ as a non-interruptible operation. Atomic regions are, however, also included in PROBMela. Due to the close relationship between Promela and PROBMela, any system specified in one of these languages can be converted into the other rather easily (of course excluding probabilistic features or replacing probabilistic choices with nondeterminism). This is of some importance because it supports selective studies with and without probabilistic behaviour of the same model with different, closely related languages. An obvious application scenario is modelling and examining functional features of a system with, e.g., Promela, and then proceeding to calculate probabilistic properties using PROBMela. A preliminary part of the material presented in this section about PROBMela has been published in [BCG04, CBGK08] and the language was implemented for the model checker LiQuor [CB07, Cie07]. LiQuor uses a bytecode-oriented approach with a stack-based assembler-like language called PASM, that is also addressed in this chapter.

There are some other probabilistic languages that share the same core features, such as pGCL [MM99] and MoDeST [dHKK01, BdHK06]. The language pGCL (Probabilistic Guarded Command Language), was introduced for formal reasoning about randomized algorithms by means of probabilistic expectations. The main ingredients of pGCL coincide with those of the core language of PROBMela for processes without channel-based communication and atomic regions. The modelling language MoDeST (MOdeling and DEscription language for Stochastic Timed systems) combines features of pGCL, process algebras with TCSP-like synchronization over common actions and other language-constructs, such as urgent actions, exception handling and process instantiation, to reason about stochastic timed systems. It supports an advanced concept of modularisation and is suitable for reasoning about functional properties as well as real-time and probabilistic features of the model under consideration. The underlying semantics of MoDeST relies on stochastic timed automata and probabilistic timed transition systems.

There exist several other types of modelling languages especially for probabilistic systems with a Markov chain or MDP semantics that serve as starting point for automated verification processes. The issue of operational and denotational semantics for probabilistic systems has been discussed extensively in the context of (data-abstract) process algebras, see e.g. [GJS90, To90, HJ90, LS92, Han94, Low95, JYL01]. Operational semantics for a C-like imperative language to specify the behaviour of synchronous probabilistic systems (without nondeterminism or asynchronous parallelism) are described in [BCHG98, HGCC99]. Formal semantics for probabilistic variants of guarded command
languages have been presented e.g. in [Jon90, HSM97, dHdV99, MM99, dH02, d’A03b] for reasoning with Hoare-like and wp-like calculi. Parallel composition with channel-based communication or atomic regions have not been considered in these papers. The probabilistic model checker PRISM [KNP02] uses a variant of reactive modules [AH99] as modelling language for the description of a Markov decision process. The input language of the tool RAPTURE [JdL02] is based on an explicit description of the step-wise behaviour of the processes which resembles the format of the intermediate language PASM we use in our implementation. PROBVERUS [HGCC99] uses a C-like imperative language with a Markov chain semantics. The concept of wait-commands in PROBVERUS has some similarities with our concept of atomic regions. Nondeterminism or asynchronous parallelism are not considered in [HGCC99].

It should be mentioned, that the language PROBMela, as presented in this chapter and implemented in LiQuor, contains some semantical differences in comparison to the semantics presented in [BCG04]. The termination of repetitive statements of the form do...od is treated here differently, using an explicit command break, which is now compatible with the semantics of the popular language Promela. Another difference relates to atomic-regions (see section 3.2.3). In [BCG04], a denotational semantics has been presented that treats the execution of such regions as a genuine single step, which is in the spirit of the denotational semantics à la [dHdV99, dH02, d’A03b], but with a different characterization of the input/output behaviour of the body of an atomic region. In this thesis, however, we chose a different approach using an operational semantics that reflects the implementation in LiQuor in a better way and is, again, compatible with the approach chosen in SPIN with Promela. This approach executes the atomic region as multiple steps, but without parallel interleaving. The non-probabilistic virtual machine NIPS [Web10] which shares some similarities with the virtual machine of LiQuor and its bytecode language PASM treats atomic regions also as the sequential execution of prioritized actions with the interleaving effectively "turned off".

### 3.1. The syntax of the core language

A PROBMela program $\mathcal{P}$ consists of finitely many processes $P_1, \ldots, P_n$ that run in parallel and communicate over shared variables and/or channels, for which we formally write $\mathcal{P} = P_1 \parallel \ldots \parallel P_n$. In what follows, we present the PROBMela syntax to specify the stepwise behaviour of the $P_i$’s. The semantics of parallel composition will be discussed later in section 3.2. For a simpler presentation, we restrict ourself to the core features of PROBMela and its MDP-semantics, but omit some concepts of PROBMela that are well known from imperative programming languages and do not give new insights. For example, the precise
syntax of the variable and channel declarations and the predefined data types of the current version of the tool LiQuor are irrelevant for the general concepts of the modelling language PROBMela. Such rather technical aspects will not be addressed in this thesis, but can be found in the user manual of LiQuor. Some additional features that are not included in this core language (see Definition 3.1.1 below) will, however, be summarized at the end of this section with a brief explanation on their treatment in our SOS-style semantics.

PROBMela is a typed language where any program starts with a list of declarations of the program variables and FIFO channels. In what follows, \( \text{Var} \) denotes a fixed set of program variables (with typical element \( v \)) and \( \text{Chan} \) is a fixed set of channels. The symbol \( c \) will be used for FIFO channels, while \( d \) denotes a synchronous channel. We write \( \text{cap}(c) \) for the capacity, i.e., the number of cells in the buffer of \( c \) and denote the current grade of a channel’s filling by \( |c| \). \( \text{Dom}(v) \) denotes the domain (type) of variable \( v \). Furthermore, we suppose that each variable type is equipped with a set of operations that allow to build expressions from variables and constants (e.g., arithmetic expressions for integer variables or Boolean expressions for Boolean variables). For a synchronous channel \( d \), \( \text{Dom}(d) \) denotes the domain of the channel, and we have that \( \text{cap}(d) = 0 \). If \( c \) is a FIFO channel then \( \text{Dom}(c) \) denotes the type of messages that can be send via channel \( c \). If \( c \) is a lossy FIFO channel then \( \lambda(c) \in [0, 1] \) denotes its failure rate, i.e., the probability that a send-operation along channel \( c \) fails. The initial value of any variable \( v \in \text{Var} \) can be user specified in the context of the declaration of \( v \), or set to some default value in \( \text{Dom}(v) \). The initial configuration for any (asynchronous) FIFO channel \( c \in \text{Chan} \) is, that the channel is empty.

For the syntax and the rules that specify the MDP semantics of PROBMela, the cardinalities of the domains \( \text{Dom}(\cdot) \) do not play any role. However, to ensure that the MDP for a PROBMela program is finite, we suppose that the domain of each program variable and FIFO channel is finite and that all FIFO channels have finite capacity. We now define the syntax of PROBMela processes.

**Definition 3.1.1. [Abstract syntax of PROBMela (core elements)]** The syntax of the PROBMela processes \( P \) is defined as follows, where \( v \) denotes a variable, \( d \) represents a synchronous channel, \( c \) denotes an asynchronous (FIFO) channel and \( p_1, \ldots, p_n \) stand for probability values:
\[ P ::= g \rightarrow P \mid A \mid \text{prob\_choice} \mid \text{nondet\_choice} \mid P_1; P_2 \mid c?v \mid c!\ expr \mid d?v \mid d!\ expr \]
\[
\leftrightarrow ::= \rightarrow \mid \Rightarrow
\]
\[
A ::= v = expr \mid v = \text{random}(V) \mid \text{skip}
\]
\[
\text{nondet\_choice} ::= \text{if} :: g_1 \leftarrow 1 P_1 \ldots \leftarrow n P_n \text{fi} \mid \\
\text{do} :: g_1 \leftarrow 1 B_1 \ldots \leftarrow n B_n \text{od}
\]
\[
g ::= \text{bool\_expr} \mid \text{full}(c) \mid \text{empty}(c)
\]
\[
B ::= P \mid \text{break}
\]
\[
\text{prob\_choice} ::= \text{pif} : p_1 : C_1 \ldots : p_n : C_n \text{fip}
\]
\[
C ::= \rightarrow A \mid \rightarrow A; P \mid \Rightarrow P
\]

In this grammar, \( \mid \) stands for an alternative choice in the production rule. Symbols written in \textit{teletype} stand for terminal symbols. Except for one-character symbols they are \underline{underlined}. Symbols written in \textit{emphasized font} are non-terminal. We use this kind of notation for grammars throughout the chapter.

Statements of the form \( g \rightarrow P \), with \( \rightarrow \in \{\rightarrow, \Rightarrow\} \), are called guarded commands. These commands are only executable if \( g \), a Boolean guard, is satisfied by the current evaluation of the program variables and channel configurations. Such guards are Boolean combinations of atoms that impose conditions on the current value of variables, such as \( v > 5 \) or \( v_1 < v_2 \) or conditions on the enabledness of receive or send events along some FIFO channel \( c \) given by the predicates \texttt{full}(c) and \texttt{empty}(c). The difference between one-step guarded commands \( g \rightarrow P \) and two-step guarded commands \( g \Rightarrow P \) already has been explained intuitively in the introduction to this chapter.

In assignments \( v = expr \), type-consistency of variable \( v \in \text{Var} \) and the outcome of expression \( expr \) is required. The precise syntax of the expressions is not of importance for understanding the main features of PROBMela. They may be built from constants in \( \text{Dom}(v) \), variables \( w \) of a type that is compatible to \( v \) and operators on \( \text{Dom}(v) \), e.g., arithmetical operators for integer variables. The effect of a random assignment \( v = \text{random}(V) \) is that a value \( v \in V \) is probabilistically chosen and assigned to variable \( v \) (uniform distribution for \( V \) is assumed). In this context it is also worth noting that there exist different ways of treating situations where an operation on a variable \( v \), e.g. assigning the outcome of an
expression, exceeds Dom(v). We do not insist on a particular rule for these situations. Possible solutions are of course considering the exceeding of Dom(v) as an error or falling back to some default or "wrap-around" value, as it is done in many popular programming languages. In LiQuor, the user has the choice between these very two options. The command skip stands for a command that does not alter the variable evaluation or channel evaluation and just skips to the next statement. Sequential compositions \( P_1; P_2 \) have the meaning of concatenated multiple statements. Commands \( c?expr \) and \( c!expr \) represent channel operations on FIFO channels, where reading (receiving) a value from channel \( c \) into variable \( v \) is denoted by \( c?x \) and writing (sending) the value, that results from evaluating the expression \( expr \), to channel \( c \) is denoted by \( c!expr \). Synchronous communication (handshaking) is possible by sending and receiving values through synchronous channels, the difference to ordinary (asynchronous) channels being that sending and receiving is performed simultaneously (in one aggregated step).

The intuitive semantics of the commands if and do is roughly the same as in Promela and other guarded command languages. The conditional command

\[
\text{if} :: g_1 \mapsto_1 P_1 \ldots \mapsto_n P_n \text{fi}
\]

stands for a non-deterministic choice between those processes \( P_i \) where the guard \( g_i \) is satisfied in the current state. If none of the guards is satisfied, the if-fi statement blocks until at least one of the conditions is satisfied. Repetitive statements

\[
\text{do} :: g_1 \mapsto_1 P_1 \ldots \mapsto_n P_n \text{od}
\]

stand for the repeated execution of the non-deterministic choice between the guarded commands \( g_i \mapsto_1 P_i \). As in Promela and like conditional commands, do...od-loops block if no guard is satisfied. The innermost do...od statement is left with the command break. Again, the guard operators \( \mapsto \) and \( \Rightarrow \) rely on one-step and two-step semantics respectively, similarly to single guarded commands. Note that PROBMela allows for mixing one- and two-step guarded commands in single if...fi and do...od statements.

The probabilistic choice operator pif...fip can be viewed as an analogue to if...fi-statements where the former chooses its alternatives according to given probabilities instead of Boolean conditions. Intuitively,

\[
pif :: p_1 : C_1 \ldots : p_n : C_n \text{ fip}
\]

performs a stochastic experiment with \( n \) possible events occurring with probabilities given by \( p_i \), followed by the execution of the (probabilistically) chosen process \( C_i \). For this, we require, that \( p_1, \ldots, p_n \in ]0, 1[ \) such that \( p_1 + \ldots + p_n = 1 \). In practice, using decimal
point numbers for stating probabilities $p_i \in ]0, 1[ \), making it difficult to express rational values with an infinite representation, e.g., the probability $\frac{1}{3}$. This can be avoided by allowing for $p_i \cdot \ldots : q_i : C_i \ldots \top$, where $q_i \in \mathbb{N}_{\geq 1}$ and then using the weighted sum

$$p_i = \sum_{1 \leq j \leq n} q_i$$

as the corresponding probability. For the sake of simplicity, we will keep using probability values $p_i \in ]0, 1[ \)$ from now on, although the corresponding implementation in LiQuor allows for the general weighted-sum notation. The syntactic restrictions of one-step guarded commands of $p_i \cdot \ldots$-statements has been made to ensure than an enabled atomic step follows the probabilistic choice.

**Example.** An example, known as Knuth’s dice [KY76], serves to illustrate the one-step behaviour in connection with the probabilistic choice operator. The PROBMela program in listing 3.1 specifies a process that simulates a fair (six-sided) dice by only using fair coin tosses.

```
Listing 3.1: Knuth’s dice

v = 0;
do :: v = 0 ->
        pif 0.5:->
            do :: v = 0 ->
                pif 0.5:->
                    pif 0.5:->v = 1
                    :0.5:->skip
            fip
            :0.5:->random(2,3)
        fip
        :: v!=0 -> break
    od
    :0.5:->
do :: v = 0 ->
        pif 0.5:->
            pif 0.5:->v = 4
            :0.5:->skip
    fip
    :0.5:->random(5,6)
    fip
    :: v!=0 -> break
    od
    fip
    :: v!=0 -> break
```


3.2. Operational semantics for the core language

In this section, we present a two-layered approach to provide the MDP-semantics for a given PROBMela program.

\[ \mathcal{P} = P_1 || \ldots || P_n \]

\[ \text{PCGs for } P_1, \ldots, P_n \]

\( \text{MDP } \mathcal{M} \)

\[ \mathcal{M} = \text{(probabilistic control graph)} \]

Figure 3.1: Scheme of the two-layered semantics

In the first layer, we assign to each PROBMela-process a probabilistic control graph (PCG, see section 3.2.1), that formalizes the control flow. The MDP of a PROBMela program \( \mathcal{P} = P_1 || \ldots || P_n \) then results by ”unfolding” the product of the PCGs for \( P_1, \ldots, P_n \) when taking the variable evaluations and channel configurations and effects of events in the PCG into account (see section 3.2.2).

The two-layered approach using PCGs also serves as conceptual basis for the implementation of the model checker LiQuor. In the context of LiQuor, the execution of process code written in the intermediate probabilistic assembler language PASM (see section 3.3) essentially mimics the structural behaviour of a probabilistic control graph. In the implementation, these PASM processes are derived directly (i.e., generated by a compiler for the
language PROBMela) from PROBMela processes, according to the first layer of semantics. The LiQuor virtual machine then executes these PASM processes in an interleaved setting, thus generating the MDP for the PROBMela program in an on-the-fly fashion according to the second layer of semantics.

### 3.2.1. Probabilistic control graph semantics for PROBMela programs

Given a PROBMela process \( P \) (as one of the processes \( P = P_1 \) of a PROBMela program \( P = P_1 \| \ldots \| P_n \)), we use the classical SOS-style [Plo81] to provide the operational control flow semantics of process \( P \). The resulting structure is a directed graph (called probabilistic control graph) where the nodes are control components (called locations) and the edges represent the potential steps. In the MDP for the composite program \( P = \ldots \| P \| \ldots \), these steps might or might not be enabled, depending on the current variable and channel evaluation (that might or might not satisfy the guard of the corresponding edge) or the availability of synchronization partners in case of synchronous message passing actions (formalized by means of the synchronization rule for the MDP-semantics of the composite program \( P \)).

As in the core syntax, the guards of the edges are Boolean combinations of predicates stating conditions on the current values of variables or the current status of a FIFO channel (predicates \( \text{full}(c) \) and \( \text{empty}(c) \)). In what follows, we write \( \text{Cond}(\text{Var} \cup \text{Chan}) \) for the set of guards. Each step can have a randomized effect on the values of the program variables, the contents of the buffers of the FIFO channels and the successor location. Thus, edges in the program control graph have the form \( \ell \xrightarrow{\nu} \nu \) where \( \ell \in \text{Cond}(\text{Var} \cup \text{Chan}) \) and \( \nu \) is a distribution over pairs consisting of a process event and a location. The process events (denoted by the symbols \( a, b, c, \ldots \)) stand for atomic instructions such as variable assignments, skip, communication events, the evaluation of the guard, or the coin tossing experiment that is inherent in the probabilistic choice operator.

**Definition 3.2.1. [Probabilistic control graph]** A probabilistic control graph (PCG) over a set of typed variables and channels \( \text{Var} \cup \text{Chan} \) is a tuple

\[
\Psi = (\text{Loc}, \text{Events}, \sim, \ell_0),
\]

where

- \( \text{Loc} \) is a nonempty set of program locations,
- \( \text{Events} \) is a nonempty set of process events,
- \( \sim \subseteq \text{Loc} \times \text{Cond}(\text{Var} \cup \text{Chan}) \times \text{Distr}(\text{Events} \times \text{Loc}) \) is the edge relation, and

---

43
• \( \ell_0 \in \text{Loc} \) is the initial location.

The edge relation \( \rightsquigarrow \) assigns to each location a finite set of distributions over pairs of process events and locations. These potential events might or might not be enabled, depending on the current variable evaluation and the availability of synchronization partners in case of synchronous message passing events \( d?x \) or \( d!expr \) via synchronous channel \( d \). Note, however, that concrete variable evaluations are not integral part of the definition of a PCG. The enabled events can have a randomized effect on the program variables (data). For the control structure, we just need to specify the (possibly randomized) effect on the locations through the edge relation \( \rightsquigarrow \).

Given a PROBMela process \( P \) of the core language, we now define the PCG for \( P \) using the classical SOS-style notations. When needing a second symbol for a (sub-)process we may use \( Q \). The definition of the set \( \text{Loc}(P) \) of locations in the PCG for \( P \) can formally be provided by structural induction:

- If \( P \in \{ \text{skip}, v = expr, c?v, c! expr, v = \text{random}(V) \} \), then \( \text{Loc}(P) = \{ P, \text{exit} \} \).
- \( \text{Loc}(\text{break}) = \{ \text{skip}, \text{exit} \} \).
- \( \text{Loc}(g \hookrightarrow P') = \{ g \hookrightarrow P' \} \cup \text{Loc}(P') \).
- If \( P \) is a conditional command of the form

\[
\text{if} :: g_1 \hookrightarrow_1 P_1 \ldots \hookrightarrow_n P_n \text{ if},
\]

or a probabilistic choice

\[
\text{pif} : p_1 : \hookrightarrow_1 P_1 \ldots : p_n : \hookrightarrow_n P_n \text{ fip},
\]

then \( \text{Loc}(P) = \{ P \} \cup \bigcup_{1 \leq i \leq n} \text{Loc}(P_i) \).

- If \( P \) is a repetitive command of the form

\[
\text{do} :: g_1 \hookrightarrow_1 P_1 \ldots \hookrightarrow_n P_n \text{ od},
\]

then \( \text{Loc}(P) = \{ P \} \cup \bigcup_{1 \leq i \leq n} \{ Q : P : Q \in \text{Loc}(P_i) \} \).

\footnote{The definition of \( \text{Loc}(P) \) that we provide here treats both types of guarded commands (\( \rightarrow \) and \( \Rightarrow \)) in the same manner. This simplifies the recursive definition and leads to a set \( \text{Loc}(P) \) that covers all locations relevant for \( P \), but might also include locations that are unreachable from \( P \). However, these locations are utilized in PASM to realize the one-step guarded commands.}
Sequential composition is carried out with respect to the rule
\[ \text{Loc}(P_1;P_2) = \{P_1;P_2\} \cup \{Q;P_2 : Q \in \text{Loc}(P_1)\} \cup \text{Loc}(P_2). \]

Then, the PCG of \( P \) is defined by
\[ \Psi_P = (\text{Loc}(P), \text{Events}_P, \sim, P), \]
where the initial location is the process \( P \) itself. In what follows, subterms of \( P \) are often denoted by the symbol \( Q \). For convenience, we write \( P \overset{g}{\sim} \nu \) instead of \((P, g, \nu) \in \sim\) and furthermore write \( P \overset{g}{\sim} \mu \) as a shortform for \( P \overset{g}{\sim} \nu \), where \( \nu(\langle a, Q \rangle) = \mu(Q) \) and \( \nu(\langle b, Q \rangle) = 0 \) if \( b \neq a \). Furthermore we drop \( g \) if it is a tautology and use the event name \( t \) to denote an event that does not have any effect on variables or channels. Beside statements of the type \text{pif}..\text{fip}, the PROBMela commands impose a nonprobabilistic effect on the location and process events, in which case we write \( Q \overset{g}{\sim} Q' \) as a shortform notation for \( Q \overset{g}{\sim} \mu \), where \( \mu \) is the unique (Dirac) distribution over \( \text{Loc}(P) \) such that \( \mu(Q') = 1 \).

We denote the set of all edges in \( P \) as \( \text{Edges}_P \). With \( v \in \text{Var} \), \( c \) denoting non-synchronous channels and \( d \) referring to synchronous channels, the following axioms and rules determine the edge relation \( \sim \):

<table>
<thead>
<tr>
<th>Axioms for skip, deterministic assignments and communication events</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{skip} \overset{t}{\sim} \text{exit} \quad \text{(skip)}</td>
</tr>
<tr>
<td>( v = \text{expr} ) \overset{\sim}{\sim} \text{exit}, \quad \text{(deterministic assignment)}</td>
</tr>
<tr>
<td>( v = \text{random}(V) ) \overset{\sim}{\sim} \text{exit} \quad \text{(random assignment)}</td>
</tr>
<tr>
<td>( c!\text{expr} ) \overset{\sim}{\sim} \text{exit} \quad \text{(sending along FIFO channel)}</td>
</tr>
<tr>
<td>( c?v ) \overset{\sim}{\sim} \text{exit} \quad \text{(receiving from FIFO channel)}</td>
</tr>
<tr>
<td>( d!\text{expr} ) \overset{\sim}{\sim} \text{exit} \quad \text{(send with handshake)}</td>
</tr>
<tr>
<td>( d?v ) \overset{\sim}{\sim} \text{exit} \quad \text{(receive from handshake)}</td>
</tr>
</tbody>
</table>

Note that these commands have a deterministic effects on the control location, i.e., a Dirac distribution over \( \text{Loc}(P) \). This is especially the case for the \text{random()} command, which imposes a randomized effect on the variables, but not on the program flow.
### Axioms and rules for guarded commands, choices and repetitive statements

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ g \Rightarrow P \stackrel{g}{\Rightarrow} P ]</td>
<td>(two-step guarded command)</td>
</tr>
<tr>
<td>[ \frac{P \xrightarrow{h} \nu}{g \Rightarrow P \stackrel{g &amp; h}{\Rightarrow} \nu} ]</td>
<td>(one-step guarded command)</td>
</tr>
<tr>
<td>[ \text{if } \ldots : g_i \Rightarrow P_i \ldots \text{ fi } \stackrel{g_i}{\Rightarrow} P_i ]</td>
<td>(choice with two-step guarded command)</td>
</tr>
<tr>
<td>[ \frac{\text{if } \ldots : g_i \Rightarrow P_i \ldots \nu \stackrel{g_i &amp; h}{\Rightarrow} \nu}{\text{if } \ldots : g_i \Rightarrow P_i \ldots \nu} ]</td>
<td>(choice with one-step guarded command)</td>
</tr>
<tr>
<td>[ \text{If } P = \text{ do } \ldots : g_i \Rightarrow P_i \ldots \text{ od } \text{ then } \frac{P \stackrel{g_i}{\Rightarrow} P_i ; P}{P} ]</td>
<td>(two-step guarded command in a loop)</td>
</tr>
<tr>
<td>[ \text{If } P = \text{ do } \ldots : g_i \Rightarrow P_i \ldots \text{ od } \text{ then } \frac{P \stackrel{g_i}{\Rightarrow} P_i ; P}{P} , \quad \frac{\sigma(a, P_i) = \nu(\langle a, P_i \rangle)}{\sigma(a, \cdot) = 0 \text{ otherwise.}} ]</td>
<td>(one-step guarded command in a loop)</td>
</tr>
</tbody>
</table>

\[ \text{do} \ldots : g_i \leftrightarrow \text{ break } \ldots \text{ od } \stackrel{g_i}{\Rightarrow} P' \]

where \( P' = \begin{cases} \text{skip} & \text{if } \leftrightarrow \text{ is } \Rightarrow \\ \text{exit} & \text{if } \leftrightarrow \text{ is } \Rightarrow \end{cases} \) (termination of a loop)

Again, these commands involve - in most situations - deterministic effects on the control location (Dirac distribution over \( \text{Loc}(P) \)). Note however, that it is possible to use the one-step operator in \( \text{if } \ldots : \text{fi} \) or \( \text{do } \ldots \text{od} \) statements. Due to this, a nondeterministic choice may also incorporate a proper probabilistic effect by including a probabilistic choice \( \text{pif} \ldots \text{fip} \) (see below) via a one-step guarded command.

#### Rule for probabilistic choice

Let \( P = \text{pif } : p_1 : C_1 \ldots : p_n : C_n : \text{fip} \), where \( C_i \) is \( \rightarrow A_i ; P_i \), or \( C_i \) is \( \Rightarrow P_i \) for \( 1 \leq i \leq n \) and let \( a_i \) be the unique process event such that \( C_i \stackrel{a_i}{\Rightarrow} P_i \).

Here, we identify \( \rightarrow A_i ; P_i \) with the guarded command \( \rightarrow A_i ; P_i \) and \( \Rightarrow P_i \) with the guarded command \( \Rightarrow P_i \). Indeed, for each of these guarded commands, there is a unique enabled process event \( a_i \), that moves control from \( C_i \) to \( P_i \).

Then \( P \sim \nu \), where \( \nu(\langle a, Q \rangle) = \sum_{i \in I_{a,Q}} p_i \), with \( I_{a,Q} = \{ i \in \{ 1, \ldots, n \} : a = a_i \text{ and } Q = P_i \} \).
Chapter 3. PROBMela specification language

3.2. Operational semantics for the core language

<table>
<thead>
<tr>
<th>Rule for sequential composition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_1 \overset{h}{\rightarrow} \mu ) ( \quad ) ( P_1; P_2 \overset{h}{\rightarrow} \nu )</td>
</tr>
</tbody>
</table>

where \( \nu(\langle a, P'_1; P_2 \rangle) = \mu(\langle a, P'_2 \rangle) \) and we identify \( \text{exit}; P \) with \( P \),

\[ \nu(\langle a, \cdot \rangle) = 0 \] otherwise.

The set \( \text{Events}_P \) of event names in the program graph for \( P \) is then obtained by the set of all event names that appear as label of the relevant axioms and rules.

**Effect of events.** The unfolding of the program graphs into an MDP (see section 3.2.2) will rely on a function

\[ \text{Effect} : \text{Events} \times \text{Eval}(\text{Var} \cup \text{Chan}) \rightarrow \text{Distr}(\text{Eval}(\text{Var} \cup \text{Chan})) \]

that specifies the effect of the event on the program variables and the buffers of the FIFO channels. As above, \( \text{Var} \cup \text{Chan} \) denotes the set of variables and FIFO channels, that are used in the given PROBMela program. The notation \( \text{Eval}(\text{Var} \cup \text{Chan}) \) refers to the set of all functions \( \eta \) that assign a value to each variable and a sequence of values to each FIFO channel representing a possible configuration of its buffer. There are the obvious side-constraints (type-consistency for the variables, i.e., \( \eta(v) \in \text{Dom}(v) \) for each variable \( v \in \text{Var} \) and type- and capacity-consistency for the channels, i.e., if \( c \in \text{Chan} \) is a FIFO channel of capacity \( k \) and domain \( D \) then \( \eta(c) \) is a word over \( D \) of length at most \( k \)). In the sequel, we refer to the elements of \( \text{Eval}(\text{Var} \cup \text{Chan}) \) as variable and channel evaluations, or briefly evaluations.

We often identify an evaluation \( \eta \) with the distribution that assigns probability 1 to \( \eta \) and probability 0 to all other evaluations. That is, if \( a \) is an event and \( \eta, \eta' \in \text{Eval}(\text{Var} \cup \text{Chan}) \) then the notation \( \text{Effect}(a, \eta) = \eta' \) indicates that the execution of \( a \) in a state where the variable and channel evaluation is \( \eta \) affects the variables and channels in a deterministic manner and the resulting variable and channel evaluation is \( \eta' \).

For most events, the definition of the effect function \( \text{Effect}(a, \cdot) \) is as expected. Given an evaluation \( \eta \in \text{Eval}(\text{Var} \cup \text{Chan}) \), then \( \text{Effect}(\text{assign}(v, \text{expr}), \eta) \) returns the unique evaluation \( \eta[v \leftarrow \text{value(expr)}] \) that agrees with \( \eta \) for all variables \( v' \neq v \) and all channels \( c \) and assigns the current value of \( \text{expr} \) under \( \eta \) to variable \( v \). The effect of a random assignment \( \text{assign}(v, \text{random}(V)) \) where \( V \) is a nonempty and finite subset of \( \text{Dom}(v) \) is specified by the uniform distribution over \( V \), i.e.,

\[ \text{Effect}(\text{assign}(v, \text{random}(V)), \eta) = \sigma \text{ where} \]

\[ \sigma(\eta[v \leftarrow w]) = \frac{1}{|V|} \] for all \( w \in V \) and

\[ \sigma(\eta') = 0 \] in all other cases.
Asynchronous channels have a buffer that is organized according to the FIFO principle. The effect of events \( \text{receive}(c) \) and \( \text{send}(c, expr) \) relies on the standard operations on queues:

- \( \text{front}(c) \) access to the front, i.e., first element,
- \( \text{remove}(c) \) deletion of the front element, and
- \( \text{insert}(c, v) \) inserting \( v \) as the last element.

Thus, if \( c \) is a perfect (i.e., not lossy) FIFO channel of capacity \( k \) and \( \eta \) a variable evaluation such that \( |\eta(c)| < k \) (i.e., the buffer for \( c \) is not full) then \( \text{Effect}(\text{send}(c, expr), \eta) = \eta[\text{insert}(c, \text{value}(expr))] \). The effect of a send events \( \text{send}(c, expr) \) for some lossy FIFO channel \( c \) with failure rate \( \lambda = \lambda(c) \in [0, 1[ \) is as follows. (The failure rate \( \lambda \) is supposed to be specified in the channel declaration.)

\[
\text{Effect}(\text{send}(c, expr), \eta) = \sigma \text{ where } \sigma(\eta[\text{insert}(c, \text{value}(expr))]) = 1 - \lambda \text{ and } \sigma(\eta) = \lambda
\]

The effect of a receive event \( \text{receive}(c) \) for a FIFO channel \( c \) and the current variable and channel evaluation \( \eta \) such that \( \eta(c) \neq \varepsilon \) (i.e., the buffer for FIFO channel \( c \) is nonempty) corresponds to the assignment \( v \leftarrow \text{front}(c) \) and the removal of the front element of \( c \). Thus,

\[
\text{Effect}(\text{assign}(v, \text{receive}(c)), \eta) = \eta[v \leftarrow \text{front}(c); \text{remove}(c)].
\]

The special event symbol \( t \) is used for events that have no effect on the variables or channels, i.e., \( \text{Effect}(t, \eta) = \eta \) for all \( \eta \).

Since the effect of communication events \( d?v \) and \( d!expr \) over synchronous channels affect multiple processes, their effect cannot be specified directly. Instead, their effect is taken into account in the composition rule for the transition relation of the MDP of a composite PROBMela program (see below).

**Example.** For an illustrative example, we now revisit the fair dice program (see listing \[3.1\] on page 41) and give a PCG for the program. In figure \[3.2\] the locations of the PCG are identified with the line number of the corresponding PROBMela command in the listing. In location 2 a probabilistic choice is made in the PCG due to the one-step guarded command that proceeds from line 2 directly over the probabilistic choice (pif) in line 3 to locations 4 and 14. In terms of semantical rules, this is a combination of the rules *one-step guarded command in a loop* and *probabilistic choice* (both see page 46). The location pif...fip, i.e., line 3 exists in the PCG due to the rule of *probabilistic choice*, but is omitted from the figure because it is not reachable from the initial location. Similarly, a second combination of do and pif lead to an event from location 4 to locations 6 or 4. Returning to line 4 (with probability \( \frac{1}{2} \)) involves a cascade of two one-step guarded commands (\( v = 0 \rightarrow \text{pif} \) and : \( 0.5 : \rightarrow v = \text{random}(2, 3) \)). Note that random (see rule on page 45) has a deterministic
Figure 3.2: Graph for the stepwise execution of program locations in Knuth’s dice

effect on the successor location. The argumentation is symmetrical for location 14. Note
that the guarded commands in lines 11/21 and 24 are used to exit the program after a dice
value was assigned to \( v \).

3.2.2. MDP semantics for Probabilistic control graphs

Given a PROBMela program \( P = P_1||...||P_n \), where each \( P_i, 1 \leq i \leq n \) falls under the re-
striction of the PROBMela core syntax, we assume that the corresponding PCGs \( \Psi_1, ..., \Psi_n \)
are generated according to the semantics given above. Using these PCGs, we define an
MDP

\[ M_P = (S, \text{Act}, \rightarrow, s_0, AP, L) \]

that specifies the stepwise operational semantics of \( P \). The state space of \( M_P \) consists of tuples

\[ S = \text{Loc}_1 \times ... \times \text{Loc}_n \times \text{Eval}(\text{Var} \cup \text{Chan}), \]

holding the current process locations \( \text{Loc}_i \) and the current variable evaluation, where \( \text{Loc}_i = \text{Loc}(P_i) \). The initial state is \( s_0 = (P_1, ..., P_n, \eta_0) \in S \), where \( \eta_0 \) is the evaluation that agrees
with all initial values of variables \( v \in \text{Var} \) and all initial channel configurations of channels \( c \in \text{Chan} \) (see explanations in section 3.1).
The set of actions in the MDP is,
\[
\text{Act} = \bigcup_{1 \leq i \leq n} \text{Act}_i,
\]
where the sets \( \text{Act}_i \) are obtained by using pairwise distinct action names for the edges in the PCG for \( P_i \). The only exception are the action names for synchronous message passing which are shared between the synchronization partners.

Formally, the action sets \( \text{Act}_i \) can be defined as follows. Let \( \text{SyncEdges}_i \) denote the set of edges of the form \( Q \xrightarrow{\text{send}(d, \text{expr})} Q' \) or \( Q \xrightarrow{\text{assign}(v, \text{receive}(d))} Q' \) in the PCG for process \( P_i \), where \( d \) ranges over all synchronous channels. Then, the action set \( \text{Act}_i \) consists of

- pairwise distinct abstract action names for all edges \( Q \xrightarrow{\sim} \nu \) in the PCG for \( P_i \) that do not belong to \( \text{SyncEdges}_i \), and

- the elements in the set \( \text{SyncAct}_i \) of action names in the MDP for \( \mathcal{P} \) that represent some handshaking where process \( P_i \) is involved as sender or receiver. Formally, \( \text{SyncAct}_i \) consists of abstract action names for all pairs \( (\alpha, \beta) \) where
  
  - \( \alpha \) is an edge \( Q \xrightarrow{\text{send}(d, \text{expr})} Q' \) in the PCG for \( P_i \) and
  - \( \beta \) an edge \( Q \xrightarrow{\text{assign}(v, \text{receive}(d))} Q' \) in the PCG for some other process \( P_j \),

  or vice versa, i.e.,
  
  - \( \alpha \) is an edge \( Q \xrightarrow{\text{send}(d, \text{expr})} Q' \) in the PCG for some process \( P_j \) where \( j \neq i \)
  - \( \beta \) is an edge \( Q \xrightarrow{\text{assign}(v, \text{receive}(d))} Q' \) in the PCG for \( P_i \).

In what follows, we often identify the abstract action names with the corresponding edge or pair of edges. The purpose of the abstract action names is simply that we can choose them in such a way that for different indices \( i \) and \( j \) the intersection \( \text{Act}_i \cap \text{Act}_j \) agrees with the set \( \text{SyncAct}_i \cap \text{SyncAct}_j \) which represents exactly the set of actions where processes \( P_i \) and \( P_j \) communicate via some synchronous channel. In particular, if \( \mathcal{P} \) does not use synchronous channels then \( \text{Act}_i \cap \text{Act}_j = \emptyset \), provided that \( i \neq j \). We denote the set of all synchronization actions by
\[
\text{SyncAct} = \bigcup_{1 \leq i < j \leq n} \text{Act}_i \cap \text{Act}_j.
\]

The set of atomic propositions \( \text{AP} \) consists of labels for the evaluation of all variables and channels while the labeling function \( L \) maps these labels to their states in the expected way. The transition relation \( \rightarrow \) is defined according the two following rules:
Chapter 3. PROBMLa specification language

3.2. Operational semantics for the core language

Interleaving rule

\[ \alpha = Q_i \overset{\nu}{\sim} \nu \in \text{Act}_i \setminus \text{SyncAct}_i \land \eta \models g \]
\[ \langle Q_1, \ldots, Q_i, \ldots, Q_n \rangle \xrightarrow{\alpha} \mu \]

where \( \mu \) is the following distribution

\[ \mu(\langle Q_1, \ldots, Q_{i-1}, R, Q_{i+1}, \ldots, Q_n, \eta' \rangle) = \sum_{a \in \text{Events}_i} \nu(\langle a, R \rangle) \cdot \text{Effect}(a, \eta)(\eta') \]

Synchronization rule

\[ \alpha = Q_i \overset{d^x}{\rightarrow} Q'_i \land \beta = Q_j \overset{d^{expr}}{\rightarrow} Q'_j \]
\[ \langle Q_1, \ldots, Q_i, \ldots, Q_j, \ldots, Q_n, \eta \rangle \xrightarrow{\gamma} \langle Q_1, \ldots, Q'_i, \ldots, Q'_j, \ldots, Q_n, \eta[v \leftarrow \text{value(expr)}] \rangle \]

where \( \gamma \in \text{SyncAct} \) is the action name for handshaking via \( \alpha \) and \( \beta \)

Examples. We conclude this section by first completing the dice example, where the given MDP semantics is applied to the PCG from figure 3.2. Secondly, we give an example that illustrates the interleaving of multiple processes.

According to the MDP semantics given above, the MDP of the PROBMLa program \( P = P \) (PROBMLa program on page 41, PCG page 49) has the state space \( S = \text{Loc}(P) \times \text{Eval}(\text{Var} \cap \text{Chan}) \). Consequently the MDP and the PCG are identical for the states where \( v = 0 \). Note that the probabilistic effects of the random commands are combined with the probabilistic effects on the locations of the associated \( \text{pif} \) commands according to the multiplication in the interleaving rule to obtain probabilities \( \frac{1}{4} = \frac{1}{2} \cdot \frac{1}{2} \) for different values for \( v \).

A second example shows parallel interleaving between two processes that compete for the mutual exclusive access to a critical resource, that is controlled by a probabilistic arbiter (see listings 3.2, 3.4, figures 3.3 (PCG) and 3.6 (MDP)).

Here, each process holds an own synchronous channel for communication with the arbiter, which is channel \( d_1 \) for process \( P_1 \), and channel \( d_2 \) for process \( P_2 \). Consequently, edges that access these channels are synchronous (see legend in figure 3.6 and synchronisation rule on page 51). Since the processes chose nondeterministically between skipping (wich, e.g., models some abstract behaviour that stands independently from the mutex protocol) and entering for access to the shared resource by sending a token to the arbiter, each node also possesses a self loop for the \( \text{skip} \) edge of the process. \( \square \)
3.2. Operational semantics for the core language

### 3.2.3. Advanced language features

To simplify the notations in the axioms and rules for the PCG of PROBMela processes, some features of PROBMela have been omitted in the grammar for the core language. The most important additional construct are atomic regions, that will be explained now. Other additional features, e.g., jump commands (goto), relaxed rules for using the command break to terminate loops and the special guard else may be considered as “syntactic sugar”, but are useful in practise. Since these concepts are known from other languages and do not cause any semantical problems that are specific for the probabilistic case, we just give brief explanations on their syntax and semantics.

**Atomic regions.** Atomic-regions, defined by the keyword atomic\{P\}, where P is a process according to the core syntax, are sequences of commands that are conceptually executed without interruption. They are used as

- an abstract, built-in mutual-exclusion protocol without the necessity of using additional semaphores or other means of concurrency control, and
Chapter 3. PROBMela specification language  3.2. Operational semantics for the core language

Listing 3.2: Process 1

```plaintext
1 do :: true -> skip
2 :: true -> d1?x;
3  skip ;
4 d1!(x-1)
5 od
```

Listing 3.3: Arbiter

```plaintext
1 do :: true -> pif :
2   0.5 =>
3   d1!1;
4   d1?v
5 : 0.5 =>
6   d2!2;
7   d2?v
8 fip od
```

Listing 3.4: Process 2

```plaintext
1 do :: true -> skip
2 :: true -> d2?x;
3  skip ;
4 d2!(x-2)
5 od
```

Figure 3.4: A mutual exclusion protocol using a probabilistic arbiter

Figure 3.5: PCGs with edges according to axioms for communication events and skip

- a technique that shrinks the state space by letting a process execute a sequence of steps without the interleaving effects of other processes’ executions.

It is required, that there is no atomic region inside atomic regions, i.e., for any term atomic\{P\}, we suppose that P is a PROBMela process that does not contain any subprocess atomic\{Q\}.

Semantics of atomic regions. To provide a formal definition of the MDP-semantics of PROBMela-programs with atomic regions we follow the same approach as for the core language, i.e., we assign a PCG to the processes, that formalizes the control flow, and then unfold the product of the PCG for \(P_1, \ldots, P_n\) into an MDP for \(P\) by taking the current values of the program variables and current content of the FIFO channels into account. To treat atomic regions, we enrich the set \(Var\) of program variables by a fresh variable atomic with domain \(\{0, 1, \ldots, n\}\), where \(n\) is the number of processes. Intuitively, atomic = \(i \in \{1, \ldots, n\}\) indicates that process \(i\) is in one of its atomic regions, in which case the
interleaving and synchronization rules are suspended such that only process $i$ may perform the next step. If $\text{atomic} = 0$, then none of the processes is in an atomic region and the transitions are obtained in “normal mode” according to the interleaving and synchronization rule as for the core language.

**Program control graphs for atomic regions.** The set of locations in the PCG of PROBMela-processes with atomic regions are obtained by the function $\text{Loc}(\cdot)$ which is defined inductively as before with the following additional case:

$$\text{Loc}(\text{atomic}\{P\}) = \{\text{atomic}\{P\}\} \cup \{[Q] : Q \in \text{Loc}(P)\}$$

Here, $[Q]$ stands for a process that behaves to a large extent as $Q$, the difference being that $[Q]$’s steps are non-interruptible. The PCG for $\text{atomic}\{P\}$ results from the program graph for $P$ by renaming location $Q$ into $[Q]$ and adding edges for entering and leaving the atomic region. Entering an atomic region is accompanied with the assignment $\text{atomic} \leftarrow \text{PID}$ where we think of $\text{PID}$ as a fictive variable, that is the process identifier of the (unique) process that is entering its atomic region. Similarly, leaving an atomic region is conducted with the assignment $\text{atomic} \leftarrow 0$. Thus, the set of process events in the PCG of $\text{atomic}\{P\}$ as a subprocess of the $i$-th process of a given PROBMela-program $P = P_1 \parallel \ldots \parallel P_n$ contains the process events $a \in \text{Events}_P$ and additionally the events $\text{enter}$ and $\text{release}$ with the effect

\footnote{In LiQuor there is indeed a process variable, i.e., a local variable (named \texttt{pid}) implicitly defined in each process, that uniquely identifies a process.}
Chapter 3. PROBMela specification language 3.2. Operational semantics for the core language

Effect\(\text{enter}, \eta\) = \(\eta[\text{atomic} \leftarrow \text{PID}]\) and

Effect\(\text{release}, \eta\) = \(\eta[\text{atomic} \leftarrow 0]\).

The edges of the PCG for \(\text{atomic}\{P\}\) arise by the following SOS-axioms and rules:

| Axiom for entering an atomic region: | \(\text{atomic}\{P\} \xrightarrow{\text{enter}} [P]\) |
| Axiom for leaving an atomic region: | \([\text{exit}] \xrightarrow{\text{release}} \text{exit}\) |
| Behaviour inside an atomic region: | \(\frac{Q \xrightarrow{\alpha} \nu}{[Q] \xrightarrow{\alpha} \nu}\) |

where \([\nu](a, [R]) = \nu(a, R)\) for any process \(R\) and event \(a\)

**MDP-semantics.** Suppose that we are given a PROBMela-program \(\mathcal{P} = P_1 \parallel \ldots \parallel P_n\). As before, let \(\text{Events}_i\) denote the set of process events in the PCG for \(P_i\). The action set of the MDP for \(\mathcal{P}\) is \(\mathcal{Act}\) consisting of pairwise distinct abstract action names for the edges in the PCG for \(P_1, \ldots, P_n\), except for shared action names representing synchronization events (see page 51). Thus, if \(\mathcal{Act}_i\) is used to denote the set of actions of process \(P_i\), then \(\mathcal{Act} = \bigcup_{1 \leq i \leq n} \mathcal{Act}_i\), and \(\mathcal{Act}_i \cap \mathcal{Act}_j\) consists of all actions that represent some synchronous message passing between processes \(P_i\) and \(P_j\). The MDP for PROBMela-program \(\mathcal{P} = P_1 \parallel \ldots \parallel P_n\) where some of the processes \(P_i\) might contain atomic regions is obtained as for the core language, the only difference are additional constraints in the premises of the interleaving and synchronization rule. In the premise of the interleaving rule we add the condition

\[
\text{atomic} = 0 \lor (\text{atomic} = i \land \alpha \in \mathcal{Act}_i \setminus \mathcal{SyncAct})
\]

to ensure that the interleaving rule respects the priority of processes that are inside their atomic regions. Since we treat synchronization requests inside atomic regions as deadlocks, the synchronization rule is restricted to the case where \(\text{atomic} = 0\).
Interleaving rule with respect to atomic

\[ \alpha = Q_i \overset{\gamma}{\rightarrow} \nu \land \eta = g \land (\text{atomic} = 0 \lor (\text{atomic} = i \land \alpha \in \text{Act}_i \setminus \text{SyncAct})) \]

\[ \langle Q_1, \ldots, Q_i, \ldots, Q_n \rangle \overset{\alpha}{\rightarrow} \mu \]

where \( \mu \) is the following distribution

\[ \mu(\langle Q_1, \ldots, Q_{i-1}, R, Q_{i+1}, \ldots, Q_n, \eta' \rangle) = \sum_{a \in \text{Events}_i} \nu(\langle a, R \rangle) \cdot \text{Effect}(a, \eta)(\eta') \]

Synchronization rule with respect to atomic

\[ \alpha = Q_i \overset{\gamma}{\rightarrow} Q'_i \land \beta = Q_j \overset{\delta}{\rightarrow} Q'_j \land \text{atomic} = 0 \]

\[ \langle Q_1, \ldots, Q_i, \ldots, Q_j, \ldots, Q_n, \eta \rangle \overset{\gamma}{\rightarrow} \langle Q_1, \ldots, Q'_i, \ldots, Q'_j, \ldots, Q_n, \eta\langle v \leftarrow \text{value(expr)}\rangle \rangle \]

where \( \gamma \in \text{SyncAct} \) is the action name for handshaking via \( \alpha \) and \( \beta \)

**The goto statement.** For the goto statement, we allow, that labels \( \ell \) can be placed before a PROBMela subprocess \( Q \) by stating \( \ell : Q \). As it is common in many programming languages, these processes can then be jump targets for the command goto. For a fixed PROBMela process \( P \), we require that the labels that appear in labeled commands \( \ell : Q \) of \( P \) must be pairwise distinct and for each target-label \( \ell \) of a jump command goto \( \ell \) of \( P \) there exists a labeled command of the form \( \ell : Q \) in \( P \). The first assumption allows to assign to each label \( \ell \) the unique process \( \text{proc}(\ell) = Q \) of \( P \) such that \( \ell : Q \) is a labeled command in \( P \). The PCG semantics of \( P \) can then be defined as for processes of the core language with the following additional axiom and rule:

\[ \text{goto } \ell; P \overset{\gamma}{\rightarrow} \text{proc}(\ell) \]

\[ Q \overset{\alpha}{\rightarrow} Q' \overset{\ell}{\rightarrow} Q' \]

**Relaxed use of the break command.** A slightly more liberal use of the statement break is permitted in PROBMela than indicated in the grammar of the core language (see definition [3.1.1]). As in Promela or C, the break statement may appear at an arbitrary position in the commands \( P_i \) of a loop \( \text{do} :: g_1 \leftrightarrow_1 P_1 \ldots :: g_n \leftrightarrow_n P_n \text{ od} \). The semantic effect of this break statement is then the immediate termination (i.e., involving a single event) of the nearest enclosing do-od-loop in which the break statement appears.

**Guards of type else.** In choice statements \( \text{if} :: g_1 \leftrightarrow_1 P_1 \ldots :: g_n \leftrightarrow_n P_n \text{ fi} \) and loops \( \text{do} :: g_1 \leftrightarrow_1 P_1 \ldots :: g_n \leftrightarrow_n P_n \text{ od} \), we allow for one guard \( g_i = \text{else} \) which specifies the behaviour if none of the guards \( g_j \) fulfil the current variable evaluation.
3.3. PASM bytecode and LiQuor’s virtual machine

In this section, we outline the approach for an implementation of the MDP semantics given in the sections above and sketch the actual implementation in the model checker LiQuor. For this, a language called PASM (probabilistic assembler language) is employed that is capable of representing all aspects of PCGs with a textual, command based approach. In fact, processes written in PASM can be seen as just a special, command oriented representation of a PCG. Because of this, such processes play a central role in the automated static analysis of PCGs, that precedes partial order reduction (see chapter 6) and the application of advanced fast numerical solving methods (see chapter 4). Moreover, it serves as a basis for the on-the-fly MDP generation, that is prerequisite for the model checking procedures addressed in this thesis. The main advantage of this concept emerges from the fact that, since PASM processes contain all PCG information, the PROBMela program is no longer needed, and the static analysis and the model checking procedures can operate directly on PASM, which is far more suitable for the automatic procession by a computer than the syntactical and semantical more complex PROBMela program. Moreover, PASM appeals as an intermediate language for exchanging models between different model checkers. In [CBGP08], an automated translation process was proposed, that translates PASM processes into the input language of the model checker PRISM, which is strongly inspired by the language of reactive modules. Vice versa, modelling languages that share PCG semantics can be translated into PASM comparably easy and analysed using LiQuor or any further translation process.

Probabilistic assembler language (PASM). The nature of the PASM language is as follows. PASM is a stack oriented bytecode-based language, that is executed on a virtual machine, similar to the virtual machines used in Sun’s implementation of JAVA [GJS05] or Microsofts .NET platform. The virtual machine holds copies of all PASM processes for PCGs $\Psi_1, \ldots, \Psi_i$ of the PROBMela program, where a single PASM process consists of an arbitrary number of parametrized procedures of two types, called edges and fire. A procedure of type edges($Q$) stands for a single PCG location and its outgoing edges $Q \sim^{\not\rightarrow} \mu$. For each $\langle a, Q' \rangle$, where $\mu(\langle a, Q' \rangle) > 0$, a procedure fire($a, Q'$) exists and stands for taking event $a$ and then moving to location $Q'$. Note that the set of locations of a PROBMela process $P$ is given by the set $\text{Loc}(P)$ as defined on page 44. The static structure of the PCG, i.e., the precise interconnection of the locations given by the edge relation $\sim \not\rightarrow$ in the PCG, is maintained by special PASM commands that refer from an edges($Q$) procedure to the corresponding fire($a, Q'$) procedure(s), while each fire procedure contains instructions that refer to its subsequent edges($Q'$) procedure. This is realized by using labels for the
edges and fire procedures that are internally translated into addresses in the address space
of the virtual machine. Since the PASM instructions are of very low syntactical complexity
- they resemble primitive assembler instructions, each with very simple functionality - a
static analysis of the PCG is effectively reduced to a comparably simple parsing step of the
relevant edges and fire procedures and their PASM commands.

For the on-the-fly generation of the MDP, this method is extended from parsing to execut-
ing the PASM procedures when needed in order to generate states from the MDP in an
on-the-fly fashion. For this the virtual machine provides the capability of holding a variable
evaluation and channel configuration $\eta$, that is compatible to the variables and channels de-
clared in the PROBMela program. Procedures edges($Q$) then contain PASM instructions
for checking all guards $g$ against $\eta$ and then inserting triples $\langle p, a, Q' \rangle$ for the enabled edges
into an internal list of enabled edges. The task of determining possible steps of the en-
abled events in a particular state of parallel processes $s = \langle Q_1, ..., Q_n, \eta \rangle$ is achieved by
executing the edges($Q_i$) procedures for all processes $1 \leq i \leq n$. A side effect of these
des-procedure calls inside the virtual machine then is that enabled actions $Q \xrightarrow{g} \mu$ of the
corresponding process are inserted into an internal local list as triples $\langle p, a, Q'_i \rangle$. Similarly,
taking a particular transition in the MDP is achieved by executing the corresponding fire
procedure, which contains PASM instructions to apply potentially necessary changes to $\eta$
and proceed to the subsequent location of the PCG.

Generating the MDP. A simple DFS procedure that generates the MDP of $P = P_1 | | ... | | P_n$
by using the virtual machine (via an application programming interface, API) thus involves
the following steps as a central mechanism. For this note that a state in the MDP is rep-
resented by a tuple $\langle Q_1, ..., Q_n, \eta \rangle$. For now we omit synchronizations, they are covered
below.

⇒
Procedure Generate_MDP:

Input: state \( s = \langle Q_1, ..., Q_n, \eta \rangle \)

Purpose: generate the MDP of \( P = P_1 || | | ... || P_n \)

for \( i = 1, ..., n \) do
  mark \( s \) as being visited
  execute PASM procedures edges\((Q_i)\) (* all enabled events are stored in a local list.
  For an enabled edge \( Q \xrightarrow{a} \mu \), the corresponding
  events are stored as triples \( \langle p, a, Q'_i \rangle \),
  where \( \mu(a, Q'_i) = p > 0. * \)
  for all \( \langle p, a, Q'_i \rangle \) in the local list do
    call PASM procedure fire\((a, Q'_i)\) (* the current state changes from \( s = \langle Q_1, ..., Q_n, \eta \rangle \)
    to \( s' = \langle Q_1’, ..., Q_n’, \eta’ \rangle. *\)
    if \( s' \) has not been visited yet then
      recursively call procedure Generate_MDP\((s’)\).
    end if
  end for
end for

In order to determine whether a state has yet been visited by the procedure the virtual
machine provides an API for storing and retrieving complete state machine configurations
\( \langle Q_1, ..., Q_n, \eta \rangle \) using a fast, hash-table based storing facility. These machine configurations
can be seen as the states of the MDP. A schematic illustration of this principle can be found
in figure 3.7.

Descriptive example. We now give an example that illustrates the on-the-fly generation of
the MDP using PASM in greater detail. Consider the PROBMela program in figure 3.8, that
just involves one single PROBMela process \( P \). In this example, locations \( Q_1, Q_2, Q_3, Q_4 \)
stand for further arbitrary subprocesses of \( P \) that are not covered by our explanations. It
is sufficient to focus on the first events. Locations with numbers in the PCG (right part in
figure 3.8) refer to subprocesses of the according lines in the PROBMela process (left part
of figure). According to the inductive definition of \( \text{Loc}(P) \) (see page 44), we obtain

\[
\text{Loc}(P) = \{ \ P, \\
\text{pif} : 0.3 \mapsto v = v + 1; Q_1 : 0.3 \mapsto c?v; Q_2 : 0.4 \mapsto c?v; Q_3 \text{ fip}, \\
v = v + 1; Q_1, \quad Q_1 \\
c?v; Q_2, \quad Q_2, \\
c?v; Q_3, \quad Q_3, \\
v = 0; Q_4, \quad Q_4 \} ,
\]

59
and we have a procedure $\text{edges}(Q)$ for each $Q \in \text{Loc}(P)$. For the example, we concentrate on the two possible edges of $P$, where location 1 corresponds with $P \in \text{Loc}(P)$, location 7 corresponds with subprocess $c?v; Q_2$ and location 10 corresponds with subprocess $c?v; Q_3$. We then have the following two edges

$$
P \xrightarrow{\text{empty}(c)} \mu, \text{ with } \begin{cases} 
\mu((a, Q_1)) = 0.3 \\
\mu((t, c?v; Q_2)) = 0.3 \\
\mu((t, c?v; Q_3)) = 0.4
\end{cases}
$$

and

$$
P \xrightarrow{v > 5:t} v = 0; Q_4.
$$

where $a = \text{assign}(v, v + 1)$. Recall that $t$ denotes an action that does not have any effect on the evaluations of variables or the configuration of channels (see page 45). The PASM
Listing 3.5: Example Process $P$

```
if ::!empty(c) ->
  pif
  :0.3: ->
    v = v + 1;
    Q1
  :0.3: =>
    c?v;
    Q2
  :0.4: =>
    c?v;
    Q3
fip
  :: v > 5 =>
    v = 0;
    Q4
fi
```

Figure 3.8: Example PROBMela process and corresponding PCG

procedures $\text{edges}(P)$ and

$$
\text{edges}(\text{pif} : 0.3 : v = v + 1; Q_1 : 0.3 : \Rightarrow c?v; Q_2 : 0.4 : \Rightarrow c?v; Q_3 \text{ fip})
$$

together with the PASM procedures

$$
\text{fire}(a, Q_1), \text{fire}(t, c?v; Q_2), \text{fire}(t, c?v; Q_3), \text{ and fire}(t, \text{assign}(v, 0); Q_4)
$$

form the necessary PASM process for the first two potential events in location $P$. These PASM procedures contain instructions for the following virtual-machine operations, that are here presented as (easier to read) pseudo code:

**PASM-procedure** $\text{edges}(P)$:

```
if channel c is not empty then
  call PASM-procedure $\text{edges}(\text{pif} : 0.3 : v = v + 1; Q_1 : 0.3 : \Rightarrow c?v; Q_2 : 0.4 : \Rightarrow c?v; Q_3 \text{ fip})$
end if
if v > 5 then
  add triple $(t, v = 0; Q_4, 1)$ to the local list of enabled actions
end if
return
```
3.3. PASM bytecode and LiQuor’s virtual machine

Chapter 3. PROBMeLa specification language

PASM-procedure edges(pif: 0.3:→ v = v + 1; Q₁: 0.3:→ c?v; Q₂: 0.4:→ c?v; Q₃: fip):

  add triple (a, Q₁, 0.3) to the list of enabled actions
  add triple (t, c?v; Q₂, 0.3) to the list of enabled actions
  add triple (t, c?v; Q₃, 0.4) to the list of enabled actions
  return

PASM-procedure fire(a, Q₁):

  add 1 to the value of v
  (* a = assign(v, v + 1) *)
  set procedure edges(Q₁) as next program location

PASM-procedure fire(t, c?v; Q₂):

  set procedure edges(c?v; Q₂) as next program location

PASM-procedure fire(t, c?v; Q₃):

  set procedure edges(c?v; Q₃) as next program location

PASM-procedure fire(t, assign(v, 0); Q₄):

  set procedure edges(assign(v, 0); Q₄) as next program location

First, recall that procedures edges(Q) stand for PCG locations and their edges $Q \xrightarrow{g} \mu$, where procedures fire(a, Q') stand for executing event a and then moving to location Q'.

Then, by applying a procedure in the spirit of Generate_MDP the effect of semantical rules for one-step guarded commands and two-step guarded commands (see page 46)

$$g \Rightarrow P \xrightarrow{\mu} P \quad \text{and} \quad \frac{P \xrightarrow{h \mu} \nu}{\Rightarrow P \xrightarrow{g \cup h} \nu}$$

are precisely mimicked. This can be seen as follows: For each outgoing edge of $P$ there is an IF-clause that handles the task of determining whether the edge is enabled. In case of the second IF-clause which deals with the guarded command of line 13 the triple that when executed skips to location 14 is inserted. This corresponds to a two-step guarded command. The first IF-clause handles a one-step guarded command by first checking for the satisfaction of guard $\neg \text{empty}(c)$ and then calling procedure

$$\text{edges(pif: 0.3:→ v = v + 1; Q₁: 0.3:→ c?v; Q₂: 0.4:→ c?v; Q₃: fip).}$$

Edges are enabled only if both guards $g$ and $h$ are satisfied, which matches precisely the implication $g \Rightarrow P \xrightarrow{g \cup h} \nu$ of the one-step rule. The procedures fire(...) refer to their
successor locations with special PASM commands at the end of each routine. Procedures that stand for action \( \text{t} \), an event that does not have any effect on variables or channels, do not contain any commands further than that.

**Synchronous actions.** In PROBMela, synchronous communication actions consist of pairs of matching send- and receive events \( d!\text{expr} \) and \( d?v \) that are executed for both involved processes in one step. By doing so, the outcome of \( \text{expr} \) is transferred to variable \( v \) and both processes advance to their successor statement. In the corresponding semantical rule (synchronization rule, see page 51) the channel name is the identifier for any such pairing of send- and receive events. In the corresponding edges-procedures of these events special PASM-commands announce these events to the virtual machine together with the channel-id. After the execution of all edges-procedures for the current state (see, e.g., procedure Generate_MDP on page 58) the announced events are combined by the virtual machine to matching pairs of triples \( \langle 1, d!\text{expr}, Q' \rangle, \langle 1, d?v, Q'' \rangle \) for any synchronous channel \( d \) involved. After that, only triple \( \langle 1, d!\text{expr}, Q' \rangle \) is inserted into the local list of enabled events, and when procedure \( \text{fire}(1, d!\text{expr}, Q') \) is chosen to be executed, the synchronous event is executed as a whole according to the explanations above.

This concludes the explanations on the modelling language PROBMela.
In the previous chapter, we explained the core features of the PCG- and MDP semantics of the probabilistic modelling language PROBMela. We now address methods for the efficient quantitative analysis of probabilistic systems represented by MDPs that have been implemented in the tool LiQuor. Figure 4.1 illustrates the quantitative model checking approach as it is treated in this thesis and shows how the previous and the following chapters fit into figure 1.1 (see page 8).

For a given PROBMela program $P_1 \parallel \ldots \parallel P_n$, we employed a formal operational semantics in the previous chapter 3 to generate PCGs, that are represented as PASM programs. These PASM programs contain assembler-like micro commands for guards $g$ and events $a$ that mimick the behaviour and the effect of PCG transitions $l \xrightarrow{g,a} \nu$ and are designed for an automated generation of the corresponding MDP by the virtual machine implementation of
In this chapter (chapter 4) as well as in chapter 6 different reduction techniques are discussed. In general, these reduction techniques aim at reducing the size of $\mathcal{M}$ under consideration of the context. Thus, the different techniques cope with the omnipresent state explosion problem, but in different stages of the analysis. The general approach, however, is that while they reduce the size of the problem they preserve maximal and minimal probabilities for stutter-invariant LT-properties.

The reduction technique discussed in chapter 6 is the ample set method for partial order reduction of MDPs, which reduces the system $\mathcal{M}$ while preserving its stutter-invariant $\omega$-regular properties ([BGC04], see also [dN04] and [Gro08]). More precisely, instead of the original MDP crossproduct $\mathcal{M} \otimes \mathcal{A}$ it generates a reduced sub-MDP $\widehat{\mathcal{M}} \otimes \mathcal{A}$ starting from the PASM processes for PROBMela processes $P_1, \ldots, P_n$ and the PASM representation of the property automaton $\mathcal{A}$ such that the worst-case probabilities in $\mathcal{M} \otimes \mathcal{A}$ and $\widehat{\mathcal{M}} \otimes \mathcal{A}$ for the acceptance condition agree. Since $\mathcal{M}$ is generated without generating $\mathcal{M}$ in its entirety, it is an on-the-fly method. The main challenge in the context of the implementation in the model checker LiQuor is providing algorithmical procedures that check the ample-set conditions (locally) in $\mathcal{M}$ or $\widehat{\mathcal{M}}$ respectively, that need to be satisfied in order to preserve all stutter-invariant $\omega$-regular properties.

This chapter (chapter 4) is dedicated to some drastic algorithmical improvements in the later stage of the model checking process, namely of the worst-case reachability analysis of the product MDP $\mathcal{M} \otimes \mathcal{A}$. We show how this system can be reduced drastically before the reachability analysis is executed (section 4.3), thus leading to improved performance of the numerical solving process. In this context, we report on algorithmical improvements of the solving step itself (sections 4.1 and 4.2). Most of these results were originally published in [CBGK08] and [BGC09b].

In chapter 5 we show how the concept of fairness can be applied to the worst-case analysis of the product MDP $\mathcal{M} \otimes \mathcal{A}$.

**Probabilistic reachability analysis.** Calculating the values $\Pr_s^{\text{max}}(\Diamond B)$, i.e., calculating the maximum probability for reaching a goal set $B \subseteq S$ in an MDP $\mathcal{M} = (S, \text{Act}, \rightarrow, s_0, AP, L)$, can be done via the solving of a linear optimization problem, see Theorem 2.3.2 of the form

- If $s \in B$, then $x_s = 1$.
- If $s \notin B$, then

\[
    x_s = \max \left\{ \sum_{t \in S} \Pr(s \xrightarrow{\alpha} t) \cdot x_t \mid \alpha \in \text{Act}(s) \right\},
\]
where the vector \((x_s)_{s \in S}\), with \(x_s = \Pr_{s}^{\text{max}}(\Diamond B)\), is the solution vector and the goal is to compute \(\Pr_{s_0}^{\text{max}}(\Diamond B)\) (for the initial state \(s_0\) of \(M\)). To compute the value \(\Pr_{s_0}^{\text{max}}(\Diamond B)\) in practice, this equation system can be rephrased as the following linear program:

- If \(s \in B\), then \(x_s = 1\).
- If \(s \notin B\), then \(0 \leq x_s \leq 1\) and for all actions \(\alpha \in \text{Act}(s)\):
  \[
x_s \geq \sum_{t \in S} \Pr(s \xrightarrow{\alpha} t) \cdot x_t
  \]

With the objective to minimize \(\sum_{s \in S} x_s\), the vector \((x_s)_{s \in S}\) with \(x_s = \Pr_{s}^{\text{max}}(\Diamond B)\) yields a solution for the problem. The case of calculating \(\Pr_{s}^{\text{min}}(\Diamond B)\) can be formulated very similarly, such that we concentrate most of the presentation on the \(\Pr_{s}^{\text{max}}\) case, knowing that the arguments carry over in a straightforward way. In this chapter, several improvements are discussed that concentrate on algorithmical procedures for solving of this linear optimization problem. The methods discussed here assume the standard preprocessing to compute the sets \(S_1\) consisting of all states \(s\) where \(\Pr_{s}^{\text{max}}(\Diamond B) = 1\) and \(S_0\) consisting of all states \(s\) that cannot reach \(B\) (i.e., where \(\Pr_{s}^{\text{max}}(\Diamond B) = 0\)). The set \(S_0\) can be obtained by a standard reachability analysis in the underlying graph of \(M\). The set \(S_1\) can be computed by means of a graph algorithm in time \(O(\text{size}(M) \times |S|)\) [CY95]. After having computed \(S_0\) and \(S_1\), there is no need to distinguish the states in \(S_0\) and \(S_1\), respectively. Furthermore, their outgoing transitions are not of interest anymore. Thus, we can safely collapse all states in \(S_0\) resp. \(S_1\) to a single terminal state \(s_0\) resp. \(s_1\).

The interesting step is the computation of the values \(x_s = \Pr_{s}^{\text{max}}(\Diamond B)\) for the states \(s \in S_? = S \setminus (S_0 \cup S_1)\), which can be done by using classic algorithms, e.g., the simplex algorithm. In the context of quantitative model checking, such classic methods perform comparably bad. There are two major reasons for that: 1. The particular structures of the linear program and the associated MDP \(M\) are not exploited by these algorithms, and 2. the widely applied strategy to use highly optimized solving libraries such as lpsolve [Ber08] creates a bottleneck, since the linear optimization problem needs to be exported to these libraries (see, e.g., [BBC+08]). In practice, a (in our context comparably simple) method called value iteration ([Bel58] see also, e.g., [Put94]) performs much better. Here, the solution vector \((x_s)_{s \in S_?}\) is iterated according to the requirement

\[
x_s^{(0)} = 1 \text{ for all states } s \in B, \text{ and } x_s^{(n+1)} = \max \left\{ \sum_{t \in S} \Pr(s \xrightarrow{\alpha} t) \cdot x_t^{(n)} \mid \alpha \in \text{Act}(s) \right\}
\]

until a fixpoint is reached. For fixed \(s\), the sequence \(x_s^{(0)}, x_s^{(1)}, x_s^{(2)}, \ldots\) is increasing and its limit is \(x_s\). If convergence of the solution vector \((x_s)\) is reached, then a deterministic
memoryless scheduler $\mathcal{U}$ for which it holds that $\Pr_{s_0}^{M_\mathcal{U}}(\Diamond B) = \Pr_{s_0}^{\text{max}}(\Diamond B)$ can simply be constructed by setting
\[
\mathcal{U}(s) = \arg \max_{\alpha \in \text{Act}} \left\{ \sum_{t \in S} \mathbf{P}(s \xrightarrow{\alpha} t) \right\}.
\]
The halting condition of an approximation algorithm is given by a threshold $\varepsilon \in [0, 1]$ and requires that the elementwise difference between the current vector $(x_s)^{(i)}$ and the previous vector $(x_s)^{(i-1)}$ is less than $\varepsilon$.

Table 6.2 contains a brief comparison of solving times obtained with the external library `lpsolve` [Ber08] that includes a highly efficient implementation of the Simplex algorithm [Dan66] for solving linear programs and a rather naïve implementation of value iteration, both done with models that where generated by LiQuor using PROBMela. The abbreviations in the first column stand for the models Randomized Dining Philosophers (DP) [LR81], Dining Cryptographers (DC) [Cha88], Leader Election (LE) [IR90] and a proprietary model of a failure-scheme in the UMTS key-synchronisation protocol developed with cooperation with a large mobile communication company (UMTS). The values in brackets () correspond to the number of processes that are present in the model.

<table>
<thead>
<tr>
<th>model</th>
<th>states</th>
<th>transitions</th>
<th>inequations</th>
<th>Simplex</th>
<th>naïve value iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP(6)</td>
<td>$3.3 \cdot 10^6$</td>
<td>$26 \cdot 10^6$</td>
<td>$655465$</td>
<td>$&gt;1h$</td>
<td>$699s$</td>
</tr>
<tr>
<td>DC(5)</td>
<td>$1.64 \cdot 10^6$</td>
<td>$6.32 \cdot 10^6$</td>
<td>$394358$</td>
<td>$&gt;1h$</td>
<td>$302s$</td>
</tr>
<tr>
<td>RG(4)</td>
<td>$488902$</td>
<td>$661307$</td>
<td>$602796$</td>
<td>$&gt;1h$</td>
<td>$689s$</td>
</tr>
<tr>
<td>LE(6)</td>
<td>$1.1 \cdot 10^7$</td>
<td>$6.2 \cdot 10^7$</td>
<td>$971874$</td>
<td>—</td>
<td>$&gt;1h$</td>
</tr>
<tr>
<td>UMTS</td>
<td>$244233$</td>
<td>$253629$</td>
<td>$165393$</td>
<td>$&gt;1h$</td>
<td>$286s$</td>
</tr>
</tbody>
</table>

Table 4.1: Comparison of solving LPs with `lpsolve` (simplex) and value iteration

The table clearly illustrates that in the context of solving linear programs for quantitative model checking as illustrated above, even a naïve implementation of value iteration outperforms a highly sophisticated implementation of the Simplex algorithm by several orders of magnitude. Since value iteration turns out to be much more efficient than the linear programming approach the focus of this chapter will lie on high-performant implementations of the value iteration method in the model checker LiQuor.

Remark: An approach closely related to value iteration is a method called policy iteration [How60], which is widely regarded as being a method superior to value iteration (see 1In the literature, value iteration is typically used to compute discounted average rewards in MDPs. In our setting, we need a reformulation of the value iteration algorithm to compute maximal reachability probabilities.}
In policy iteration, starting with an initial policy $U^{(0)}$, the process of computing a policy $U$ for which $\text{Pr}^{M_{\pi_0}}(\Diamond B)$ is maximal, involves two separate steps that are executed iteratively (for $n = 0, 1, \ldots$):

**Step 1:** Calculate $x_s = \sum_{s' \in S} P(s \xrightarrow{\pi_0^{(n)}} s') \cdot x_{s'}$ for all $s \in S$

**Step 2:** Set $U^{(n+1)}(s) = \arg \max_{\alpha \in \text{Act}} \left\{ \sum_{s' \in S} P(s \xrightarrow{\alpha} s') \cdot x_{s'} \right\}$ for all $s \in S$

The solution is then calculated by repeatedly solving step 1, followed by an iteration of step 2 to obtain (a possibly improved) policy $U^{(n+1)}$. This is repeated until convergence is reached, i.e., until $U^{(n+1)}$ brings no further changes to $U^{(n)}$. The advantage of this approach in general lies in its increased flexibility, since step 1 represents an isolated sub-task of solving a system of linear equations over $(x_s)$, for which an arbitrary method of choice for solving equation systems may be applied. This approach was further improved by [PS78] and [vN76] in a sense that the equation system in step 1 is not necessarily solved exactly before an iteration in step 2 is done, such that policy iteration may in many cases need less overall calculation steps than value iteration. It is, however, also the case that the optimal number of calculation steps cannot be found easily and depends on the concrete numerical problem and its real-world application in the background. Consider for example a setting in the context of motion planning for an agent, that needs to make movement decisions in a spacial environment that is represented by an MDP that contains implicit information about an abstract cost function relevant to the agent. Then, an optimal policy $U$ is sought for the best movement, but there may be only little interest in the exact values of $(x_s)$. It may even be the case, that finding the best policy is not necessary and only a "good" policy is needed to succeed, i.e., a policy with a low Hamming distance to the optimal policy. The authors of [PRD96] show that in such cases policy iteration is usually the better, more flexible choice. However, in the context of quantitative model checking based on Markov decision processes, we have face the situation that we want to calculate $\text{Pr}^{\pi_0^{\text{max}}}(\Diamond B)$, and thus the vector $(x_s)$, as exact as possible. The authors argue, that in this case, especially for larger problem instances, value iteration performs better or equally good. Note also, that an implementation of policy iteration requires the maintaining of an additional array for the current policy $U^{(n)}$ in the implementation. Of course it is still very interesting to perform experiments with policy iteration in the context of quantitative analysis, e.g., because policy iteration yields a concrete scheduler, which is important in the context of counterexamples. The research on suitable forms for the presentation of counterexamples in quantitative model checking is still in an earlier, but very promising.

---

By "exactly" we mean here either mathematical exact or approximated to a high degree.
stage of development (see, e.g., [AL09, HK07, Kat10]). In this thesis we report only on results that were obtained with variants of value iteration and postpone an investigation of policy iteration for quantitative model checking to future work.

4.1. Value iteration

In the naïve approach to value iteration, each iteration step sweeps through all states \( s \in S \) in some fixed order and recalculates (attempts to improve) the values for \( x_s = \text{Pr}_s^{\text{max}}(\diamond B) \).

A variant of the value iteration algorithm, that is more suitable for a productive implementation, is given with algorithm \text{value iteration} which can be understood as a repeated backward analysis that starts in the states in the goal set \( B \) and successively propagates positive probabilities to reach \( B \), but ignores states where the values of the successors remained unchanged. More precisely, Algorithm \text{value iteration} only touches those states \( s \) that have a successor \( t \) where the value \( x_t \) has been changed since the last modification of \( x_s \). The calculations are started with states \( s \in S \). The following example illustrates this:

**Example 4.1.1.** Consider figure 4.2, where the naïve value iteration and value iteration, as described above, are compared with respect to an example MDP. Assume, that we calculate \( \text{Pr}_s^{\text{max}}(\diamond B) \), and \( B = \{s_3, s_6\} \). Consequently we obtain, that \( S_1 = \{s_1, s_3, s_6\} \) and the value iteration iterates over the set \( S_7 = \{s_0, s_2, s_5\} \). When naïve value iteration is applied, the algorithm has to go through 4 iterations (iterations \((x_s)^{(0)}\) to \((x_s)^{(3)}\)) to reach convergence, which accumulates to a total of \( 4 \cdot 3 = 12 \) updates, since all elements of \( S_7 \) are updated in the 4 iterations. In contrast to this, the backward-aware value iteration approach

\[
\begin{align*}
S_0 = \{s_3, s_7\}, S_1 = \{s_1, s_2\}, S_7 = \{s_0, s_3, s_4, s_5\} \\
\text{Iterations of naïve implementation:} \\
\begin{array}{cccccccc}
\text{iter} & x_{s_0} & x_{s_1} & x_{s_2} & x_{s_3} & x_{s_4} & x_{s_5} & x_{s_6} \\
(x_s)^{(0)} & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
(x_s)^{(1)} & 1 & 1 & 1 & 0.5 & 1 & 1 & 1 \\
(x_s)^{(2)} & 1 & 1 & 0.45 & 1 & 0.5 & 1 & 1 \\
(x_s)^{(3)} & 1 & 1 & 0.45 & 1 & 0.5 & 1 & 1 \\
\end{array}
\end{align*}
\]

\[4 \cdot 3 = 12 \text{ updates}\]

\[
\begin{align*}
\text{Iterations of "faster" value iteration:} \\
x_{s_0} = \frac{1}{10}, x_{s_5} = \frac{1}{2}, x_{s_2} = 0.45, x_{s_0} \text{ gets value 1} \\
4 \text{ updates}
\end{align*}
\]

Figure 4.2: MDP for comparison of naïve value iteration and "faster" value iteration

\begin{center}
\text{Figure 4.2: MDP for comparison of naïve value iteration and "faster" value iteration}
\end{center}
just needs 4 updates to complete its task.

**Procedure** `value_iteration`.

<table>
<thead>
<tr>
<th>Purpose</th>
<th>calculate the solution for $\Pr^\max(\Diamond B)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>sets $S_0$, $S_1$, $S_?$</td>
</tr>
<tr>
<td>Output</td>
<td>the solution vector $(x_s)<em>{s \in S</em>?}$, $x_s = \Pr^\max_s(\Diamond B)$</td>
</tr>
</tbody>
</table>

```
for all states $s \in S_?$ do
    $p(s) := \max\{\sum_{s_1 \in S_1} \mu(s_1) \mid s \rightarrow \mu\}; \Delta(s) := p(s)$
end for
$T := \{t \in S_? \mid p(t) > 0\}$;
repeat
    if $T \neq \emptyset$ then
        pick some state $t \in T$ and remove $t$ from $T$
        for all transitions $s \rightarrow \mu$ where $\mu(t) > 0$ do
            $q := \mu(S_1) + \sum_{s' \in S_?} \mu(s') \cdot p(s')$;
            if $q > p(s)$ then
                $\Delta(s) := q - p(s)$ and $p(s) := q$, and add $s$ to $T$
            end if
        end for
    end if
until $T = \emptyset$ or $\max_{s \in S_?} \Delta(s) < \epsilon$
```

### 4.2. Prioritized updates

There are several possibilities to realize Algorithm `value_iteration` that differ in the organization of the set $T$ and picking a state $t \in T$. In example 4.1.1, $T$ was organized to pick states according to the ascending numbers of the state indices, which could correspond for instance with the DFS visiting order of the state space generation process. This order agrees with realizing $T$ as a queue and generates successively the same approximations $(x_s)^{(i)}$ as naïve value iteration, only with dramatically less updates. There are, however, cases where this order can be further improved by using data structures different from a queue. If there are small cyclic areas in the MDP where convergence is slow, while other areas of the MDP just need a few iterations, then with an implementation of $T$ as a queue many unnecessary calculations might take place to improve the values $p(s)$ for states outside this cyclic area, even if the computed values for them are already sufficiently precise. An attempt to avoid this phenomenon uses a representation of $T$ by means of a priority queue where the differences $\Delta(s)$ between the new and the old values for $p(s)$ are used to derive the priorities. This approach is closely related to the concept of priority sweeping [MA93] which has been
proposed in the context of prediction and reinforcement learning. Such priority-based variants of value iteration have been shown to be more efficient than ordinary value iteration for the computation of discounted expected rewards \cite{DG06}. For this, consider the following example:

**Examples.** In figure 4.3, an MDP is given, that contains a cycle with relatively slow convergence behaviour (state $s_0$ with selfloop with probability $\frac{1}{10}$). In addition to that, a longer chain from state $s_0$ to $s_7$ connects state $s_0$ with a relatively high probability source. Applying only faster value iteration to this example leads to 11 updates, when $T$ is organized as a queue, and to 9 updates, when organizing $T$ as a priority queue. (We use $\epsilon = 10^{-4}$ as the halting condition and write $s_i$ instead of $x_{s_i}$ in the diagram.)

![Diagram](image)

In figure 4.3, an MDP is given, that contains a cycle with relatively slow convergence behaviour (state $s_0$ with selfloop with probability $\frac{1}{10}$). In addition to that, a longer chain from state $s_0$ to $s_7$ connects state $s_0$ with a relatively high probability source. Applying only faster value iteration to this example leads to 11 updates, when $T$ is organized as a queue, and to 9 updates, when organizing $T$ as a priority queue. (We use $\epsilon = 10^{-4}$ as the halting condition and write $s_i$ instead of $x_{s_i}$ in the diagram.)

<table>
<thead>
<tr>
<th>updates</th>
<th>$T$ as a queue</th>
<th>$T$ as a priority queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>first init: $s_0 = 0.1, s_5 = 0.9$</td>
<td>$(s_0, 0.1)$, $(s_5, 0.9)$</td>
<td>$(s_0, 0.1)$, $(s_5, 0.9)$</td>
</tr>
<tr>
<td>update 3 $s_0 = 0.11$</td>
<td>$(s_0, 0.9)$, $(s_5, 0.01)$</td>
<td>update 3 $s_4 = 0.9$</td>
</tr>
<tr>
<td>update 4 $s_4 = 0.9$</td>
<td>$(s_0, 0.01)$, $(s_4, 0.9)$</td>
<td>update 4 $s_3 = 0.9$</td>
</tr>
<tr>
<td>update 5 $s_5 = 0.111$</td>
<td>$(s_4, 0.9)$, $(s_5, 0.001)$</td>
<td>update 5 $s_8 = 0.83$</td>
</tr>
<tr>
<td>update 6 $s_3 = 0.9$</td>
<td>$(s_0, 0.001)$, $(s_3, 0.9)$</td>
<td>update 6 $s_9 = 0.903$</td>
</tr>
<tr>
<td>update 7 $s_0 = 0.8311$</td>
<td>$(s_0, 0.99)$, $(s_3, 0.7201)$</td>
<td>update 7 $s_5 = 0.9103$</td>
</tr>
<tr>
<td>update 8 $s_8 = 0.90311$</td>
<td>$(s_8, 0.7201)$, $(s_8, 0.07201)$</td>
<td>update 8 $s_0 = 0.91103$</td>
</tr>
<tr>
<td>update 9 $s_9 = 0.910311$</td>
<td>$(s_9, 0.07201)$, $(s_7, 0.007201)$</td>
<td>update 9 $s_0 = 0.911103$</td>
</tr>
<tr>
<td>update 10 $s_0 = 0.9110311$</td>
<td>$(s_0, 0.007201)$, $(s_0, 0.0007201)$</td>
<td>update 10 $s_0 = 0.911103$</td>
</tr>
<tr>
<td>update 11 $s_0 = 0.91110311$</td>
<td>$(s_0, 0.0007201)$, $(s_0, 0.00007201)$</td>
<td>update 11 $s_0 = 0.911103$</td>
</tr>
</tbody>
</table>

The reason for this reduction in costs is that with a priority queue, the high probability values are relayed faster from states $s_5$ to $s_3$ with updates 3, 4 and 5 without intermediate updates of state $s_0$. As a result, the probability value of $s_0$ comes earlier close to its convergence value ($\frac{90}{100} = 0.91$) and thus the calculation needs less updates to converge below $\epsilon$. This strategy can generally lead to further reduction of the amount of needed updates when the prioritized higher probability values lead to an early termination, as illustrated in figure 4.4. Here, $s_0$ adopts the value of $s_3$ as soon as possible with update 5. Without action $\beta$, the probability value of $s_0$ would slowly converge to only $\frac{1}{5}$, which is less than its designated
Chapter 4. Efficient probabilistic reachability analysis

4.2. Prioritized updates

Figure 4.4: Second comparison of $T$ realized as queue (left) and as priority queue (right)

An artificially extreme and more general example which illustrates this aspect is given in figure 4.5, where $s_0$ receives updates via actions $\alpha, \beta, \gamma, \ldots, \iota$. All these actions cause a new costly convergence process in state $s_0$, when the new value exceeds the old value in $s_0$. In the example, the chains "behind" these actions are chosen to yield ascending probability values, i.e., the topmost chain will yield a low probability $10^{-4}$, the second topmost a slightly higher value $2/11$, etc. (the shaded states on the right belong to $S_1$ as in the example above). Apparently the best strategy is to propagate probability value $9/11$ from the chain at the bottom before all others, followed by successively propagating the next chain, etc. This order is chosen by the prioritized value iteration.

Note, that when calculating $x_a = P_{s\rightarrow s}^{\min}(\Delta B)$ with

$$x_a^{(0)} = 1 \text{ for all states } s \in B, \text{ and } x_a^{(n+1)} = \min \left\{ \sum_{t \in S} P(s \rightarrow t) \cdot x_t^{(n)} \mid \alpha \in \text{Act}(s) \right\},$$

the priority queue that favors large updates over smaller ones is also the data structure of choice for $T$, as the convergence arguments are the same.

Since maintaining an ideal priority queue is unnecessarily inefficient compared to the benefit, an approach that represents $T$ by a fixed number of buckets $T_0, \ldots, T_{m-1}$ seems feasible.
4.2. Priorized updates

Chapter 4. Efficient probabilistic reachability analysis

By doing so, $T_i$ can be seen as a list containing all elements of $T$ where $\theta_i \leq \Delta(s) < \theta_{i+1}$ and where the thresholds $\theta_i$ are chosen in any way such that $\theta_0 = 0 < \theta_1 \leq \epsilon < \ldots < \theta_m = 1$ (e.g., we may deal with $\theta_i = \frac{i}{m}$ if $\frac{1}{m} \leq \epsilon$, or simply $m = 2$ and $\theta_1 = \epsilon$). In each iteration of Algorithm value_iteration, we pick an element of the bucket $T_i$ where $i$ is the maximal index such that $T_i$ is nonempty. The halting condition of Algorithm value_iteration can be replaced by the requirement that the buckets $T_1, \ldots, T_{m-1}$ are empty. With this bucket approach, we still have the advantages of the priority-based approach (faster convergence), but avoid the time-consuming organization of the priority queue.

Compared to standard value iteration, the memory requirements of Algorithm value_iteration are slightly higher, since in addition to the probability vector a representation for the set $T$ (together with the values $\Delta(s)$ in case of an implementation with priority queues) is needed. Note that the approaches with a queue or buckets do not require the explicit representation of the values $\Delta(s)$.\footnote{A single probability vector $(x_s)$ suffices in an implementation of the standard value iteration. Note that when computing $x_s^{(i+1)}$ then one might use the already computed values $x_s^{(i+1)}$ rather than $x_s^{(i)}$. Furthermore, the maximal difference between the new value $x_s$ and the previous value of $x_s$ can be stored in a single variable.}

Figure 4.5: Extreme example for prioritized fast value iteration
4.3. Structured solving approaches

In the preceding paragraphs, techniques were discussed that improve solving times for linear programs which emerge in quantitative model checking of LT-properties. In the following we discuss related techniques that tackle this problem differently not by improvements in the efficiency of the numerical solving step, but by constructing special reduced linear programs, that are much easier to solve, thus supporting the problem indirectly. As we will see, these techniques rely on graph-based insights rather than on numerical aspects and are meant to replace the original MDP $\mathcal{M}$ by a reduced one (i.e., an MDP $\mathcal{M}'$ with fewer states $S' \subset S$), while preserving the maximal and minimal reachability probabilities. Thus, these techniques can be applied before the numerical solution step, performed, e.g., by means of value iteration. In contrast to methods like partial order reduction, these techniques are not applied in an on-the-fly fashion. This reduction, i.e., the construction of $\mathcal{M}'$, takes place after the state space search has established the complete MDP $\mathcal{M}$.

The MEC-quotient approach. The MEC-quotient approach is a reduction technique that collapses all states that belong to the same maximal end component in $\mathcal{M}$ to a single meta state before constructing the linear program, which is justified by Lemma 4.3.1 given below. This technique appears particularly useful in the context of model checking of LT-properties for MDPs, because the computation of the maximal end components is, anyway, an integral part of the model checking process (see [dA97] and section 2.2).

Lemma 4.3.1. Given an MDP $\mathcal{M} = (S, Act, \rightarrow, s_0, AP, L)$ and a target set $B \subseteq S$, then for all states $s, t \in S$ that belong to the same end component $\mathcal{E} = (T, Act_\mathcal{E})$ in $\mathcal{M}$: $s$ and $t$ belong to $\mathcal{E}$, then

$$Pr_{s}^{\text{max}}(\Diamond B) = Pr_{t}^{\text{max}}(\Diamond B).$$

Proof. Suppose that $Pr_{s}^{\text{max}}(\Diamond B) > Pr_{t}^{\text{max}}(\Diamond B)$. Let $\mathcal{U}$ be a (memoryless) scheduler that maximizes the probability to reach $B$ for all states. According to Lemma 2.2.9 there exists a scheduler $\mathcal{S}$ that ensures that

$$Pr_{u}^{\mathcal{S}}(\{\pi \in \text{Paths} : \lim(\pi) = \mathcal{E}\}) = 1$$

for any state $u \in T$. We now assume, that $t$ is the starting state and regard the third scheduler $\mathcal{V}$ which first behaves identical to $\mathcal{S}$ as long as state $s$ is not reached. As soon as $\mathcal{V}$ reaches state $s$ (which happens with probability 1 when imitating scheduler $\mathcal{S}$), $\mathcal{V}$ behaves as $\mathcal{U}$, thus maximizing the probability to reach $B$. Clearly, the probability to reach $B$ from state $t$ when using scheduler $\mathcal{V}$ agrees with the probability to reach $B$ from $s$ under scheduler $\mathcal{U}$. But then $Pr_{s}^{\text{max}}(\Diamond B) \leq Pr_{t}^{\text{max}}(\Diamond B)$, which is a contradiction to the premise. \qed
4.3. Structured solving approaches

Chapter 4. Efficient probabilistic reachability analysis

Justified by Lemma 4.3.1, we now define the MEC-quotient $\mathcal{M}_{/\text{mec}}$ as the MDP resulting from $\mathcal{M}$ by identifying the states of each maximal end component. More precisely, we define $\text{mec}: S \rightarrow S \cup \mathcal{MEC}$ as follows:

$$\text{mec}(s) = \begin{cases} \mathcal{E}, & \text{if there is a maximal end component } \mathcal{E} = (T, \text{Act}_\mathcal{E}), \text{ such that } s \in T \\ s, & \text{otherwise.} \end{cases}$$

Note that each state $s$ is contained in at most one maximal end component. For $U \subseteq S$ we define $\text{mec}(U) = \bigcup_{s \in U} \text{mec}(s)$, where we identify any end component $\mathcal{E} = (T, \text{Act}_\mathcal{E})$ with the set $T$ of states in $\mathcal{E}$. With this, the state space of the MDP $\mathcal{M}_{/\text{mec}} = (S', \text{Act}', \rightarrow_{\text{mec}}, s'_0, \text{AP}', L')$ is $S' = \text{mec}(S)$. The initial state $s'_0$ of $\mathcal{M}_{/\text{mec}}$ is $\text{mec}(s_0)$. Since the MEC-quotient $\mathcal{M}_{/\text{mec}}$ is used only for constructing a very efficient pendant of the linear program of $\mathcal{M}$, the action names in $\text{Act}'$ and the labelings $L'$ are not relevant for our purposes and not addressed any further. Formally, we use singletons for $\text{AP}'$ and $\text{Act}'$. The transition relation of $\mathcal{M}_{/\text{mec}}$ is the smallest relation $\rightarrow_{\text{mec}}$ such that the following property holds.

- For any transition $s \rightarrow \mu$ in $\mathcal{M}$ there exists a transition $\text{mec}(s) \rightarrow_{\text{mec}} \nu$, where $\nu \in \text{Distr}(\text{mec}(S))$ is given by $\nu(D) = \sum_{t \in D} \mu(t)$ for all $D \in \text{mec}(S)$.

Furthermore, $\text{mec}(B)$ replaces the target set $B$ for $\mathcal{M}_{/\text{mec}}$.

**Theorem 4.3.2.** For all states $s$ in $\mathcal{M}$ we have:

$$\max_{U \in \text{Sched}} \Pr_{s \sim \mathcal{M} U} (\diamond B) = \max_{U' \in \text{Sched}} \Pr_{s \sim \mathcal{M}_{/\text{mec}} U'} (\diamond \text{mec}(B)).$$

**Proof.**

$\leq$: For any scheduler $U$ of $\mathcal{M}$ and finite each path $\pi \in \text{Paths}_{\text{fin}}^{\mathcal{M} U}$ induced by $U$ there is clearly a path $\hat{\pi}$ in the quotient system induced by a scheduler $U'$ with $P(\pi) = P(\hat{\pi})$.

$\geq$: For any scheduler $U'$ of $\mathcal{M}_{/\text{mec}}$ and finite each path $\hat{\pi} \in \text{Paths}_{\text{fin}}^{\mathcal{M}_{/\text{mec}} U'}$ induced by $U'$ there is clearly a path $\pi$ in the quotient system induced by a scheduler $U$ with $P(\hat{\pi}) = P(\pi)$.

From this it follows immediately that the maximum probabilities of the theorem agree.

Thanks to Theorem 4.3.2, maximal reachability probabilities in $\mathcal{M}$ can be obtained by first applying graph-algorithms to obtain the maximal end components [CY95, dA97] and the partition of $S' = \text{mec}(S)$ into the sets $S'_0$, $S'_1$ and $S'_?$ where the maximal probability to reach $B$ in the MEC-quotient is 0,1 or strictly between 0 and 1, respectively. Finally, all variants
of value iteration presented in the first part of this chapter can be applied to calculate the
maximal reachability probabilities for the states in \( S' \) in the MEC-quotient \( \mathcal{M}_{/\text{mec}} \). The
length of the probability vector agrees with the size of \( S' \) which is the number of states
\( \min_{\text{mec}}(S) \) that are contained in \( S' \). Thus, it is shorter than the probability vector for \( \mathcal{M} \).
However, the number of transitions in the MEC-quotient is the same as in \( \mathcal{M} \), but obviously
contains many unnecessary transitions. These transitions are self-loops that represent in
\( \mathcal{M}_{/\text{mec}} \) transitions that connected states of the same maximal end component in
\( \mathcal{M} \) (see figure 4.6 for an example MDP).

![Diagram](image)

Figure 4.6: An MEC-quotient example

Under certain side conditions, a large amount of these transitions can be removed. In order
to do so, we switch from \( \mathcal{M}_{/\text{mec}} \) to the MDP \( \mathcal{M'}_{/\text{mec}} \) that results from \( \mathcal{M}_{/\text{mec}} \) by the
following modifications:

1. **Remove self-loops**: each edge \( s \to \mu \) in \( \mathcal{M}_{/\text{mec}} \) with \( \mu(s) = 1 \) is removed, and

2. **Normalize self-references**: each edge \( s \to \mu \) in \( \mathcal{M}_{/\text{mec}} \) with \( 1 > \mu(s) > 0 \) is replaced
   with the edge \( s \to \nu \) where \( \nu(s) = 0 \) and \( \nu(t) = \frac{\mu(t)}{1-\mu(s)} \) for all \( t \in S \setminus \{s\} \).

This modification of \( \mathcal{M}_{/\text{mec}} \) does also not affect the maximal or minimal reachability prob-
abilities, but they can increase the efficiency of the value iteration procedure.

Finally, to further reduce the number of outgoing edges for some states \( s \in S' \), we use the
following observation. Let \( s \to \mu \) and \( s \to \nu \) be edges in \( \mathcal{M}_{/\text{mec}} \). If \( \mu(S_1) \leq \nu(S_1) \) and
\( \mu(t) \leq \nu(t) \) for all states \( t \in S' \) then the maximal reachability probabilities do not change if
the edge \( s \to \mu \) is removed. Note that these modifications may not be applied in preceding
stages of the analysis such as the calculation of maximal accepting end components. On
the other hand, this technique may be applied to any MDP \( \mathcal{M} \) and is not exclusive for the
quotient system \( \mathcal{M}_{/\text{mec}} \).
For minimal reachability probabilities, analogous results can be established, provided that the states in $B$ have been made terminal. Then, each state in $B$ constitutes a singleton maximal end component and we get

$$
Pr_{s}^{\min}(\Diamond B) = Pr_{t}^{\min}(\Diamond B)
$$

for all states $s, t$ where mec$(s) = \text{mec}(t)$ as in Lemma 4.3.1 and

$$
Pr_{s,\text{mec}}^{\min}(\Diamond B) = Pr_{t,\text{mec}}^{\min}(\Diamond \text{mec}(B))
$$

as in Theorem 4.3.2.

**Blockwise value iteration in the MEC-quotient.** After having discussed improved variants of value iteration and the construction of a reduced quotient system $M'/\text{mec}$ respectively $M'_{\text{mec}}$ from a given MDP $M$ with the goal of a drastically smaller numerical problem to be solved, we now discuss a third improvement in methodology called blockwise value iteration.

Since all maximal end components were collapsed in the quotient system $M'/\text{mec}$, it does not contain any further maximal end components, but it still might contain cycles (see. figure 4.8).

We now consider the underlying digraph $G$ of the MEC-quotient $M'/\text{mec}$, where the nodes of $G$ are the states in the MEC-quotient, i.e., the maximal end components of $M$. The edge relation $\rightarrow_{G}$ is given by $E \rightarrow_{G} D$ if and only if $E \neq D$ and there is a transition $E \rightarrow_{\text{MEC}} v$ in $M'/\text{mec}$ such that $\nu(D) > 0$.

We now compute a topological order $C_{1}, C_{2}, \ldots , C_{k}$ of the strongly connected components (SCCs) in $G$. Thus, the $C_{i}$’s are strongly connected components of $G$ and whenever $E \rightarrow_{G} D$ and $E \in C_{i}$, $D \in C_{j}$ then $i < j$. The fixed point equation for $p(E) = Pr_{E,\text{mec}}^{\max}(E, \Diamond F)$ where $E \in C_{i} \cap S'$ has the form
Chapter 4. Efficient probabilistic reachability analysis  4.3. Structured solving approaches

\[
p(E) = \max \left\{ \nu(S'_i) + \sum_{D \in S'_i} \nu(D) \cdot p(D) \mid E \to_{\text{MEC}} \nu \right\}.
\]

As \( E \to_{\text{MEC}} \nu \) and \( \nu(D) > 0 \) implies \( D \in C_i \cup \ldots \cup C_k \), we can split the fixed point equations into blocks that correspond to the SCCs of \( G \). The last SCCs in the topological order represent \( S'_0 \) and \( S'_1 \) and their maximal probability to reach \( F \) is 0 or 1. Assume now that for all \( D \in C_{i+1} \cup \ldots \cup C_k \), the values \( p(D) \) are known. The equation for the maximal end components \( E \in C_i \cap S'_i \) for \( 1 \leq i < k \) do not refer to the values for the maximal end components in \( C_1 \cup \ldots \cup C_{i-1} \) and the values \( p(D) \) for \( D \in C_{i+1} \cup \ldots \cup C_k \cup S'_0 \cup S'_1 \) can be treated as constants. Hence, we obtain an equation system for the values \( p(E) \), \( E \in S'_i \cap C_i \) that just depends on \( p(E') \) for \( E \in S'_i \cap C_i \). This equation system can be solved by value iteration.

Obviously, the decomposition technique can also be applied to \( \mathcal{M} \), rather than its MEC-quotient. The resulting approach is similar to the topological value iteration suggested in [DG07] to reason about discounted average rewards in MDPs.

Remark: We argue that the decomposition technique applied to \( \mathcal{M} \) can also be seen as a generalization of the concept of essential states proposed in [dJJL02]. Essential states \( s \) are maximal according to the least partial order \( \preceq \) such that \( t \preceq s \) if and only if \( s = t \) or \( t \) is non-terminal and \( u \preceq s \) for all successors \( u \) of \( t \). The states \( t \neq s \) with \( t \preceq s \) are said to be dominated by \( s \). [dJJL02] observe that \( p(t) = p(s) \) whenever \( t \preceq s \), and hence it suffices to compute \( p(s) \) for the essential states. In our approach, the topological order is consistent with \( \preceq \) and any essential state \( s \) is treated before the states dominated by \( s \). But then, with the decomposition technique the value \( p(s) \) is simply propagated to the states dominated by \( s \) (without any iteration). In this sense, the decomposition technique covers the effect of the essential state abstraction. However, the algorithm to compute the essential state abstraction presented [dJJL02] is rather different from our approach. While we work with a topological order, [dJJL02] deals with a union-find algorithm to compute the essential states.

Another observation of [dJJL02] is that if a state \( s \) has just one transition \( s \to \mu \) then Gauss
elimination can be applied to remove the equation for \( s \). Our decomposition technique in combination with the removal of self-loops has a similar effect.

### 4.4. Experimental results

Table 4.2 shows results on experiments concerning the usage of priorized updates in the value iteration process. For this, we used the test models already employed in the comparison of the value iteration approach and the simplex algorithm (see table 6.2, page 125). Note, that the tables report only on the solving part of the model checking process, not on the building part.

<table>
<thead>
<tr>
<th>model</th>
<th>states</th>
<th>transitions</th>
<th>ineq.</th>
<th>queue time</th>
<th>queue updates</th>
<th>priority queue time</th>
<th>priority queue updates</th>
<th>bucket based time</th>
<th>bucket based updates</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP(6)</td>
<td>3.3 \cdot 10^6</td>
<td>2.6 \cdot 10^7</td>
<td>655465</td>
<td>302 s</td>
<td>4.5 \cdot 10^7</td>
<td>351 s</td>
<td>3.0 \cdot 10^7</td>
<td>281 s</td>
<td>3.2 \cdot 10^7</td>
</tr>
<tr>
<td>DC(5)</td>
<td>1.64 \cdot 10^6</td>
<td>6.32 \cdot 10^6</td>
<td>394358</td>
<td>65s</td>
<td>8.2 \cdot 10^6</td>
<td>86s</td>
<td>7.3 \cdot 10^6</td>
<td>65s</td>
<td>7.4 \cdot 10^6</td>
</tr>
<tr>
<td>RG(4)</td>
<td>488902</td>
<td>661307</td>
<td>602796</td>
<td>68s</td>
<td>8.6 \cdot 10^6</td>
<td>75s</td>
<td>6.8 \cdot 10^6</td>
<td>59s</td>
<td>6.9 \cdot 10^6</td>
</tr>
<tr>
<td>LE(6)</td>
<td>1.1 \cdot 10^7</td>
<td>6.2 \cdot 10^7</td>
<td>971874</td>
<td>813s</td>
<td>1.1 \cdot 10^8</td>
<td>1067s</td>
<td>8.9 \cdot 10^7</td>
<td>751s</td>
<td>9.3 \cdot 10^7</td>
</tr>
<tr>
<td>UMTS</td>
<td>244233</td>
<td>253629</td>
<td>165393</td>
<td>60s</td>
<td>8.9 \cdot 10^6</td>
<td>47 s</td>
<td>4.1 \cdot 10^6</td>
<td>52s</td>
<td>6 \cdot 10^6</td>
</tr>
</tbody>
</table>

Table 4.2: Comparison of different priority-based update strategies

The results clearly indicate that a priorized handling of the updated values is unlikely to bring a disadvantage and leads to some extra speed-up, that is, however, not another order-of-magnitudes improvement. This indicates that a queue (without priorities) carries already an order of update values that has some quality. The priority queue is often considerable slower, although the number of updates is always reduced. The implementation of a priority queue in LiQuor relies on a standard STL implementation. Whether it can be optimized was not target of further investigation. The bucket approach needs more updates than the exact priority queue but is faster, because the implementation is very simple. The advantage of the priority approach is more moderate than in the theoretical motivation since the practical models used here do not exhibit extreme structural properties that are illustrated in figures 4.3, 4.5. An artificial example that contains such extreme structural properties was not constructed since the focus of the experimental results lies on models that are representative for a wide range of practical problems.

A necessary remark on value iteration concerns a memory-versus-speed tradeoff. Consider the following two lines of Procedure value iteration (see page 71):

80
pick some state \( t \in T \) and remove \( t \) from \( T \) for all transitions \( s \rightarrow \mu \) where \( \mu(t) > 0 \) do

Now recall the mechanisms from chapter 3 that were used to construct the MDP \( M \) from the PROBMela program. In section 3.2.2 the SOS rules for interleaving and synchronization uses transitions in the PCG (\( \leadsto \), premise of the rules) to construct the MDP (\( \rightarrow \), consequence of the rule). Since this is not reversible, i.e., the set of predecessor states \( \text{Pre}(s) \) of a state \( s \) in MDP \( M \) cannot be derived by these rules the model checker, needs to store the transitions during the construction phase of \( M \), which adds to the total memory requirement of the model checking procedure. For instance storing an MDP transition in LiQuor will cost at least between 16 bytes on a machine using 32-bit architecture.

<table>
<thead>
<tr>
<th>model</th>
<th>states</th>
<th>transitions</th>
<th>ineq.</th>
<th>bucket based time</th>
<th>bucket based memory</th>
<th>naïve value iteration time</th>
<th>naïve value iteration memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP(6)</td>
<td>( 3.3 \cdot 10^9 )</td>
<td>( 2.6 \cdot 10^7 )</td>
<td>655465</td>
<td>281 s</td>
<td>724 MB</td>
<td>699 s</td>
<td>326 MB</td>
</tr>
<tr>
<td>DC(5)</td>
<td>( 1.64 \cdot 10^7 )</td>
<td>( 6.32 \cdot 10^7 )</td>
<td>394358</td>
<td>65 s</td>
<td>2.8 GB</td>
<td>302 s</td>
<td>1.8 GB</td>
</tr>
<tr>
<td>RG(4)</td>
<td>488902</td>
<td>661307</td>
<td>602796</td>
<td>59 s</td>
<td>71 MB</td>
<td>689 s</td>
<td>58 MB</td>
</tr>
<tr>
<td>LE(6)</td>
<td>( 1.1 \cdot 10^7 )</td>
<td>( 6.2 \cdot 10^7 )</td>
<td>971874</td>
<td>751 s</td>
<td>2.2 GB</td>
<td>&gt;1 h</td>
<td>1.2 GB</td>
</tr>
<tr>
<td>UMTS</td>
<td>244233</td>
<td>253629</td>
<td>165393</td>
<td>47 s</td>
<td>35 MB</td>
<td>286 s</td>
<td>30 MB</td>
</tr>
</tbody>
</table>

Table 4.3: Comparison of memory requirements for backward-oriented value iteration

Table 4.3 illustrates the time-versus-memory tradeoff. In models that consist of comparably many transitions per state it might be necessary to dispense with the backward-oriented value iteration and use the naïve approach. A semi-automatic option that is implemented in LiQuor for this works as follows: If the user wants to use backward-oriented value iteration the transitions needed to perform the backward steps are stored. In case the machine runs out of memory the user has the option to reclaim most of the memory for continuing the run with the consequence that only a naïve value iteration can be performed at the end. This is a experimental feature since LiQuor uses its own memory management strategy where normally only freed blocks of the same size can be reclaimed for new purposes. Of course the user has always the option to start the model checking run without storing any transitions in the first place.

We now proceed to results concerning the MEC-quotient approach. Table 4.4 shows for all models considerable reductions of the number of inequations of the linear optimization problem when turning on the MEC-quotient construction. The potential of this reduction is also strongly related to the memory-speed tradeoff for the ability to "go backwards" in
the MDP (see above). In cases where the additional memory needed to store the backward edges for a backward-oriented value iteration is unavailable the MEC-quotient approach might still allow for a reasonable solving time. The results suggest the following recipe when analysing a new model. First, try to employ both the backward-oriented value iteration and the MEC-quotient approach. If the memory runs out, then abandon the backward edges (i.e., dispense with the hope of using backward-oriented value iteration) but try to apply the MEC-quotient approach anyway. Even with the naïve value iteration the solving time might still be tolerable.

Table 4.5 contains results for a comparison of different value iteration strategies in connection with the MEC-quotient approach with and without the blockwise extension. The results show that for some models this approach gains some extra speed.

### Table 4.5: Results of applying the MEC-quotient approach to the case studies

<table>
<thead>
<tr>
<th>Dining Cryptographers</th>
<th>MDP building</th>
<th>solving</th>
<th>solving (MEC-quotient)</th>
</tr>
</thead>
<tbody>
<tr>
<td>N states</td>
<td>transitions</td>
<td>time</td>
<td>ineq</td>
</tr>
<tr>
<td>3</td>
<td>9865</td>
<td>36544</td>
<td>0.9 s</td>
</tr>
<tr>
<td>4</td>
<td>95753</td>
<td>415687</td>
<td>4.9 s</td>
</tr>
<tr>
<td>5</td>
<td>1.64 - 10^6</td>
<td>6.32 - 10^6</td>
<td>66 s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Randomized Gossiping</th>
<th>MDP building</th>
<th>solving</th>
<th>solving (MEC-quotient)</th>
</tr>
</thead>
<tbody>
<tr>
<td>N states</td>
<td>transitions</td>
<td>time</td>
<td>ineq</td>
</tr>
<tr>
<td>3</td>
<td>4015</td>
<td>5298</td>
<td>0.9 s</td>
</tr>
<tr>
<td>4</td>
<td>488902</td>
<td>661307</td>
<td>4.4 s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Randomized Leader Election</th>
<th>MDP building</th>
<th>solving</th>
<th>solving (MEC-quotient)</th>
</tr>
</thead>
<tbody>
<tr>
<td>N states</td>
<td>transitions</td>
<td>time</td>
<td>ineq</td>
</tr>
<tr>
<td>4</td>
<td>53621</td>
<td>156072</td>
<td>1.1 s</td>
</tr>
<tr>
<td>5</td>
<td>896231</td>
<td>3.2 - 10^7</td>
<td>34 s</td>
</tr>
<tr>
<td>6</td>
<td>1.1 - 10^7</td>
<td>6.2 - 10^7</td>
<td>795 s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UMTS Sync Failure Problem</th>
<th>MDP building</th>
<th>solving</th>
<th>solving (MEC-quotient)</th>
</tr>
</thead>
<tbody>
<tr>
<td>model</td>
<td>states</td>
<td>transitions</td>
<td>time</td>
</tr>
<tr>
<td>UMTS</td>
<td>61696</td>
<td>64131</td>
<td>1.7 s</td>
</tr>
<tr>
<td>UMTS</td>
<td>244233</td>
<td>253629</td>
<td>3.8 s</td>
</tr>
</tbody>
</table>

Table 4.4: Results of applying the MEC-quotient approach to the case studies
### Value iteration strategies with MEC-quotient approach

| model | ineq. | queue time | updates $|10^6$ | priority queue VI time | updates $|10^6$ | bucket based time | updates $|10^6$ |
|--------|-------|------------|--------|-------------------------|--------|-------------------|--------|
| DC(5)  | 78023 | 21 s       | 3.0    | 25 s                    | 2.1    | 16 s              | 2.3    |
| RG(4)  | 37601 | 6.9 s      | 6.45   | 10 s                    | 8.72   | 4.1 s             | 5.01   |
| LE(6)  | 105896| 59 s       | 8.5    | 78 s                    | 6.8    | 48 s              | 7.3    |
| UMTS   | 65393 | 6.1 s      | 8.45   | 7.5 s                   | 6.42   | 4.8 s             | 7.85   |

### Value iteration strategies with MEC-quotient approach and blockwise solving

| model | ineq. | queue time | updates $|10^6$ | priority queue VI time | updates $|10^6$ | bucket based time | updates $|10^6$ |
|--------|-------|------------|--------|-------------------------|--------|-------------------|--------|
| DC(5)  | 78023 | 19 s       | 2.8    | 23 s                    | 2.01   | 15 s              | 1.75   |
| RG(4)  | 37601 | 7.2 s      | 5.98   | 6.9 s                   | 5.77   | 3.9 s             | 1.59   |
| LE(6)  | 105896| 55 s       | 8.1    | 69 s                    | 5.8    | 43 s              | 4.8    |
| UMTS   | 65393 | 5.6 s      | 8.36   | 7.3 s                   | 6.32   | 4.2 s             | 4.86   |

Table 4.5: The MEC-quotient approach with and without blockwise value iteration
Quantitative analysis under fairness constraints

When model checking systems that use nondeterministic behaviour to model features as underspecification of behaviour, liberty in process scheduling orders or free interaction with the environment (e.g., a human user), problems arise concerning extreme unrealistic behaviours that favour particular modules or processes of the system while disregarding some other. Without a notion of fairness there almost always exist such "unrealistic" schedulers that ignore certain processes which trivially can render even simple liveness properties to not being satisfied.

For instance consider two processes that compete for some shared resource and the liveness property

"It is always the case that every process gets access to the shared resource eventually".

The ordinary worst-case / best-case analysis (see section 2.2) ranges over the set of all schedulers, including the unfair instances that prefer a particular process in such a way that the property is not satisfied only because of this. Consider the MDP for the mutual exclusion example from chapter 3 as it is revisited in figure 5.1. Recall, that here a probabilistic arbiter manages the access to the shared resource. For this example, we re-equip the MDP with more figurative atomic propositions. Labels $\text{crit}_1$ and $\text{crit}_2$ indicate that processes

![Rabin automaton $A$](image)

![MDP $M \times \mathcal{A}$](image)

Figure 5.1: MDP of the mutex example revisited for fairness

85
5.1. Fairness notions for probabilistic systems

Chapter 5. Quantitative analysis under fairness constraints

Process 1 and Process 2, respectively, are in their critical sections, which means that in the corresponding PROBMela process (see page 53) the process execution is in line 4. Granting a process the permission to access the shared resource is done by sending a corresponding token over synchronous channel \(d_1\) and \(d_2\) in lines \(x\) and \(y\) of the arbiter specification, which is indicated by atomic propositions \(\text{grant}_1\) and \(\text{grant}_2\). The deterministic Rabin automaton \(A\) shown on the left of figure 5.1 encodes an LT-property \(E\) stating

"it is the case that Process 1 will never visit its critical section".

Clearly, \(\Pr^{\max}(M | E) = 1\), which can be seen, for example, by considering paths that are marked bold in figure 5.1. Intuitively, to achieve \(\Pr^{\mathcal{U}}(M | E) = 1\), scheduler \(\mathcal{U}\) chooses to stay in state \(\text{grant}_2\) by executing \(P_1\) infinitely. Thus, the system does not make any progress whatsoever. In fact, any such cycles of the states from

\[ \{\text{waiting}, \text{grant}_1, \text{grant}_2, \text{release}_1, \text{release}_2, \text{crit}_2\} \]

can be used this way to avoid progress of the system. This kind of behaviour is often considered as unrealistic or unfair and may be removed from the set of paths under consideration by the concept of fairness.

While in the context of functional LTL verification incorporating fairness is an easy task - fairness conditions are essentially LTL conditions - for CTL a more sophisticated approach based on an analysis of SCCs has to be applied [EL85, CES86]. Compared to models without fairness this leads to an increase of the complexity linear in the number of fairness conditions. For probabilistic systems the importance of fairness carries directly over. For the branching time case (PCTL) fairness algorithms were proposed in [BK98].

In this chapter, we describe an approach that covers the probabilistic analysis of LT-properties. Unfortunately, in the quantitative setting the combination of fairness constraints and LT-property is not as straightforward as in the functional case, e.g., with LTL. The fact that fairness constraints are essentially LTL conditions does not help and furthermore, the necessary concept of realizability forces the property and the fairness constraints to be treated separately in stage preliminary to the main analysis step [BGC09b].

5.1. Fairness notions for probabilistic systems

Given \(k\) strong fairness assumptions of the form \(\Box \Diamond H_i \Rightarrow \Box \Diamond K_i\) for \(1 \leq i \leq k\) there are \(2^k\) "ways" to satisfy the fairness condition (in addition to the fact that the number of end components in an MDP is exponential). To avoid the exponential blow-up, we propose a reduction to a probabilistic reachability problem that examines maximal end components rather than SCCs resp. end components. For this we propose algorithms that, given an
arbitrary number of strong and weak fairness constraints, increase the complexity of the main analysis in probabilistic model checking only by a factor linear in the number of fairness constraints (see section 5.3). We use a formalism of weak and strong fairness constraints \((I, J)\), \(I, J \subseteq 2^{\mathbb{AP}} \cup \text{Act}\) over infinite paths \(\pi \in \text{Paths}_{\text{inf}}\), more precisely over paths that may contain information about labelings and actions of an MDP. We call \(I\) a premise and \(J\) an implication of the constraint. Intuitively the constraints require for

- **weak fairness** that if continuously from some point in time (i.e., continuously beginning at some index \(i \in \mathbb{N}\) in the path \(\pi = s_0, \alpha_0, ..., s_i, \alpha_i, ... \in (S \times \text{Act})^\omega\)) only labelings and actions from \(I\) occur, then infinitely often labelings or actions from \(J\) will occur, too.

- **strong fairness** that if infinitely often an action or labeling of \(I\) occurs in the path, then also infinitely often a labeling or action from \(J\) occurs.

We now fix the notion for fairness conditions that hold multiple fairness constraints and give corresponding satisfaction relations for sets \(X \subseteq 2^{\mathbb{AP}} \cup \text{Act}\). At the end of these definitions, the satisfaction relation is extended to traces in order to link fairness constraints with paths in an MDP, which fixes central the notion of fair path in an MDP.

**Definition 5.1.1. [Fairness constraints, general fairness condition and satisfaction]** Given an MDP \(\mathcal{M} = (S, \text{Act}, \rightarrow, s_0, \mathbb{AP}, L)\), a single fairness constraint is a pair \((I, J)\) such that \(I, J \subseteq 2^{\mathbb{AP}} \cup \text{Act}\). Given a set \(X \subseteq 2^{\mathbb{AP}} \cup \text{Act}\), we distinguish different satisfaction relations between state-action sets and fairness constraints, namely weak and strong fairness.

- **Weak fairness**: \(X\) weakly satisfies a fairness constraint \((I, J)\), written \(X \vdash_{\text{weak}} (I, J)\), if and only if
  \[
  X \setminus I \neq \emptyset \text{ or } X \cap J \neq \emptyset.
  \]

- **Strong fairness**: \(X\) strongly satisfies a fairness constraint \((I, J)\), written \(X \vdash_{\text{strong}} (I, J)\), if and only if
  \[
  X \cap I = \emptyset \text{ or } X \cap J \neq \emptyset.
  \]

Strong and weak fairness conditions are sets of such fairness constraints

\[
\mathcal{SF} = \{(I_1, J_1), ..., (I_m, J_m)\},
\]

and

\[
\mathcal{WF} = \{(I_n, J_n), ..., (I_q, J_q)\},
\]

and a general fairness condition is a tuple of such strong and weak fairness constraints

\[
\mathcal{F} = (\mathcal{WF}, \mathcal{SF}).
\]
If $X \subseteq 2^{AP} \cup \text{Act}$ then we write $X \models^{\text{fair}} \mathcal{F}$ to denote that ”$X$ satisfies $\mathcal{F}$” in the following sense:

$X \models^{\text{weak}} (I, J)$ for all $(I, J) \in \mathcal{WF}$

and

$X \models^{\text{strong}} (I, J)$ for all $(I, J) \in \mathcal{SF}$.

It is useful to extend these relations directly to paths $\pi \in \text{Paths}_{\text{inf}}$ such that

$\pi \models^{\text{fair}} \mathcal{F} \overset{\text{def}}{=} X_\pi \models^{\text{fair}} \mathcal{F},$

where

$X_\pi \overset{\text{def}}{=} \{ L(t) : t \in \inf(\pi) \} \cup \{ \alpha \in \text{Act} : \alpha \text{ has been taken infinitely often in } \pi \}.$

For end components $\mathcal{E} = (T, \text{Act}_\mathcal{E})$, we write $\mathcal{E} \models^{\text{fair}} \mathcal{F}$ if and only if for all paths $\pi$ where $\lim(\pi) = \mathcal{E}$ it holds that $\pi \models^{\text{fair}} \mathcal{F}$. An equivalent set based characterization of $\mathcal{E} \models^{\text{fair}} \mathcal{F}$, that is more suitable to the actual implementation of algorithms, is given by the notation

$X_\mathcal{E} = \{ L(t) : t \in T \} \cup \bigcup_{t \in T} \text{Act}_\mathcal{E}(t)$

to obtain the union of an end component’s states and actions. It obviously holds that $\mathcal{E} \models^{\text{fair}} \mathcal{F}$ iff $X_\mathcal{E} \models^{\text{fair}} \mathcal{F}.$

Note, that fairness constraints operate on labelings and actions rather that on states and actions. For instance in the context of a high-level modelling language as PROBMela, fairness conditions argue about labelings such as ”the process is in a waiting state” or ”the process is the critical section” rather than on concrete states in the MDP. Therefore, paths are lifted to their traces and augmented with their taken actions.

When using the LTL-like notation $\Box \Diamond$ for ”always eventually”\footnote{we do not give formal LTL-semantics for paths here} it becomes evident, that $\models^{\text{strong}}$ indeed relates to a classic strong fairness constraint $\Box \Diamond a \Rightarrow \Box \Diamond b$, i.e., a path property. If $I, J \subseteq 2^{AP}$, then:

$\pi \models^{\text{strong}} (I, J) \overset{\text{def}}{=} X_\pi \models^{\text{strong}} (I, J)$

iff $X_\pi \cap I = \emptyset \lor X_\pi \cap J \neq \emptyset$

iff $\pi \models \Box \Diamond \neg I \lor \Box \Diamond J$

iff $\pi \models \Box \Diamond I \Rightarrow \Box \Diamond J.$

Similarly, for weak fairness we have

$\pi \models^{\text{weak}} (I, J) \overset{\text{def}}{=} X_\pi \models^{\text{weak}} (I, J)$

iff $X_\pi \setminus I \neq \emptyset \lor X_\pi \cap J \neq \emptyset$

iff $\pi \models \Box \Diamond \neg I \lor \Box \Diamond J$

iff $\pi \models \Box \Diamond I \Rightarrow \Box \Diamond J.$
5.2. Fair schedulers and realizability of fairness conditions

We now define the notion of realizability of fairness conditions and fair schedulers, as it can be found in the literature. Intuitively, realizability requires that in every state of the MDP, there is a way to continue a path in a fair manner. In subsection 5.4, we motivate an alternative to this standard definition that is less strict, and discuss some interconnections between them.

**Definition 5.2.1. [Realizability of fairness conditions and fair schedulers]**

Given an MDP \( M = (S, \text{Act}, \rightarrow, s_0, \text{AP}, L) \) and a fairness condition \( F \), a scheduler \( U \) is called *fair*, if for all states \( s \in S \) it holds that \( \Pr^M_{s_0}([\pi \in \text{Paths} : \pi \vdash \text{fair } F]) = 1 \). A fairness condition \( F \) is called *realizable* if there exists a fair scheduler, i.e.,

\[
\Pr^\text{fairmax}_{s_0}([\pi \in \text{Paths} : \pi \vdash \text{fair } F]) = 1
\]

for all states \( s \in S \). The set of fair schedulers (with respect to a fairness condition \( F \)) is written \( \text{FairSched}_{F} \), or simply \( \text{FairSched} \). If a fairness condition is not realizable, it is called *unrealizable*.

If \( E \) is an LT-property representing the desired behaviour, then the worst-case probability for \( E \) assuming fair resolution of the nondeterminism according to \( F \) is:

\[
\Pr^\text{fairmax}(M|E) \overset{\text{def}}{=} \sup_{U \in \text{FairSched}_{F}} \Pr^M_{s_0}([\pi \in \text{Paths} : \pi \vdash E]).
\]

We will later see (section 5.3, Theorem 5.3.1) that the task of calculating the maximum probability under all fair schedulers \( \Pr^\text{fairmax}(M|E) \) to satisfy an \( \omega \)-regular property \( E \) can be calculated by means of methods already described for \( \Pr^\text{max} \), provided that \( F \) is realizable. Prior to this we will explain how to check for the realizability of a fairness constraint \( F \) algorithmically.

Intuitively, it is clear that deciding the existence of a fair scheduler (i.e., deciding whether a given fairness condition is realizable in a system) relies on checking graph-based criteria, namely searching for end components where the strong and weak fairness constraints of \( F \) hold. Again, the relevant end components might very well be non-maximal and the naive approach to consider all end components has exponential time complexity. Recall that a state can belong to many end components and the total number of end components can be exponentially in the number of states of \( M \). However, appropriate argumentations about maximal end components and their sub-end components (see Lemma 2.2.11) avoid the bad complexity.

**Theorem 5.2.2.** Given an MDP \( M = (S, \text{Act}, \rightarrow, s_0, \text{AP}, L) \) and a general fairness condition \( F = (W, F, S, F) \), let
FMEC = \{ t \mid t \in T \text{ for some maximal end component } E = (T, \text{Act}_E) \\
that contains a fair sub-end component, \\
i.e., a sub-end component } E' \text{ with } E' \vdash \text{fair } F \}. \\

Then, there exists a finite-memory scheduler } \mathcal{W} \text{ such that } \\
\Pr_{\mathcal{W}}^t(\{ \pi \in \text{Paths} : \pi \vdash \text{fair } F \}) = 1 \\
for all states } t \in \text{FMEC}.

**Proof.** Each state in } t \in \text{FMEC} \text{ belongs to a maximal end component } E. \text{ Let } E' \text{ be a fair sub-end component of } E. \text{ According to Lemma 2.2.11 there exists a finite memory scheduler } \mathcal{W} \text{ such that } E' \text{ is reached with probability } 1, \text{ and } \Pr_{\mathcal{W}}^t(\{ \pi \in \text{Paths} | \lim(\pi) = E' \}) = 1, \text{ and consequently } \Pr_{\mathcal{W}}^t(\{ \pi \in \text{Paths} : \pi \vdash \text{fair } F \}) = 1. \\

Similar to the case when checking the satisfaction of } \omega \text{-regular properties, Theorem 5.2.2 suggests that for checking realizability of a given fairness constraint } F \text{ the probability } \Pr_s(\{ \pi \in \text{Paths} | \pi \vdash \text{fair } F \}) \text{ for a state } s \in S \text{ can be calculated through finding } \Pr_{\max}^{s}(\Diamond \text{FMEC}). \text{ However, since realizability is a property that must hold for all states in } s \in S, \text{ the approach of calculating this value for each state requires solving a linear program. The following theorem leads to a simpler method: }

**Theorem 5.2.3.** Let } F \text{ be a given general fairness condition and } \mathcal{M} = (S, \text{Act}, \rightarrow, s_0, \text{AP}, L) \text{ an MDP. Then, } F \text{ is realizable if and only if } \text{FMEC} \text{ is reachable from all states } s \in S, \text{i.e., if - using an intuitive CTL-like notation - we have that } \{ s \mid s \models \exists \Diamond \text{FMEC} \} = S.

**Proof.** “⇒”: Suppose } F \text{ is realizable. Let } \mathcal{U} \text{ be a fair scheduler and } s \in S \text{ a state. Theorem 2.2.8 yields the existence of some end component } E = (T, \text{Act}_E) \text{ such that } \Pr_{\mathcal{U}}^t(\{ \pi : \lim(\pi) = E \}) > 0. \text{ Since } \mathcal{U} \text{ is fair, this end component } E \text{ must be fair. But then } T \subseteq \text{FMEC} \text{ (as each end component is contained in some maximal end component). Thus, } \Pr_{\mathcal{U}}^t(\Diamond \text{FMEC}) > 0 \text{ and therefore } \text{FMEC} \text{ must be reachable from } s.

“⇐”: Suppose } \text{FMEC} \text{ is reachable from all states } s \in S. \text{ For each state } s \text{ we pick a shortest path } \pi_s = s, \alpha, s_1, \alpha_1, s_2, ..., s_n \text{ from } s \text{ to some state in } s_n \in \text{FMEC}. \text{ We now construct a memoryless scheduler } \mathcal{U}, \text{ that takes in each state } s \in S \text{ the first action } \alpha \text{ of } \pi_s \text{ The resulting finite Markov chain } \mathcal{M}_{\mathcal{U}} \text{ therefore does not contain any state, from which } \text{FMEC} \text{ is not reachable, and it follows from standard arguments from Markov chain theory, that } \Pr_{\mathcal{U}}^t(\Diamond \text{FMEC}) = 1 \text{ for each state } s \in S_{\mathcal{M}_{\mathcal{U}}}, \text{ and therefore, } \Pr_{\max}^{s}(\Diamond \text{FMEC}). \text{ Thus, } F \text{ is realizable.}

Thus, the problem of checking realizability can be solved by the following algorithm:
Chapter 5. Quantitative analysis under fairness constraints

5.2. Fair schedulers and realizability of fairness conditions

**Procedure** check_realizability.

<table>
<thead>
<tr>
<th>Purpose</th>
<th>check whether $F$ is realizable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>general fairness condition $F$</td>
</tr>
<tr>
<td>Output</td>
<td>Boolean value</td>
</tr>
</tbody>
</table>

let $FMEC := \text{calc}_FMEC(F)$

if all maximal end components belong to $FMEC$, then return true  (* $F$ is realizable *)

calculate set $S' := \{s | \Pr^\max_{s}(\Diamond FMEC) = 0\}$  (* see page 32, section 2.3 *)

if $S' = \emptyset$ then return true

return false  (* there are states for which $FMEC$ is unreachable *)

The details of calculating $FMEC$, the first step in procedure check_realizability, are as follows:

**Procedure** calc_FMEC.

<table>
<thead>
<tr>
<th>Purpose</th>
<th>calculate set $FMEC$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>general fairness condition $F$</td>
</tr>
<tr>
<td>Output</td>
<td>set $FMEC$</td>
</tr>
</tbody>
</table>

let $FMEC := \emptyset$

calculate set $MEC$  (* see reference below *)

for all $E \in MEC$ do

if $E$ contains a fair sub-end component then $FMEC := FMEC \cup E$  (* see procedure below *)

end for

return $FMEC$

The maximal end components themselves, the set $MEC$, can be computed by applying the iterative algorithm presented in [CY95, JA97], which is based on an algorithm that calculates strongly connected components (SCCs), e.g., Tarjan’s SCC algorithm [Tar72] that fits nicely into the DFS setting. Checking whether an end component $E$ contains a fair sub-end component can be realized by a recursive algorithm that successively tests all fairness constraints in $F$. We denote by $E \ominus I$ the sub-MDP $(U, B)$ that arises from an end component $E = (T, \text{Act}E)$, where $U = \{t \in T : L(t) \notin H\}$ and $B : U \rightarrow 2^{\text{Act}}$, where

$$B(t) = \{\alpha \in \text{Act}(t) : \sum_{u \in U} \Pr(t, \alpha, u) = 1\} \text{ for all } t \in U.$$  

Intuitively, $E \ominus I$ results from $E$ by removing all elements that contribute to $E$ satisfying $H$. The resulting sub-MDP then becomes a "candidate" for an end component. The algorithm can then be formulated as follows, where we assume a fairness condition $F = (WF, SF)$, with $WF = \{(I_1, J_1), \ldots, (I_m, J_m)\}$ and $SF = \{(I_1, J_1), \ldots, (I_m, J_m)\}$:
**5.2. Fair schedulers and realizability of fairness conditions**

Chapter 5. Quantitative analysis under fairness constraints

---

**Procedure** `check_fair(\mathcal{E}, \mathcal{F})`.

**Purpose**: Check the existence of a fair sub-end component in end component \(\mathcal{E}\) under fairness condition \(\mathcal{F}\).

**Input**: end component \(\mathcal{E}\), general fairness condition \(\mathcal{F}\).

**Output**: Boolean value.

\[
\text{if } \mathcal{E} \not\models_{\text{weak}} (I,J) \text{ for some } (I,J) \in \mathcal{WF} \text{ then}
\]

\[
\quad \text{return false} \quad (* \mathcal{E} \text{ does not contain a fair sub-end component. *})
\]

\[
\text{else}
\]

\[
\quad \text{if } \mathcal{E} \models_{\text{strong}} (I,J) \text{ for all } (I,J) \in \mathcal{SF} \text{ then}
\]

\[
\quad \quad \text{return true}
\]

\[
\text{else}
\]

\[
\quad \text{pick } (I,J) \in \mathcal{SF} \text{ such that } \mathcal{E} \not\models_{\text{strong}} (I,J)
\]

\[
\quad \text{let } \mathcal{E}_{\text{cand}} := \mathcal{E} \ominus I
\]

\[
\quad \text{remove } (I,J) \text{ from } \mathcal{SF} \text{ to form fairness condition } \mathcal{F}'
\]

\[
\quad \text{for all maximal end component } \mathcal{E}' \text{ in } \mathcal{E}_{\text{cand}} \text{ do}
\]

\[
\quad \quad \text{call algorithm check_fair with input } \mathcal{E}' \text{ and fairness condition } \mathcal{F}'
\]

\[
\quad \quad \text{if check_fair returned true then}
\]

\[
\quad \quad \quad \text{return true} \quad (* \mathcal{E} \text{ contains a fair sub-end component *})
\]

\[
\quad \quad \text{else}
\]

\[
\quad \quad \quad \text{return false} \quad (* \mathcal{E} \text{ contains no fair sub-end components *})
\]

\[
\quad \text{end if}
\]

\[
\text{end for}
\]

\[
\text{end if}
\]

---

This algorithm works similar to approaches that were presented in the context of (non-probabilistic) fair CTL model checking (see for instance [EL86, BK98]), where fair SCCs are the target of investigation rather than end components. When the structure under investigation (SCC or maximal end component) is not satisfying the current strong fairness constraint, states and actions that match the constraint’s premise are removed before the remaining sub-structures are processed recursively with the remaining constraints. The correctness of algorithm `check_fair` can be seen as follows:

**Lemma 5.2.4** (Soundness of Algorithm `check_fair`). If the procedure call

\[
\text{check_fair}(\mathcal{E}, \mathcal{F} = (WF, SF))
\]

returns “true” then \(\mathcal{E}\) contains a fair sub-end component. If it returns “false” then \(\mathcal{E}\) has no fair sub-end component.

**Proof.** We provide a proof by induction on the number \(f\) of strong fairness constraints in \(\mathcal{F} = (WF, SF)\).
If \( f = 0 \) then the algorithm correctly returns the answer “true” in the then-branch of the second if-statement.

For \( f \geq 1 \), the soundness crucially relies on the following facts:

- If \( \mathcal{E} \not\vdash \text{weak} (I,J) \) for some weak fairness constraint \((I,J) \in \mathcal{WF} \) then \( X_{\mathcal{E}} \subseteq H \) and \( X_{\mathcal{E}} \cap J = \emptyset \). It is clear that the same holds for each sub-end component of \( \mathcal{E} \). Thus, \( \mathcal{E} \) cannot contain a fair sub-end component and the answer “false” returned in the first if-statement is correct.

- If \( \mathcal{E} \vdash \text{weak} (I,J) \) for all \((I,J) \in \mathcal{WF} \) and \( \mathcal{E} \vdash \text{strong} (I,J) \) for all \((I,J) \in \mathcal{SF} \) then \( \mathcal{E} \) is fair and the algorithm correctly halts with the answer “true” in the then-branch of the second if-statement.

- In case that the algorithm must pick a strong fairness constraint \((I,J) \) with \( \mathcal{E} \not\vdash \text{strong} (I,J) \), we have \( X_{\mathcal{E}} \cap I \neq \emptyset \) and \( X_{\mathcal{E}} \cap J = \emptyset \). Since \( X_{\mathcal{E}} \cap J = \emptyset \), the only chance to satisfy \((I,J)\) is to remove \( H \) from \( \mathcal{E} \), i.e., form \( \mathcal{E} \oplus I \), and check for sub-end components. Note that we have then \((U,B) \vdash \text{strong} (I,J)\) for any sub-end component \((U,B)\) of the subgraph \( \mathcal{E} \oplus I \). On the other hand, any sub-end component \((U,B)\) of \( \mathcal{E} \) such that \((U,B) \vdash \text{strong} (I,J)\) is also contained in \( \mathcal{E} \oplus I \). Hence, if there exists a sub-end component \((U,B)\) in \( \mathcal{E} \oplus I \) such that \((U,B)\) meets all other strong fairness constraints in \( \mathcal{SF} \setminus \{(I,J)\} \) and all weak fairness constraints in \( \mathcal{WF} \), then we have also that there is a fair end component in \( \mathcal{E} \).

Figure 5.2 illustrates the argumentation in the correctness proof for algorithm check_fair by providing a graphical view on its recursive calls when called with a maximal end component \( \mathcal{E} \) and a fairness condition \( \mathcal{F} \). (The diagram only contains information for the strong-fairness case). Note, that only constraints that are not satisfied by the current end component are removed from \( \mathcal{F} \). Once a fairness constraint has been removed by the algorithm, we have that all new maximal end components \( \mathcal{E}_1', \mathcal{E}_2', \ldots, \mathcal{E}_n' \) that may evolve from the candidate \( \mathcal{E}_{\text{cand}} = \mathcal{E} \oplus I \) also satisfy the removed constraint \((I,J)\) since \( X_{\mathcal{E}} \cap I = \emptyset \), which trivially also holds for any sub-component of \( \mathcal{E} \). Note also, that satisfying the implication \( J \) is the only chance for \( \mathcal{E} \) to satisfy \((I,J)\) since there are no \( I \)-states in \( \mathcal{E} \).

**Lemma 5.2.5** (Complexity of algorithm check_fair). Checking realizability of a fairness condition \( \mathcal{F} = (\mathcal{WF}, \mathcal{SF}) \) in an MDP \( \mathcal{M} \) is solvable by an algorithm that runs in time quadratic in the size of \( \mathcal{M} \) and linear in the number of fairness constraints.

**Proof.** The recursion depth of Algorithm check_fair is bounded by the total number of strong fairness constraints. For each recursive call check_fair(\( \mathcal{E}, \mathcal{F}' \)) inside check_fair(\( \mathcal{E}, \mathcal{F} \)),
\[ E = (V \hookrightarrow \text{Act}_E) \]

\[ X_E \cap I \subseteq \text{strong} (I \hookrightarrow J) \iff \neg (X_E \cap I = \emptyset \lor X_E \cap J \neq \emptyset) \]

\[ \iff X_E \cap I \neq \emptyset \land X_E \cap J = \emptyset \]

\[ E' = \text{check fair}(E', F(I, J)) \]

\[ \mathcal{E'} = \text{end component of the subgraph } \mathcal{E} \hookrightarrow I \text{ of } \mathcal{E} \text{ where } (I, J) \text{ is a strong fairness constraint such that } \mathcal{E} \not\leftarrow \text{strong} (I, J) \text{. But then } X_\mathcal{E} \cap I \neq \emptyset \text{. Let the size of a subgraph } (U, B) \text{ be defined by } \text{size}(U, B) = |U| + |\{ \alpha \in \text{Act} : \alpha \in B(u) \text{ for some } u \in U \}| \text{. Then, for } \mathcal{E} = (T, \text{Act}_E) \text{ and } \mathcal{E}' = (T', \text{Act}_{E'}) \text{, we have } \text{size}(\mathcal{E}, \text{Act}_E) < \text{size}(\mathcal{E}', \text{Act}_{E'}) \text{. With } n \text{ being the total size of an appropriate list representation for } M \text{, an upper bound for the asymptotic cost for checking } \mathcal{E} \not\leftarrow \text{strong} (I, J) \text{ for all } (I, J) \in \mathcal{S F} \text{ can be provided by the recurrence} \]

\[ T(n, f) = O(n^2) + \max \{ T(n_1, f-1) + \ldots + T(n_k, f-1) : (n_1, \ldots, n_k) \in \mathcal{N}(n) \} \]

where \( \mathcal{N}(n) \) consists of all tuples \( (n_1, \ldots, n_k) \in \mathbb{N}^k \) where \( k \geq 1, n_1, \ldots, n_k \geq 1 \) and \( n_1 + \ldots + n_k < n \). The summand \( O(n^2) \) stands for the time required to compute the maximal end components of \( M \). We also assume that \( |\mathcal{S F}| + |\mathcal{W F}| \leq n \). Then, the time required to check whether some weak or strong fairness constraint is violated is also covered by \( O(n^2) \). The terms \( T(n_i, f-1) \) stand for the cost that are caused by the recursive calls for the maximal end components of the subgraph \( \mathcal{E} \hookrightarrow I \). If \( \mathcal{E} \hookrightarrow I \) has \( k \) maximal end components and their sizes are \( n_1, \ldots, n_k \) then \( n_1 + \ldots + n_k \) is bounded by the size of \( \mathcal{E} \hookrightarrow I \) which is at most \( n-1 \). The solution of this recurrence is \( O(n^2 \cdot f) \). \[ \square \]
5.3. Fair satisfaction of $\omega$-regular LT-properties

As mentioned above, the maximal probabilities for an MDP $\mathcal{M}$ satisfying an $\omega$-regular LT-property $E$ (see section 2.1.8) as defined by

$$\Pr_{\text{fairmax}}(\mathcal{M} \models E) \triangleq \sup_{U \in \text{FairSched}_{\mathcal{F}}} \Pr_{\mathcal{M}U}(\{\pi \in \text{Paths} : \pi \models E\})$$

can be calculated similar to the case without fairness, provided that the fairness condition is realizable.

**Theorem 5.3.1** (Calculating fair satisfaction of LT-properties). Given a labeled MDP $\mathcal{M} = (S, \text{Act}, \rightarrow, s_0, \text{AP}, L)$, a general fairness condition $\mathcal{F} = (\mathcal{WF}, \mathcal{SF})$ and an LT-property $E$, the task of calculating the maximum probability of $\mathcal{M}$ satisfying $E$ with respect to $\mathcal{F}$ can be reduced to the unfair case, i.e.,

$$\Pr_{\text{fairmax}}(\mathcal{M} \models E) = \Pr_{\text{max}}(\mathcal{M} \models E \land \mathcal{F})$$

where for this equation we treat $\mathcal{F}$ as a collection of path properties (see page 88).

The standard automata-based approach aims to compute $\Pr_{\text{max}}(\mathcal{M} \models E)$ by representing $E$ by a deterministic $\omega$-automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, \text{Acc})$. The product $\mathcal{M} \times \mathcal{A}$ can then (also) be viewed as an MDP and the task is to compute the maximal probability for the (Rabin- or Streett-) acceptance condition $\text{Acc}$ of $\mathcal{A}$ in the product (see section 2.2.4). In this section we present algorithms for computing

$$\Pr_{\text{fairmax}}(\mathcal{M} \models \text{Acc}) \triangleq \sup_{U \in \text{FairSched}_{\mathcal{F}}} \Pr_{\mathcal{M}U}(\{\pi \in \text{Paths} : \pi \models \text{Acc}\})$$

Due to the duality of Streett and Rabin acceptance, the techniques presented here are at the same time applicable to reason about minimal probabilities since:

$$\Pr_{\text{fairmin}}(s \models \text{StreettAcc}) \triangleq \inf_{U \in \text{FairSched}_{\mathcal{F}}} \Pr_{\mathcal{M}U}(\{\pi \in \text{Paths}_{\text{inf}}^{\mathcal{M}} : \pi \models \text{StreettAcc}\}) = 1 - \Pr_{\text{fairmax}}(s \models \text{StreettAcc})$$

and vice versa, $\Pr_{\text{fairmin}}(s \models \text{RabinAcc}) = 1 - \Pr_{\text{fairmax}}(s \models \text{StreettAcc})$. Since both deterministic Streett and Rabin automata have the full power of $\omega$-regular languages, it would have even been sufficient to consider just one type of acceptance condition. However, the experimental results in [KB07] show that for many relevant properties the freedom to choose between deterministic Streett and Rabin acceptance can be essential for efficiency reasons.
Definition 5.3.2. [Fair accepting end components] Given an MDP \( M = (S, \text{Act}, \rightarrow, s_0, \text{AP}, L) \), a general fairness condition \( F = (WF, SF) \) and a deterministic \( \omega \)-automaton \( A = (Q, \Sigma, \delta, Q_0, \text{Acc}) \) we call an end component \( E = (V, \text{Act}_E) \) of \( M \otimes A \) fair accepting if and only if both of the following conditions are satisfied:

1. \( V \vert \mathcal{Q} \) is accepting, and
2. \( V \vert \mathcal{S} \cup \bigcup_{t \in T} \text{Act}_E(t) \models \text{fair } F \),

where \( V \vert \mathcal{Q} = \{ q \in Q \mid \exists s \in S : (s, q) \in V \} \) and \( V \vert \mathcal{S} = \{ s \in S : (s, q) \in V \} \).

In the spirit of the notations \( \models \text{Rabin} \), \( \models \text{Streett} \), and \( \models \text{fair} \) we write \( E \models \text{Streett}_{\text{fair}}(F, F) \) and \( E \models \text{Rabin}_{\text{fair}}(F, F) \) if an end component \( E \) is fair accepting with respect to a fairness condition \( F = (WF, SF) \) and an \( \omega \)-automaton \( A = (Q, \Sigma, \delta, Q_0, \text{Acc}) \).

Theorem 5.3.3 (Reaching maximal end components). Given an MDP \( M \) and a deterministic \( \omega \)-automaton \( A \), let

\[
\text{FAMEC} = \{ \langle s, q \rangle \mid \exists \text{ maximal end component } E = (V, \text{Act}_E) \text{ in } M \otimes A : \\
\langle s, q \rangle \in V \text{ and } E \text{ contains a fair accepting sub-end component. } \}
\]

be the set of states that are contained in the set of maximal end components that contain a fair and accepting end component.

Then if \( E = \mathcal{L}(A) \):

\[
\Pr_{s}^{\text{fairmax}}(M \models E) = \Pr_{\langle s, A(q_0, L(s)) \rangle}^{\text{max}}(\Diamond \text{FAMEC}). \tag{5.1}
\]

Computing the set \( \text{FAMEC} \). Thanks to Theorem 5.3.3 the maximal probability for an \( \omega \)-regular property \( E \) under all fair schedulers can be obtained by linear programming techniques for maximal reachability probabilities that range over all schedulers in the product of \( M \) and a deterministic \( \omega \)-automaton \( A \) for \( E \). We now address the question of computing the target set \( \text{FAMEC} \) of the reachability objective.

\[ \Rightarrow \]
Chapter 5. Quantitative analysis under fairness constraints 5.3. Fair satisfaction of \( \omega \)-regular LT-properties

Procedure \texttt{calc\_FAMEC}.

<table>
<thead>
<tr>
<th>Purpose</th>
<th>calculate the set FAMEC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>MDP ( \mathcal{M} ), deterministic ( \omega )-automaton ( \mathcal{A} = (Q, \Sigma, \delta, Q_0, \text{Acc}) ) and fairness condition ( \mathcal{F} = (WF, SF) )</td>
</tr>
<tr>
<td>Output</td>
<td>set FAMEC</td>
</tr>
<tr>
<td>Local variable</td>
<td>Boolean flag result</td>
</tr>
</tbody>
</table>

Let FAMEC := \( \emptyset \)

For all end components \( \mathcal{E} = (T, \text{Act}_\mathcal{E}) \) in \( \mathcal{M} \otimes \mathcal{A} \) do

let result := false

if \( \mathcal{A} \) is a Rabin automaton then call result := check\_fair\_rabin with input \( \mathcal{E}, F \) and \( \mathcal{F} \)

if \( \mathcal{A} \) is a Streett automaton then call result := check\_fair\_streett with input \( \mathcal{E}, F \) and \( \mathcal{F} \)

if result = true then let FAMEC := FAMEC \( \cup \) \( T \)

end for

We present algorithms check\_fair\_rabin and check\_fair\_streett for both cases where the acceptance condition Acc of \( \mathcal{A} \) is a Rabin- or a Streett-condition. For this (see section 2.2.2) recall that Acc = \( \{ (L_1, U_1), \ldots, (L_n, U_n) \} \), where \( L_i, U_i \subseteq Q \) for all \( 1 \leq i \leq n \). The following procedures utilize procedure check\_fair to calculate Rabin- or Streett-acceptance.

Procedure check\_fair\_rabin.

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Check end component ( \mathcal{E} = (T, \text{Act}_\mathcal{E}) ) for Rabin acceptance under fairness constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>end component ( \mathcal{E} ), Rabin acceptance condition ( F = { (L_1, U_1), \ldots, (L_n, U_n) } ), and general fairness condition ( \mathcal{F} = (WF, SF) )</td>
</tr>
<tr>
<td>Output</td>
<td>Boolean value indicating whether ( \mathcal{E} ) is Rabin_fair ( (F, F) )</td>
</tr>
</tbody>
</table>

For all \( i = 1, \ldots, n \) do

remove all states in \( U_i \cap Q \) and all actions in \( \{ \alpha \mid \alpha \) leads to a state in \( U_i \cap Q \} \) from \( \mathcal{E} \) to obtain a candidate \( \mathcal{E}' \)

add \( (\emptyset, L_i) \) as a new constraint to \( \mathcal{WF} \) to obtain fairness condition \( \mathcal{F}' \)

call check\_fair with input \( \mathcal{E}' \) and \( \mathcal{F}' \)

if check\_fair (see page 92) then

return true

end if

end for

return false
5.4. Checking realizability on-the-fly

In the previous sections checking for realizability of a fairness condition $F$ and calculating $Pr_{f_{max}}^{fair}(M \models E)$ were treated as two independent steps. This separation comes from the fact that realizability and $E$ are essentially different properties. Realizability is a property that imposes a condition for each state of $M$ while $Pr_{f_{max}}^{fair}(M \models E)$ is a quantitative result that relates to only one state (the starting state $s_0$). A disadvantage with two separate analysis steps is that certain stages of the model checking process, especially calculating the set $PROB0MAX$, is needed to be carried out twice, namely for realizability check (with target set $FMEC$) and during a precalculation step for solving the linear program (with target set $FAMEC$). In order to avoid this, we present a variant of algorithm check_fair that checks for the realizability of $F$ on-the-fly while searching fair end components.

**Definition 5.4.1. [Bottom maximal end components]** Given an MDP $M = (S, Act, \rightarrow, s_0, AP, L)$, we call a maximal end component $E = (T, Act_E)$ **bottom**, if

$$\forall s \in T : Act_E(s) = Act(s).$$

Intuitively, an end component is bottom, if there is no action in $M$ that allows for leaving $E$. We then have the following criterion for realizability, that is based only on local conditions of maximal end components:

**Lemma 5.4.2 (Realizability).**

$F$ is realizable $\iff$ for all bottom maximum end components $E = (T, Act_E)$:

$E$ contains a fair sub-end component.
Proof.

⇒: There exists a scheduler $U$ such that $\Pr_s^{M_{U}}(\{\pi : \inf(\pi)\models_{\text{fair}} F\}) = 1$ for all states $s \in S$, especially for all states $s \in T$, where $E = (T, \text{Act}_E)$ is an arbitrary bottom maximal end component. Since there is no way to leave $E$ from $T$, there must be a fair sub-end component in $E$, otherwise the fairness condition would not be realizable.

⇐: Suppose that each bottom maximum end component $E$ contains a fair sub-end component. Lemma 2.2.11 ensures the existence of a scheduler that yields $\Pr_t^{\text{max}}\{\pi \in \text{Paths} | \inf(\pi)\models_{\text{fair}} F\} = 1$, for each $t \in T$. Hence, $E$ is fair.

For each state $s \in S$ any scheduler $U$ eventually reaches a maximum end component $E$ (Theorem 2.2.8). If $E$ is not bottom, we modify $U$ such that we leave $E$ to reach a different maximum end component $E'$. We repeat this until a bottom end component is reached. In this way, we obtain a scheduler $U$ such that almost all $U$-paths eventually enter a bottom MEC. Furthermore we can assume that the limit of all $U$-paths contains a fair sub-component of a bottom MEC. Hence, $U$ is fair and $F$ is realizable.

An on-the-fly reachability check can simply be performed by checking whether there exist bottom maximal end components $E$ with $E \not\models_{\text{fair}} F$. If (and only if) at least one such bottom maximal end component is found, the fairness condition is unrealizable.

In order to handle also the case when mixing fairness constraints with Streett-acceptance pairs that belong to an automaton $A$ in $S\mathcal{F}$ (as it is done in procedure check\_fair\_streett), procedure check\_fair must be slightly modified such that these two groups of fairness constraints are indeed processed in the order “first fairness, then the automaton”. For this we assume that $S\mathcal{F} = \{(I_1, J_1), \ldots, (I_n, J_n)\}$ is indexed such that constraints $(I_1, J_1), \ldots, (I_p, J_p)$ are original fairness constraints and constraints $(I_{p+1}, J_{p+1}), \ldots, (I_n, J_n)$ correspond to acceptance conditions of $A$. In addition to that we assume for each bottom maximal end component a global flag $\text{fair} \in \{\text{true}, \text{false}\}$ that is initially set to false. Then algorithm check\_fair can be modified to set this flag as follows:

⇒
### 5.5. Handling of unrealizable fairness conditions

The common characterization of a fair scheduler, namely that a scheduler \( \mathcal{U} \) is fair if it can produce a fair path starting in every state in \( s \in S \) with probability 1 (see definition 5.2.1 and also [BK98, BGC09b]), appears very strict compared to the setting of model checking under fairness constraints for non-probabilistic model checking against LTL or CTL. In context of LTL a fairness condition can always be encoded as an LTL property in the spirit of definition 5.1.1 provided, of course, that the corresponding atomic propositions and actions are made available in the model. A precondition that is comparable to the concept of realizability is

#### 5.5. Handling of unrealizable fairness conditions

As indicated above, if during the analysis with check\textsubscript{\text{fair}} the fair-flag remains false for a maximum end component \( E \) and \( E \) is bottom, then it immediately follows that \( F \) is not realizable.

---

**Procedure check\textsubscript{\text{fair}} (modification for on-the-fly realizability check).**

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Detect a fair sub-end component in end component ( E ) under fairness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>end component ( E ), general fairness condition ( F )</td>
</tr>
<tr>
<td>Output</td>
<td>Boolean value</td>
</tr>
<tr>
<td>Local variable</td>
<td>Boolean flag fair with initial value false</td>
</tr>
</tbody>
</table>

```
if \( E \not\models_{\text{weak}}(I, J) \) for some \((I, J) \in \mathcal{WF}\) then
    return false (* \( E \) does not contain a fair sub-end component *)
else
    if \( E \models_{\text{strong}}(I, J) \) for all \((I, J) \in \mathcal{SF}\) then
        return true
    else
        pick \((I_i, J_i) \in \mathcal{SF}\) with the lowest possible index \(i \leq n\) such that \( E \not\models_{\text{strong}}(I_i, J_i) \)
        let \( E_{\text{cand}} := E \circ I_i \)
        remove \((I_i, J_i) \) from \( \mathcal{SF} \) to form fairness condition \( F' \)
        if \( i > p \) and \( E_{\text{cand}} \) contains at least one end component then set flag fair to true
        for all maximal end component \( E' \) in \( E_{\text{cand}} \) do
            call algorithm check\textsubscript{\text{fair}} with input \( E' \) and fairness condition \( F' \)
            if check\textsubscript{\text{fair}} returned true then
                return true (* \( E \) contains a fair sub-end component *)
            else
                return false (* \( E \) contains no fair sub-end components *)
        end if
    end if
end if
```

---

100
not required, since states from which no fair path origins, do not have influence on the result of the model checking process. In CTL, the situation is similar \cite{EL85}.

The following alternative definition of fair schedulers, which is not compatible with definition 5.2.1 and the presented approaches for calculating $\Pr^{\text{fairmax}}$, agrees with this spirit and shall serve as a basis for further discussions:

**Definition 5.5.1. [Initial fairness]** Given a fairness condition $\mathcal{F}$ and an MDP $\mathcal{M} = (S, \text{Act}, \rightarrow, s_0, \text{AP}, \text{L})$, a scheduler $\mathcal{U}$ is denoted *initially fair*, if

\[
\Pr_{s_0}^M \{ \pi \in \text{Paths} | \pi \models \text{fair } \mathcal{F} \} = 1.
\]

We call the fairness condition $\mathcal{F}$ *initially realizable*, if there exists an initially fair scheduler and denote the set of initially fair schedulers by $\text{iFairSched}$.

The process of checking whether a given fairness condition $\mathcal{F}$ is initially realizable obviously means checking whether $\Pr^{\text{max}}(\mathcal{M} \models \mathcal{F}) = 1$ and we observe, that an on-the-fly approach which relies only on local criteria of maximal end components, like in section 5.4, seems not possible.

The goal is to compute

\[
\Pr^{\text{ifairmax}}_\mathcal{F}(\mathcal{M} \models \mathcal{E}) \overset{\text{def}}{=} \sup_{\mathcal{U} \in \text{iFairSched}} \Pr_{s_0}^{\mathcal{M}_\mathcal{U}}(\{ \pi \in \text{Paths} : \pi \models \mathcal{E} \}).
\]

We also observe, that in general

$\mathcal{F}$ initially realizable $\not\Rightarrow \left( \Pr^{\text{ifairmax}}_\mathcal{F}(\mathcal{M} \models \mathcal{E}) = \Pr^{\text{max}}(\mathcal{M} \models \mathcal{E} \land \mathcal{F}) \right)$,

which is illustrated by the following two examples:

**Examples.** In figure 5.3(1) a schematic view of an MDP $\mathcal{M}_1$ is presented, for which its fairness condition is unrealizable, since maximal end component $\mathcal{E}_2$ is not fair (and not bottom). Because of this, the class of fair schedulers $\text{FairSched}$ is empty. However, the class of initially fair schedulers $\text{iFairSched}$ is not empty and, intuitively speaking, contains all schedulers that avoid entering maximum end component $\mathcal{E}_2$ and rather remain in $\mathcal{E}_1$. In such cases, it then obviously holds, that $\Pr^{\text{ifairmax}}_\mathcal{F}(\mathcal{M}_1 \models \mathcal{E}) = \Pr^{\text{max}}(\mathcal{M}_1 \models \mathcal{E} \land \mathcal{F}) = 1$.

As mentioned above, this does not hold in general. Figure 5.3(2) shows a second MDP $\mathcal{M}_2$, whichs fairness condition is also unrealizable, but initially realizable. Here we have $\Pr^{\text{ifairmax}}_\mathcal{F}(\mathcal{M} \models \mathcal{E}) = 0$. Though we have an accepting end component $\mathcal{E}_3$, there exists no initially fair scheduler that reaches $\mathcal{E}_3$. Consequently, $\mathcal{E}_3$ must not taken into account for the calculation of $\Pr^{\text{ifairmax}}_\mathcal{F}$, and we have

\[
\Pr^{\text{ifairmax}}_\mathcal{F}(\mathcal{M}_2 \models \mathcal{E}) = 0 \neq \Pr^{\text{max}}(\mathcal{M}_2 \models \mathcal{E} \land \mathcal{F}).
\]
To handle this situation, the case of calculating \( \Pr_{\text{ifairmax}}(M \models E) \) can be reduced to calculating \( \Pr_{\text{fairmax}}(M' \models E) \) by constructing from the initially realizable MDP \( M \) a (reduced) MDP \( M' \) that is realizable. The MDP \( M' \) results from \( M \) by deleting all states \( \{s \in S : \Pr_{\text{s,max}}^{\text{fairmax}}(\pi \in \text{Paths} | \pi \vdash \text{fair}F) < 1\} \) from \( M \), thus enforcing realizability for all states in \( M' \). Similar to the algorithm for checking realizability this can be done by applying algorithm PROB1MAX with target set FMEC to determine states \( S' = \{s | \Pr_{s}^{\text{max}}(\circ \text{FMEC}) = 1\} \). The reduced MDP \( M' \) then has the state space \( S' \).

**Theorem 5.5.2** (Calculating \( \Pr_{\text{ifairmax}}(M \models E) \)). Given an MDP \( M \) and the reduced system \( M' \) as described above, a general fairness condition \( F \) and an \( \omega \)-regular property \( E \), it holds that

\[
\Pr_{\text{ifairmax}}(M \models E) = \Pr_{\text{fairmax}}(M' \models E).
\]

This shows that initial fairness, a concept that is more rooted in the setting of a traditional non-probabilistic analysis can be reduced to the setting described in this chapter where fairness conditions need to be realizable. In this context it is worth noting that \( M' \) is a reduced version of \( M \). We will not present experimental results on this as postpone an implementation as future work.
Partial order reduction of Markov decision processes

Partial order reduction methods reduce the size of interleaving models while preserving stutter-invariant (path- or branching-) properties and were proposed in various variants for coping with the state explosion problem in model checking. This is effectively achieved through safely restricting to a certain execution order of enabled concurrent independent actions $\alpha, \beta, \gamma, \ldots \in \text{Act}(s)$ in some state $s$ rather than exploring all possible execution orders (see scheme in figure 6.1). This procedure is feasible under certain circumstances, namely if the property under investigation does not depend on the order of these actions. Because interleaving models represent all execution orders of independent actions, the partial order reduction method can lead to enormous reductions. In this chapter, we address the ample set method for partial order reduction of MDPs that preserves stutter-invariant $\omega$-regular properties in MDPs, as proposed in [BGC04, dN04], see also [Gro08]. This method can be seen as an extension of Peled’s ample set method [Pel93, HP94, Pel96], that is implemented for instance in the model checker SPIN [Hol97]. After giving preliminary definitions concerning the ample set method, we discuss its implementation. Finally, we also discuss the compatibility of the partial order reduction and its implementation with the concept of fairness. Parts of the material presented here have been published in [CBGK08] and [BGC09b].

6.1. The ample-set method for MDPs

Starting from an MDP $\mathcal{M} = (S, \text{Act}, \rightarrow, s_0, \text{AP}, \text{L})$ that is analyzed against a stutter-invariant $\omega$-regular property, the main idea of the ample set method is to assign to any (reachable) state $s \in S$ a set $\text{ample}(s) \subseteq \text{Act}(s)$ and construct a reduced MDP $\hat{\mathcal{M}}$ by using action sets $\text{ample}(s)$ rather than $\text{Act}(s)$. The state space of the reduced MDP

$$\hat{\mathcal{M}} = (\hat{S}, \text{ample}, \rightarrow, s_0, \text{AP}, \hat{\text{L}})$$

induced by ample is the smallest set $\hat{S} \subseteq S$ that contains $s_0$ and any state $t$ where $P(s \xrightarrow{\alpha} t) > 0$ for some $s \in \hat{S}$ and $\alpha \in \text{ample}(s)$. The labeling function $\hat{\text{L}}: \hat{S} \rightarrow 2^\text{AP}$ is the restriction of the labeling function $\text{L}$ to the state-set $\hat{S}$. In the cases where $\text{ample}(s) = \text{Act}(s)$ we call $s$ fully expanded. If for comparably many states $s$, $\text{ample}(s)$ is smaller than $\text{Act}(s)$, the state
space of $\hat{M}$ becomes a proper subset of the state space of $M$ and might be much “easier” to analyse. Especially, the linear program that needs to be solved for a quantitative analysis of $\hat{M}$ against $\omega$-regular properties (see sections 2.2 and chapter 4) may contain significantly less variables and inequations.

Conditions C1 - C5 for ample. We now briefly explain the five conditions for ample shown in figure 6.2 that have been presented in [BGC04] (see also [Gro08]) and shown to preserve stutter-invariant $\omega$-regular properties. This means that for any scheduler $\mathcal{U}$ of $M$ there exists a scheduler $\hat{\mathcal{U}}$ of $\hat{M}$ such that

$$P^{M,\mathcal{U}}(E) = P^{\hat{M},\hat{\mathcal{U}}}(E)$$

for any stutter-invariant $\omega$-regular property $E$, and vice versa (see Theorem 6.1.2 below). Conditions C1 - C3 agree with Peled’s conditions in the non-probabilistic setting. Condition C1 states that ample sets are only empty if the original action set $\text{Act}(s)$ is empty, and condition C2 ensures stuttering of all actions (see definition 2.1.9 on page 15 in ample(s)) unless $s$ is fully expanded. The purpose of condition C3 is mainly to ensure that any path in the original MDP $M$ can be generated from a path in $\hat{M}$ by permuting independent actions and possible inserting stutter actions.

Condition C3 utilizes the notion of independence of probabilistic actions and is the natural extension of the independence relation for actions in non-probabilistic systems [Pel93, HP94]. In non-probabilistic systems, independence of two actions $\alpha$ and $\beta$ means that for any state $s$ where both $\alpha, \beta \in \text{Act}(s)$, the execution of $\alpha$ does not affect the enabledness of $\beta$ (i.e., the $\alpha$-successor of $s$ has an outgoing $\beta$-transition), and vice versa. Additionally it is required that the action sequences $\alpha\beta$ and $\beta\alpha$ lead to same state. In operational models which originate from systems that are based on the stepwise execution of parallel processes
C1 For all states $s \in S$, ample$(s) = \emptyset$ iff Act$(s) = \emptyset$.

C2 If $s \in \hat{S}$ and ample$(s) \neq \text{Act}(s)$ then all actions $\alpha \in \text{ample}(s)$ are stutter actions.

C3 For each path $\sigma = s \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_n \xrightarrow{\gamma} \ldots$ in $M$ where $s \in \hat{S}$ and $\gamma$ is dependent on ample$(s)$ there exists an index $i \in \{1, \ldots, n\}$ such that $\alpha_i \in \text{ample}(s)$.

C4 For each end component $(T, A)$ in $\hat{M}$: $\alpha \in \bigcap_{t \in T} \text{Act}(t)$ implies $\alpha \in \bigcup_{t \in T} \text{ample}(t)$.

C5 If $\sigma = s \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_n \xrightarrow{\gamma} \ldots$ is a path in $M$ where $s \in \hat{S}$, $\alpha_1, \ldots, \alpha_n, \gamma \notin \text{ample}(s)$ and $\gamma$ is probabilistic then $|\text{ample}(s)| = 1$.

Figure 6.2: Five conditions for the ample-sets

(as it is the case in this thesis, see chapter 3), independent actions are typically actions of different processes that only refer to local variables. The following definition of action independence for the probabilistic case is an extension of Peled’s condition with the additional requirement, that $\alpha \beta$ and $\beta \alpha$ must impose identical probabilistic effects.

**Definition 6.1.1. [Independence of actions]** Given an MDP $M = (S, \text{Act}, \rightarrow, s_0, \text{AP}, L)$, two actions $\alpha, \beta \in \text{Act}$, $\alpha \neq \beta$ are called independent (in $M$), written $\alpha \nleftrightarrow \beta$, if and only if for all states $s, t, u \in S$ with $\{\alpha, \beta\} \subseteq \text{Act}(s)$:

1. $P(s \xrightarrow{\alpha} t) > 0$ implies $\beta \in \text{Act}(t)$,
2. $P(s \xrightarrow{\beta} u) > 0$ implies $\alpha \in \text{Act}(u)$ and
3. for all states $w \in S$:

$$\sum_{t \in S} P(s \xrightarrow{\alpha} t) \cdot P(t \xrightarrow{\beta} w) = \sum_{u \in S} P(s \xrightarrow{\beta} u) \cdot P(u \xrightarrow{\alpha} w)$$

Two different actions $\alpha$ and $\beta$ are called dependent, written $\alpha \nabla \beta$, if they are not independent. If $A \subseteq \text{Act}$ and $\alpha \in \text{Act} \setminus A$ then $\alpha$ is called independent from $A$, denoted $\alpha \nleftrightarrow A$, if for all actions $\beta \in A$, $\alpha$ and $\beta$ are independent. Otherwise, $\alpha$ is called dependent on $A$.

Note that applying the above definition to non-probabilistic actions $\alpha$ and $\beta$ (i.e., where $P(s \xrightarrow{\alpha} t), P(s \xrightarrow{\beta} t) \in \{0, 1\}$ for all states $s, t$) yields the definition of independence of actions (see, e.g., [CGP99]) in ordinary (non-probabilistic) transition systems. We now give an example.
Example. The left part of fig. 6.3 shows a fragment of an MDP \( \mathcal{M}_1 \) representing the parallel execution of independent actions \( \alpha \) and \( \beta \), where
\[
\alpha = \text{“roll a 6-sided dice and assign value 1 or 0 to local boolean variable } x, \\
\text{depending on whether the outcome is 1 or in } \{2,3,4,5,6\}”, \text{ and} \\
\beta = \text{“roll a 6-sided dice and assign value 1 or 0 to boolean variable } y, \\
\text{depending on whether the outcome is in } \{1,2,3\} \text{ or in } \{4,5,6\}”.
\]
In general, whenever actions \( \alpha \) and \( \beta \) stand for stochastic experiments (i.e., experiments with fixed probabilities) that are also independent in the classical sense, then \( \alpha \downarrow \beta \) holds according to definition 6.1.1. However, there are also other situations where two actions can be independent. For example, actions \( \alpha \) and \( \beta \) of MDP \( \mathcal{M}_2 \) in Fig. 6.3 are independent, though the probabilities may vary for a single action. To see this, note that \( s \) is the only state, such that \( \text{Act}(s) = \{\alpha, \beta\} \). The \( \alpha \)-successors \( t, s \) of \( s \) have a \( \beta \)-transition to state \( u \), while the \( \beta \)-successor \( u \) has an \( \alpha \)-transition to itself. The effect under the action sequences \( \alpha \beta \) and \( \beta \alpha \) is the same. There are the two \( \alpha \beta \)-paths \( s \xrightarrow{\alpha} s \xrightarrow{\beta} u \) and \( s \xrightarrow{\alpha} t \xrightarrow{\beta} u \) that both lead with probability 1/2 to state \( u \), while the \( \beta \alpha \)-path \( s \xrightarrow{\beta} u \xrightarrow{\alpha} u \) has probability 1. Hence, state \( u \) is reached with probability 1 in both cases.

Condition C3 in figure 6.2 states that for each path \( \sigma = s \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_n \xrightarrow{\gamma} \ldots \) in \( \mathcal{M} \) where \( s \in \hat{S} \) and \( \gamma \) is dependent on \( \text{ample}(s) \) there exists an index \( i \in \{1, \ldots, n\} \) such that \( \alpha_i \in \text{ample}(s) \). Informally speaking, this condition requires that when starting in some state \( s \), on every path of the original MDP \( \mathcal{M} \) some action in \( \text{ample}(s) \) has to appear before any action that is dependent on \( \text{ample}(s) \). Thus, to realize C3 algorithmically, one needs global information about the unreduced system, which is at the same time avoided being built. This problem will be addressed in section 6.2.

Condition C4 can be viewed as the probabilistic pendant to Peled’s cycle condition. The cycle condition requires that an action \( \alpha \) that is enabled in all states on a cycle of \( \hat{\mathcal{M}} \) be-
longs to the ample set of at least one of the states on this cycle [Pel93]. Not including \( \alpha \) in \( \text{ample}(s) \) for at least one state on the cycle would postpone \( \alpha \) ad infinitum and effectively eliminate the scheduler that leaves the cycle via \( \alpha \). This could possibly have the undesired effect of cutting off subgraphs that might contain interesting "future" behaviour that is dependent on the execution of \( \alpha \). In the probabilistic setting, the definition of \( \text{C4} \) is lifted from cycles to end components in an MDP. The probabilistic condition \( \text{C4} \) applied to an ordinary (non-probabilistic) transition system viewed as an MDP (i.e., where all actions are non-probabilistic and yield unique successor states) reduces to Peled’s non-probabilistic cycle-condition, because for such “non-probabilistic MDPs”, the end components correspond to cycles.

The character of condition \( \text{C5} \) is similar to condition \( \text{C3} \), but concerns probabilistic actions, i.e., actions that involve a distribution \( \mu \) with \( |\text{support}(\mu)| > 1 \), rather than dependent actions. If \( \sigma = s \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_n \xrightarrow{\gamma} \ldots \) is a path in \( \mathcal{M} \) where \( s \in \hat{S} \), \( \alpha_1, \ldots, \alpha_n, \gamma \notin \text{ample}(s) \) and \( \gamma \) is probabilistic then \( |\text{ample}(s)| = 1 \). This means that whenever a probabilistic action \( \gamma \) occurs before an action \( \alpha \in \text{ample}(s) \) in \( \mathcal{M} \), the ample set must be a singleton (see [BGC04, Gro08]). The ratio behind \( \text{C5} \) is as follows: If \( \text{Act}(s) = \{\alpha, \beta, \gamma\} \), where \( \alpha \) is a probabilistic action with \( \alpha \xrightarrow{\lambda} \beta \) and \( \alpha \xrightarrow{\lambda} \gamma \), then the stochastic experiments induced by "execute \( \alpha \) and then, depending on the outcome of it, select \( \beta \) or \( \gamma \) for execution" or "select \( \beta \) or \( \gamma \) and then execute \( \alpha \)" might be different.

The main theorem of [BGC04] on the ample set method for partial order reduction of MDPs is, that these five condition are sufficient for preserving probabilities of stutter-invariant LT-properties:

**Theorem 6.1.2** (Ample set method for MDPs). Let \( \mathcal{M} = (S, \text{Act}, \xrightarrow{\cdot}, s_0, \text{AP}, L) \) be an MDP, \( \text{ample} : S \to 2^{\text{Act}} \) a function satisfying the conditions \( \text{C1-C5} \) given above and \( \hat{\mathcal{M}} \) the reduced MDP that emanates from the MDP \( \mathcal{M} \) and the ample sets defined by the function ample. Then for each scheduler \( \mathcal{U} \in \text{Sched of } \mathcal{M} \) there exists a scheduler \( \hat{\mathcal{U}} \in \text{Sched of } \hat{\mathcal{M}} \) such that

\[
\Pr_{\mathcal{M}\mathcal{U}}(E) = \Pr_{\hat{\mathcal{M}}\hat{\mathcal{U}}}(E)
\]

for each stutter-invariant measurable LT property \( E \subseteq (2^{\text{AP}})^\omega \). \( \square \)

Vice versa, any scheduler \( \hat{\mathcal{U}} \in \text{Sched of } \hat{\mathcal{M}} \) is also a scheduler for \( \mathcal{M} \) (as \( \hat{\mathcal{M}} \) is a sub-MDP of \( \mathcal{M} \)). Hence we obtain:

**Corollary 6.1.3.** Given an MDP \( \mathcal{M} = (S, \text{Act}, \xrightarrow{\cdot}, s_0, \text{AP}, L) \), a function \( \text{ample} : S \to 2^{\text{Act}} \) satisfying the conditions \( \text{C1-C5} \), where \( \hat{\mathcal{M}} \) is the reduced MDP, then, it holds that

\[
\sup_{\mathcal{U} \in \text{Sched}_{\mathcal{M}}} \Pr_{s_0 \mathcal{U}}(E) = \sup_{\hat{\mathcal{U}} \in \text{Sched}_{\hat{\mathcal{M}}}} \Pr_{s_0 \hat{\mathcal{U}}}(E)
\]
6.2. Efficient calculation of ample sets

We now explain the main concepts that have been developed for an efficient implementation of the ample-set method for MDPs based on conditions C1-C5 and report on results concerning their implementation in the model checker LiQuor which illustrates the practical relevance. From a more abstract point of view, an algorithm that realizes the ample-set method needs procedures that check whether conditions C1-C5 are met by a given candidate set ample(s) in \( \hat{\mathcal{M}} \). It is clear that conditions C1 and C2 (emptiness rule and stutter rule) can be checked easily by using only local information in the vicinity of a state \( s \) of \( \mathcal{M} \).

Conditions C3-C5 on the other hand have a global character, which makes them harder to check efficiently, especially in an on-the-fly based setting. Condition C3 is a global condition that refers to all actions \( \alpha_i \) of paths \( \pi = s \stackrel{\alpha_1}{\rightarrow} s_1 \stackrel{\alpha_2}{\rightarrow} \ldots \in \text{Paths}_s^{\mathcal{M}} \) in the (unreduced) MDP \( \mathcal{M} \). It was shown in [CGP99, Pel93], that checking C3 in the non-probabilistic case is at least as hard as solving the reachability problem in \( \mathcal{M} \) itself, but since \( \mathcal{M} \) is subject to reduction in the first place, building \( \mathcal{M} \) as a whole has to be avoided and is therefore not available for an analysis in its entirety. Condition C4 refers to end components in \( \hat{\mathcal{M}} \) (see page 27). End components are potentially large structures (in the order of magnitude as \( \hat{\mathcal{M}} \) itself), which renders C4 being global condition, too. Similarly to \( \mathcal{M} \), a potentially complete analysis of \( \hat{\mathcal{M}} \) to check C4 for a single candidate set ample(s) seems not practical, since checking C4 is PTIME hard in the size of \( \hat{\mathcal{M}} \) as we show in Lemma 6.2.4 (see page 116). Finally, condition C5 is, again, a global condition in \( \mathcal{M} \), similar to C3. We show that the argumentation for the complexity of checking condition C3 carries directly over to C5 in the sense that checking C5 is as hard as solving the reachability problem in \( \mathcal{M} \) (see Lemma 6.2.5 on page 123).

A practical implementation approach for checking conditions C3-C5 is the employment of stronger conditions S3-S5 that are sufficient for ensuring the original conditions C3-C5 on the one hand, but can be checked efficiently with only local information on \( \mathcal{M} \) or with information on the global structure of \( \hat{\mathcal{M}} \) that is inherently obtained by the DFS-based on-the-fly generation of \( \hat{\mathcal{M}} \) on the other hand. This kind of approach is strongly related to the successful on-the-fly partial order reduction approach realized in the popular model checker SPIN [HP94, Pel93]. There, local information about transition system states and their corresponding Promela statements is used to check stronger sufficient criteria for the
ample set conditions. For quantitative analysis as discussed here we aim to follow that spirit. Since the starting point of the analysis is a PROBMela program and with this a set of PCGs according to the operational semantics given in chapter 3 we give stronger conditions sufficient for C1-C5 that can be checked exploiting local information in the MDP and information gained from a preanalysis of the probabilistic control graphs (PCGs) for $M$ instead of analyzing global conditions in $M$ or $\hat{M}$ directly.

In the sequel, we suppose that we are given a PROBMela program with $n$ processes $P_1, \ldots, P_n$ and sets $\text{Var}$ and $\text{Chan}$ of variables and channels. Let $\Psi_i = (\text{Loc}_i, \text{Events}_i, \sim_i, P_i)$ be the probabilistic control graph (PCG) for processes $P_i, 0 \leq i \leq n$ (see section 3.2.1) equipped with their MDP semantics as given in section 3.2.2. Before now giving stronger conditions $S3$-$S5$ we discuss a stronger criterion for independence of two actions $\alpha, \beta \in \text{Act}$ in an MDP $M$, which is the basis of condition $S3$.

**Overapproximation of the independence relation.** Since independence itself is a global condition on actions in $M$ we switch to a stronger syntactic notion of independence that can easily be derived from a PCG analysis. For PCG edges $\alpha = (Q \overset{g,a}{\rightarrow} \nu), \nu \in \text{Distr}(\text{Loc}_i)$ we denote by $\text{read}(\alpha) \subseteq \text{Var} \cup \text{Chan}$ the set of variables and channels that are read by guard $g$ or event $a$. By “reading a variable” it is meant that the variable is involved in guard $g$ or in an expression that reads the variable when firing event $a$. By “reading a channel” we refer to an access to a channel by operation $\text{front}$ in $a$. Similarly, we denote by $\text{write}(\alpha) \subseteq \text{Var} \cup \text{Chan}$ the set of variables that are written by $\alpha$, which identifies the variables that are target of an assignment when firing event $a$. In the sequel, we use the notion data object to refer to either a variable in $\text{Var}$ or a channel in $\text{Chan}$.

**Formal definition of $\text{read}(\alpha)$, $\text{write}(\alpha)$.** Let $\alpha = Q \overset{g,a}{\rightarrow} \nu$ be an edge in the PCG $\Psi_i$. We define
\[
\begin{align*}
\text{read}(\alpha) &= \text{read}(g) \cup \text{read}(a) \\
\text{write}(\alpha) &= \text{write}(a) \\\n\text{data}(\alpha) &= \text{read}(\alpha) \cup \text{write}(\alpha)
\end{align*}
\]
where the data object sets $\text{read}(g)$, $\text{read}(a)$ and $\text{write}(a)$ are defined as follows.

For the guard $g$, we define $\text{read}(g)$ as the set of data objects where $g$ refers to. Hence, if $\eta, \eta'$ are evaluations for the data objects, i.e., $\eta, \eta' \in \text{Eval}(\text{Var} \cup \text{Chan})$, such that $\eta(x) = \eta'(x)$ for all $x \in \text{read}(g)$ then $\eta \models g$ iff $\eta' \models g$.

For the event $a$, $\text{read}(a)$ denotes the set of data objects $x$ where $a$ accesses to, possibly without affecting the current data item hold in $x$, while $\text{write}(a)$ denotes the set of all data objects $x$ where the current data item hold in $x$ might change when event $a$ fires. Formally:
6.2. Efficient calculation of ample sets

Chapter 6. Partial order reduction of Markov decision processes

\[ x \in \text{write}(\alpha) \iff \exists \eta, \zeta \in \text{Eval}(\text{Var} \cup \text{Chan}) \text{ s.t.} \]
\[ \text{Effect}_i(\alpha, \eta)(\zeta) > 0 \text{ and } \eta(x) \neq \zeta(x) \]

\[ x \in \text{read}(\alpha) \iff \exists \eta, \eta', \zeta \in \text{Eval}(\text{Var} \cup \text{Chan}) \text{ s.t.} \]
\[ (1) \quad \eta(y) \neq \eta'(y) \text{ for all data objects } y \neq x \]
\[ (2) \quad \text{Effect}_i(\alpha, \eta)(\zeta) \neq \text{Effect}_i(\alpha, \eta)(\zeta') \]
\[ \text{where } \zeta' = \begin{cases} 
\zeta & : \text{if } x \in \text{write}(\alpha) \\
\zeta[x := \eta'(x)] & : \text{if } x \notin \text{write}(\alpha)
\end{cases} \]

Lemma 6.2.1 (Sufficient criterion for the independence of actions). Let \( \alpha \) be an edge in the PCG \( \mathcal{P}_i \) and \( \beta \) an edge in the PCG \( \mathcal{P}_j \) such that \( i \neq j \) and

\[ \text{write}(\alpha) \cap \text{data}(\beta) = \emptyset \text{ and } \text{write}(\beta) \cap \text{data}(\alpha) = \emptyset. \]

Furthermore, we suppose that neither \( \alpha \) nor \( \beta \) stands for synchronous message passing. Then, \( \alpha \) and \( \beta \) viewed as actions in the MDP \( \mathcal{M} \) are independent (i.e., \( \alpha \wedge \beta \)).

**Proof.** Let \( \alpha = Q_{\alpha}^{a, \nu_\alpha} \) and \( \beta = Q_{\beta}^{h, \nu_\beta} \). Suppose that \( s \) is a state in \( \mathcal{M} \) such that \( \{\alpha, \beta\} \subseteq \text{Act}(s) \). Then, \( s \) has the form \( (Q_1, \ldots, Q_n, \eta_s) \) where \( Q_k \) is the control state of the \( k \)-th process and \( \eta_s \) an evaluation for the variables and asynchronous channels. Since \( \alpha \) and \( \beta \) are enabled in \( s \) we have \( Q_i = Q_{\alpha}, Q_j = Q_{\beta}, \text{ and } \eta_s \models g \wedge h \).

We now check that the three conditions imposed in the definition of the independence relation (Definition 6.1.1) hold.

1. Let \( t \) be an \( \alpha \)-successor of \( s \). \( t = (Q'_1, \ldots, Q'_n, \eta_t) \) where \( Q'_k = Q_k \) if \( k \in \{1, \ldots, n\} \setminus \{i\} \), \( \nu_\alpha(Q'_i) > 0 \) and \( \eta_t \) arises from \( \eta_s \) when firing event \( a \). Hence, \( \eta_s(x) = \eta_t(x) \) for all data objects \( x \notin \text{write}(\alpha) \). As \( \text{read}(\beta) \cap \text{write}(\alpha) = \emptyset, \eta_s \) and \( \eta_t \) agree on all variables and channels that are relevant for the truth value of guard \( h \) of action \( \beta \). As \( \eta_s \models h \) we obtain \( \eta_t \models h \). This yields \( \beta \in \text{Act}(t) \).

2. By symmetry, we get that each \( \beta \)-successor of state \( s \) has the form \( u = (Q'_1, \ldots, Q'_n, \eta_u) \) where \( Q'_k = Q_k \) if \( k \in \{1, \ldots, n\} \setminus \{j\} \), \( \nu_\beta(Q'_j) > 0 \) and \( \eta_u \) arises from \( \eta_s \) when firing event \( b \).

3. It remains to show that the probabilistic effect of the action sequences \( \alpha \beta \) and \( \beta \alpha \) is the same, when executed in state \( s \). Let \( \mu_{\alpha \beta}, \mu_{\beta \alpha} : S \to [0, 1] \) be the functions given by:

\[ \mu_{\alpha \beta}(w) = \sum_{t \in S} P(s \xrightarrow{\alpha} t) \cdot P(t \xrightarrow{\beta} w) \]
\[ \mu_{\beta \alpha}(w) = \sum_{u \in S} P(s \xrightarrow{\beta} u) \cdot P(u \xrightarrow{\alpha} w) \]
The goal is to show that \( \mu_{\alpha\beta}(w) = \mu_{\beta\alpha}(w) \) for all states \( w \). Let

\[
w = (Q''_1, \ldots, Q''_n, \eta_w)
\]

be a state in \( \mathcal{M} \). If \( Q''_k \neq Q_k \) for some \( k \in \{1, \ldots, n\} \setminus \{i, j\} \) then \( \mu_{\alpha\beta}(w) = \mu_{\beta\alpha}(w) = 0 \). Similarly, if \( \eta_w(x) \neq \eta_k(x) \) for some data object \( x \notin \text{write}(\alpha) \cup \text{write}(\beta) \) then \( \mu_{\alpha\beta}(w) = \mu_{\beta\alpha}(w) = 0 \). Suppose now that

- \( Q''_k \neq Q_k \) for all \( k \in \{1, \ldots, n\} \setminus \{i, j\} \)
- \( \eta_w(x) = \eta_k(x) \) for all data objects \( x \notin \text{write}(\alpha) \cup \text{write}(\beta) \).

Then:

\[
\mu_{\alpha\beta}(w) = \sum_{\eta} \nu_\alpha(Q''_i) \cdot \text{Effect}_i(a, \eta_k)(\eta_w) \cdot \nu_\beta(Q''_j) \cdot \text{Effect}_j(b, \eta)(\eta_w)
\]

where \( \eta \) ranges over all evaluations for \( \text{Var} \cup \text{Chan} \), and the analogous formula for \( \mu_{\beta\alpha}(w) \). Thus, it suffices to show that

\[
\sum_{\eta} \text{Effect}_i(a, \eta_k)(\eta) \cdot \text{Effect}_j(b, \eta)(\eta_w) = \sum_{\zeta} \text{Effect}_j(b, \eta_k)(\zeta) \cdot \text{Effect}_i(a, \zeta)(\eta_w)
\]

For better readability of the following calculations, we split the set \( \text{Var} \cup \text{Chan} \) into

- the data objects that are written by \( \alpha \),
- the data objects that are written by \( \beta \)
- and the remaining data objects that are not affected by \( \alpha \) or \( \beta \).

The condition imposed on \( \alpha \) and \( \beta \) ensures that \( \text{write}(\alpha) \cap \text{write}(\beta) = \emptyset \). Hence, these three sets of data objects are pairwise disjoint. For simplicity, we write evaluations for \( \text{Var} \cup \text{Chan} \) in the form \([x=v_x, y=v_y, z=v_z]\) where \( y \) stands for the tuple of variables and channels in \( \text{write}(\alpha) \), \( z \) for the tuple of variables and channels in \( \text{write}(\beta) \), and \( x \) for the tuple of all other data objects in \( \text{Var} \cup \text{Chan} \), and \( v_y, v_x \) and \( v_x \) denote tuples of concrete data items assigned to \( z, x \) and \( x \). Let us assume that

\[
\begin{align*}
\eta_k &= [x=v_x, y=v_y, z=v_z] \\
\eta_w &= [x=v_x, y=v'_y, z=v'_z]
\end{align*}
\]

Recall that both \( a \) and \( b \) have no effect on \( x \). Event \( a \) can affect \( y \), but not \( z \). Hence, we get:

\[
\begin{align*}
\text{Effect}_i(a, [x=v_x, y=v_y, z=v_z])|_{[x=v_x, y=v'_y, z=v'_z]} &= \text{Effect}_i(a, [x=v_x, y=v'_y, z=v'_z])|_{[x=v_x, y=v_y, z=v_z]} \\
\end{align*}
\]
Similarly, event \( b \) might modify \( z \), but not \( y \). Thus:

\[
\text{Effect}_j(b, [x=v_x, y=v'_y, z=v_z]) [x=v_x, y=v'_y, z=v_z]
\]

\[
= \text{Effect}_j(b, [x=v_x, y=v_y, z=v'_z]) [x=v_x, y=v_y, z=v'_z]
\]

Putting things together we obtain:

\[
\sum_{\eta} \text{Effect}_i(a, \eta_s)(\eta) \cdot \text{Effect}_j(b, \eta)(\eta_w)
\]

\[
= \sum_{v'_y} \text{Effect}_i(a, [x=v_x, y=v_y, z=v_z]) [x=v_x, y=v'_y, z=v_z] \cdot \\
\text{Effect}_j(b, [x=v_x, y=v'_y, z=v'_z]) [x=v_x, y=v'_y, z=v'_z]
\]

\[
= \sum_{v'_y} \text{Effect}_j(b, [x=v_x, y=v'_y, z=v'_z]) [x=v_x, y=v'_y, z=v'_z] \cdot \\
\text{Effect}_i(a, [x=v_x, y=v_y, z=v'_z]) [x=v_x, y=v_y, z=v'_z]
\]

\[
= \sum_{\zeta} \text{Effect}_j(b, \eta_s)(\zeta) \cdot \text{Effect}_i(a, \zeta)(\eta_w)
\]

**Preprocessing PCGs for calculating** \( \text{read}(\cdot) \) and \( \text{write}(\cdot) \): In LiQuor, the sets \( \text{read}(\alpha) \) and \( \text{write}(\alpha) \) can be derived easily from the PASM code of a PCG. For each corresponding edge the corresponding set of PASM operations can simply be searched for read and write operations. For an efficient implementation, the overapproximation of the independence relation can be calculated in advance and be stored in a suitable data structure, such that the overhead during state space exploration is reduced to a minimum. Consider the following example.

**Example.** Reconsider the arbiter model from section 3.2.2, page 53. In table 6.1, only the relevant edges \( \alpha \in \text{Edges} \) with \( \text{read}(\alpha) \neq \emptyset \) or \( \text{write}(\alpha) \neq \emptyset \) are shown. Locations in \( \text{Loc}(\cdot) \) are displayed as their corresponding line number of the PROBMela process. In the first column, we name these edges.

Since \( x \) is a local variable, the actions of Process\(_1\) and Process\(_2\) are pairwise independent. We have the following dependent action relationships:
Choosing ample($s$). We now proceed with sufficient criteria for C3-C5, that can be checked in an on-the-fly DFS-based setting. In the implementation we aim at constructing $\hat{M}$ on-the-fly with a DFS-based procedure similar to Procedure Generate_MDP. For instance it is possible to use Procedure Generate_MDP without changes and ensure that the local list of enabled events (see page 58) will always form a valid ample set. We choose the ample set such that it is unreduced or it consists of the actions of one particular process, i.e.,

$$\text{ample}(s) = \text{Act}_i(s) \text{ for some } i \in \{1, ..., n\} \text{ or ample}(s) = \text{Act}(s).$$

We assume that the states in the MDP have the form $(Q_1, \ldots, Q_n, \eta)$, were $Q_i$ is the (sub-)process with process-index $i$. Recall that the set of edges $Q \xrightarrow{\nu} Q'$ in the PCG $P_i$ of process $P_i$ induced by relation $\sim$ is denoted as $\text{Edges}_{P_i}$. According to the MDP semantics for PCGs presented in chapter 3, the set $\text{Edges}_{P_i}$ corresponds to the set $\text{Act}_i$ of MDP actions in the sense that each action in $\alpha \in \text{Act}_i$ of $\mathcal{M}$ and $\hat{M}$ origins from a specific control edge $\alpha = (Q \xrightarrow{\nu} Q') \in \text{Edges}_{P_i}$ with $\nu(Q', \alpha) > 0$ that goes from location $Q$ over to $Q'$, where $Q, Q' \in \text{Loc}(P_i)$ (see page 45 ff.). In the following we denote sequences of edges

$$\alpha_1 \xrightarrow{\gamma_1} \alpha_2 \xrightarrow{\gamma_2} \alpha_3 \xrightarrow{\gamma_3} \alpha_4 \xrightarrow{\gamma_4} \alpha_5 \xrightarrow{\gamma_5} \cdots$$

Table 6.1: read and write sets for the mutual exclusion example from section 3.2.2
We then have the following two lemmas to prove that $S_3$ as control paths

Paths $s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_n \in \text{Paths}^M$ and their corresponding sequences of actions $\alpha_1, \alpha_2, \ldots, \alpha_n$ in the MDP are associated with their control paths as follows:

$$Q_0\xrightarrow{\alpha_1}\ldots\xrightarrow{\alpha_n}Q_n.$$ 

We discuss stronger conditions that are sufficient criteria for conditions C3-C5.

**Sufficient criterion S3 for C3:** Condition C3 requires that on every path of the original (or unreduced) model $M$ starting in state $s$ some action in $\text{ample}(s)$ has to appear before any action that is dependent on $\text{ample}(s)$. Condition S3 consists of two parts S3.1 and S3.2 that are together sufficient for C3 and are formulated in terms of control paths in PCGs $\Psi_1, \ldots, \Psi_n$ of a PROBMela program $P$.

**Condition S3 sufficient for C3:**
Let $\text{ample}(s) = \text{Act}(s)$. If $\alpha \in \text{ample}(s)$, then the following two subcriteria hold:

**S3.1:** For all $j \in \{1, \ldots, n\} \setminus \{i\}$ and all control paths

$$\ell_j = l_0 \xrightarrow{g_1 : \alpha_1} l_1 \xrightarrow{g_2 : \alpha_2} \ldots \xrightarrow{g_n : \alpha_n} l_m.$$ 

in $\Psi_j$ we have $\alpha \land \gamma_k$ for all $\alpha \in \text{Act}(s)$ and $1 \leq k \leq m$.

**S3.2:** Whenever $\beta = (\ell_i \xrightarrow{g_i : \nu} \nu)$ is a control transition in $\Psi_i$ such that $\beta \notin \text{Act}(s)$ then for all $j \in \{1, \ldots, n\} \setminus \{i\}$ and all control paths

$$\ell_j = l_0 \xrightarrow{g_1 : \alpha_1} l_1 \xrightarrow{g_2 : \alpha_2} \ldots \xrightarrow{g_n : \alpha_n} l_m.$$ 

in $\Psi_j$ we have that there is no $k \in \{1, \ldots, m\}$ such that a variable $x \in \text{write}(\gamma_k)$ appears in $g$ of $\beta$.

We then have the following two lemmas to prove that S3 (i.e., S3.1 together with S3.2) is indeed sufficient for C3. Lemma 6.2.2 states that in any state $s$ of $\hat{M}$ there is no disabled action $\beta$ of the process which actions are taken as $\text{ample}(s)$ that may be enabled by actions of other processes, that may be executed later. With disabled actions we mean actions $\beta = (\ell_i \xrightarrow{g_i : \nu} \nu)$ for which there is a control edge in the PCG for the current location $\ell_i$ of $P_i$, but for which the guards are not satisfied by the current variable evaluation. Lemma 6.2.3 then shows, that S3 implies C3.
Lemma 6.2.2. Suppose that S3.2 holds. Then, for each path \( s = s_0 \xrightarrow{\gamma_1} s_1 \xrightarrow{\gamma_2} \cdots \xrightarrow{\gamma_m} s_m \) in \( \mathcal{M} \) where \( \gamma_1, \ldots, \gamma_m \notin \text{Act}_i \) we have \( \beta \notin \text{Act}_i \) or \( \beta \in \text{Act}_i(s) \).

Proof. For \( h = 0, 1, \ldots, m \), let \( \ell_{h, i} \) be the location of the \( i \)th process in the global state \( s_h \). Then, state \( s_h \) has the form

\[
s_h = \langle \ell_{h,1}, \ldots, \ell_{h,n}, \eta_h \rangle.
\]

Since \( \gamma_1, \ldots, \gamma_m \) do not belong to \( \text{Act}_i \), we have

\[
\ell_{1,i} = \ell_{2,i} = \cdots = \ell_{m,i}
\]

Since \( \beta \in \text{Act}_i \) is enabled in state \( s_m \) there exists a control transition

\[
\beta = (\ell_{m,i} \xrightarrow{g} \nu)
\]

in \( \mathcal{P}_i \). Since \( \ell_{1,i} = \ell_{m,i} \) and \( \gamma_m \notin \text{Act}_i(s) \) the variable valuation \( \eta_0 \) of state \( s = s_0 \) does not fulfil the guard of action \( \beta \). But as \( \beta \) is enabled in state \( s_m \), the variable valuation \( \eta_m \) of state \( s_m \) meets the guard of \( \beta \). Hence, at least one of the actions \( \gamma_1, \ldots, \gamma_m \), say \( \gamma_k \), must modify the value of a variable \( \nu \) that appears in the guard of \( \beta \). Let \( \gamma_k \in \text{Act}_j \). We then have \( x \in \text{write}(\gamma_k) \) and \( x \) appears in guard \( g \) of \( \beta \), and

\[
\ell_{j,0} \xrightarrow{g} \ell_{j,k-1} \quad \text{and} \quad \gamma_k = (\ell_{j,k-1} \xrightarrow{g} \ell_{j,k})
\]

This contradicts condition S3.2.

\[ \square \]

Lemma 6.2.3. If S3.1 and S3.2 hold then C3 holds for \( \text{ample}(s) = \text{Act}_i(s) \).

Proof. Suppose by contradiction that C3 is violated, while S3.1 and S3.2 hold. Then, there exists a path

\[
s = s_0 \xrightarrow{\gamma_1} s_1 \xrightarrow{\gamma_2} \cdots \xrightarrow{\gamma_m} s_m
\]

in \( \mathcal{M} \) and an action \( \alpha \) is in \( \text{ample}(s) = \text{Act}_i(s) \) such that \( \alpha \not\subseteq \gamma \) and \( \{\gamma_1, \ldots, \gamma_m\} \cap \text{Act}_i(s) = \emptyset \). This yields \( \gamma_m \in \text{Act}_i \) by S3.1. Let \( k \in \{1, \ldots, m\} \) be the smallest index such that \( \gamma_k \in \text{Act}_i \). But this yields \( \gamma_k \in \text{Act}_i(s) \) by Lemma 6.2.2. This contradicts the assumption that \( \{\gamma_1, \ldots, \gamma_m\} \cap \text{Act}_i(s) = \emptyset \).

\[ \square \]

The criteria S3.1 and S3.2 depend on requirements on actual control paths in the PCGs involved in the building of the MDP \( \mathcal{M} \) and \( \hat{\mathcal{M}} \) respectively. Of course, in practice we just process the set of control states in the PCG and their corresponding edges rather than actual control paths. So for S3.1 we check whether \( \alpha \not\subseteq \gamma \) for all \( \alpha \in \text{Act}_i(s) \), where \( \gamma = (l \xrightarrow{g} l') \) such that \( \ell \xrightarrow{g} l \) in \( \mathcal{P}_i \). For condition S3.2 we require that whenever \( \beta = (\ell \xrightarrow{g} \nu) \) is a control transition in \( \mathcal{P}_i \) such that \( \beta \not\subseteq \text{Act}_i(s) \) then for all \( j \in \{1, \ldots, n\} \setminus \{i\} \) and all
locations $l \in \text{Loc}_j$ such that $\ell_j \sim^*_j l$ and all control transitions $\gamma = (l \overset{R\alpha_j}{\rightarrow} l')$ there is no variable $x \in \text{write}(\gamma)$ that appears $g$ of $\beta$.

From this it follows that a practical realization of a treatment of (a tighter) condition $C3$ can be done by investigating PCGs instead of the MDP $\mathcal{M}$.

**Stronger conditions $S4$, $S4'$ and $S4''$ sufficient for $C4$ and realizations $R4-1$ to $R4-3$:**

Let us first observe that if given the reduced MDP then $C4$ can be checked in time polynomial in the size of $\hat{\mathcal{M}}$ as follows. For each action $\alpha$ we compute the maximal end components of $\hat{\mathcal{M}}^\alpha$ which results from $\mathcal{M}$ by considering the states where $\alpha \in \text{Act}(s) \setminus \text{ample}(s)$ only. Then, $C4$ holds if and only if none of the MDPs $\hat{\mathcal{M}}^\alpha$ has a proper maximal end component. We now establish PTIME-hardness of the problem to check whether $C4$ holds.

**Lemma 6.2.4.** The problem

"given a PROBMela-program $\mathcal{P}$ and an atomic proposition $a$ that only refers to variables of $\mathcal{P}$, check whether some state $s$ with $s \models a$ belongs to a proper end component of $\mathcal{M}_P"$

is PTIME-hard.

**Proof.** We establish a log-space reduction to the problem monotone circuit value which is known to be complete for PTIME [Pap94]. Monotone circuit value asks whether for a given Boolean circuit $C$ with and- and or-gates and truth values as inputs the value of its output node is true. The circuit is formalized by an acyclic finite directed graph where the and- and or-nodes have in-degree 2, the input nodes have no predecessors. The output node has exactly one incoming edge, has no successors and is reachable from all nodes. For the log-space reduction, we construct a PROBMela-program $\mathcal{P}$ with just one PCG $\Psi$ and a single boolean variable $x$. We deal with five actions $\tau_1$, $\tau_2$, $\tau$ (which have the trivial guard “true” and do not affect the value of $x$), set (which has the trivial guard “true” and sets $x$ to 1), and reset (which has the guard $x = 1$ and resets $x$ to 0). We treat the nodes in $C$ as locations in $\Psi$. The control transitions arise by reversing all edges in $C$ (which yields the graph $C^{-1}$). The outgoing edges of an or-node in $C^{-1}$ are treated as nondeterministic choices, i.e., if $\ell$ is an or-node in $C$ and $\ell_1, \ell_2$ are the predecessors of $\ell$ in $C$ then we have two control transitions

\[ \ell \overset{\tau_j}{\rightarrow} \mu_1 \quad \text{and} \quad \ell \overset{\tau_j}{\rightarrow} \mu_2 \quad \text{where} \quad \mu_i(\ell_i) = 1. \]

The and-nodes just have one transition where both successor can be reached with equal probability. I.e., if $\ell$ is an and-node in $C$ and $\ell_1, \ell_2$ are the predecessors of $\ell$ in $C$ then we have one control transitions
\[ \ell \overset{\tau}{\to} \mu \quad \text{where } \mu(\ell_1) = \mu(\ell_2) = \frac{1}{2}. \]

If \( \ell \to \ell_{\text{out}} \) is the unique edge in \( C \) that leads to the output node then \( \ell_{\text{out}} \sim \ell \). The input nodes of \( C \) representing the truth value “false” are terminal in \( P \). The input nodes of \( C \) labeled by “true” can loop to the output node, via resetting \( x \) to 0, i.e., if \( \ell \) is an input node with value “true” then

\[ \ell \overset{\text{reset}}{\sim} \mu_{\text{out}} \quad \text{where } \mu_{\text{out}}(\ell_{\text{out}}) = 1. \]

The initial variable condition is \( x = 0 \) and atomic proposition \( a \) stands for the requirement “\( x = 1 \)”.

It is now easy to see that the only reachable \( a \)-state \( (\ell_{\text{out}}, x = 1) \) belongs to a proper end component if and only if the value of \( C \) is “true”.

In order to realize algorithms for checking \( C_4 \) that can be used in an on-the-fly setting, we use a stronger condition \( S_4 \), that relies on the concept of cycles rather than end components and exploit the fact that cycles are much easier to handle in a DFS based setting. This agrees with the corresponding condition proposed in [dN04] for the branching time case and with the cycle condition from the nondeterministic setting.

### Condition \( S_4 \) sufficient for \( C_4 \):

For each cycle \( s_0 \overset{\alpha_1}{\sim} s_1 \overset{\alpha_2}{\sim} \ldots \overset{\alpha_k}{\sim} s_k = s_0 \) in the reduced MDP \( \hat{M} \) it holds that

\[
\bigcap_{1 \leq i \leq k} \text{Act}(s_i) \subseteq \bigcup_{1 \leq i \leq k} \text{ample}(s_i)
\]

It is clear that \( S_4 \) is a sufficient criterion for \( C_4 \), since each end component contains at least one cycle. \( S_4 \) forms the conceptual basis for three different realizations \( R_4-1 \)-\( R_4-3 \) that are presented in the following. All these realizations meet stronger variants of \( S_4 \) each. The first two realizations \( S_4.1 \) and \( S_4.2 \) realize condition \( S_4" \) which ignores information about enabled actions and only aims at ensuring that on every cycle in \( \hat{M} \) there is at least one state \( s \) which is fully expanded, thus ensuring that \( C_4 \) is not violated.

### Condition \( S_4" \) sufficient for \( S_4 \):

For each cycle \( s_0 \overset{\alpha_1}{\sim} s_1 \overset{\alpha_2}{\sim} \ldots \overset{\alpha_k}{\sim} s_k = s_0 \) in the reduced MDP \( \hat{M} \) it holds that

\[
\exists i \in \{1, ..., k\} \text{ such that } \text{ample}(s_i) = \text{Act}(s_i)
\]

The afterwards discussed realization \( S_4-3 \) ensures a condition \( S_4' \) that is stronger than \( S_4 \) but weaker than \( S_4" \). It also takes into account information about enabled actions on the
cycles by ensuring a fully expanded state on only those cycles in $\hat{M}$ for which there is an action that is enabled in every state.

**Condition S4' sufficient for S4:**

For each cycle $s_0 \overset{\alpha_1}{\leadsto} s_1 \overset{\alpha_2}{\leadsto} \ldots \overset{\alpha_k}{\leadsto} s_k = s_0$ in the reduced MDP $\hat{M}$ it holds that

$$\bigcap_{1 \leq i \leq k} \text{Act}(s_i) \neq \emptyset,$$

then $\exists i \in \{1, \ldots, k\}$ such that $\text{ample}(s_i) = \text{Act}(s_i)$

**Realizations R4-1 - R4-3:** For the presented methods we assume a DFS based setting in which the state space is built up. Since we have to identify backward edges during the DFS, the implementation of the DFS stack must be equipped with an operation for checking whether a rediscovered state, i.e., a state, that was already visited before, lies on the stack. It is then the case that the edge to this rediscovered state is a backward edge.

**Realization R4-1 (full expansion when visiting backward edges):** Assuming a DFS based state space search, a simple and practical on-the-fly approach to meet condition S4’’ is to fully expand a state $t$, whenever a backward edge $t \rightarrow s$ is encountered, where a full expansion means, that $\text{ample}(t) = \text{Act}(t)$. The example in figure 6.4 features a path $s \overset{\beta_0}{\rightarrow} s_1 \overset{\beta_2}{\rightarrow} \ldots \overset{\beta_k}{\rightarrow} t \overset{\gamma}{\rightarrow} s_0$, that forms a cycle. Since $\gamma$ is a backward edge, state $t$ is expanded fully to include a potentially existing action $\alpha \in \bigcap_{0 \leq i \leq k} \text{Act}(s_i) \cap \text{Act}(t)$ that is enabled in all states on the cycle. Consequently, with this approach at least one state on every cycle becomes fully expanded and condition S4’’ is met. This approach is the method of choice implemented in the popular model checker SPIN for non probabilistic transition system $s$, and despite the fact that this approach appears to be quite straightforward, it shares the advantage of being a fast method that is easy to implement [PeL93].

If a cycle is closed from $s$ over $t$ to $s$ then state $t$ will be fully expanded

![Figure 6.4: Full expansion when encountering a backward edge](image)

**Realization R4-2 (refinement of R4-1):** Method R4-1 can be weakened to avoid some unnecessary full expansions while still meeting condition S4’’. Suppose that a state on the
cycle, e.g., state $s_1$, (see also figure 6.5) was already selected for full expansion (because of conditions other than $S4''/C4$), such that full expansion of state $t$ is unnecessary for meeting $S4''$, despite the fact that $\gamma$ is a backward edge. Such situations can be detected on-the-fly by means of the DFS visiting number. We use an additional data structure $D$ that stores all states $u$ that are currently in the DFS stack and that will be fully expanded (according to the implemented heuristics for C1-C5), i.e., for which it holds that $\text{ample}(u) = \text{Act}(u)$.

If the DFS finds a backward edge $t \rightarrow s$, emanating from some state $t$ and leading to state $s$, realization R4-2 checks whether $D$ contains some state $s'$ such that $s'$ has been visited after $s$ in the DFS. If such a state is not found, it follows that no cycle containing $t$ has been found so far and state $t$ will be fully expanded (and also inserted into $D$ since it is a state that is fully expanded). If such a state is found, then the backward edge from $t$ to $s$ will be ignored in the sense that it is not seen as a reason to expand $t$ fully. Otherwise, state $t$ remains in $D$ until the DFS for $t$ in which moment $t$ is removed from the DFS stack. The required operations for $D$ are just checking for emptiness and access to the elements of $D$ in the LIFO-principle. Hence, an efficient realization of $D$ is obtained by using a stack for this, too.

If a backward edge $t \rightarrow s$ is encountered, then we just have to compare the DFS visiting number of $s$ with the DFS visiting number of the top element $s'$ of $D$. For this, we denote by $\text{DFSNumber}(s')$ the DFS visiting number of a state $s'$. As $s'$ is an element of $D$, full expansion of state $s'$ has already been decided, i.e., $\text{ample}(s') = \text{Act}(s')$. We have to distinguish two cases:

**Case 1 (full expansion not necessary):** Suppose that $\text{DFSNumber}(s) \leq \text{DFSNumber}(s')$.

State $s'$ must be an element of the DFS stack, i.e., the DFS call for $s'$ is still running, in the moment where the backward edge $t \rightarrow s$ has been found. (Note that otherwise $s'$ would have been removed from $D$ just before the DFS call for $s'$ has terminated.) But then, the discovered cycle has the form $s \rightarrow \ldots \rightarrow s' \rightarrow \ldots \rightarrow t \rightarrow s$.

As $\text{ample}(s') = \text{Act}(s')$, full expansion of $t$ is not necessary for satisfying condition $S4''$. Realization R4-2 therefore ignores the backward edge $t \rightarrow s$ with respect to full expansion.

**Case 2 (full expansion of state $t$):** If $\text{DFSNumber}(s) > \text{DFSNumber}(s')$, then $s'$ and all other states stored in $D$ have been visited before $s$. The just discovered cycle $s \rightarrow \ldots \rightarrow t \rightarrow s$ consists of states that have been visited during the ongoing DFS call for state $s$. Hence, none of the states in the discovered cycle $s \rightarrow \ldots \rightarrow t \rightarrow s$ is currently stored in $D$. Thus, in the moment where the backward edge $t \rightarrow s$ has been found, for all states along the cycle $s \rightarrow \ldots \rightarrow t \rightarrow s$ there is not yet a definite decision for full expansion. In this case we do full expansion of state $t$ to ensure that
S4” holds.

In the example in figure 6.5 (top), when encountering backward edge \( t \xrightarrow{\gamma} s \), the last state that is selected for full expansion and stored in data structure \( D \) is state \( s_1 \). Since \( \text{DFSNumber}(s) \leq \text{DFSNumber}(s_1) \), we know that state \( s_1 \) is on the cycle \( s \rightarrow s_1 \rightarrow ... \rightarrow s_k \rightarrow s \) and that it was already selected for full expansion. According to case 1 above, selecting state \( t \) for full expansion is not necessary. In figure 6.5 (bottom), \( s_1 \) is again the last state that was selected for full expansion and stored in \( D \) when backward edge \( t \xrightarrow{\gamma} s_i \) is encountered. Since \( \text{DFSNumber}(s_i) > \text{DFSNumber}(s_1) \) there is no state on the cycle \( s_k \rightarrow ... \rightarrow t \rightarrow s_k \) that was selected for full expansion yet. According to case 2 above we select therefore state \( t \) for being fully expanded to meet condition S4”. This method performs very good in practice.

Case 1: backward edge leads from \( t \) to \( s \) with fully expanded state \( s_1 \) is on the cycle ⇒ no full expansion

Case 2: backward edge leads from \( t \) to \( s \) with fully expanded state \( s_1 \) off the cycle ⇒ ample \( (t) = \text{Act}(t) \)

Figure 6.5: Full expansion depends on data structure \( D \) (before / after last full expansion)

Realization S4-3 (action-wise expansion on backward edges): Realizations R4-1 and R4-2 both meet the stronger condition S4”, but lack exploiting any information about the enabledness of particular actions \( \alpha \in \text{Act}(s) \) for a state \( s \) on a cycle, in contrast to what conditions C4 and S4 are suggesting. We now focus on criterion S4’, which is weaker than S4” since it does not require the existence of fully expanded states on a cycle unless there is an action which is indeed enabled continuously.

In what follows we explain an algorithmic schema that works for one ”selected” action. After that, the general schema is explained. Consider the example in figure 6.6. Here a full expansion of state \( t \) is not necessary because \( \alpha \notin \text{Act}(s_1) \) and hence \( \alpha \notin \bigcap_{t \in \text{cycle}} \text{Act}(t) \). We first focus on one action and explain the general case for arbitrarily many actions afterwards. We denote an action \( \alpha \) suppressed, if and only if \( \alpha \in \text{Act}(s) \setminus \text{ample}(s) \). A fresh data
structure $D_\alpha$ is employed and used in a similar fashion as data structure $D$ in method S4.2 to store the states $s$ on the DFS stack, where $\alpha$ was not suppressed. This means, that when state $s$ is stored on the DFS stack, it is also stored in $D_\alpha$ if $\alpha \notin \text{Act}(s)$ or $\alpha \in \text{ample}(s)$.

Only case 1 is shown: backward edge from $t$ to $s$ closes cycle, but $\alpha$ is not enabled in every state of the cycle (here: in $s_1$) ⇒ no full expansion necessary

Figure 6.6: Full expansion not necessary because $\alpha$ is not enabled in state $s_1$

Then $D_\alpha$ is used as follows: If the DFS finds a backward edge $t \rightarrow s$, emanating from state $t$ and pointing to state $s$, we check the top element $s'_\alpha$ of $D_\alpha$ and distinguish two cases:

**Case 1 (full expansion not necessary):** If DFSNumber$(s) \leq$ DFSNumber$(s'_\alpha)$, then state $s'_\alpha$ is an element of the DFS stack. It follows that the discovered cycle has the form $s \rightarrow \ldots \rightarrow s'_\alpha \rightarrow \ldots \rightarrow t \rightarrow s$. Since $\alpha$ was not suppressed in $s'_\alpha$ full expansion of $t$ is not necessary (see example in figure 6.6).

**Case 2 (full expansion of state $t$):** If DFSNumber$(s) >$ DFSNumber$(s'_\alpha)$, then $s'_\alpha$ and all other states stored in $D_\alpha$ have been visited before $s$. The cycle $s \rightarrow \ldots \rightarrow t \rightarrow s$ consists of states that have been visited during the ongoing DFS call for state $s$. Hence, none of the states in the discovered cycle $s \rightarrow \ldots \rightarrow t \rightarrow s$ is currently stored in $D_\alpha$. Thus, when the backward edge $t \rightarrow s$ has been found, action $\alpha$ was suppressed in each state on the cycle. In this case, $t$ becomes fully expanded and S4' holds.

Like in method 4.2 with data structure $D$, any state that is removed from the DFS stack is also immediately removed from $D_\alpha$.

As mentioned above, for the sake of a simpler presentation the explanations above only cover one data structure $D_\alpha$ for one action $\alpha$ that may occur on the cycle. For the general case of arbitrary actions, data structures $D_\gamma$ have to be maintained for every possible action $\gamma \in \text{Act of } \mathcal{M}$. This means that for every control edge of every PCG / process that is involved such a data structure must be available. Please note that it is not sufficient to just push actions $\text{Act}(s) \setminus \text{ample}(s)$ for states $s$ on a potential cycle. Actions $\alpha \notin \text{Act}(s)$ count also as "not suppressed" in state $s$. Then, with data structures $D_\gamma$ for all $\gamma \in \text{Act of}
6.2. Efficient calculation of ample sets

Chapter 6. Partial order reduction of Markov decision processes

\( \mathcal{M} \), case 1 in our distinction of cases occurs only if it holds for all data structures \( D_\gamma \) that \( \text{DFSNumber}(s) < s'_\gamma \).

However, maintaining a separate data structure \( D_\gamma \) for each action \( \gamma \in \text{Act} \) leads to considerable overhead, but the situation can be improved significantly if a mixed scenario, where data structures \( D_\gamma \) and the single data structure \( D \) from method 4.2 are used simultaneously. Then, the insertion process for \( D \) and \( D_\gamma \) is organized as follows: If a state \( s \) in fully expanded (due to possibly other criteria or for meeting \( S4 \)), then when \( s \) is pushed on the DFS stack \( s \) is stored only in \( D \) and the data structures \( D_\gamma \) are not used. Only if \( \text{ample}(s) \neq \text{Act}(s) \), data structures \( D_\gamma \) are used store all actions that are not suppressed.

Then, if the DFS finds a backward edge \( t \rightarrow s \), emanating from state \( t \) to state \( s \), we check the top elements \( s'_\alpha \) of \( D_\alpha \) and \( s' \) of \( D \) as follows:

**Case 1 (full expansion not necessary):** This case is identical to case 1 of realization \( R4-2 \), i.e., \( \text{DFSNumber}(s) \leq \text{DFSNumber}(s') \) and a full expansion is not necessary.

**Case 2:** If \( \text{DFSNumber}(s) > \text{DFSNumber}(s') \), then we consider data structures \( D_\gamma \). We distinguish two subcases:

**Subcase 2.1 (full expansion not necessary):** Let \( s'_\gamma \) be the top element of \( D_\gamma \). If \( \text{DFSNumber}(s) \leq \text{DFSNumber}(s'_\gamma) \) for all \( \gamma \in \text{Act} \), then for each action \( \gamma \in \text{Act} \), the discovered cycle has the form \( s \rightarrow \ldots \rightarrow s'_\gamma \rightarrow \ldots \rightarrow t \rightarrow s \).

Since no action was suppressed in every state on the cycle, full expansion of \( t \) is not necessary.

**Subcase 2.2 (full expansion of state \( t \)):** If \( \text{DFSNumber}(s) > \text{DFSNumber}(s'_\alpha) \) for at least one action \( \alpha \in \text{Act} \) then none of the states in the discovered cycle \( s \rightarrow \ldots \rightarrow t \rightarrow s \) is currently stored in \( D_\alpha \), action \( \alpha \) was suppressed in each state on the cycle. In this case, \( t \) becomes fully expanded and \( S4' \) holds.

**Sufficient criteria \( S5 \) and \( S5' \) for \( C5 \):** As explained above, if \( \text{Act}(s) = \{ \alpha, \beta, \gamma \} \), where \( \alpha \) is a probabilistic action, then the stochastic experiments induced by ”execute \( \alpha \) and the select \( \beta \) or \( \gamma \) for execution” or ”select \( \beta \) or \( \gamma \) and then execute \( \alpha \)” might be different even when \( \alpha \prec \beta \) and \( \alpha \prec \gamma \) (see \[BGC04\]). In such cases it is necessary to enforce that either \( |\text{ample}(s)| = 1 \) or \( \text{ample}(s) = \text{Act}(s) \).

The problem of deciding this condition in practice shares similarities with condition \( S3 \).

Both conditions are global conditions of \( \mathcal{M} \) that depend on the reachability of states that execute may execute dependent actions in case of \( C3 \) or probabilistic actions in case of \( C5 \). As it is the case with \( C3 \), checking \( C5 \) is at least as hard as deciding reachability in \( \mathcal{M} \):
Lemma 6.2.5. Checking condition C₅ is at least as hard as checking whether a state s is reachable via a path in M starting in s₀ with some specified labeling a ∈ L(s), or, written in standard CTL notation, checking whether the property s₀ |= ∃a holds in M.

Proof. We show that if we are given an algorithm A for checking whether C₅ holds with time complexity T(N), where N is the total size of the PCGs of the given program, then there is an algorithm A’ for the above form of the reachability problem with time complexity T’(N) = O(max{T(N), N}). Since N is an obvious lower bound for the reachability problem, this yields the statement of the lemma.

We adapt the reduction of the reachability problem to the problem of checking whether C₃ is violated, presented in [CGP99], for condition C₅.

Let (P, s₀, a) be an instance of the reachability problem. Since for the reachability problem s₀ |= ∃a the probabilistic choices are irrelevant and can be replaced with nondeterministic choices, we can assume that all actions in the PCGs of P are deterministic. Note that there is a one-to-one correspondence between the transitions in M and control edges in the PCGs of P. Then, all transitions in M yield a unique successor state, i.e., have the form s −→ µ where µ(t) = 1 for some state t in S.

Let α₀, α₁ and αₚrob ∈ Act be three fresh actions and x ∈ Var a fresh boolean variable. We assume that α₀ and α₁ are always executable and do not affect the values of the variables (i.e., Effect(α, η)(η) = 1 for all η ∈ Eval(Var)). The guard of action αₚrob is given by the atomic proposition a such that prob is a (probabilistic) action that is enabled exactly in the states where a holds. The effect of αₚrob is specified by the tossing a fair coin:

“if coin shows head then x becomes false, otherwise x becomes true”.

We now extend P to a new PROBMela-program P’ with additional processes Q₀, Q₁ and Qₚ. Their PCGs consist of a single location ℓ₀, ℓ₁ and ℓₚ, respectively, and the control transitions

α₀ = (ℓ₀ ~ ℓ₀),  α₁ = (ℓ₁ ~ ℓ₁),  αₚrob = (ℓₚ a: a ℓₚ),

where process event a has the semantics of tossing a coin and setting Boolean variable x as described above.

Since Q₀, Q₁, Qₚ just have one location each, these locations can be dropped from the tuples representing the states in the MDP M’ for P’. Thus, the states in M’ have the form ⟨s, x = 0⟩ or ⟨s, x = 1⟩ where s is a state in M = Mₚ. There are three types of transitions in the MDP M’ for P’:

1. M’ contains all transitions of M. More precisely, if s −→ t is a transition in M then M’ contains the transitions ⟨s, x = b⟩ −→ ⟨t, x = b⟩ where b ∈ {0, 1}. 

123
2. \( \mathcal{M}' \) has “nonprobabilistic self-loops” labeled by one the action \( \alpha_0 \) or \( \alpha_1 \) for all states, i.e., transitions \( \langle s, x = b \rangle \xrightarrow{\alpha_0} \langle s, x = b \rangle \) and \( \langle s, x = b \rangle \xrightarrow{\alpha_1} \langle s, x = b \rangle \) where \( b \in \{0, 1\} \).

3. If \( s \) is a state in \( \mathcal{M} \) where \( a \) holds then

\[
\langle s, x = b \rangle \xrightarrow{\alpha_{\text{prob}}} \mu
\]

where \( \mu(\langle s, x = 0 \rangle) = \mu(\langle s, x = 1 \rangle) = \frac{1}{2} \).

Thus, \( \alpha_{\text{prob}} \) is the only probabilistic action in \( \mathcal{M}' \). Furthermore, for each path \( s_0 \xrightarrow{\gamma_1} s_1 \xrightarrow{\gamma_2} \ldots \xrightarrow{\gamma_m} s_m \) in \( \mathcal{M} \), \( \langle s_0, x = 0 \rangle \xrightarrow{\gamma_1} \langle s_1, x = 0 \rangle \xrightarrow{\gamma_2} \ldots \xrightarrow{\gamma_m} \langle s_m, x = 0 \rangle \) is a path in \( \mathcal{M}' \). Vice versa, given a path in \( \mathcal{M}' \) then we can drop all transitions with the label \( \alpha_0, \alpha_1 \) or \( \alpha_{\text{prob}} \) and the additional component for variable \( x \) to obtain a path in \( \mathcal{M} \).

Obviously, given \( \mathcal{P}, s_0 \) and \( a \), the PROBMela-program \( \mathcal{P}' \) can be derived in time linear in the total size of the PCGs of \( \mathcal{P} \). Moreover, we have:

\[
\text{ample}(\langle s_0, x = 0 \rangle) = \{\alpha_0, \alpha_1\} \quad \text{for the ample set for state } \langle s_0, x = 0 \rangle \text{ in } \mathcal{M}' \text{ satisfies condition C5 if and only if } s_0 \not\models \exists \diamond a \text{ in } \mathcal{M}.
\]

“\( \Rightarrow \)”:

Suppose that \( \text{ample}(\langle s_0, x = 0 \rangle) = \{\alpha_0, \alpha_1\} \) satisfies C5. Then, state \( \langle s_0, x = 0 \rangle \) cannot reach a state in \( \mathcal{M}' \) where action \( \alpha_{\text{prob}} \) is enabled. Hence, state \( s_0 \) in \( \mathcal{M} \) cannot reach a state where \( a \) holds. Thus, \( s_0 \not\models \exists \diamond a \) in \( \mathcal{M} \).

“\( \Leftarrow \)”:

Suppose that \( s_0 \models \exists \diamond a \) in \( \mathcal{M} \). Let \( s_0 \xrightarrow{\gamma_1} s_1 \xrightarrow{\gamma_2} \ldots \xrightarrow{\gamma_m} s_m \) be a shortest path in \( \mathcal{M} \) such that \( s_m \models a \). Then:

\[
\langle s_0, x = 0 \rangle \xrightarrow{\gamma_1} \ldots \xrightarrow{\gamma_m} \langle s_m, x = 0 \rangle \xrightarrow{\beta} \langle s_m, x = 0 \rangle
\]

is a path in \( \mathcal{M}' \) where \( \gamma_1, \ldots, \gamma_m, \beta \notin \text{ample}(\langle s_0, x = 0 \rangle) \) and \( \beta \) is probabilistic. As \( \text{ample}(\langle s_0, x = 0 \rangle) \) is not a singleton, condition C5 does not hold. \( \Box \)

Since calculating condition C5 optimally is too hard, we now proceed with sufficient weaker criteria for C5, that can be checked efficiently in an on-the-fly DFS-based setting. As in the discussion of realizations for C3 we focus on a stronger condition that argues about sequences of control edges \( Q_0 \overset{\gamma_1; a_1}{\sim} Q_1 \overset{\gamma_2; a_2}{\sim} \ldots \overset{\gamma_m; a_m}{\sim} Q_n \) and their associated actions \( a_i \text{Act} \) in the MDP, i.e., about sequences of the form:

\[
Q_0 \overset{\gamma_1; a_1}{\sim} Q_1 \overset{\gamma_2; a_2}{\sim} \ldots \overset{\gamma_m; a_m}{\sim} Q_n.
\]
The following criterion $S5$ is sufficient for condition $C5$:

**Condition $S5$ sufficient for $C5$:**

Let $ample(s) = Act_i(s)$. If $\alpha \in ample(s)$, then the following holds:

For all $j \in \{1, \ldots, n\} \setminus \{i\}$ and all control paths

$$\ell_j = l_0 \xrightarrow{g_1} j_1 \xrightarrow{g_2} j_2 \cdots \xrightarrow{g_m} j_m \xrightarrow{g_{m+1}} l'$$

where $\beta$ is a probabilistic action, then $|Act_i(s)| = 1$.

Similar to the situation with condition $S3$ and $S3'$ criterion $S5$ depends on requirements on control paths in the PCGs. In practice we just process the set of control states in the PCG and their corresponding edges rather than actual control paths. For $S5$ we require that $|Act(s)| > 1$ only if for all $j \in \{1, \ldots, n\} \setminus \{i\}$ and all locations $l \in Loc_j$ such that $\ell_j \xrightarrow{g} l$ and all control transitions $\gamma = (l \xrightarrow{g} j \xrightarrow{g} l')$ it holds that $\gamma$ is not probabilistic action.

Under certain circumstances, it can be more efficient to just chose the ample sets such that $|ample(s)| = 1$ regardless of the actions in the PCGs or in $M$, such that a processing of PCGs can be omitted completely. We denote this simple variant of realization as $S5'$.

### 6.3. Experimental results

The following experimental results show the different potential of the proposed realizations $S4-1 \cdot S4-3$ for condition $C4$. For this their performance with different probabilistic parallel models (see page 68).

<table>
<thead>
<tr>
<th></th>
<th>states</th>
<th>transitions</th>
<th>time</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>LE(6)</td>
<td>$1.1 \cdot 10^7$</td>
<td>$6.2 \cdot 10^7$</td>
<td>813 s</td>
<td>1.9 GB</td>
</tr>
<tr>
<td>DP(5)</td>
<td>$2.2 \cdot 10^6$</td>
<td>$1.4 \cdot 10^7$</td>
<td>126 s</td>
<td>490 MB</td>
</tr>
<tr>
<td>DC(5)</td>
<td>$1.64 \cdot 10^6$</td>
<td>$6.32 \cdot 10^6$</td>
<td>66 s</td>
<td>302 MB</td>
</tr>
<tr>
<td>RG(4)</td>
<td>488902</td>
<td>661307</td>
<td>4.4 s</td>
<td>49 MB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>states</th>
<th>transitions</th>
<th>time</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>LE(6)</td>
<td>$3.7 \cdot 10^6$</td>
<td>$9.4 \cdot 10^6$</td>
<td>164 s</td>
<td>504 MB</td>
</tr>
<tr>
<td>DP(5)</td>
<td>$1.3 \cdot 10^6$</td>
<td>$3.9 \cdot 10^6$</td>
<td>86 s</td>
<td>223 MB</td>
</tr>
<tr>
<td>DC(5)</td>
<td>701254</td>
<td>$1.39 \cdot 10^6$</td>
<td>11 s</td>
<td>109 MB</td>
</tr>
<tr>
<td>RG(4)</td>
<td>4424</td>
<td>5380</td>
<td>1.3 s</td>
<td>15 MB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>states</th>
<th>transitions</th>
<th>time</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>LE(6)</td>
<td>$3.6 \cdot 10^6$</td>
<td>$8.5 \cdot 10^6$</td>
<td>192 s</td>
<td>473 MB</td>
</tr>
<tr>
<td>DP(5)</td>
<td>$1.3 \cdot 10^6$</td>
<td>$3.5 \cdot 10^6$</td>
<td>113 s</td>
<td>217 MB</td>
</tr>
<tr>
<td>DC(5)</td>
<td>691360</td>
<td>$1.33 \cdot 10^6$</td>
<td>16 s</td>
<td>107 MB</td>
</tr>
<tr>
<td>RG(4)</td>
<td>4424</td>
<td>5380</td>
<td>1.3 s</td>
<td>15 MB</td>
</tr>
</tbody>
</table>

Table 6.2: Comparison of different realizations of the end-component condition $C4$. 

125
The results suggest that realization $R4-2$ brings a significant improvement compared to $R4-1$. However, the reduction potential of realization $R4-3$ seems to be more dependent on the model’s structure. This realization requires that if we have $|\text{ample}(s) < |\text{Act}(s)|$ during the system construction, then all actions that are not suppressed need to be considered for a stack operation. This includes actions $\alpha \in \text{Act}\backslash \text{Act}(s)$. Consequently, since in systems with a high grade of reduction this happens more often, high reduction slows down the process. In such cases the additional effort seems to lead to a considerable increase of reduction not in all parallel models an if so with a comparably high time-payoff.

The following results show the effect of the partial order reduction on the whole process of model checking, which includes different combinations of priority-based value iteration approach and the MEC-quotient approach (see chapter 4).

### 6.4. Partial order reduction and fairness

In this section we explain how to treat partial order reduction for MDPs and stutter-invariant LT-properties together with strong and weak fairness conditions (see chapter 5). Theorem 6.4.2 shows that stutter trace equivalence between two MDPs $\mathcal{M}$ and $\hat{\mathcal{M}}$ is compatible with strong and weak fairness conditions. Hence, an implementation of the partial order reduction based on any criteria that ensure $C1$-$C5$ (which ensures stutter trace equivalence) is also adequate for reasoning about extremal probabilities for stutter-invariant LT properties in the presence of strong or weak process fairness constraints. Furthermore, we again address the particular notion of process fairness, as it is a very natural form of fairness in parallel process based settings and avoids at the same time a check for realizability (see chapter 5). This particular variant of fairness ensures a fair treatment of process execution in the interleaved setting in the sense, that any process, that is executable infinitely often (strong process fairness) respectively continuously from some point in time on (weak process fairness) eventually gets executed. We show that the fairness conditions $C1$-$C5$ (see chapter 5) are too weak for strong process fairness. However, the stronger conditions given in chapter 6.2 are compatible with strong and weak process fairness, as we will see in the following.

**Definition 6.4.1. [Stutter-equivalent MDPs]** Given two MDPs $\mathcal{M}_1$ and $\mathcal{M}_2$ with the same set of atomic propositions $AP$, then $\mathcal{M}_1$ and $\mathcal{M}_2$ are called **stutter-equivalent**, denoted $\mathcal{M}_1 \equiv_{st} \mathcal{M}_2$, if for each scheduler $U_1$ of $\mathcal{M}_1$ there exists a scheduler $U_2$ of $\mathcal{M}_2$ such $\text{Pr}^{\mathcal{M}_1 U_1}(E) = \text{Pr}^{\mathcal{M}_2 U_2}(E)$ for all stutter-invariant measurable LT-properties $E \subseteq (2^{AP})^\omega$, and vice versa.

**Theorem 6.4.2. [BGC09b]** Let $\mathcal{M} = (S, \text{Act}, \rightarrow, s_0, AP, L)$ be an MDP, $\mathcal{F} = (WF, SF)$ a general fairness condition as above that uses only atomic propositions from $AP$. Then,
for any reduction technique that derives from the MDP \( \mathcal{M} \) a reduced MDP \( \hat{\mathcal{M}} \) such that \( \mathcal{M} \equiv_{\text{st}} \hat{\mathcal{M}} \) and for each stutter-invariant measurable LT property \( E \subseteq (2^{\text{AP}})^{\omega} \) it holds that

\[
\sup_{U \in \text{FairSched}^{\mathcal{M}}} \Pr^{\mathcal{M}}_U (E) = \sup_{U' \in \text{FairSched}^{\hat{\mathcal{M}}}} \Pr^{\hat{\mathcal{M}}}_{U'} (E)
\]

**Proof.** “\( \geq \)” is obvious as \( \hat{\mathcal{M}} \) is a sub-MDP of \( \mathcal{M} \). To show “\( \leq \)” let \( U \) be a fair scheduler of \( \mathcal{M} \). As \( \mathcal{M} \) and \( \hat{\mathcal{M}} \) are stutter-equivalent there exists a scheduler \( U' \) for \( \hat{\mathcal{M}} \) such that \( \Pr^{\mathcal{M}}_U (E) = \Pr^{\hat{\mathcal{M}}}_{U'} (E) \) for all stutter-invariants measurable properties \( E \subseteq (2^{\text{AP}})^{\omega} \).

Since \( \mathcal{F} \) is also a stutter-invariant LT-property, and \( \mathcal{F} \) is realizable, we have \( \Pr^{\hat{\mathcal{M}}}_{U'} (\mathcal{F}) = 1 \), Thus, \( U' \) is also a fair scheduler of \( \hat{\mathcal{M}} \).

Theorem 6.4.2 holds in the same way for the infimum instead of the supremum. It shows that with respect to strong or weak fairness conditions, the partial order reduction based on the ample set conditions from sections 6.1 and 6.2 is still applicable for \( r \) quantitative model checking against stutter-invariant LT properties under fairness assumptions. A disadvantage of this approach is, however, that the atomic propositions used in the general fairness condition \( \mathcal{F} = (\mathcal{W}, \mathcal{S}) \) add up to the atomic propositions that may contribute to non-stuttering actions, thus rendering them unavailable for partial order reduction because of violating condition \( C2 \). This may lead to a larger reduced MDP \( \hat{\mathcal{M}} \) than the one obtained for the analysis without fairness against the same formula. We now explain that for an important instance of fairness, namely process fairness, this disadvantage can be avoided.

**Process fairness.** Intuitively speaking, process fairness ensures that the parallel execution of processes \( P_1, ..., P_n \) is carried out in a fair way in the sense that if a process is executable infinitely often or continuously from some moment on, then it eventually gets executed. More precisely, we assume for each process \( P_i \) a fresh atomic proposition \( \text{Enabled}_i \) that serves to characterize the states where at least one action of process \( P_i \) (i.e., at least one action in \( \text{Act}_i \)) is enabled. The atomic propositions \( \text{Enabled}_i \) might not be contained in the original set \( \text{AP} \). Let \( \text{AP}' = \text{AP} \cup \{ \text{Enabled}_i : 1 \leq i \leq n \} \). The labeling function of the MDP \( \mathcal{M} \) is extended to a function \( \mathcal{L}' : S \to 2^{\text{AP}'} \) by \( \text{Enabled}_i \in \mathcal{L}'(s) \) if and only if \( \text{Act}_i \cap \text{Act}(s) \neq \emptyset \). Furthermore, we suppose that \( \mathcal{M} \) possesses a fairness condition \( \mathcal{F} = (\mathcal{W}, \mathcal{S}) \) that imposes strong or weak fairness for processes, in the following way: For all \( (I, J) \in \mathcal{S} \cup \mathcal{W} \) there is an index \( i \) such that \( I = \text{Enabled}_i \) and \( J = \text{Act}_i \), where \( \text{Enabled}_i \) denotes the set of all subsets \( \mathfrak{A} \) of \( \text{AP}' \) such that \( \text{Enabled}_i \in \mathfrak{A} \). Then, such a pair describes strong (resp. weak) fairness for one process \( P_i \) in the sense that if \( P_i \)'s actions are infinitely often (resp. continuously from some moment on) enabled, then \( P_i \) performs also infinitely many actions.

In contrast to strong fairness conditions that use only atomic propositions in \( \text{AP} \), we cannot hope in general that systems reduced according to the ample set conditions \( \mathbf{C1 - C5} \) are com-
patible with our notion of strong process fairness as presented above. Under strong process fairness, the extremal probabilities for stutter-invariant LT properties in the original MDP $\mathcal{M}$ and the reduced MDP $\hat{\mathcal{M}}$ may not agree, as the conditions are too weak. By applying the ample set conditions, there may be strongly process-fair probabilistic executions that are removed from $\mathcal{M}$, where their stutter-equivalent pendants, that remain in $\hat{\mathcal{M}}$, may very well be strongly unfair. The following counterexample illustrates this, for the sake of a simple argumentation, without using probabilistic actions.

**Example.** Consider the MDPs $\mathcal{M}$ and $\hat{\mathcal{M}}$ in figure 6.7, both equipped with the obvious labeling function $L : S \rightarrow \{\Box, \lozenge\}$ and the action set $\text{Act} = \{\alpha, \beta, \gamma, \delta, \epsilon\}$:

![Figure 6.7: Counterexample for ample-set method with process fairness](image)

We assume three processes $P_1$, $P_2$ and $P_3$ with action sets

$$\text{Act}_1 = \{\alpha, \epsilon\}, \text{Act}_2 = \{\beta\}, \text{Act}_3 = \{\gamma, \delta\}$$

and attach a strong process fairness condition

$$\mathcal{SF} = \{(\text{Enabled}_1, \text{Act}_1), (\text{Enabled}_2, \text{Act}_2), (\text{Enabled}_3, \text{Act}_3)\}.$$

For the ample set method we assume that $\alpha \land \gamma \beta$, that $\epsilon$ is independent of all other actions and that $\gamma$ and $\delta$ are dependent on all other actions. Thus, ample($s_0$) = $\{\beta\}$ is legal according to ample set conditions $C1-C5$, which leads to $\hat{\mathcal{M}}$.

Now, with the intuitive LTL notion of $\Box\lozenge\bullet$ for the LT-property "infinitely often black", observe that in $\mathcal{M}$ we have $\Pr_{s_0}^{\text{fairmax}}(\Box\lozenge\bullet) = 1$, since for example the action sequence $\alpha, \beta, \epsilon, \alpha, \beta, \epsilon, \ldots$ induces a fair path that satisfies the property. On the other hand, observe that $\mathcal{SF}$ is realizable in $\hat{\mathcal{M}}$ and that in $\hat{\mathcal{M}}$ no such fair path exists, since all fair paths must take the action $\gamma$ eventually to include process $P_3$, as required by the strong process fairness condition. Thus, in $\hat{\mathcal{M}}$ we have $\Pr_{s_0}^{\text{fairmax}}(\Box\lozenge\bullet) = 0$.

We now show that process fairness can, however, be used when the stronger conditions $S3$ $S4'$ and $SS'$ are employed instead of the (weaker) original ample-set conditions $C1-C5$. 

Theorem 6.4.3 (Process fairness with weaker conditions S4*, S5 and overapproximation of independence). Let $\mathcal{M} = (S, \text{Act}, \rightarrow, s_0, \text{AP}, L)$ be an MDP and $\mathcal{F} = (\mathcal{W}, \mathcal{F}, \mathcal{S})$ a process fairness condition as above, ample : $S \rightarrow 2^{\text{Act}}$ a function satisfying conditions S3, S4*, S5 and $E \subseteq (2^{\text{AP}})^\omega$ be a stutter-invariant measurable LT property. Then it holds that:

$$\sup_{U \in \text{FairSched}_{\mathcal{F}}^M} \mathbf{Pr}^{M, U}(E) = \sup_{U' \in \text{FairSched}_{\mathcal{F}}^{\hat{U}}(E)}$$

Proof. As in the proof of Theorem 6.4.2, it is sufficient to show direction “$\leq$”. Given a fair scheduler $U$ for $\mathcal{M}$, we apply the construction of [BGC04, Gro08] to obtain a corresponding scheduler $\hat{U}$ for $\hat{\mathcal{M}}$, such that $\Pr^{\mathcal{M}, U}(E) = \Pr^{\hat{\mathcal{M}}, \hat{U}}(E)$ for all stutter-invariant LT-properties $E$. And show, that $\hat{U}$ is also fair. The construction of $\hat{U}$ relies on an iterative approach where an infinite sequence $\mathcal{U} = U_0, U_1, U_2, \ldots$ of schedulers for $\mathcal{M}$ is constructed such that $U_0, \ldots, U_i$ agree on all finite paths of length $< i$. The scheduler $\hat{U}$ is then defined to be the "limit" of all schedulers $U_i$.

The transformations $U_i \sim U_{i+1}$ all rely on the same schema and we will sketch here the transformation for a single $U_0$-path into a set of stutter-equivalent $U_1$-paths. We will show that these $U_1$-paths are fair w.r.t. $\mathcal{F} = (\mathcal{W}, \mathcal{F}, \mathcal{S})$ if $\pi$ is fair w.r.t. $\mathcal{F} = (\mathcal{W}, \mathcal{F}, \mathcal{S})$. Given a $U_0$-path $\pi = s_0 \alpha_1 s_1 \alpha_2 s_2 \alpha_3 \ldots$ of $\mathcal{M}$, let $n$ be the smallest number such that $\alpha_n \in \text{ample}(s)$. If no $\alpha_i$ is an ample-action of $s_0$, let $n = \infty$.

Suppose first that $n$ is finite. If $n = 1$ (e.g., if ample($s_0$) = Act($s_0$)) then $\pi$ stays unchanged. If $n > 1$ then condition S3 and Lemma [6.2.1] ensure that $\alpha_n$ is enabled in each of the states $s_1, \ldots, s_{n-1}$. Indeed, assuming that $\{\alpha_n\} = \text{Act}_j(s_i)$, for $1 \leq i \leq n-1$, $\alpha_1 \not\in \text{Act}_j$ as PROBMela commands are executed consecutively. By S3 and Lemma [6.2.1] the execution of the actions $\alpha_1, \alpha_2, \ldots, \alpha_{n-1}$ does not change any variable that is relevant for the guard of $\alpha_n$. Hence, $\alpha_n$ remains enabled in the states $s_1, \ldots, s_{n-1}$. Furthermore, $\alpha_1$ is enabled in every $\alpha_n$-successor of $s_0$ and we can switch from the action sequence $\alpha_1 \alpha_2 \ldots \alpha_{n-1} \alpha_n$ to the action sequence $\alpha_n \alpha_1 \alpha_2 \ldots \alpha_{n-1}$. Both action sequences start in state $s_0$ and yield identical distributions over the states that can be reached, which is illustrated by figure [6.8].

We define $U_1$ such that $\pi_1 = s_0 \alpha_n t_0 \alpha_1 t_1 \ldots \alpha_{n-2} t_{n-2} \alpha_{n-1} s_n \alpha_n s_{n+1} \alpha_{n+2} \ldots$ is a $U_1$-path.

Let us now suppose that $n = \infty$. By similar arguments, we define $U_1$ such that $\pi_1 = s_0 \alpha t_0 \alpha t_1 \alpha \ldots$ is a $U_1$-path, where ample($s$) = $\{\alpha\}$. Moreover, $U_1$ is defined in such a way that almost surely every $U_1$-path originates from a $U_0$-path via such a transformation.

In summary, given a $\mathcal{U}$-path $\pi$ starting in state $s_0$, the basic idea is to permute the first ample action of $s$ that occurs along $\pi$ to the beginning of the action sequence of $\pi$. If no such action exists, an arbitrary ample action of $s$ is prepended to the action sequence of $\pi$. This step is then repeated ad infinitum to yield a scheduler $\hat{U}$ of $\hat{\mathcal{M}}$. 

129
Condition (S4") guarantees that each action of a $U_0$-path $\pi$ will almost surely be eventually executed by the “corresponding” $U$-path $\hat{\pi}$. We therefore get that almost surely

$$\inf(\pi) \cap \text{Act} \subseteq \inf(\hat{\pi}) \cap \text{Act}.$$  \hspace{1cm} (6.1)

Moreover, we observe that if $n \neq \infty$ in the above described transformation, then $\pi$ and $\pi_1$ have a common suffix. Thus:

$$\inf(\pi) = \inf(\pi_1).$$  \hspace{1cm} (6.2)

If $n = \infty$ and $\text{ample}(s_0) = \text{Act}_j(s_0) = \{\alpha\}$ then

$$\text{Act}_i(s_k) = \text{Act}_i(t_k), \text{ for } i \neq j \text{ and } k \geq 0.$$  \hspace{1cm} (6.3)

Indeed, conditions S3 and Lemma 6.2.1 ensure that the execution of $\alpha$ does not change the variables that are relevant for the execution of the actions $\beta \in \text{Act}_i$ for $i \neq j$. As each $t_k$ is an $\alpha$-successor of $s_k$, equation 6.3 follows.

Let $(I, J)$ be a fairness constraint stating fairness for process $P_\ell$, i.e., $K = \text{Act}_\ell$ and $H$ consists of all subsets $\mathcal{A}$ of $\mathcal{AP}'$ such that $\text{Enabled}_\ell \in \mathcal{A}$. Let $\pi = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \ldots$ be a path in $\mathcal{M}$. If $\inf(\pi) \cap K \neq \emptyset$ then equation (1) ensures that $\inf(\hat{\pi}) \cap K \neq \emptyset$. Now assume that $\inf(\pi) \cap K = \emptyset$ and $\inf(\pi) \cap H = \emptyset$. If in the above described transformation from $\pi$ to $\pi_1$, the index $n$ is equal to $\infty$ then the chosen ample action $\alpha$ is not in $\text{Act}_\ell$. Otherwise by S3 and Lemma 6.2.1 $\alpha$ would be enabled in each state $s_k$ for $k \geq 0$ (see the above picture), but then $\inf(\pi) \cap H \neq \emptyset$ which contradicts our assumption. Hence, if $n = \infty$, the chosen ample action is in some $\text{Act}_j$ with $j \neq \ell$ and equation 6.3 ensures that $\text{Act}_\ell(s_k) = \text{Act}_\ell(t_k)$ for $k \geq 0$.

Altogether, equations 6.2 and 6.3 ensure that $(\inf(\pi) \cap K = \emptyset \land \inf(\pi) \cap H = \emptyset)$ implies that $\inf(\pi_1) \cap H = \emptyset$. Condition S4" then ensures that almost surely $U$-paths $\pi$ with $\pi \vdash_{\text{strong}} (I, J)$ are transformed into $\hat{U}$-paths $\hat{\pi}$ with $\hat{\pi} \vdash_{\text{strong}} (I, J)$. With similar
arguments, we get that $U$-paths $\pi$ with $\pi \vdash \text{weak} (I, J)$ transformed into $\hat{U}$-paths $\hat{\pi}$ with $\hat{\pi} \vdash \text{weak} (I, J)$. We conclude that for a fair scheduler $U$ of $\mathcal{M}$, the scheduler $\hat{U}$ of $\hat{\mathcal{M}}$ is also fair.

Note that in Theorem 6.4.3 the auxiliary atomic propositions $\text{Enabled}_i$ were only needed to formalize the fairness constraints, but they do not have to be taken into account for the construction of the reduced MDP, i.e., they are not used in the definition of stutter actions.
We conclude this thesis with the following remarks on the different parts.

**Modelling probabilistic nondeterministic systems.** In chapter 3, we presented an expressive input language for probabilistic reactive models. This language, which we called PROBMela, is suited in a parallel process-based setting and inherits some of its features from the well-known non-probabilistic modelling language Promela. It offers operational behaviour with a guarded-command approach, shared variables of different types, buffered message passing between processes via channels that may be lossy as well as perfect, conditional execution of alternatives, randomized assignments and probabilistic choice, loops and atomic regions, i.e., program sections composed out of multiple commands that are executed as one single step. We argue that this language is particularly suitable for modelling randomized nondeterministic communication protocols on a level that is relevant in practical settings. The language offers clear operational semantics that are given two steps with a "detour" over probabilistic program graphs (PCGs). Crucial for linking this elegant approach to its practical implementation is the language PASM, which defines the low-level operational behaviour of probabilistic programs graphs in terms of virtual-machine operations in LiQuor. Normally, the implementation of a non-trivial high-level modelling language is not straightforward. PASM, however, provides a connection between the formal structure of probabilistic program graphs and the implementation by being very suitable for the execution on a virtual machine. To our knowledge the only other approach that utilizes the concept of virtual machine language in the context of formal verification is done with the non-probabilistic NIPS virtual machine (see, e.g., [Web10]). The main difference is that NIPS allows for the speculative execution of guards with an optional rollback-mechanism while PASM does allow only for readonly operations in guards, which fully agrees with PROBMela. Instead of the more sophisticated priority-scheme, e.g., for implementing atomic-regions, of NIPS, the PASM virtual machine can switch between a general (interleaved) mode and an exclusive mode in which only one dedicated process is executed.

**Quantitative analysis of probabilistic nondeterministic systems.** Supposing that the model definition is already given, for instance by a high-level specification consisting of processes defined in PROBMela leading to an MDP, the automata-theoretic approach to quantitative
model checking can be divided into two further calculation steps. The first step being identifying a target set for the probabilistic reachability analysis and the numerical solving of this reachability problem. In both of these stages it is highly desirable to apply reduction techniques that help coping with the state explosion problem.

**Efficient numerical solving.** Suppose, that the target set is already constructed such that the remaining step is solving a linear optimization problem. Some prototype comparisons suggest in chapter 4 that applying standard solving techniques as the Simplex method might not the best choice for this and create an unnecessary bottleneck since even a naïve implementation of the value iteration performs orders of magnitude better. Consequently the value iteration principle became target for further investigation. With the obvious observation that in value iteration it is only necessary to recalculate a value if the values that correspond to the successor states in the MDP have changed we presented several variants of this "backward-oriented" approach. These variants differ in the data structure that is used to pick the next value to be recalculated in the iteration process. We compared three different data structures. However, since the transition relation of PCGs is not symmetric, "going backward" is not possible without storing some further data, such that the backward-oriented analysis and its variants bound to some time-vs-memory tradeoff.

Another improvement that is tailored to exploit the MDP context in particular is the MEC-quotient approach. Here, structural observations derived from properties of end components lead us to the building of a quotient system that is typically significantly smaller than the original linear optimization problem. A significant improvement of this time-sensitive part of the model checking process is the consequence.

**Fairness.** For the identification of the target set for the probabilistic reachability analysis the model is unrolled and examined for accepting end components in an on-the-fly fashion. In chapter 5 the concept of fairness which is very important for practical model checking is addressed. For this we fixed an appropriate notion of fairness for our purpose from the literature, namely state and action based weak and strong fairness constraints. We showed how fairness can seamlessly be integrated into the automata-theoretic approach for quantitative analysis through an extension of the algorithm that checks an end component for satisfiability. We showed that when using fairness like this the complexity of the model checking process is not worsened. The problem of realizability is particularly important for fairness constraints that are based on states and atomic propositions respectively. We showed that in practice the check for realizability of a fairness constraint can be done on-the-fly while during the analysis for the property.

**Partial order reduction.** In chapter 6 we showed how to utilize the approach of partial order reduction, an on-the-fly reduction technique, for probabilistic systems. Since partial order
Chapter 7. Conclusions

reduction preserves ω-regular path properties in the reduced system it is compatible with the automata-theoretic approach. We developed and discussed several crucial heuristics - stronger criteria - that realize the important ample set conditions C1-C5. These criteria where implemented and tested against several probabilistic models in the model checker LiQuor to stress their reduction potential. In the context of this implementation it is worth noting that the virtual-machine approach allows for very clear and easy realizations of these conditions. The application of the reduction at runtime, i.e., during the unrolling process, depends on a preanalysis of the process specifications, i.e., the PROBMela program, that is done prior to the actual analysis. With the virtual-machine approach this preanalysis can be moved from the level of the specification language, e.g., PROBMela or any other input language that might be in charge, to the level of the syntactically rather simple machine oriented language PASM. While this preanalysis is very easy in PASM it would mean an additional level of complexity for an interpreter or compiler that already solves the advanced task of processing a syntactically rather complex modelling language. This decoupling of the preanalysis and the language has, from a software-engineers point of view, very positive influence on the implementation’s complexity and can be regarded as a kind of design pattern for model checkers.

Comparison with symbolic model checking. Via an automatic translator that translates PASM programs to the input language of PRISM, the PROBMela language is also available for symbolic model checking [CBGP08]. Table 7.1 contains results on a brief comparison of LiQuor and PRISM based on this translator. Note, that here the complete model checking runs are measured, i.e., building the system and solving a quantitative property.

<table>
<thead>
<tr>
<th>model</th>
<th>states</th>
<th>transitions</th>
<th>time LiQuor</th>
<th>time PRISM</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP(6)</td>
<td>3.3 · 10⁶</td>
<td>26 · 10⁶</td>
<td>761 s</td>
<td>4.2 s</td>
</tr>
<tr>
<td>DC(5)</td>
<td>1.64 · 10⁶</td>
<td>6.32 · 10⁶</td>
<td>81 s</td>
<td>&lt;1 s</td>
</tr>
<tr>
<td>RG(4)</td>
<td>488902</td>
<td>661307</td>
<td>6.9s</td>
<td>2 s</td>
</tr>
<tr>
<td>LE(6)</td>
<td>1.1 · 10⁷</td>
<td>6.2 · 10⁷</td>
<td>856 s</td>
<td>8.3 s</td>
</tr>
<tr>
<td>UMTS</td>
<td>244233</td>
<td>253629</td>
<td>8s</td>
<td>1201 s</td>
</tr>
</tbody>
</table>

Table 7.1: Comparison of LiQuor and PRISM

Among the case studies is one industrial motivated model (UMTS) that involves examining certain rare errors that occur when UMTS phones register to the network provider. In this case the symbolic model checking approach seems to experience difficulties compared to the explicit approach realized in LiQuor. This model involves complex storage behaviour in internal buffers of an UMTS end user device and uses almost every language element of
Probmela discussed in this paper. Moreover the model consists of a structure that contains many long chains that are not connected. In this case the MTBDDs size grows and the BFS oriented symbolic approach performs badly. The results show that there exist models where one tool experiences great difficulties where the other may succeed rather quickly, and vice versa. For smaller models the explicit approach of LiQuor roughly matches the performance of the symbolic approach.

This concludes the topics discussed in this thesis.
# List of Figures

1.1. Overview on quantitative automata-based model checking ........................................ 8

2.1. Hierachy of schedulers ................................................................................................. 17
2.2. Example $M, A$ and $M \otimes A$ .................................................................................. 27
2.3. Example of accepting and non-accepting end components ........................................... 30

3.1. Scheme of the two-layered semantics .............................................................................. 42
3.2. Graph for the stepwise execution of program locations in Knuth’s dice ......................... 49
3.3. MDP for the dice example a la Knuth ........................................................................... 52
3.4. A mutual exclusion protocol using a probabilistic arbiter ............................................ 53
3.5. PCGs with edges according to axioms for communication events and skip ..................... 53
3.6. MDP of the mutex example ......................................................................................... 54
3.7. Schematic overview of the virtual machine and PASM .................................................. 60
3.8. Example PROBMela process and corresponding PCG .................................................. 61

4.1. Stages of quantitative model checking discussed in this thesis .................................... 65
4.2. MDP for comparison of naïve value iteration and "faster" value iteration ......................... 70
4.3. Comparison of $T$ realized as queue (left) and as priority queue (right) ......................... 72
4.4. Second comparison of $T$ realized as queue (left) and as priority queue (right) .............. 73
4.5. Extreme example for priorized fast value iteration ....................................................... 74
4.6. An MEC-quotient example ........................................................................................... 77
4.7. Example MEC-quotient with normalized actions and without self loops ....................... 78
4.8. An MEC-quotient that (still) contains a cycle .............................................................. 79

5.1. MDP of the mutex example revisited for fairness .......................................................... 85
5.2. Tree of recursive calls during execution of procedure check_fair ................................. 94
5.3. Consequence of an unrealizable fairness conditions ..................................................... 102

6.1. Schema of partial order reduction .................................................................................. 104
6.2. Five conditions for the ample-sets .............................................................................. 105
6.3. Examples for independent actions ................................................................................ 106
6.4. Full expansion when encountering a backward edge .................................................... 118
6.5. Full expansion depends on data structure $D$ (before / after last full expansion) ........... 120
6.6. Full expansion not necessary because $\alpha$ is not enabled in state $s_1$ .......................... 121
6.7. Counterexample for ample-set method with process fairness ....................................... 128
6.8. Proof illustration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 130
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.</td>
<td>Comparison of solving LPs with lp_solve (simplex) and value iteration</td>
<td>68</td>
</tr>
<tr>
<td>4.2.</td>
<td>Comparison of different priority-based update strategies</td>
<td>80</td>
</tr>
<tr>
<td>4.3.</td>
<td>Comparison of memory requirements for backward-oriented value iteration</td>
<td>81</td>
</tr>
<tr>
<td>4.4.</td>
<td>Results of applying the MEC-quotient approach to the case studies</td>
<td>82</td>
</tr>
<tr>
<td>4.5.</td>
<td>The MEC-quotient approach with and without blockwise value iteration</td>
<td>83</td>
</tr>
<tr>
<td>6.1.</td>
<td>read and write sets for the mutual exclusion example from section 3.2.2</td>
<td>113</td>
</tr>
<tr>
<td>6.2.</td>
<td>Comparison of different realizations of the end-component condition C4</td>
<td>125</td>
</tr>
<tr>
<td>7.1.</td>
<td>Comparison of LiQuor and PRISM</td>
<td>135</td>
</tr>
</tbody>
</table>
Bibliography


[d’A03b] P. d’Argenio. Towards the Verification of Probabilistic and Nondeterministic Systems. Taken from Personal Communications with the authors, 2003.


147


Declaration

Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Affidavit

I herewith declare, in lieu of oath, that I have prepared this thesis on my own, using only the materials (devices) mentioned. Ideas taken, directly or indirectly, from other sources, are identified as such.

October 11th, 2010

_________________________________
(Frank Ciesinski)