On the Analysis of Stochastic Timed Systems

Thesis for obtaining the title of
Doctor of Engineering Science
of the Faculty of Natural Science and Technology I
of Saarland University

by

Arnd Hartmanns

Saarbrücken
February 2015

Day of Colloquium
4 February 2015

Dean of the Faculty
Univ.-Prof. Dr. Markus Bläser

Chair of the Committee
Prof. Dr. Gert Smolka

Reviewers
Prof. Dr. Holger Hermanns
Prof. Kim G. Larsen, Ph.D.
Prof. Dr. Jaco C. van de Pol

Academic Assistant
Dr. Jan Krčál

Abstract

The formal methods approach to develop reliable and efficient safety- or performance-critical systems is to construct mathematically precise models of such systems on which properties of interest, such as safety guarantees or performance requirements, can be verified automatically. In this thesis, we present techniques that extend the reach of exhaustive and statistical model checking to verify reachability and reward-based properties of compositional behavioural models that support quantitative aspects such as real time and randomised decisions.

We present two techniques that allow sound statistical model checking for the nondeterministic-randomised model of Markov decision processes. We investigate the relationship between two different definitions of the model of probabilistic timed automata, as well as potential ways to apply statistical model checking. Stochastic timed automata allow nondeterministic choices as well as nondeterministic and stochastic delays, and we present the first exhaustive model checking algorithm that allows their analysis. All the approaches introduced in this thesis are implemented as part of the MODEST TOOLSET, which supports the construction and verification of models specified in the formal modelling language MODEST. We conclude by applying this language and toolset to study novel distributed control strategies for photovoltaic microgenerators.

Zusammenfassung

Formale Methoden erlauben die Entwicklung verlässlicher und performanter sicherheits- oder zeitkritischer Systeme, indem auf mathematisch präzisen Modellen relevante Eigenschaften wie Sicherheits- oder Performance-Garantien automatisch verifiziert werden. In dieser Dissertation stellen wir Methoden vor, mit denen die Anwendbarkeit der klassischen und statistischen Modellprüfung (*model checking*) zur Verifikation von Erreichbarkeits- und Nutzenseigenschaften auf kompositionellen Verhaltensmodellen, die quantitative Aspekte wie zufallsbasierte Entscheidungen und Echtzeitverhalten enthalten, erweitert wird.
Wir zeigen zwei Methoden auf, die eine korrekte statistische Modellprüfung von Markov-Entscheidungsprozessen erlauben. Wir untersuchen den Zusammenhang zwischen zwei Definitionen des Modells des probabilistischen Zeitautomaten sowie mögliche Wege, die statistische Modellprüfung auf diese Art Modelle anzuwenden. Stochastische Zeitautomaten erlauben nichtdeterministische Entscheidungen sowie nichtdeterministische und stochastische Wartezeiten; wir stellen den ersten Algorithmus für die klassische Modellprüfung dieser Automaten vor. Alle Techniken, die wir in dieser Dissertation behandeln, sind als Teil des MODEST TOOLSETs, welches die Erstellung und Verifikation von Modellen mittels der formalen Modellierungssprache MODEST erlaubt, implementiert. Wir verwenden diese Sprache und Tools, um neuartige verteilte Steuerungsalgorithmen für Photovoltaikanlagen zu untersuchen.

Acknowledgments

There are many people that contributed directly or indirectly to the existence of this thesis, to its contents, and to the ten years of my time at Saarland University before this thesis was finally written and defended. I am grateful to all of these people for their support, advice, contributions, and friendship.

First and foremost, this thesis would not have been possible without the advice, guidance and funding provided by my advisor, Holger Hermanns. On the organisational side, Christa Schäfer's skills and continued support were invaluable. I am grateful for Gert Smolka's advice and support, from my undergraduate times ten years ago right up to taking on the task of heading my committee. When it comes to my reviewers, Kim G. Larsen not only accepted the job of reviewing this thesis, but also taught me how to deal with tough questions during presentations early on in the QUASIMODO project, while Jaco van de Pol provided a long list of deep questions and helpful comments before the defence that surely improved both my talk and this final version of the thesis document. I am grateful to Jan Krčál for joining the committee and making sure that all the reviewers arrived safely and in time for the defence.

A number of people have directly contributed to my research and thus the contents of this thesis. Although my official time as a Ph.D. candidate only began in May 2009, I had been exposed to the MODEST modelling language much earlier. Reza Pulungan helped with several technical challenges during this period. The development of the MODEST TOOLSET, which now implements the new techniques presented in this thesis, was aided in the first years by the programming work of Jonathan Bogdoll. He also initiated the work on using partial order reduction for statistical model checking, which later also benefited from input by Luis María Ferrer Fioriti. This idea was originally due to Pedro D'Argenio, who continued to help with discussions on modelling, the MODEST language, and stochastic timed systems every time he visited us in Saarbrücken. The alternative of using confluence reduction was the result of the most efficient and enjoyable scientific collaboration I was part of so far, with Mark Timmer from the University of Twente. This collaboration would not have taken place without the ROCKS project and its fruitful meetings in nice locations throughout Germany and the Netherlands. Alexandre David must be mentioned for providing insights into UPPAAL and the effective joint work that made the development of the mctau tool possible towards the end of the QUASIMODO project. Another enjoyable collaboration was with Ernst Moritz Hahn and Joost-Pieter Katoen on extending MODEST to the hybrid systems setting, which later led to the development of the first model checking technique for stochastic timed automata by specialising Moritz' results for hybrid

systems. Pascal Berrang helped evaluate the microgenerator control strategies by running the experiments and creating beautiful graphs of the results.

I would like to thank Dominique Borrione for inviting me to give a tutorial on MODEST at FDL 2012, and Thorsten Tarrach for arranging the opportunity to present my work at IST Austria in early 2014. I met David Parker at several conferences and workshops, and our discussions always left me with new insights and ideas. I am grateful for having had the opportunity to work with great colleagues like Hernán Baró Graf, Jonathan Bogdoll, Yuliya Butkova, Pepijn Crouzen, Christian Eisentraut, Alexander Graf-Brill, Luis María Ferrer Fioriti, Ernst Moritz Hahn, Vahid Hashemi, Hassan Hatefi, Jan Krčál, Gilles Nies, Lei Song, Thorsten Tarrach, Andrea Turrini and Björn Wachter at the chair of Dependable Systems and Software as well as with our visitors Pedro D'Argenio, Tugrul Dayar and Hubert Garavel, and also for the exchange of ideas with Verena Wolf's group for Modeling and Simulation, including Aleksandr Andreychenko, Thilo Krüger, Linar Mikeev and David Spieler.

Over the ten years I spent at Saarland University as a computer science student and Ph.D. candidate, I have always been most grateful for the support of my entire family: Ursula, Jörg, Anke, Annaliese, Hans-Hermann, Rolf, thank you! Furthermore, these years would have been much less enjoyable without the friendship of Mirren Augustin, Sebastiano Barbieri, Hernán & Virginia Baró Graf, Jan & Agnes Christoph, Sven Dahms, Christian Doczkal, Fabienne Eigner, Fiona Gutjahr, Pascal Gwosdek, Ernst Moritz Hahn, Michaela Hardt, Steffen & Tamara Heil, Anne Krätschmer, Markus Mainberger, Sebastian Meiser, Esfandiar Mohammadi, Markus Rabe and Raphael Reischuk—who knows what this thesis would be like without you? An important part of being a student or Ph.D. candidate is to use the flexibility that these occupations allow to explore the world. The exploration of the Northeastern U.S., as well as of the game of golf, has been kindly supported by Corrine & Rick Jurgens. The collaboration with Mark Timmer also extended, with the same ease and efficiency, to the exploration of the national parks of Arizona, California, Nevada and Utah together with Alfons and Laura. Visits to Argentina started in 2013 together with Pedro D'Argenio, Alexander Graf-Brill, Luis María Ferrer Fioriti, Vahid Hashemi, Hassan Hatefi, Holger Hermanns, Andrea Turrini and Lijun Zhang, and were continued with the amazing hospitality of Carlos Budde and his family in 2014, including activities with Pedro D'Argenio, Raúl Monti and Silvia Pelozo. Norway was explored together with Manuel Jöris, and a visit to Canada was caused and supported by Marianne Boivin. Last but very far from least, I have only been to Ireland due to Thorsten Tarrach, and would not have experienced Serbia, Bosnia and Herzegovina, Croatia and Montenegro without Thorsten Tarrach and Sanja Pavlović.

Funding

The research leading to the results presented in this thesis was supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center “Automatic Verification and Analysis of Complex Systems” (SFB/TR 14 AVACS); by the European Community’s Seventh Framework Programme under grant agreements no. 214755 (QUASIMODO), no. 295261 (MEALS) and no. 318490 (SENSATION); by the DFG/NWO Bilateral Research Program ROCKS; and by the CAS/SAFEA International Partnership Program for Creative Research Teams.

Contents

1 Introduction

Information technology enables us to rely on an increasing amount of increasingly complex systems in our individual daily lives and as the basis for our modern society. We have been able to replace manual tasks and mechanical or electrical machinery with electronic components and computer-based systems. For example, fly-by-wire technology has steadily taken the place of mechanical or hydraulic flight control systems in new commercial airliners since the early 1980s. The ensuing reduction of weight and simplification of handling improve aircraft economy. Once a luxury, flying has become a normal mode of mass transportation. Trading, formerly the realm of brokers on stock exchange floors, can today be carried out electronically with precise split-second timing. The reader's pension funds may well be in the hands of such a real-time electronic trading system, and so is our economy. The recent trend of "smart" power grids revolves around the embedding of computing and communication components in decentralised electricity producers and consumers. The goal is to cut off overproduction when necessary and dynamically schedule flexible consumers like off-peak storage heaters or air conditioning systems. The power grid of the future will thus be ready to deal with the increase of volatile renewable energy sources such as the photovoltaic generators installed on many private rooftops. The same pattern can be found over and over again in the large, where e.g. industrial automation systems are becoming networked, and in the small as our daily lives depend on more and more Internet services.

However, this increase of complexity comes at a cost. Fly-by-wire systems have failed due to small errors and incorrect assumptions, leading to loss of lives and aircraft [Job96]. Unexpected interactions combined with the sheer speed of electronic trading have caused severe stock market fluctuations and million-dollar losses for banks and funds [LSS10, SC13]. Controllers for photovoltaic microgenerators deployed on hundreds of thousands of rooftops have been shown to adversely affect grid stability by introducing oscillatory behaviour [BBZL11]. When nuclear power plants are controlled by networked

computer systems and people can no longer imagine life without constant high-speed Internet access, the necessity to use information technology not only as a part of these systems, but also to study their safety, reliability, and performance is evident.

1.1 Modelling and Verification

The field of *formal methods* [CW96] is concerned with the development of mathematically rigorous techniques for the design and analysis of complex critical systems. As illustrated in Figure 1.1, the core components of the formal methods approach are
– a precise mathematical **model** of the behaviour of the system under study,
– a set of **requirements** that formally specify the desired behaviour, and
– a **verification** procedure to check if the model satisfies the requirements.
Any result obtained in this way obviously applies to the actual system *implementation* only as much as the model correctly reproduces all aspects that are relevant for the satisfaction or violation of the requirements. *Model-based testing* [Tre08] should therefore be applied to *validate* the model, i.e. to gain confidence that the model correctly reflects what is implemented, or equivalently that the implementation actually conforms to the model.

Models A system design is usually described in a natural language text, or using an informal representation such as UML [ISO12a, ISO12b]. The *modelling* step is the creation of a formal model from such an informal description. It is usually performed manually, and often reveals inconsistencies and omissions in the design document. The process of constructing a formal model has thus been shown to already be beneficial to the system design [SSBM11]. If we study a software system that has already been implemented, we may be able to automatically extract a model from its source code.

In this thesis, we focus on *reactive systems*, i.e. processes that interact with their environment and run continuously. We thus consider *behavioural models*. This is in contrast to the analysis of transformational programs, which can be characterised by their input-output relation. Behavioural models can be represented in any of a large number of formalisms with widely differing degrees of expressiveness and conciseness: low-level automata specifications, networks of interacting symbolic automata, Petri nets, process algebras, and formal modelling languages, to name a few categories.

Requirements Just like a formal model needs to be created based on the informal design documents, the requirements of interest have to be extracted from

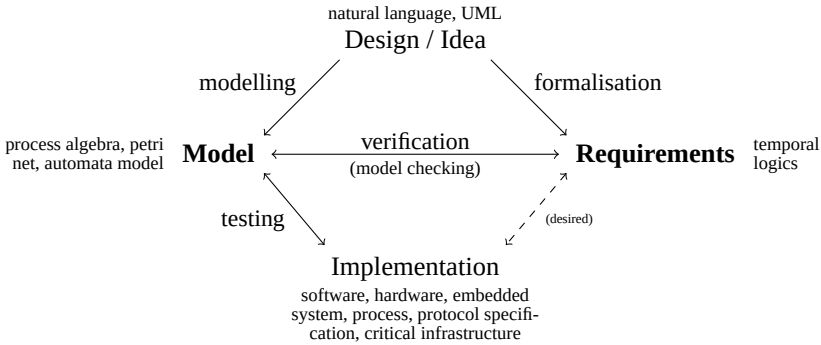


Figure 1.1: The formal methods approach

these documents as well. Again, their formalisation may highlight inconsistencies and previously ignored corner cases. For behavioural models, a natural way to express requirements formally is through the use of *temporal logics*. Abstractly speaking, a temporal logic formula characterises the desired evolutions of the system state (the *state-based* approach) or the system's desired interactions with the environment (the *action-based* approach) over time. Examples of temporal logics are LTL [Pnu77], CTL [EC82] and their various derivatives.

Verification Given a formal model and a set of requirements, a verification procedure establishes whether the model satisfies or violates the requirements. The set of concrete verification procedures can broadly be divided into those based on *theorem proving* and those based on *model checking*. Although theorem proving has been partly automated and tool support is available, it remains a manual procedure in key steps. Theorem proving tools are thus known as *interactive* theorem provers or proof *assistants*. On the other hand, very large or infinite-state models can naturally be dealt with.

The key characteristic of model checking, in contrast, is that it is a fully automatic technique (also known as a "push-button approach"), with an exhaustive exploration of a model's state space at its core. Not surprisingly, models with extremely large or infinite state spaces cannot be dealt with in a naïve model checking approach, giving rise to the *state space explosion* problem. Various techniques to reduce, minimise, truncate, abstract or otherwise transform state spaces in a way that introduces no or a clearly quantifiable error have

been developed over the last decades. Model checking has thus become applicable to real-life-size systems using one of the many available model checking tools, known as *model checkers*.

In the remainder of this thesis, we only consider verification using traditional *exhaustive* model checking techniques as described above as well as so-called *statistical* model checking. The latter is an attempt to tackle the state space explosion problem by using an entirely different approach while keeping the *quantifiable error* property that is so fundamental for the applicability of model checking to critical systems and infrastructures.

1.2 Quantitative Requirements and Models

Let us reconsider the examples of fly-by-wire control, electronic real-time trading, smart power grids, networked industrial automation systems and the Internet. Most of these are clearly safety- or mission-critical, i.e. a malfunction may lead to the loss of lives or at least significant amounts of money. We thus need to verify *functional*, or *qualitative*, requirements in the first place. Classic examples are indeed *safety*, requiring that "bad things never happen" (e.g. "thrust reversers will never engage in flight" as they did on Lauda Air Flight 004 in 1991 [Job96]), but also *liveness*, the requirement that "something good will always eventually happen" (e.g. "whenever a plane is in flight, it will safely land again in the future").

However, functional correctness is usually not enough. A correct electronic trading system, to stay with our examples, is useless if it cannot realise orders within certain time constraints; an algorithm to ensure power grid stability needs to react to changes in a matter of milliseconds to be useful; and a wireless temperature sensor in an industrial process must not empty its battery within the first hour of operation. Performance requirements therefore need to be considered as well. These are necessarily expressed as *quantitative* requirements, such as "the minimum time until the battery is empty must be more than 10 hours". In general, *time* is not the only quantity we may be interested in. The last example already includes *costs* in the form of battery usage. Additionally, the satisfaction of requirements can only be guaranteed with a certain *probability* in many scenarios, leading to probabilistic statements like "the expected time until the battery is empty must be at least 16 hours". In the quantitative setting, it is often convenient to ask for an actual value instead of wrapping it in a Boolean requirement: "What is the minimum probability of success without exceeding battery capacity?" We refer to these types of questions, or to the value they relate to, as *queries*. In the remainder of this thesis, we use the word *property* as a generic term for queries and requirements.

In order to refer to quantities in properties, they need to be included in the models in the first place. Model checking has traditionally been used to verify qualitative requirements with modelling formalisms that represent the possible behaviours of a system using *nondeterministic choices*, such as variants of labelled transition systems. Quantitative properties, on the other hand, have been studied in the field of *performance evaluation*, using Markov chains or more elaborate stochastic processes. As we have argued above, though, the analysis of today's and tomorrow's complex systems requires a unified treatment of correctness and performance. Not surprisingly, a variety of integrative approaches that combine quantitative reasoning with formal verification techniques have therefore been developed over the last decades [BHHK10]. In this thesis, we use a number of automata-based modelling formalisms that capture various kinds of quantitative aspects as highlighted in Figure 1.2:

Nondeterminism A nondeterministic choice is an *unquantified* choice between two or more alternative behaviours. It signifies the absence of any information related to the relative frequency in which the alternatives will occur or the precise conditions that influence the decision, and can be used to
– model the influence of an unknown environment on the system,
– represent the unknown scheduling of concurrent components,
– abstract details and create a coarser model from a very detailed one,
– allow implementation freedom regarding a certain choice, or simply
– represent true absence of knowledge of the modeler about a certain choice.
Nondeterministic choices are the core feature of the labelled transition system formalism (LTS [BK08], presented in Chapter 3), also known (with minor differences in purpose and definition depending on the source) as Kripke structures or finite automata. An LTS consists of a set of states, each of which has a number of outgoing labelled transitions. The nondeterminism is in the choice of the transition to take from a state. Each transition in turn leads to a single next state.

Probabilities In contrast to this, a *probabilistic choice* is a *quantified* one: it assigns a certain, precise probability to the alternatives. Probabilistic choices represent quantified uncertainty, which can be uncontrollable or inherent to the system at hand:
– the influence of the environment may be modelled as probabilistic, e.g. by assigning a probability to the event of message loss in wireless communication; on the other hand,
– randomised algorithms intentionally use probabilistic experiments to achieve correctness or improve performance.

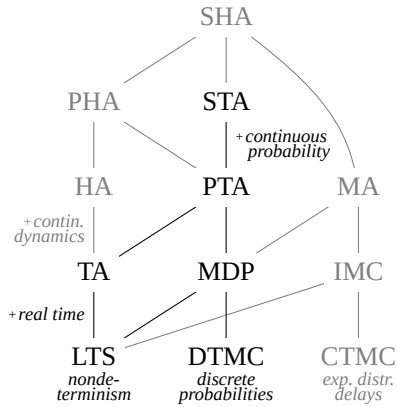
A classic example for randomised algorithms that require probabilistic choices for correctness are distributed, fully-symmetric leader election protocols.

Based on the underlying probability distribution, probabilistic choices can be categorised into discrete and continuous choices: discrete probability distributions assign probabilities to a countable (or even finite) set of alternatives, while this set is uncountably infinite—usually the set of real numbers—for continuous distributions (which therefore also require a more intricate definition taking into account the issue of measurability). We call formalisms and models that use only discrete probability distributions *probabilistic*, whereas *stochastic* ones use continuous distributions. In the remainder of this thesis, we also distinguish between a probabilistic *choice* over a finite set of outcomes, i.e. using a finite-support distribution, and the (stochastic) *sampling* of a value from an infinite-support distribution.

The most basic probabilistic modelling formalism used in this thesis is the discrete-time Markov chain (DTMC [GS01, BK08], presented in Chapter 3). In contrast to LTS, each state in a DTMC has a single, unlabelled outgoing transition that leads to a discrete and usually finite-support probability distribution over target states. Combining the features of LTS and DTMC results in the nondeterministic-probabilistic formalism of Markov decision processes (MDP [Put94], presented in Chapter 4). For our purposes, we can consider the formalism of probabilistic automata (PA [Seg95, Sto02]) as an equivalent of MDP. In a given MDP model, the result of a query depends on the way the nondeterministic choices are resolved and is thus usually given as the interval from minimum to maximum possible value. In terms of stochastic formalisms, the most basic one supporting continuous distributions in this thesis is that of stochastic timed automata (STA [BDHK06], presented in Chapter 6). STA in fact allow arbitrary distributions; there is also a significant family of stochastic formalisms based solely on the exponential distribution, the most basic of which would be the continuous-time Markov chain (CTMC [GS01], see Section 6.2). CTMC can be extended with nondeterministic choices to interactive Markov chains (IMC [Her02]), and additionally with discrete probability distributions to Markov automata (MA [EHZ10]).

Time It was evident in our earlier examples of quantitative properties that *time* plays an important role in many applications. The notion of time, or of a *delay*, is crucial

– for communication protocols that incur transmission delays and make use of timeouts to detect message loss;
– when studying performance and response times of network servers that need time to process incoming requests, which themselves occur in intervals at



Key:

SHA	stochastic hybrid automata
PHA	probabilistic hybrid automata
STA	stochastic timed automata
HA	hybrid automata
PTA	probabilistic timed automata
MA	Markov automata
TA	timed automata
MDP	Markov decision processes
IMC	Interactive Markov chains
LTS	labelled transition systems
DTMC	discrete-time Markov chains
CTMC	continuous-time Markov chains

Figure 1.2: Automata-based modelling formalisms relevant for this thesis

a certain rate, and can handle a certain number of requests per time unit (throughput); and

– when looking at time-critical control applications, where inputs need to be processed within very small delays.

There are various ways to model time, depending on the desired level of abstraction and the given requirements. They can be categorised in two dimensions: discrete- and continuous-time models, where delays can in turn be deterministic, nondeterministic or stochastic. In discrete-time models, time progresses in steps and delays are natural numbers; in continuous-time models, the time domain is the set of positive real numbers, allowing arbitrarily small delays. Notably, the semantics of a continuous-time model is an uncountably infinite-state object, but in certain cases, a finite or at least countable representation is sufficient for verification. Delays, in the simplest case, are fixed natural or real numbers, i.e. they are deterministic. However, if precise timings are unknown, we may prefer a model with stochastic delays if we have reason to assume a certain probability distribution for inter-event times, or one with nondeterministic delays in other cases. A nondeterministic delay may, for example, be that a certain event takes place at any moment in the interval $[0, 1)$; the reasons to use nondeterminism for time intervals are much the same as for nondeterministic choices.

In DTMC, as the name implies, there is an implicit notion that every transition takes one unit of model time. For the purposes of this thesis, we do not consider DTMC or MDP models as truly timed. Instead, we focus on

continuous-time models and only use MDP *for their analysis* when this is possible. The basic continuous-time formalism we consider is the timed automaton (TA [AD94, HNSY94]) and its probabilistic version, the probabilistic timed automaton (PTA [KNSS02], presented in Chapter 5). PTA allow nondeterministic and (using an encoding with probabilistic choices) probabilistic delays, but not stochastic ones; the latter become possible when moving to STA.

Continuous dynamics Continuous time can be seen as a variable that increases over time with derivative one. This leads to the obvious generalisation of "continuous" variables that evolve over time in more complex ways, for example specified by differential equations. In this way, behaviours according to physical laws can be captured in a natural fashion. For the special case of otherwise read-only continuous variables that evolve with a constant derivative that only changes with the discrete state of the system, we obtain the classic representation of time-dependent costs or *rewards*. A cost may be a system's energy consumption, which in turn may be higher or lower depending on the operational mode; for example, a wireless station consumes more power when in receiving or sending mode than when idle. A reward is looking at the same issue from the other side, where a cost is simply a negative reward.

The addition of continuous rewards to essentially all continuous-time models is sufficiently straightforward so that we do not explicitly consider, say, "timed automata with rewards" as a dedicated modelling formalism, but merely as a variant of TA. General continuous variables, however, can have a profound impact on expressiveness, but also limit verification possibilities. We therefore expressly mention hybrid automata (HA [Hen96]), probabilistic hybrid automata (PHA [ZSR+10]) and stochastic hybrid automata (SHA [FHH+11]), although we do not treat them in this thesis. SHA are a generalisation of the STA we use here; all other formalisms mentioned so far are special cases of SHA.

1.3 Exhaustive and Statistical Model Checking

As mentioned, an exhaustive exploration of a model's state space is at the core of model checking for all the formalisms described in the previous section. For those that give rise to uncountably infinite state spaces, such as TA or SHA, a finite representation that is either sufficient to precisely verify all requirements (in the case of TA) or that is a safe overapproximation of the real state space (in the case of complex hybrid models) can be used. Still, what limits the applicability of model checking is the state space explosion problem: in the worst case, introducing a new variable that can take m different values leads to a factor-m multiplication of the number of states, i.e. the state space exponentially

in the number of variables. Yet all the states need to be represented in some form in limited computer memory. Even when this is still possible, the runtime of model checking of probabilistic formalisms such as MDP is also severely affected by state space explosion because it involves numerical computations on the states to obtain probability values.

Several techniques have been introduced to stretch the limits of model checking while preserving its basic nature of performing state space exploration to obtain results that unconditionally, certainly hold for the entire state space. One class of techniques aims at finding ways to represent a state space efficiently in memory. A very successful example is the use of binary decision diagrams (BDD) to compactly represent sets of states [dAKN+00], achieving an exponential reduction in memory usage for some models. However, as with all such approaches, there exist models where the use of BDD does not yield space savings and merely increases verification runtime.

State space reduction The other significant class of approaches are state space reduction techniques. They provide procedures that, given a model, yield a state space that is smaller than what would be obtained via a naïve exploration of all reachable states while retaining the same results for all relevant properties. Ideally, state space reduction is applied *on-the-fly*, i.e. replacing the naïve approach by directly generating the reduced state space instead of reducing the full state space post-exploration. The decisive point is to try to minimise the largest intermediate state space that ever needs to be stored in memory.

The relationship between the original and the reduced model needs to be such that the verification results for the properties of interest are the same for both. Various *bisimulation* [Mil89] relations fulfill this purpose for wide ranges of properties and formalisms. For example, two MDP related by divergence-sensitive probabilistic visible bisimulation satisfy the same formulae specified in the temporal logic PCTL* as long as its "next" operator is not used [Tim13]. And in fact, various effective *minimisation* procedures are known that can be used to obtain the minimal representation of a model w.r.t. to some concrete notion of bisimulation [BK08]. Unfortunately, most of them require access to the full reachable state space and thus only lead to reductions in verification runtime, but not in memory usage.

On the other hand, there exist overapproximating approaches that lead to smaller, bisimilar models, but not necessarily minimal ones. Two of them, which will be used later in this thesis (beginning in Chapter 4), are *partial order reduction* (POR) [BDG06, God96, Pel94, Val90] and *confluence reduction* [BvdP02, TSvdP11, TvdPS13]. Both can be used on-the-fly during state space exploration. POR relies on syntactic information from the modelling

language used to work in practice, whereas confluence can easily be applied on the level of the concrete state space without such extra information. Again, both POR and confluence reduction can result in significant memory savings on some models, while they may not provide much of an improvement on others.

Finally, we mention techniques that use abstraction, e.g. counterexample-guided abstraction refinement (CEGAR) [CGJ+00, HWZ08]. They use coarser representations of the real state space, where one abstract state summarises several concrete states and their behaviour. The resulting model is related to the original not through a bisimulation, but a simulation relation, such that e.g. its behaviours are *at least* those of the original, but may include more. The verification results for an abstract model are thus *safe* in some precisely specified way depending on the exact simulation relation used; in probabilistic models, for example, the probability of reaching any set of ("unsafe") states in the abstract model is typically greater than or equal to that in the original. Techniques exist to automatically refine abstract models, and through repeated refinements, the desired tradeoff between verification precision and memory usage can be reached. In a quantitative setting, however, it is not always possible to determine how far the computed result is from the actual value. In this thesis, we only make use of abstraction where it is necessary to analyse continuous-time (chapters 5 and 6) models that would otherwise yield uncountably infinite state spaces. Other than that, we rely on bisimulation-based state space reduction techniques.

Statistical model checking Despite its name, and although it tries to meet the same goal of providing trustworthy verification results with quantifiable error, statistical model checking (SMC) [YS02, HLMP04, LDB10, BBB+10, ZPC10] is very different from model checking. When necessary for clarity, we refer to model checking as *exhaustive model checking* to distinguish it from SMC.

Instead of exploring—and storing in memory—the entire state space, or a reduced version of it, simulation is used in SMC to generate traces of paths through the state space. This comes at constant memory usage and thus circumvents state space explosion entirely, but cannot deliver results that hold with absolute certainty. Statistical methods such as sequential hypothesis testing [Wal45] are then used to make sure that e.g. the *probability* of returning the wrong result is below some threshold. Figure 1.3 contrasts model checking and SMC schematically. The accuracy and precision of the SMC results depend on the system parameters and especially (that is, logarithmically) on the number of paths explored. Here, theoretical complexity is practical complexity, and as a result, SMC is most competitive time-wise for lower accuracy analysis.

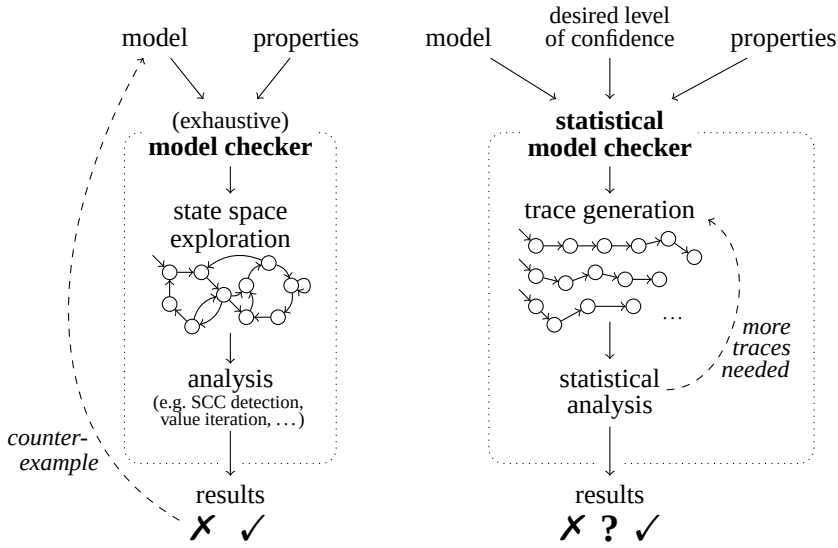


Figure 1.3: Exhaustive and statistical model checking, schematically

While some studies have made efforts to compare the effectiveness of SMC versus model checking empirically [YKNP06, KZ09], such a comparison is inherently problematic: As a simulation-based approach, SMC is limited to fully stochastic models such as Markov chains, whereas model checking is usually applied to variations of nondeterministic transition systems. Indeed, in the traditional simulation community, the unprejudiced application of simulation to models that actually exhibit nondeterministic behaviour has already led to notorious suspicions about hidden assumptions that affect the analysis results [AY06, CSS02]. Attempts to solve this problem, which would allow sound SMC for MDPs, have only been investigated very recently. The author has been involved in the development of two such approaches (presented in Chapter 4) which form a major contribution of this thesis.

1.4 The MODEST Approach

For a formal model to be analysed by a model-checking tool, it needs to be specified in a computer-readable format, i.e. a *modelling language*. Even if a model checker were to process input as close to the underlying mathematical model as possible, it would need to be given in a specific syntax. However,

MODEST:

```
process Channel()
{
  snd palt {
  :99: delay(2)
       rcv
  : 1: // msg lost
       {==}
  };
  Channel()
}
```

Guarded Commands:

```
module Channel
  l: [0..2]; // control loc
  c: clock;  // for delay
  invariant
    l=1 => c<=2
  endinvariant
  [snd] l=0  ->  0.01:(l'=0)
       + 0.99:(l'=1)&(c'=0)
  [rcv] l=1 & c>=2 -> (l'=0)
endmodule
```

UPPAAL TA:

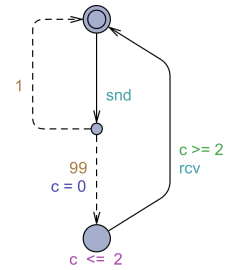


Figure 1.4: Three modelling languages

when models are to be written by humans, a higher-level language that allows modularisation and compositional modelling (i.e. the construction of large models from smaller, independently specified parts) and that has an expressive, easy-to-use and easy-to-learn syntax is desirable. For any such language, a *formal semantics* specifies the mapping from its syntax to the underlying mathematical model, for example one of the automaton formalisms presented earlier.

Today, quantitative modelling and verification is supported by a wide range of tools, most of which use their own dedicated modelling language. Examples include the CADP [GLMS11] toolkit centered around the LOTOS language [BB87] and its successor LNT; the PRISM [KNP11] model checker and similar tools [HHWZ09, HHWZ10a, HHWZ10b] operating on guarded commands; and UPPAAL [BDL04], which allows the graphical modelling of networks of TA. The variety of different languages used by tools in this area, however, is a major obstacle for new users seeking to apply formal methods in their field of work: they usually need to learn several of these languages, and still models built for one tool need to be rewritten to be usable with a different one.

The MODEST language The MODEST[1] modelling language, on the other hand, was designed to be as expressive as possible instead of being restricted to the formalism and supported analysis approaches of any specific tool. While rooted in process algebra, MODEST borrows syntax and concepts from widely-used programming languages to make it accessible to programmers and engineers. At the same time, its expressive syntax allows complex models to remain concise and readable. MODEST was originally specified with a formal semantics in terms of STA [BDHK06]. We have extended it to cover the full

[1] Originally, MODEST was an acronym for "a **mo**delling and **de**scription language for **s**tochastic **t**imed systems", and was written as MODEST.

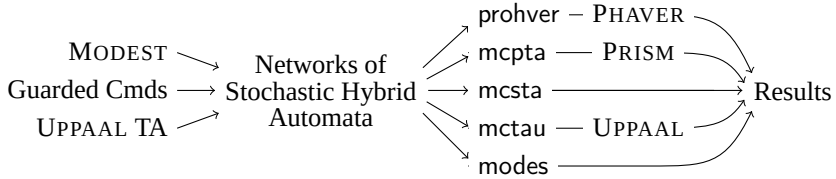


Figure 1.5: Schematic overview of the MODEST TOOLSET's components

expressiveness of SHA [HHHK13]; our presentation of MODEST in this thesis is based on this revised syntax and semantics. Figure 1.4 gives an impression of MODEST in comparison with guarded commands as used by PRISM and the graphical representation of TA as presented by UPPAAL based on its XML format. The example represents a simple communication channel as could be used in models of wireless communication protocols, with a message loss probability of 0.01 and a transmission delay of 2 time units.

The MODEST TOOLSET To support the analysis of MODEST models, we have developed the MODEST TOOLSET [HH14]. Originally intended to support the *single-formalism, multiple-solution* approach that MODEST was designed for, it has evolved into a *multiple-formalism, multiple-solution* toolset: Aside from MODEST, which is still the primary input language, the aforementioned guarded commands and UPPAAL's XML format are also supported. This is a first step to make model reuse in formal methods easier. At the core of the MODEST TOOLSET is the formalism of networks of SHA, i.e. sets of concurrent, communicating automata. Verification, using exhaustive or statistical model checking, is available through five tools, three of which reuse existing model checkers as backends in order to avoid unnecessary reimplementation:

– prohver computes upper bounds on maximum probabilities of probabilistic safety properties in SHA [HHHK13]. It relies on a modified version of PHAVER [Fre05] to model-check HA.
– mcpta performs model checking of PTA using PRISM for a probabilistic analysis; it supports probabilistic and expected-time/expected-reward reachability properties in unbounded, time- and cost-bounded variants [HH09].
– mcsta performs model checking of STA, PTA and MDP using an explicit-state engine. It supports the same kinds of properties as mcpta, but is only able to computes upper/lower bounds for true STA models [HHH14].

– mctau connects to UPPAAL for model checking of TA [BDHH12], for which it is more efficient than mcpta. It automatically overapproximates probabilistic choices with nondeterminism when given a PTA, providing a quick first check of such models.
– modes performs SMC of STA with an emphasis on the sound handling of nondeterministic models [BFHH11, BHH12, HT13]. Its trace generation facilities are useful for model debugging and visualisation.

Fig. 1.5 gives a schematic overview of the input languages and the analysis backends that form the MODEST TOOLSET. In this thesis, we briefly review the approach used by mcpta to reuse a probabilistic model checker to analyse a probabilistic-timed model in Section 5.5. We present the abstractions and analysis technique used by mcsta in detail, which is the first tool to support model checking of full STA (in Section 6.5). This is similar to the chain of abstractions used by prohver to decompose the SHA analysis problem into model checking of an MDP and a HA. We describe the techniques implemented in modes to soundly perform SMC for nondeterministic formalisms in detail in Section 4.7.

1.5 Contributions and Origins of the Thesis

This thesis is structured along the model hierarchy of Figure 1.2: After introducing basic notions and notation in Chapter 2, we define the foundational models of LTS and DTMC in Chapter 3, including the corresponding subsets of the MODEST language, and give an overview of the existing approaches to exhaustive and statistical model checking. In Chapter 4, we move to MDP, where—after defining the model and summarising how to analyse it with exhaustive model checking—we focus on ways to soundly extend statistical model checking to handle the added nondeterminism. We then add real-time modelling capabilities in Chapter 5, which covers the model of PTA. We highlight the two ways of defining PTA—with invariants or with deadlines—and investigate the respective expressiveness as well as giving a procedure to convert a large subset of PTA with deadlines into PTA with invariants. We review the existing exhaustive model checking techniques and give an overview of potential ways to perform sound SMC for PTA. It is then time to move on to STA for Chapter 6, which are syntactically a small extension of PTA that, however, requires important changes in semantics. The main part of that chapter is on a novel technique to perform exhaustive model checking for STA with continuous and discrete nondeterminism. Finally, we conclude with a report on the modelling and evaluation of distributed control strategies for power grids with significant volatile microgeneration based on renewable energy sources in Chapter 7.

Contributions The main contributions of this thesis, aside from presenting a unified view of the various quantitative models supported by the MODEST language, are the following:

– **Chapter 4**: Two approaches for sound statistical model checking of spuriously nondeterministic MDP, based on partial order and confluence reduction.
– **Chapter 5**: An in-depth comparison of PTA with deadlines vs. PTA with invariants and a procedure to convert a large subset of the former into the latter.
– **Chapter 6**: The first model checking approach for STA with discrete nondeterministic choices, nondeterministic delays, and stochastic timing.
– **Chapter 7**: The modelling and analysis of novel distributed control algorithms for photovoltaic microgenerators that are inspired by solutions to similar problems in Internet protocols.

We have implemented algorithms for sound SMC of spuriously nondeterministic MDP in the modes tool and present an evaluation of their applicability and performance as well as an extension with caching. The conversion of PTA with deadlines into PTA with invariants is implemented in the core libraries of the MODEST TOOLSET and is in particular used by mcpta. The new model checking technique for STA is implemented in the mcsta tool; we assess its effectiveness and efficiency using four varied examples. The evaluation of the new control strategies presented in Chapter 7 has been performed using MODEST models and the modes simulator.

Sources At the beginning of every chapter, we briefly summarise its origins and clearly state the author's personal contributions. At the time of publication, the main contributions of this thesis have appeared in the following conference papers (listed in chronological order):

– *Model-Checking and Simulation for Stochastic Timed Systems*, presented at FMCO 2010 [Har10] (for Chapter 5);
– *Partial Order Methods for Statistical Model Checking and Simulation*, joint work with Jonathan Bogdoll, Luis M. Ferrer Fioriti and Holger Hermanns, presented at FMOODS/FORTE 2011 [BFHH11] (for Chapter 4);
– *Modelling and Decentralised Runtime Control of Self-stabilising Power Micro Grids*, joint work with Holger Hermanns, presented at the 2012 ISoLA Symposium [HH12] (for Chapter 7);
– *A Comparative Analysis of Decentralized Power Grid Stabilization Strategies*, joint work with Pascal Berrang and Holger Hermanns, presented at the Winter Simulation Conference 2012 [HHB12] (for Chapter 7);

– *On-the-fly Confluence Detection for Statistical Model Checking*, joint work with Mark Timmer, presented at the 2013 NASA Formal Methods Symposium [HT13] (for Chapter 4);
– *The Modest Toolset: An Integrated Environment for Quantitative Modelling and Verification*, joint work with Holger Hermanns, presented at TACAS 2014 [HH14] (for the MODEST TOOLSET as a whole); and
– *Reachability and Reward Checking for Stochastic Timed Automata*, joint work with Ernst Moritz Hahn and Holger Hermanns, presented at AVoCS 2014 [HHH14] (for Chapter 6).

2 Preliminaries

Before we go into technical details of modelling and verification, we need to establish notions and notation that will occur repeatedly in the remainder of this thesis. We start with basic mathematical notation in Section 2.1 before we cover the essentials of probability theory for both the discrete and continuous setting in Section 2.2. The syntax of several models and the modelling language MODEST includes variables, expressions and various derived constructs such as assignments, which we define formally in Section 2.3.

2.1 Mathematical Notation

Let us start with a recap of basic mathematical notions—in particular related to sets, relations and functions—and the notation we use for them in this thesis:

Numbers and sets We denote the empty set by $\varnothing$. For a given set S, its power set, i.e. the set of all sets with elements only in S, is written as $\mathscr{P}(S)$. Remember that $\varnothing \in \mathscr{P}(S)$ for all S. We denote the set $\{0,1,2,\dots\}$ of the natural numbers by $\mathbb{N}$. Observe that it includes 0; we use $\mathbb{N}^+$ to refer to the set $\mathbb{N} \setminus \{0\}$ of the positive natural numbers. Similarly, we refer to the set of real numbers as $\mathbb{R}$, to the set of positive real numbers as $\mathbb{R}^+$, and to the set of non-negative real numbers as $\mathbb{R}_0^+$. The closed interval $\{z \in \mathbb{R} \mid x \le z \le y\}$ will be written as $[x,y]$, while (x,y) refers to the open interval $\{z \in \mathbb{R} \mid x < z < y\}$. Half-closed intervals follow this notation analogously. Other particular sets that we use are the set of integers $\mathbb{Z} = \{\dots,-1,0,1,\dots\}$, the set of rational numbers $\mathbb{Q}$, and the set of Boolean values $\mathbb{B} = \{\mathit{true}, \mathit{false}\}$. When space is tight, we may choose to write *tt* and *ff* in place of *true* and *false*, respectively.

For a finite number of sets $S_1,\dots,S_n$ with $n \in \mathbb{N}^+$, $S_1 \times \cdots \times S_n$ denotes the set of n-tuples $\langle s_1,\dots,s_n\rangle$, or tuples of length n, such that we have $s_i \in S_i$ for each component s_i. We also use S^n as a shorthand for the set of tuples of length n where all components are elements of S. For a given set S, we

denote by S^* the set of finite sequences of elements of S. We identify a concrete finite sequence $s_1 \dots s_n \in S^*$ with the tuple $\langle s_1, \dots, s_n \rangle \in S^n$. The set of infinite sequences $s_1 s_2 \dots$ of elements of S is referred to as S^ω.

Relations Given two sets S_1 and S_2, a *relation* R over S_1 and S_2 is a set of pairs $R \subseteq S_1 \times S_2$. Its *domain* is $\mathrm{Dom}(R) = \{ s_1 \in S_1 \mid \exists s_2 \in S_2 \colon \langle s_1, s_2 \rangle \in R \}$. R is
– *injective* if $\langle s_1, s_2 \rangle \in R$ and $\langle s_1', s_2 \rangle \in R$ implies $s_1 = s_1'$,
– *functional* if $\langle s_1, s_2 \rangle \in R$ and $\langle s_1, s_2' \rangle \in R$ implies $s_2 = s_2'$,
– *surjective* if $\forall s_2 \in S_2 \colon \exists s_1 \in s_1 \colon \langle s_1, s_2 \rangle \in R$, and
– *total* if $\mathrm{Dom}(R) = S_1$.
A *bijective* relation is one that is injective and surjective. Observe that a bijective relation is also functional. The *restriction* of the relation R to a subset $S_1' \subseteq S_1$ is written as $R|_{S_1'} = \{ \langle s_1, s_2 \rangle \in R \mid s_1 \in S_1' \}$. The *inverse* relation of R is denoted by $R^{-1} = \{ \langle s_2, s_1 \rangle \mid \langle s_1, s_2 \rangle \in R \}$. The *image* of $s_1 \in S_1$ is a subset of S_2, denoted by $R(s_1) = \{ s_2 \in S_2 \mid \exists s_1 \in S_1 \colon \langle s_1, s_2 \rangle \in R \}$, and the *preimage* of $s_2 \in S_2$ is defined as $R^{-1}(s_2) = \{ s_1 \in S_1 \mid \exists s_2 \in S_2 \colon \langle s_1, s_2 \rangle \in R \}$. We may also write $R(s_1, s_2)$ or $s_1 R s_2$ in place of $\langle s_1, s_2 \rangle \in R$. Two sets S_1 and S_2 are *isomorphic* if there exists a total bijective relation over S_1 and S_2. In that case, we write $S_1 \cong S_2$. The *identity relation* is defined as $Id(S) = \{ \langle s, s \rangle \mid s \in S \}$, or simply Id when S is clear from the context.

A relation $R \subseteq S \times S$ is *reflexive* if $R \supseteq Id(S)$, it is *symmetric* if we have $R(s, s') \Rightarrow R(s', s)$, and it is *transitive* if $R(s, s'') \wedge R(s'', s') \Rightarrow R(s, s')$. An *equivalence relation* is a reflexive, symmetric and transitive relation. Given an equivalence relation $R \subseteq S \times S$, we write $[s]_R$ for the *equivalence class* induced by s, that is, $[s]_R = \{ s' \in S \mid \langle s, s' \rangle \in R \}$. We denote the set of all such equivalence classes by S/R.

Functions A *function* is a total functional relation. We denote by $S_1 \to S_2$ the set of functions from S_1 to S_2, i.e. the set of total functional relations over S_1 and S_2. We call a functional but not necessarily total relation a *partial function*. All the notation and definitions for relations can thus be applied to functions as well. However, for clarity, we may also write the function $\{ \langle a, b \rangle, \langle c, d \rangle, \dots \}$ as $\{ a \mapsto b, c \mapsto d, \dots \}$, and we take $f(s)$ to denote the single element of the (relational) image of s when we focus on f as a function and not as a relation. A *bijection* between S_1 and S_2 is a bijective function in $S_1 \to S_2$. If $f_1 \in S_1 \to S_2$ and $f_2 \in S_2 \to S_3$, we denote by $f_2 \circ f_1$ the sequence of the two functions, i.e. the function $f \in S_1 \to S_3$ such that $f(s) = f_2(f_1(s))$. For $f \in S_1 \to S_2$, $x \in S_1$ and $y \in S_2$, we write $f[x \mapsto y]$ to denote $f|_{S_1 \setminus \{x\}} \cup \{ x \mapsto y \}$, i.e. the function f

with the image of x changed to y. This notation can be extended to sets, such that $f[X \mapsto y]$ for $X \subseteq S_1$ denotes $f|_{S_1 \setminus X} \cup \{ x \mapsto y \mid x \in X \}$.

2.2 Probability Theory

In order to formally define models that include probabilistic decisions, such as selecting an element of a finite or countable set with a certain probability or sampling a value from a continuous set, we need certain basic notions from probability theory which we define in the remainder of this section. Unless noted otherwise, these definitions are based on [HHHK13] and [GS01].

Probability Measures and Distributions

We want to use general definitions that cover phenomena where elements of finite, countable, and uncountable sets are selected with certain probabilities. In the uncountable case, the probability for any single element of the set to occur is usually zero. We thus need to use particular subsets to assign probabilities to as follows:

Definition 1 (σ-algebra, measurable space)**.** Given a set Ω (a *sample space*), a family Σ of subsets of Ω is a *σ-algebra over Ω* if $\Omega \in \Sigma$ and Σ is closed under complementation and countable union. In that case, a set $B \in \Sigma$ is called *measurable*, and the pair $\langle \Omega, \Sigma \rangle$ is called a *measurable space*. We say that a function $f \in \Omega_1 \to \Omega_2$ is *Σ_1-Σ_2-measurable* if every preimage of a measurable set is measurable, i.e. if $f^{-1}(B) \in \Sigma_1$ for all $B \in \Sigma_2$.

We also call the elements of Σ *events*. Given a family of sets $\mathscr{A}$, by $\sigma(\mathscr{A})$ we denote the σ-algebra *generated* by $\mathscr{A}$, that is the smallest σ-algebra containing all sets of $\mathscr{A}$. The *Borel σ-algebra* over Ω is generated by the open subsets of Ω, and it is denoted $\mathscr{B}(\Omega)$. We will also need the hit σ-algebra defined as follows:

Definition 2 (Hit σ-algebra [Wol12])**.** Given a σ-algebra Σ, we define the *hit σ-algebra* over Σ by

$$\mathscr{H}(\Sigma) \stackrel{\text{def}}{=} \sigma(\{ H_B \mid B \in \Sigma \}) \text{ where } H_B \stackrel{\text{def}}{=} \{ C \in \Sigma \mid C \cap B \neq \emptyset \}.$$

H_B consists of all measurable sets C which have a nonempty intersection with B. $\mathscr{H}(\Sigma)$ is then generated by all sets of sets $\{C_1, \ldots, C_n\}$ such that there is a set B which "hits" all C_i. On measurable spaces, we can now define functions that associate probabilities to events:

Definition 3 (Probability measure)**.** Given a measurable space $\langle \Omega, \Sigma \rangle$, a function $\mu \in \Sigma \to [0,1]$ is called σ-*additive* if $\mu(\cup_{i \in I} B_i) = \sum_{i \in I} \mu(B_i)$ for countable index sets I and pairwise disjoint sets $B_i \subseteq \Sigma$. We speak of a *probability measure* if additionally $\mu(\Omega) = 1$. In that case, $\langle \Omega, \Sigma, \mu \rangle$ is a *probability space*. We denote the set of probability measures on $\langle \Omega, \Sigma \rangle$ by $\mathrm{Prob}(\Omega, \Sigma)$, or just $\mathrm{Prob}(\Omega)$ when Σ is clear from the context.

We write $\mathscr{D}(s)$ for the *Dirac measure* for s, defined by $\mathscr{D}(s)(\omega) = 1 \Leftrightarrow s \in \omega$. When a probability space $\langle \Omega, \Sigma, \mu \rangle$ is assumed or clear from the context, we also use the notation $\mathbb{P}(e)$ where e describes an event $s \in \Sigma$ to refer to "the probability of e", i.e. $\mu(s)$.

In the models we define in chapters 3 to 5, we only need countable sample spaces. In this *discrete* setting, we can make the σ-algebra implicit and use the following definitions:

Definition 4 (Probability distribution)**.** A (discrete) *probability distribution* over a countable sample space Ω is a function $\mu \in \Omega \to [0,1]$ such that we have $\sum_{\omega \in \Omega} \mu(\omega) = 1$. The *support* of μ is $\mathrm{support}(\mu) \stackrel{\text{def}}{=} \{ s \in S \mid \mu(s) > 0 \}$. The corresponding probability space is $\langle \Omega, \mathscr{P}(\Omega), \mu_C \rangle$ where $\mu_C \in \mathscr{P}(\Omega) \to [0,1]$ with $\mu_C(S) = \sum_{s \in S} \mu(s)$. We denote by $\mathrm{Dist}(\Omega)$ the set of all probability distributions over Ω.

By implicitly falling back upon the corresponding probability measure, we can interchangeably treat probability distributions as functions over elements or subsets of Ω. In the other direction, we can treat e.g. the Dirac measure as a probability distribution, too.

Given two probability distributions $\mu, \mu' \in \mathrm{Dist}(\Omega)$ and an equivalence relation $R \subseteq \Omega \times \Omega$, we overload notation by writing $\langle \mu, \mu' \rangle \in R$ to denote that $\mu([s]_R) = \mu'([s]_R)$ for all $s \in S$. For a finite set $\Omega = \{ \omega_1, \ldots, \omega_n \}$, we denote by $\mathscr{U}(\Omega) = \{ \omega_1 \mapsto \frac{1}{n}, \ldots, \omega_n \mapsto \frac{1}{n} \}$ the *uniform distribution* over Ω. The *product* of two discrete probability distributions $\mu_1 \in \mathrm{Dist}(\Omega_1)$, $\mu_2 \in \mathrm{Dist}(\Omega_2)$ is determined by $(\mu_1 \otimes \mu_2)(\langle \omega_1, \omega_2 \rangle) = \mu_1(\omega_1) \cdot \mu_2(\omega_2)$.

Product Measures

While the definition of the product of two probability distributions was short and straightforward, a more involved construction is needed to achieve the same in a correct manner for probability measures. We first define the σ-algebra of probability distributions, which we need in the actual product definition later:

Definition 5 (σ-algebra of measures)**.** The set $\mathrm{Prob}(\Omega)$ of probability measures on $\langle \Omega, \Sigma \rangle$ can be endowed with the σ-algebra $\Delta(\Sigma)$ [Gir82] generated

by the measures that, when applied to $B \in \Sigma$, give a value greater than some $q \in \mathbb{Q} \cap [0,1]$:

$$\Delta(\Sigma) \stackrel{\text{def}}{=} \sigma(\{\, \Delta^{>q}(B) \mid B \in \Sigma \wedge q \in \mathbb{Q} \,\}) \text{ where } \Delta^{>q}(B) \stackrel{\text{def}}{=} \{\, \mu \mid \mu(B) > q \,\}.$$

Note that $\Delta(\Sigma)$ is a set of sets of probability measures. Together with $\text{Prob}(\Omega)$, it forms the measurable space $\langle \text{Prob}(\Omega), \Delta(\Sigma) \rangle$. We can now define the product of σ-algebras and then of probability measures:

Definition 6 (Product σ-algebra and measure)**.** Given a finite index set I and a family $(\Sigma_i)_{i \in I}$ of σ-algebras, the *product σ-algebra* $\bigotimes_{i \in I} \Sigma_i$ is defined as

$$\bigotimes_{i \in I} \Sigma_i \stackrel{\text{def}}{=} \sigma(\{\, \times_{i \in I} B_i \mid \forall\, i \in I \colon B_i \in \Sigma_i \,\}),$$

while for a family $(\mu_i)_{i \in I}$ of probability measures on Σ_i, the *product measure* is the uniquely defined probability measure $\bigotimes_{i \in I} \mu_i \in \Delta(\bigotimes_{i \in I} \Sigma_i)$ such that

$$(\bigotimes_{i \in I} \mu_i)(\times_{i \in I} B_i) \stackrel{\text{def}}{=} \prod_{i \in I} \mu_i(B_i) \text{ for all } B_i \in \Sigma_i,\ i \in I.$$

We can extend this definition to families of sets of measures $(M_i)_{i \in I}$ with $M_i \in \Delta(\Sigma_i)$ by

$$\bigotimes_{i \in I} M_i \stackrel{\text{def}}{=} \{\, \bigotimes_{i \in I} \mu_i \mid \mu_i \in M_i \text{ for all } i \in I \,\}.$$

We use $\otimes$ as an infix operator on two σ-algebras, probability measures or sets of probability measures.

Random Variables and Expectations

We are sometimes not so much interested in a particular probabilistic experiment itself but rather in some quantity derived from its outcomes. Such a quantity can be represented as a random variable:

Definition 7 (Random variable)**.** Given a measurable space $\langle \Omega, \Sigma \rangle$ and a probability measure $\mu \in \text{Prob}(\Omega, \Sigma)$, a Σ-$\mathscr{B}(\mathbb{R})$-measurable function $X \in \Omega \to \mathbb{R}$ is called a *random variable*. The *cumulative distribution function* (or cdf for short) of X is the function $F \in \mathbb{R} \to [0,1]$ given by

$$F(x) = \mathbb{P}(X \leq x) = \mu(\{\, \omega \in \Omega \mid X(\omega) \leq x \,\}).$$

F is a *discrete* random variable if $\text{Dom}(F)$ is countable. In that case, its *probability mass function* $f \in \mathbb{R} \to [0,1]$ is given by $f(x) = \mathbb{P}(X = x)$. F is a *continuous* random variable if its cdf can be expressed as

$$F(x) = \int_{-\infty}^{x} f(u)\, du$$

for $x \in \mathbb{R}$ and some integrable function $f \in \mathbb{R} \to \mathbb{R}_0^+$, which is called the *probability density function* (or pdf) of X.

For brevity, when we do not need operations such as addition for its domain, we may use discrete random variables that map to some arbitrary countable set S and implicitly assume a bijection to $\mathbb{R}$ to formally make the definitions above work. When we want to describe the evolution of a system that is subject to uncertainty, we may use stochastic processes:

Definition 8 (Stochastic process)**.** A *stochastic process* $(X)_{i\in I}$ is a collection of random variables for some index set I.

The index set I often represents a notion of time; in particular, we talk of *discrete time* if $I = \mathbb{N}$ and of *continuous time* if $I = \mathbb{R}_0^+$.

Example 1. Useful random variables include, for example, the Bernoulli variables: Let $\Omega = \{\omega_1, \omega_2\}$ and $\mu = \{\omega_1 \mapsto p, \omega_2 \mapsto 1-p\}$ for some parameter $p \in [0,1]$. μ is a probability distribution. Then $X = \{\omega_1 \mapsto 1, \omega_2 \mapsto 0\}$ is a *Bernoulli* random variable, and its cdf is

$$F(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1-p & \text{if } x \in [0,1) \\ 1 & \text{otherwise.} \end{cases}$$

We also call this cdf "the Bernoulli distribution".

When we perform a number of probabilistic experiments, we can compute the average of the outcomes if they are themselves numeric, or in any case the average of some given random variable. In the long run and for "well-behaved" random variables, we expect this average to tend to some particular *expected* value:

Definition 9 (Expected value)**.** Given a discrete random variable X with probability mass function f, its *expected value*, *expectation*, or *mean* is

$$\mathbb{E}(X) \stackrel{\text{def}}{=} \sum_{x\in\{y|f(y)>0\}} x \cdot f(x)$$

whenever this sum is absolutely convergent. If X is a continuous random variable with pdf f, then its expectation is

$$\mathbb{E}(X) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} x \cdot f(x)\, dx$$

whenever this integral exists.

Independence

The probability that some event A occurs may change when another event B occurs. In case that the two events do not influence each other, we say that they are independent. Formally, these phenomena can be defined as follows:

Definition 10 (Conditional probability, independence)**.** The *conditional probability* that $A \in \Sigma$ occurs given that $B \in \Sigma$ has occurred is

$$\mathbb{P}(A \mid B) \stackrel{\text{def}}{=} \mathbb{P}(A \cap B)/\mathbb{P}(B).$$

A family $(A_i)_{i \in I}$ of events is *independent* if

$$\mathbb{P}(\bigcap_{i \in J} A_i) = \prod_{i \in J} \mathbb{P}(A_i)$$

for all finite subsets $J \subseteq I$.

The notion of independence generalises to random variables: Two discrete variables X and Y are independent if, for all x and y, the events $\{\, \omega \in \Omega \mid X(\omega) = x \,\}$ and $\{\, \omega \in \Omega \mid Y(\omega) = y \,\}$ are independent. Likewise, two continuous variables X and Y are independent if, for all x and $y \in \mathbb{R}$, the events $\{\, \omega \in \Omega \mid X(\omega) \leq x \,\}$ and $\{\, \omega \in \Omega \mid Y(\omega) \leq y \,\}$ are independent.

2.3 Variables and Expressions

We define several *symbolic* models in this thesis whose syntax makes use of variables, expressions and various other derived notions such as assignments. In this section, we give abstract definitions of these notions and in particular introduce different classes of expressions (again based on [HHHK13]) that will be used in different parts of the symbolic models.

Variables and Valuations

A *variable* x is an object that has an associated domain (or *type*) $\mathrm{Dom}(x)$ with an (implicit) σ-algebra $\mathrm{Dom}_\Sigma(x) \subseteq \mathscr{P}(\mathrm{Dom}(x))$. We typically use the symbol *Var* to denote (finite) sets of variables. Common types of variables include Boolean variables, with $\mathrm{Dom}(x) = \{\mathit{true}, \mathit{false}\}$ and $\mathrm{Dom}_\Sigma(x) = \sigma(\{\{\mathit{true}\}, \{\mathit{false}\}\})$; integer variables, where $\mathrm{Dom}(x) = \mathbb{Z}$ and $\mathrm{Dom}_\Sigma(x) = \mathscr{P}(\mathbb{Z})$; *bounded integers*, with $\mathrm{Dom}(x) = \{a, \ldots, b\}$ for $a, b \in \mathbb{Z}$ with $a \leq b$ and $\mathrm{Dom}_\Sigma(x) = \mathscr{P}(\mathrm{Dom}(x))$; and real variables, where $\mathrm{Dom}(x) = \mathbb{R}$ and $\mathrm{Dom}_\Sigma(x) = \mathscr{B}(\mathbb{R})$.

For a set of variables *Var*, we let $\mathrm{Val}(\mathit{Var})$ denote the set of variable *valuations*, that is of functions $\mathit{Var} \to \bigcup_{x \in \mathit{Var}} \mathrm{Dom}(x)$ that map variables to values such that $v \in \mathrm{Val}(\mathit{Var}) \Rightarrow \forall x \in \mathit{Var}\colon v(x) \in \mathrm{Dom}(x)$. When *Var* is clear from the context, we may write *Val* in place of $\mathrm{Val}(\mathit{Var})$. If *Var* is finite and we are given an ordering on the variables (or we assume one), we can represent a valuation v as a tuple whose i-th component is the value of the i-th variable according to v. Two valuations $v_1 \in \mathrm{Val}(\mathit{Var}_1)$ and $v_2 \in \mathrm{Val}(\mathit{Var}_2)$ are *consistent* if $x \in \mathit{Var}_1 \cap \mathit{Var}_2 \Rightarrow v_1(x) = v_2(x)$. Then their union $v_1 \cup v_2$ is a valuation in $\mathrm{Val}(\mathit{Var}_1 \cup \mathit{Var}_2)$.

Expressions

Given a set of variables *Var*, by $\mathrm{Exp}(\mathit{Var})$ we denote the set of *expressions* over the variables in *Var*. When *Var* is clear from the context, we may write *Exp* in place of $\mathrm{Exp}(\mathit{Var})$. In this thesis, we treat expressions in an abstract manner: We assume a standard expression *syntax* as in e.g. the C programming language [ISO11] with a few extensions that we explicitly describe where necessary, but we formally work mostly with the *semantics* of expressions as functions that take a valuation over the relevant variables and return some kind of value depending on the expression class and the type of model the expression is used in. For an expression e, we denote its function semantics by $[\![e]\!]$, and consequently write its evaluation for a given valuation v as $[\![e]\!](v)$. Just like a variable has a domain, an expression has a type $t = \mathit{type}(e)$ with an assumed corresponding σ-algebra Σ_t.

We distinguish three important subsets of *Exp* based on the expressions' types, whether subexpressions with nondeterministic values (using the any operator) are allowed—we call such expressions nondeterministic—and whether sampling of values according to probability measures is possible. The three subsets are

– *Sxp*: *sampling expressions* that may be nondeterministic and use sampling, for example
$$x + \textsc{Uniform}(0,2) + \mathsf{any}(y \mid x + y == z)$$
where $\textsc{Uniform}(x,y)$ denotes sampling from the continuous uniform distribution over the interval $[x,y]$ and $\mathsf{any}(x \mid e)$ nondeterministically selects a value y such that $e[x/y]$ (see below) evaluates to *true* in the current valuation;
– *Bxp*: simple *Boolean expressions* such as $i = 1$ that contain neither nondeterminism nor sampling and have type $\mathbb{B}$; and
– *Axp*: *arithmetic expressions* such as $2.5 + x$ or $\texttt{ceil}(y)$, which evaluate to values in $\mathbb{R}$ and, like Boolean expressions, contain no nondeterminism and no sampling.

As for *Exp*, we may write $\mathrm{Sxp}(\mathit{Var})$ or just *Sxp* and so on for all the expression classes depending on whether *Var* is clear from the context. We denote by $e[x/e']$ the replacement of all occurrences of variable x in the syntax of the expression e by the expression e'. Formally, this means that
$$\forall v \in \mathit{Val}\colon [\![e[x/e']]\!](v) = [\![e]\!](v[x \mapsto [\![e']\!](v)]).$$
For Chapters 3 to 5, we restrict sampling expressions to contain *no* nondeterminism and *not* use sampling. The expression semantics for these chapters can thus formally be defined as follows:

Definition 11 (Deterministic expression semantics)**.** The semantics $[\![e]\!]$ of an expression e over variables in *Var* with $\mathit{type}(e) = t$ is such that $[\![e]\!] \in \mathit{Val} \to t$.

In Chapter 6, we for the first time allow expressions in *Sxp* to be nondeterministic and actually contain sampling. From that point on, we thus need to use the following expression semantics:

Definition 12 (Stochastic-nondeterministic expression semantics)**.** The semantics $[\![e]\!]$ of an expression e over variables in *Var* with $type(e) = t$ is such that
– $[\![e]\!] \in Val \to \Delta(\Sigma_t)$ is a Σ_{Var}-$\mathscr{H}(\Delta(\Sigma_t))$-measurable function if $e \in Sxp$,
– $[\![e]\!] \in Val \to \mathbb{B}$ is a Σ_{Var}-$\mathscr{P}(\mathbb{B})$-measurable function if $e \in Bxp$, and
– $[\![e]\!] \in Val \to \mathbb{R}$ is a Σ_{Var}-$\mathscr{B}(\mathbb{R})$-measurable function if $e \in Axp$
where $\Sigma_{Var} = \bigotimes_{x \in Var} \mathrm{Dom}_\Sigma(x)$ is the product σ-algebra of the variable domains.

The semantics of an expression in *Sxp* maps a variable valuation to a set of measures over values, thereby combining a nondeterministic selection (of an element of the set) and a stochastic one (via the selected measure). The restrictions to measurable functions (here and in the remainder of this chapter) are technical requirements that we do not dwell on further in this thesis. They are of particular importance when it comes to modelling stochastic hybrid systems; see e.g. [FHH+11, HHHK13] for more detailed explorations of these issues.

Assignments and Consistency

An *assignment*, which we can write as $x := e$, is formally a pair $\langle x, e \rangle$ of a variable x and a sampling expression e. The set of assignments to variables in *Var* is $\mathrm{Asgn}(Var) = Var \times \mathrm{Sxp}(Var)$, or just *Asgn* if *Var* is clear from the context, such that if $\langle x, e \rangle \in Asgn$, then for all valuations $v \in Var$ either $[\![e]\!](v) \in \mathrm{Dom}(x)$ in the deterministic case or $[\![e]\!](v) \in \Delta(\Sigma_t)$ in the stochastic-nondeterministic case with $t = type(e)$. Two assignments $\langle x_1, e_1 \rangle$ and $\langle x_2, e_2 \rangle$ are *consistent* if $x_1 \neq x_2$ or $[\![e_1]\!](v) = [\![e_2]\!](v)$ for all valuations v. A finite set of pairwise consistent assignments is called an (atomic) *update*, and two updates are consistent if their union is an update. The set of all updates to variables in *Var* is $\mathrm{Upd}(Var)$, or just *Upd* when *Var* is clear from the context. We can identify the assignment u with the update $\{u\}$. Due to consistency, we can also treat an update $U = \{\langle x_1, e_1 \rangle, \ldots, \langle x_n, e_n \rangle\}$ consisting of $n \in \mathbb{N}$ assignments where possibly $x_i = x_j$ for some $i \neq j$ as a function $U \in Var \to Sxp$ with

$$U = \{\langle x, e \rangle \mid (\exists i \colon x = x_i \wedge e = e_i \wedge \nexists j < i \colon x = x_j) \oplus (e = x \wedge \nexists i \colon x = x_i)\}$$

where $\oplus$ denotes the exclusive disjunction operator. This merely means that we ignore assignments that are syntactically different, but semantically equivalent.

For the deterministic assignments of Chapters 3 to 5, the formal semantics of an update U (and thus also of a single assignment u via its corresponding update) can simply be defined as follows:

Definition 13 (Deterministic assignment semantics)**.** The semantics $[\![U]\!]$ of update U is such that $[\![U]\!] \in \mathit{Val} \to \mathit{Val}$, and it is defined by $[\![U]\!](v)(x) \stackrel{\text{def}}{=} [\![U(x)]\!](v)$.

For Chapter 6 and beyond, where expressions in *Sxp* may contain nondeterminism and sampling, the semantics of an update U combines all the possible distributions to the variables:

Definition 14 (Stochastic-nondeterministic assignment semantics)**.** The semantics $[\![U]\!]$ of an update U is such that $[\![U]\!] \in \mathit{Val} \to \Delta(\Sigma_{\mathit{Var}})$, and it is defined by $[\![U]\!](v) \stackrel{\text{def}}{=} \bigotimes_{x \in \mathit{Var}} [\![U(x)]\!](v)$.

An element of Σ_{Var} is a set of $|\mathit{Var}|$-tuples and thus corresponds to a set of valuations. The semantics of an update thus maps a valuation to a set of probability measures over valuations. The measurability restrictions on the expressions guarantee that the semantics of an assignment is Σ_{Var}-$\mathscr{H}(\Delta(\Sigma_{\mathit{Var}}))$-measurable.

The set of variables of an expression, assignment or update y is

$$\mathrm{Var}(y) = \{\, x \in \mathit{Var} \mid \exists v \in \mathit{Val}, z \in \mathrm{Dom}(x) \colon [\![y]\!](v) \neq [\![y]\!](v[x \mapsto z]) \,\},$$

that is the set of variables that $[\![y]\!]$ depends on. It is usually a subset of the variables that appear in y on the syntactical level.

3 Basic Models

At the very foundation of the automata-based formalisms used in this thesis as outlined in Figure 1.2 are the two concepts of nondeterminism and probabilistic choices. The corresponding formalisms are labelled transition systems (LTS) and discrete-time Markov chains (DTMC). In this chapter, we define both formally. For each, we introduce the corresponding subset of the MODEST modelling language and review the standard ways to specify and verify properties. We show how to analyse reachability properties on LTS using exhaustive and stateless model checking as well as randomised testing. For DTMC, we focus on probabilistic reachability properties. We show how to compute the corresponding probabilities or compare them to fixed bounds. This can be achieved through exhaustive probabilistic model checking as well as by using simulation to perform statistical model checking.

In order to support compositional modelling on the level of automata, i.e. the creation of complex models from smaller interacting components, we define a parallel composition operator for each formalism. Parallel composition allows the specification of networks of automata. After having defined the two basic models of this thesis, we therefore briefly explore the concept of compositionality in general terms.

3.1 Labelled Transition Systems

Labelled transition systems are the fundamental nondeterministic modelling formalism. An LTS consists of a set of *states*, which are connected by *transitions*. Transitions in turn are *labelled* with elements from a given *alphabet*. These labels are usually called *actions*.

States are opaque and can be any kind of concrete mathematical object that represents a “snapshot”, or *configuration*, of the system under study in a more or less abstract fashion. Ideally, the set of states—and the underlying class of mathematical objects—is chosen such that it is as small as possible without

mapping two concrete configurations to the same state when they would, in reality, lead to relevant differences in behaviour.

Transitions connect states. They model the behaviour of the system as it evolves from state to state over time. One state can have any number of outgoing transitions. The nondeterminism of LTS is in the choice of transition to take from a state, each of which in turn leads to a single next state. The transition labels are used for communication between the component LTS in a parallel composition (see Section 3.1.1 below).

Additionally, we equip each LTS with a set of *atomic propositions* and a *labelling function* that assigns atomic propositions to states. The propositions that this labelling assigns define what is "visible" of the otherwise opaque states. In properties, we will use atomic propositions to refer to (sets of) states.

Definition

Our following definition of LTS is largely inspired by the way transition systems are defined in the excellent textbook by Baier and Katoen [BK08], which provides a very detailed description of LTS, the associated classes of properties and (exhaustive) model checking approaches. In this section, we give a comparatively compact overview of these aspects as far as they are relevant for the remainder of this thesis.

Definition 15 (LTS)**.** A *labelled transition system* (LTS) is a 6-tuple

$$\langle S, A, T, s_{init}, AP, L \rangle$$

where
– S is a countable set of states,
– $A \supseteq \{\tau\}$ is the system's alphabet, a countable set of transition labels (or *actions*) that includes the *silent action* τ,
– $T \in S \to \mathscr{P}(A \times S)$ is the transition function,
– $s_{init} \in S$ is the initial state,
– AP is a set of atomic propositions, and
– $L \in S \to \mathscr{P}(AP)$ is the state labelling function.

Note that what we intuitively described as states having a number of outgoing labelled transitions is implemented by assigning a *set* of action/successor-state pairs $\langle a, s' \rangle$ to each state s. In the remainder of this thesis, we require all models that have a transition (or edge) function whose domain is some powerset to be *finitely branching* unless we say otherwise, i.e. we restrict to functions where $|T(s)| \in \mathbb{N}$ for all states s (and analogously for edges and locations later on).

For a given LTS M with transition function T, we will also write $s \xrightarrow{a} s'$ instead of $\langle a, s' \rangle \in T(s)$, or say that "$M$ has a transition labelled a from s to s'."

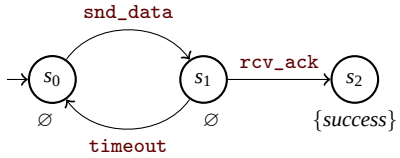


Figure 3.1: An LTS model of a sender in a simple communication protocol

In case we are only interested in the successor states and not in the transition labels, we also use the abbreviations $T_a(s)$ to denote the set $\{ s' \mid \langle a, s' \rangle \in T(s) \}$ for some $a \in A$, and $T_*(s)$ for $\cup_{a \in A} T_a(s)$. If $T(s) = \varnothing$, we say that s is *deadlocked* or a *deadlock state*. We call a self-loop, i.e. the case where $s \in T_*(s)$, simply a *loop*, and the possibility to return to some state via one or more transitions is a *cycle*.

In this thesis, an important aspect of models specified in a formalism that allows nondeterministic choices is whether the model at hand actually contains any such choices:

Definition 16 (Deterministic LTS)**.** An LTS $M = \langle S, A, T, s_{init}, AP, L \rangle$ is *deterministic* if and only if $|T(s)| \in \{0, 1\}$ for all $s \in S$, and nondeterministic otherwise.

Depending on context, an LTS may also be called a *Kripke structure* (usually when parallel composition is not considered and the transition labels can thus be left out) or a *finite automaton* (when it is primarily its language, i.e. the set of action traces (see below), that is of interest).

Example 2. A very abstract LTS model of the sending component in a simple communication protocol is shown in Figure 3.1: The sender first tries to send data via an unspecified communication channel to an equally unspecified receiver. This is modelled by action `snd_data`. As the channel may lose messages, it then waits for either a `timeout`, which causes the data to be resent, or the receipt of an acknowledgment from the receiver through action `rcv_ack`. On receiving the acknowledgment, the sender moves to a deadlocked state. That state is labelled to indicate the successful transmission. Formally, this LTS can be written as

$$M_{Snd} = \langle \{ s_0, s_1, s_2 \}, \{ \texttt{snd\_data}, \texttt{timeout}, \texttt{rcv\_ack} \}, T_{Snd}, s_0, \{ \text{success} \}, L_{Snd} \rangle$$

where the transition function T_{Snd} is given by $T_{Snd}(s_0) = \{(\texttt{snd\_data}, s_1)\}$, $T_{Snd}(s_1) = \{(\texttt{timeout}, s_0), (\texttt{rcv\_ack}, s_2)\}$ and $T_{Snd}(s_2) = \varnothing$, and the state labelling function is given by $L_{Snd}(s_0) = L_{Snd}(s_1) = \varnothing$ and $L_{Snd}(s_2) = \{success\}$. M_{Snd} is nondeterministic. It has no loops, one cycle, and one deadlock state (s_2).

Paths

The individual behaviours of an LTS—where the nondeterministic choices regarding which transition to take next have been resolved—can be represented as *paths*. Formally,

Definition 17 (Finite paths in LTS)**.** Given $M = \langle S, A, T, s_{init}, AP, L\rangle$, a (finite) *path in M from s_0 to s_n* of length $n \in \mathbb{N}$ is a finite sequence $s_0\, a_1\, s_1\, a_2\, s_2 \ldots a_n\, s_n$ where $s_i \in S$ for all $i \in \{0, \ldots, n\}$ and $a_i \in A \wedge s_{i-1} \xrightarrow{a_i} s_i$ for all $i \in \{1, \ldots, n\}$.

The length of a path π is denoted $|\pi|$, its last state is $\mathrm{last}(\pi)$. For $s, s' \in S$, $\mathrm{Paths}(s, s')$ is the set of all paths in M from s to s'. If $\mathrm{Paths}(s, s')$ is non-empty, we say that s' is *reachable* from s. For an entire LTS, $s \in S$ is reachable in M if $\mathrm{Paths}(s_{init}, s) \neq \varnothing$, and $\mathrm{Reach}(M) = \{s \in S \mid \mathrm{Paths}(s_{init}, s) \neq \varnothing\}$ is the set of reachable states of M. We refer to the set of all finite paths of M as $\mathrm{Paths}_{\mathrm{fin}}(M) \stackrel{\mathrm{def}}{=} \cup_{s \in S} \mathrm{Paths}(s_{init}, s)$.

Definition 18 (Infinite paths in LTS)**.** Given an LTS $M = \langle S, A, T, s_{init}, AP, L\rangle$, an (infinite) *path in M starting from s_0* is an infinite sequence $s_0\, a_1\, s_1\, a_2\, s_2 \ldots$ where $s_i \in S$ for all $i \in \mathbb{N}$ and $a_i \in A \wedge s_{i-1} \xrightarrow{a_i} s_i$ for all $i \in \mathbb{N}^+$.

For $s \in S$, $\mathrm{Paths}(s)$ is the union of the set of all paths in M starting from s and the sets $\mathrm{Paths}(s, s')$ for all deadlock states $s \in S$. We call the paths in $\mathrm{Paths}(s)$ *maximal*. For the entire LTS M as above, we define its set of paths as $\mathrm{Paths}(M) = \mathrm{Paths}(s_{init})$, i.e. the maximal paths starting in its initial state.

Example 3. In the simple sender LTS given as Example 2, there is one infinite path (which continuously cycles between s_0 and s_1) and infinitely many maximal finite paths (which all end in deadlock state s_2). As examples for path sets (using ω-regular expression notation [BK08, Section 4.3.1] where convenient), we have
– $\mathrm{Paths}(s_0, s_1) = [\![s_0\ \texttt{snd\_data}\ s_1\ (\texttt{timeout}\ s_0\ \texttt{snd\_data}\ s_1)^*]\!]$ and
– $\mathrm{Paths}(M_{Snd}) = [\![s_0\ \texttt{snd\_data}\ s_1\ (\texttt{timeout}\ s_0\ \texttt{snd\_data}\ s_1)^\omega]\!] \cup \mathrm{Paths}(s_0, s_2)$.

Traces

As the states of an LTS are opaque objects, paths contain more information than necessary e.g. for the verification of properties or the definition of the language

of an LTS seen as a finite automaton. Instead, *traces* are used, which represent the projections of paths on either the actions or the atomic propositions corresponding to the states:

Definition 19 (Traces in LTS)**.** Given an LTS $M = \langle S, A, T, s_{init}, AP, L\rangle$ and a (finite or infinite) path $\pi = s_0\, a_1\, s_1\, a_2\, s_2 \ldots$, the *trace* of the path π is defined as $\mathrm{trace}(\pi) \stackrel{\text{def}}{=} L(s_0)\, L(s_1)\, L(s_2) \ldots$, and its *action trace* is $\mathrm{atrace}(\pi) \stackrel{\text{def}}{=} a_1\, a_2 \ldots$.

The sets of paths $\mathrm{Paths}(s, s')$, $\mathrm{Paths}(s)$ and $\mathrm{Paths}(M)$ can be lifted to the sets of traces $\mathrm{Tr}(s, s')$, $\mathrm{Tr}(s)$ and $\mathrm{Tr}(M)$ by applying the $\mathrm{trace}(\cdot)$ operation to all their elements. Since we focus on state-based verification in this thesis and consequently use transition labels merely for parallel composition, we actually need (the atomic proposition-based) traces only. We note that when an LTS is seen as a finite automaton where all states are accepting, then the set of action traces is the *language* of the automaton.

Example 4. The sets of traces corresponding to the sets of paths given in Example 3 are $\mathrm{Tr}(s_0, s_1) = [\![\varnothing^*]\!]$ and $\mathrm{Tr}(M_{Snd}) = [\![\varnothing^\omega \mid \varnothing^*\{\mathrm{success}\}]\!]$.

3.1.1 Parallel Composition

Modelling a large, complex system as a single, atomic LTS quickly becomes an exercise in frustration. We need a way to build a complex LTS out of smaller, manageable parts. A common approach in modelling reactive systems is therefore to separate the model into a number of components that execute in parallel and communicate only via well-defined interfaces.

For LTS, this approach can be realised with a *parallel composition* operation that takes two (component) LTS and gives a well-defined meaning for the parallel execution and communication. The notion of parallel composition was originally introduced with process algebras, in particular CCS [Mil80] and CSP [Hoa85]. There are various ways to define a concrete parallel composition operator w.r.t. how the parallel components communicate. CCS, for example, provides for *binary* communication between two processes only, which naturally includes a notion of "direction" resp. of a sender and a receiver. This style of communicating has been adopted in current tools like UPPAAL, which additionally provides for a *broadcast* extension that allows multiple receivers for a single sender. CSP's *multi-way* communication, on the other hand, makes no distinction between the communicating parties, and has been adopted by languages like LOTOS [BB87], the PRISM model checker's [KNP11] guarded commands, and MODEST [BDHK06]. The latter is why, in the remainder of this thesis, we use CSP-style parallel composition operators.

In the operator that we now introduce for LTS, it is the transition labels that define the communication structure: Transitions bearing a label that is in the intersection of both components' alphabets (the *shared alphabet*, which does not include τ) must synchronise, i.e. when one component performs such a transition, the other component must also do so at the same time. For all other transitions, an *interleaving semantics* is used to represent the asynchronous parallel execution: When each component LTS can perform some non-synchronising transitions, then it is not specified in which order the two components do so. Instead, a nondeterministic choice between the available transitions of both components is used. The result of this semantics is that the parallel composition represents all possible orderings of transitions of the two components as long as no synchronisation takes place. Intuitively, this means that the relative execution speeds of the components are unspecified. Verification results for a parallel composition are thus independent of the actual way the asynchronous execution is implemented. For example, it does not matter if the components run as threads on the same processor with fair scheduling, or if alternatively they run on different machines of different execution speeds.

Formally, the parallel composition operator $\|$ for LTS is defined as follows:

Definition 20 (Parallel composition of LTS)**.** The *parallel composition of two LTS* $M_i = \langle S_i, A_i, T_i, s_{init_i}, AP_i, L_i \rangle$, $i \in \{1,2\}$, is the LTS

$$M_1 \parallel M_2 = \langle S_1 \times S_2, A_1 \cup A_2, T, \langle s_{init_1}, s_{init_2} \rangle, AP_1 \cup AP_2, L \rangle$$

where

– $T \in (S_1 \times S_2) \to \mathscr{P}((A_1 \cup A_2) \times (S_1 \times S_2))$
  s.t. $\langle a, \langle s_1', s_2' \rangle \rangle \in T(\langle s_1, s_2 \rangle) \Leftrightarrow a \notin B \wedge \langle a, s_1' \rangle \in T_1(s_1) \wedge s_2 = s_2'$
  $\vee\ a \notin B \wedge \langle a, s_2' \rangle \in T_2(s_2) \wedge s_1 = s_1'$
  $\vee\ a \in B \wedge \langle a, s_1' \rangle \in T_1(s_1) \wedge \langle a, s_2' \rangle \in T_2(s_2)$
  with $B = (A_1 \cap A_2) \setminus \{\tau\}$, and

– $L \in (S_1 \times S_2) \to \mathscr{P}(AP_1 \cup AP_2)$ s.t. $L(\langle s_1, s_2 \rangle) = L_1(s_1) \cup L_2(s_2)$.

We call the transitions resulting from the first two cases of the definition of T *interleaved*, while the others are *synchronising* transitions. We also say that the two LTS *synchronise on* labels in the shared alphabet.

Observe the role that nondeterministic choices play in this definition of parallel composition: On the one hand, they are essential to realise the interleaving semantics. In this way, parallel composition may introduce additional nondeterministic choices into a model. Even if the components were deterministic to start with, their composition can thus be nondeterministic. On the other hand, nondeterministic choices can be removed by the synchronisation mechanism: transitions may be disabled when the synchronisation partner is

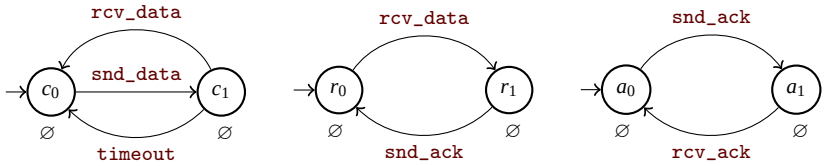


Figure 3.2: LTS for communication channels (left, right) and receiver (middle)

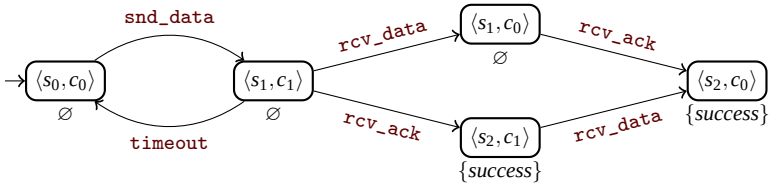


Figure 3.3: LTS for the parallel composition of sender and left channel

not ready. Thus, the composition of two nondeterministic LTS can also turn out to be deterministic.

Example 5. Let us revisit the LTS modelling the sender in a simple communication protocol scenario introduced in the previous examples. The as yet unspecified components of this model are, as mentioned, the communication channel and the receiver. It is natural to model them as separate LTS and use parallel composition to combine them.

Figure 3.2 shows the individual components. We actually use two communication channels: The one on the left models the transmission of data from sender to receiver, while the one on the right takes acknowledgments back from receiver to sender. We see that message loss is included in the data channel model by a nondeterministic choice between a transition with label `rcv_data`, which would hand the data to the receiver, and a transition with label `timeout`, which indicates message loss to the sender. For simplicity, we do not allow the loss of acknowledgments in the other channel.

The parallel composition of sender (Figure 3.1) and data channel (Figure 3.2) is shown in Figure 3.3. We see that the two LTS synchronise on `snd_data` and `timeout`, while the transitions labelled `rcv_data` and `rcv_ack` are interleaved—they will later synchronise with the receiver and the acknowledgment channel. For now, however, they form a classic *interleaving diamond*:

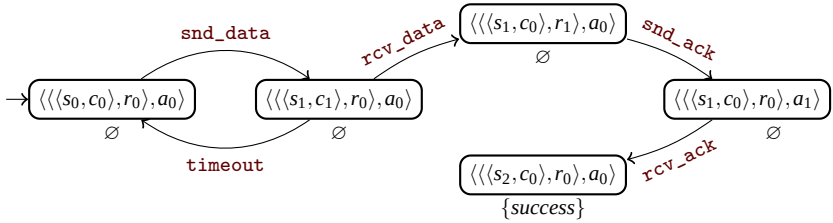


Figure 3.4: Parallel composition of sender, receiver and both channels

Aside from not synchronising, they do not disable each other and they are commutative (i.e. both execution orders end in the same state). This is an example where parallel composition introduces new nondeterminism.

Finally, we see the parallel composition of all four components in Figure 3.4. Here, the nondeterministic choices introduced in the intermediate LTS have been resolved again. All remaining nondeterminism was already present in the sender LTS. The composition with sender and receiver has therefore removed nondeterministic choices.

3.1.2 Variables

Even though the availability of a parallel composition operation is a significant improvement in ease of modelling, parallel LTS in their pure form remain a poor choice in practice: realistic systems typically have a large number of possible configurations that would need to be explicitly represented as opaque states.

A solution to this problem is to add a level of abstraction on top of LTS in the form of labelled transition systems *with variables* (VLTS), which have a semantics in terms of LTS. VLTS retain the basic labelled transition system structure in that they contain *locations* (instead of states) and labelled *edges* (instead of transitions). Additionally, they include a set of *variables* that can be used in

– *guards* on edges, which specify when an edge is enabled and the corresponding transition is thus present in the underlying LTS semantics; in
– *updates*, to be executed when an edge is taken in order to modify not only the current location, but also the values of the variables; and in
– *visible expressions*, which represent the aspects of the system visible to properties and thus replace the labelling with atomic propositions of LTS.

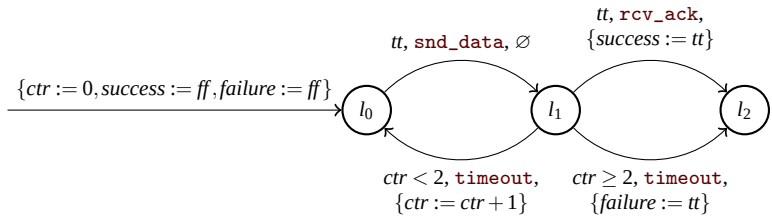


Figure 3.5: A VLTS model of a sender in a bounded retransmission protocol

Definition

Formally, we define VLTS as follows:

Definition 21 (VLTS)**.** A *labelled transition system with variables* (VLTS) is a 7-tuple $\langle Loc, Var, A, E, l_{init}, V_{init}, VExp \rangle$ where
– Loc is a countable set of locations,
– Var is a finite set of variables with countable domains,
– $A \supseteq \{\tau\}$ is the system's alphabet,
– $E \in Loc \to \mathscr{P}(Bxp \times A \times Upd \times Loc)$ is the edge function, which maps each location to a set of edges, which in turn consist of a guard, a label, an update and a target location,
– $l_{init} \in Loc$ is the initial location,
– $V_{init} \in \mathrm{Val}(Var)$ is the initial valuation of the variables, and
– $VExp \subseteq Bxp$ is the set of visible expressions.

For a given VLTS M with edge function E, we also write $l \xrightarrow{g,a,U} l'$ instead of $\langle g, a, U, l' \rangle \in E(l)$ in the remainder of this thesis.

Example 6. Let us return to the sender in our simple communication protocol built in the previous examples. We now want to remove the infinite path where a timeout occurs, the data is resent, another timeout occurs, the data is resent again, and so on. Since we cannot realistically assume that messages may not get lost and thus no timeouts will occur (although the current channel model is unrealistically *unfair* in also allowing all messages ever sent to be lost), we make the sender simply give up and report failure after a certain number of timeouts. We concisely model this as a VLTS with an integer variable *ctr* to count timeouts as shown in Figure 3.5. In this model, the sender gives up after its transmission has timed out for the third time. To be able to observe whether the sender succeeds or gives up, we include two Boolean variables *success* and *failure* and use $\{success, failure\}$ as the set of visible expressions. In terms of notation, we represent the initial valuation by an update on the arrow leading

into the initial location. In the remainder of this thesis, we omit constant *true* guards and empty assignment sets when drawing automata with variables.

Semantics

The LTS semantics of a VLTS can be obtained by using pairs of locations and variable valuations as states and evaluating the edges' guards and updates on these valuations:

Definition 22 (Semantics of VLTS)**.** The *semantics of a VLTS*

$$M = \langle Loc, Var, A, E, l_{init}, V_{init}, VExp \rangle$$

is the LTS

$$[\![M]\!] = \langle Loc \times Val, A, T, \langle l_{init}, V_{init} \rangle, VExp, L \rangle$$

where

– $T \in Loc \times Val \to \mathscr{P}(A \times (Loc \times Val))$ such that

$$\langle a, \langle l', v' \rangle \rangle \in T(\langle l, v \rangle) \Leftrightarrow \exists \langle g, a, U, l' \rangle \in E(l) \colon [\![g]\!](v) \wedge v' = [\![U]\!](v),$$

and

– $L \in Loc \times Val \to \mathscr{P}(VExp)$ such that, for all $\langle l, v \rangle \in Loc \times Val$, we have that $L(\langle l, v \rangle) = \{ e \in VExp \mid [\![e]\!](v) \}$.

It directly follows from this definition that the number of states and thus the worst-case number of reachable states of the LTS corresponding to a VLTS as above is $|Loc| \cdot \prod_{v \in Var} |\mathrm{Dom}(v)|$. Liberal use of variables with large domains can thus quickly lead to state space explosion.

Example 7. The semantics of the VLTS modelling the sender with retransmission bound 3 from the previous example is shown in Figure 3.6. Valuations are written as tuples for variable order $\langle ctr, success, failure \rangle$.

Parallel Composition

As for plain LTS, we can define a parallel composition operator for VLTS. This operator has the same effect on the graph structure (of locations and edges) as the one for LTS (on states and transitions), including synchronisation on the shared alphabet. However, symbolic operations on the expression level are needed to combine guards and assignments:

Definition 23 (Parallel composition of VLTS)**.** The *parallel composition of two consistent VLTS* $M_i = \langle Loc_i, Var_i, A_i, E_i, l_{init_i}, V_{init_i}, VExp_i \rangle$, $i \in \{1, 2\}$, is the VLTS

$$M_1 \parallel M_2 = \langle Loc_1 \times Loc_2, Var_1 \cup Var_2, A_1 \cup A_2, E, \\ \langle l_{init_1}, l_{init_2} \rangle, V_{init_1} \cup V_{init_2}, VExp_1 \cup VExp_2 \rangle$$

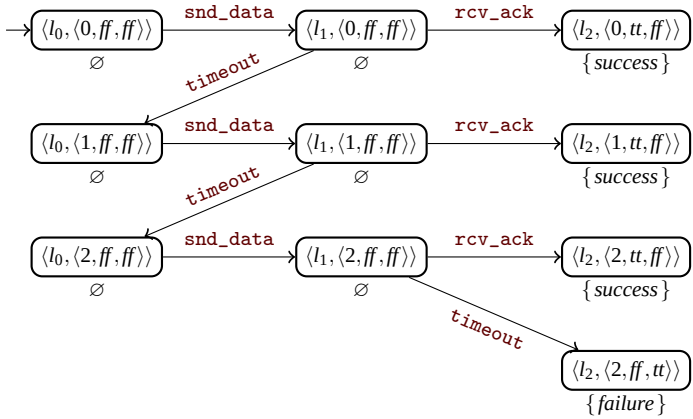


Figure 3.6: LTS semantics of the VLTS for the bounded retransmission sender

where $E \in (Loc_1 \times Loc_2) \to \mathscr{P}(Bxp \times A_1 \cup A_2 \times Upd \times (Loc_1 \times Loc_2))$ s.t.

$$\begin{aligned}\langle g,a,U,\langle l_1',l_2'\rangle\rangle \in E(\langle l_1,l_2\rangle) \Leftrightarrow\ & a \notin B \wedge \langle g,a,U,l_1'\rangle \in E_1(l_1) \wedge l_2 = l_2' \\ \vee\ & a \notin B \wedge \langle g,a,U,l_2'\rangle \in E_2(l_2) \wedge l_1 = l_1' \\ \vee\ & a \in B \wedge \exists g_1,g_2,U_1,U_2 : \\ & \quad \langle g_1,a,U_1,l_1'\rangle \in E_1(l_1) \\ & \wedge\ \langle g_2,a,U_2,l_2'\rangle \in E_2(l_2) \\ & \wedge\ (g = g_1 \wedge g_2) \wedge (U = U_1 \cup U_2)\end{aligned}$$

with $B = (A_1 \cap A_2) \setminus \{\tau\}$.

We do not require that $Var_1 \cap Var_2 = \varnothing$, i.e. we allow shared variables. This is why we need the two VLTS to be consistent:

Definition 24 (Consistent VLTS)**.** Two VLTS M_i as in the previous definition are consistent if their initial valuations are consistent and, for all $l_1 \in Loc_1$, $l_2 \in Loc_2$ and $a \in (A_1 \cap A_2) \setminus \{\tau\}$ we have that

$$U_1 \in f_1(l_1,a) \wedge U_2 \in f_2(l_2,a) \Rightarrow U_1 \text{ and } U_2 \text{ are consistent}$$

where $f_i(l,a) \stackrel{\text{def}}{=} \{U \mid \langle g,a,U,l'\rangle \in E_i(l)\}$ are the sets of all updates on the edges labelled with a from a location l.

Consistency as defined above is a sufficient condition to ensure the absence of concurrent assignments to shared variables that assign the same values. Alternative definitions that keep this property but classify more pairs of VLTS as consistent are possible, for example by also taking the guards of the edges into account. The parallel composition of two inconsistent VLTS is not defined. In

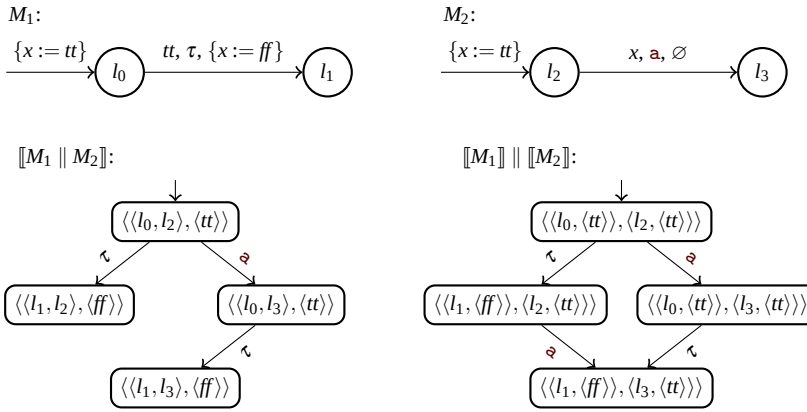


Figure 3.7: LTS parallel composition does not work with shared variables

practice, we consider an attempt to do so as a modelling error; tools should simply reject such models.

Distributivity

Now that we have defined a semantics for VLTS in terms of LTS as well as a parallel composition operator for both formalisms, an obvious question is: Are the semantics operation $[\![\cdot]\!]$ and parallel composition $\parallel$ for both VLTS and LTS distributive? The answer is, unfortunately, *no* [BK08]:

Theorem 1. For two VLTS M_1 and M_2, we have

$$[\![M_1 \parallel M_2]\!] \cong [\![M_1]\!] \parallel [\![M_2]\!]$$

when M_1 and M_2 have no shared variables, but it does not hold in general.

Proof. A counterexample for the shared variable case is given in Figure 3.7. For the case when there are no shared variables, let M_i, $i \in \{1,2\}$, be two VLTS

$$M_i = \langle Loc_i, Var_i, A_i, E_i, l_{init_i}, V_{init_i}, VExp_i \rangle$$

with $Var_1 \cap Var_2 = \varnothing$. The semantics of their parallel composition $[\![M_1 \parallel M_2]\!]$ is

$$\langle S_{\text{Sem}}, A_1 \cup A_2, T_{\text{Sem}}, \langle \langle l_{init_1}, l_{init_2} \rangle, V_{init_1} \cup V_{init_2} \rangle, VExp_1 \cup VExp_2, L_{\text{Sem}} \rangle$$

where $S_{\text{Sem}} = (Loc_1 \times Loc_2) \times (\text{Val}(Var_1 \cup Var_2))$, and the parallel composition of their individual semantics $[\![M_1]\!] \parallel [\![M_2]\!]$ is

$$\langle S_{\parallel}, A_1 \cup A_2, T_{\parallel}, \langle \langle l_{init_1}, V_{init_1} \rangle, \langle l_{init_2}, V_{init_2} \rangle \rangle, VExp_1 \cup VExp_2, L_{\parallel} \rangle$$

where $S_{\|} = (Loc_1 \times \mathrm{Val}(Var_1)) \times (Loc_2 \times \mathrm{Val}(Var_2))$. We need to show that there is an isomorphism between $[\![M_1 \parallel M_2]\!]$ and $[\![M_1]\!] \parallel [\![M_2]\!]$. We thus need to show that a bijection exists between all states and for all other parts of both systems.

First, observe that the alphabets and the sets of atomic propositions are identical, so we can use relation *Id* for them. For states, we relate $\langle\langle l_1, l_2\rangle, v\rangle$ to $\langle\langle l_1, v_1\rangle, \langle l_2, v_2\rangle\rangle$ for all $l_i \in Loc_i$, $v_i \in \mathrm{Val}(Var_i)$, $v \in \mathrm{Val}(Var_1 \cup Var_2)$ if and only if $v = v_1 \cup v_2$. This is a bijection because the sets of variables of both components are disjoint. For simplicity, we will directly denote the valuations of the reachable states of $[\![M_1 \parallel M_2]\!]$ by $v_1 \cup v_2$ in the remainder of this proof.

Recall that, by Definitions 20 and 22, we have

$$T_{\mathrm{Sem}} \in S_{\mathrm{Sem}} \to \mathscr{P}(A_1 \cup A_2 \times S_{\mathrm{Sem}})$$

and $T_{\|} \in S_{\|} \to \mathscr{P}(A_1 \cup A_2 \times S_{\|})$. We show that both functions agree on all related states $\langle\langle l_1, l_2\rangle, v_1 \cup v_2\rangle \in S_{\mathrm{Sem}}$ and $\langle\langle l_1, v_1\rangle, \langle l_2, v_2\rangle\rangle \in S_{\|}$:

$$\begin{aligned}
&\langle a, \langle\langle l_1', l_2'\rangle, v_1' \cup v_2'\rangle\rangle \in T_{\mathrm{Sem}}(\langle\langle l_1, l_2\rangle, v_1 \cup v_2\rangle) \\
&\Leftrightarrow \exists \langle g, a, U, \langle l_1', l_2'\rangle\rangle \in E(\langle l_1, l_2\rangle) \colon [\![g]\!](v_1 \cup v_2) \wedge (v_1' \cup v_2') = [\![U]\!](v_1 \cup v_2) && \text{(Def. 22)} \\
&\Leftrightarrow \exists \langle g, a, U, \langle l_1', l_2'\rangle\rangle \colon \quad [\![g]\!](v_1 \cup v_2) \wedge (v_1' \cup v_2') = [\![U]\!](v_1 \cup v_2) \\
&\qquad\qquad \wedge \big(\quad a \notin B \wedge \langle g, a, U, l_1'\rangle \in E_1(l_1) \wedge l_2 = l_2' \\
&\qquad\qquad\quad \vee a \notin B \wedge \langle g, a, U, l_2'\rangle \in E_2(l_2) \wedge l_1 = l_1' \\
&\qquad\qquad\quad \vee a \in B \\
&\qquad\qquad\qquad \wedge \langle g_1, a, U_1, l_1'\rangle \in E_1(l_1) \wedge \langle g_2, a, U_2, l_2'\rangle \in E_2(l_2) \\
&\qquad\qquad\qquad \wedge (g = g_1 \wedge g_2) \wedge U = U_1 \cup U_2 \big) && \text{(Def. 23)} \\
&\Leftrightarrow \exists \langle g, a, U, \langle l_1', l_2'\rangle\rangle \colon \quad a \notin B \wedge \langle g, a, U, l_1'\rangle \in E_1(l_1) \\
&\qquad\qquad\qquad\qquad \wedge [\![g]\!](v_1) \wedge v_1' = [\![U]\!](v_1) \wedge l_2 = l_2' \\
&\qquad\qquad \vee a \notin B \wedge \langle g, a, U, l_2'\rangle \in E_2(l_2) \\
&\qquad\qquad\qquad\qquad \wedge [\![g]\!](v_2) \wedge v_2' = [\![U]\!](v_2) \wedge l_1 = l_1' \\
&\qquad\qquad \vee a \in B \wedge \langle g_1, a, U_1, l_1'\rangle \in E_1(l_1) \\
&\qquad\qquad\qquad\qquad \wedge \langle g_2, a, U_2, l_2'\rangle \in E_2(l_2) \\
&\qquad\qquad\qquad\qquad \wedge [\![g_1]\!](v_1) \wedge [\![g_2]\!](v_2) \\
&\qquad\qquad\qquad\qquad \wedge v_1' = [\![U_1]\!](v_1) \wedge v_2' = [\![U_2]\!](v_2) \\
&\qquad\qquad\qquad\qquad \wedge (g = g_1 \wedge g_2) \wedge U = U_1 \cup U_2 \\
&\qquad\qquad\qquad\qquad\quad \text{(distributivity } \wedge/\vee \text{ and } Var_1 \cap Var_2 = \varnothing)
\end{aligned}$$

$$
\begin{aligned}
\Leftrightarrow\quad & a \notin B \wedge \big(\exists \langle g,a,U,l_1'\rangle \in E_1(l_1)\colon [\![g]\!](v_1) \wedge v_1' = [\![U]\!](v_1)\big) \\
& \qquad \wedge \langle l_2, v_2\rangle = \langle l_2', v_2'\rangle \\
\vee\ & a \notin B \wedge \big(\exists \langle g,a,U,l_2'\rangle \in E_2(l_2)\colon [\![g]\!](v_2) \wedge v_2' = [\![U]\!](v_2)\big) \\
& \qquad \wedge \langle l_1, v_1\rangle = \langle l_1', v_1'\rangle \\
\vee\ & a \in B \wedge \big(\exists \langle g,a,U,l_1'\rangle \in E_1(l_1)\colon [\![g]\!](v_1) \wedge v_1' = [\![U]\!](v_1)\big) \\
& \qquad \wedge \big(\exists \langle g,a,U,l_2'\rangle \in E_2(l_2)\colon [\![g]\!](v_2) \wedge v_2' = [\![U]\!](v_2)\big) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(distributivity } \exists/\vee\text{)} \\
\Leftrightarrow\quad & a \notin B \wedge \langle a, \langle l_1', v_1'\rangle\rangle \in T_1(\langle l_1, v_1\rangle) \wedge \langle l_2, v_2\rangle = \langle l_2', v_2'\rangle \qquad \text{(Def. 22)} \\
\vee\ & a \notin B \wedge \langle a, \langle l_2', v_2'\rangle\rangle \in T_2(\langle l_2, v_2\rangle) \wedge \langle l_1, v_1\rangle = \langle l_1', v_1'\rangle \\
\vee\ & a \in B \wedge \langle a, \langle l_1', v_1'\rangle\rangle \in T_1(\langle l_1, v_1\rangle) \wedge \langle a, \langle l_2', v_2'\rangle\rangle \in T_2(\langle l_2, v_2\rangle) \\
\Leftrightarrow\ & \langle a, \langle\langle l_1', v_1'\rangle, \langle l_2', v_2'\rangle\rangle\rangle \in T_{\|}(\langle\langle l_1, v_1\rangle, \langle l_2, v_2\rangle\rangle) \qquad \text{(Def. 20)}
\end{aligned}
$$

We finally show that the labelling functions agree for all related states, i.e. that $L_{\mathrm{Sem}}(\langle\langle l_1, l_2\rangle, v_1 \cup v_2\rangle) = L_{\|}(\langle\langle l_1, v_1\rangle, \langle l_2, v_2\rangle\rangle)$ for all $l_i \in Loc_i$, $v_i \in \mathrm{Val}(Var_i)$:

$$
\begin{aligned}
L_{\|}(\langle\langle l_1, v_1\rangle, \langle l_2, v_2\rangle\rangle) &= L_1(\langle l_1, v_1\rangle) \cup L_2(\langle l_2, v_2\rangle) && \text{(Def. 20)} \\
&= \{\, e \in VExp_1 \mid [\![e]\!](v_1)\,\} \cup \{\, e \in VExp_2 \mid [\![e]\!](v_2)\,\} && \text{(Def. 22)} \\
&= \{\, e \in VExp_1 \cup VExp_2 \mid [\![e]\!](v_1 \cup v_2)\,\} && (Var_1 \cap Var_2 = \varnothing) \\
&= L_{\mathrm{Sem}}(\langle\langle l_1, l_2\rangle, v_1 \cup v_2\rangle) && \text{(Def. 22)}
\end{aligned}
$$

□

The underlying problem is that the modification of a global variable in one component directly influences the other components, but that influence is not taken into account when constructing the parallel composition of the component semantics. In other words: While the semantics of a VLTS and the LTS parallel composition operator together preserve the possibility for further (V)LTS to influence the system via synchronisation on shared actions, they remove the possibility for similar influences via changes to global variables.

This problem can be avoided by changing the VLTS semantics to generate an overapproximation of the local state space w.r.t. the unknown influence of other parallel components on global variables; see [ZZ14] for an example of such an approach in one specialised setting. In this thesis, we do not focus on compositional verification and minimisation, which would require the application of VLTS semantics followed by LTS parallel composition. We instead assume all VLTS to be small enough for their parallel composition to

be constructed explicitly, followed by an (ideally on-the-fly) generation of the composition's LTS semantics during verification.

3.1.3 Modelling

VLTS can be seen as a simple graphical modelling formalism. Although VLTS with parallel composition already allow the creation of relatively manageable models, the only operation that allows reuse of parts of models is parallel composition. MODEST, on the other hand, is a text-based modelling language that provides a rich syntax for modelling and, in particular, component reuse. It allows very concise models to be built in a fashion similar to standard programming languages. In this section, we introduce the subset of the MODEST syntax that allows the modelling of LTS and VLTS. While the full semantics of MODEST maps to SHA, we present it step-by-step in this thesis, always using as simple a submodel of SHA as possible. The subset of MODEST introduced in this thesis is presented as a whole in Appendix A. Beyond that, the full MODEST syntax and semantics for SHA can be found in [HHHK13].

MODEST for LTS

A MODEST model consists of a number of declarations followed by a *process behaviour* according to the following grammar (for the LTS subset):

$$
\begin{aligned}
P ::= {} & \mathit{act} \mid \texttt{stop} \mid \texttt{abort} \mid \texttt{break} \mid P_1 ; P_2 \mid \\
& \texttt{alt}\,\{ :: P_1 \ldots :: P_k \} \mid \texttt{do}\,\{ :: P_1 \ldots :: P_k \} \mid \\
& \texttt{throw}(\mathit{excp}) \mid \texttt{try}\,\{ P \}\ \texttt{catch}\ \mathit{excp}_1\,\{ P_1 \} \ldots \texttt{catch}\ \mathit{excp}_k\,\{ P_k \}
\end{aligned}
$$

Its semantics is the LTS $\langle S, \mathit{Act} \uplus \mathit{Excp}, T, s_{\mathit{init}}, \mathit{AP}, L \rangle$ where
– Act contains all declared *actions* plus the predefined actions τ, $\flat$ and $\perp$,
– Excp contains all declared *exceptions*,
– S, the set of states, is the set of all (syntactically valid) process behaviours using labels in $\mathit{Act} \cup \mathit{Excp}$,
– $s_{\mathit{init}} \in S$ is the model's process behaviour itself,
– AP and L are determined by the properties introduced as part of the declarations, and finally
– T is the largest set of transitions that satisfy the inference rules that we present in the remainder of this section and thesis.

We introduce the inference rules together with a running example, where we model a communication protocol similar to the one presented in the previous examples of this section:

Actions To create action-labelled transitions, the actions need to be declared first. This is done using the keyword `action`. The silent action τ, which can also be written as `tau` in MODEST, the special *break action*, written as `break` in MODEST and as $\flat$ in inference rules, and the error action $\bot$, created by the `abort` keyword (see the paragraph on exception below), cannot and need not explicitly be declared.

A transition labelled with some action is then created by simply "calling" that action as shown in Figure 3.8 for the `snd_data` step of the communication protocol's sender. The corresponding inference rule is

$$\frac{(a \in Act \setminus \{\bot\})}{a \xrightarrow{a} \checkmark} \ (act)$$

where $\checkmark$ represents the *successfully terminated process*, a syntactically valid process behaviour that is only used in the inference rules and that must not be part of a model.

Sequential composition Two process behaviours are executed one after the other when combined by the *sequential composition* operator ;. In Figure 3.9, the sender now uses this to wait for an acknowledgment after having sent the data. The inference rules are

$$\frac{P \xrightarrow{a} P' \quad (P' \neq \checkmark)}{P; Q \xrightarrow{a} P'; Q} \ (seq_A) \qquad \text{and} \qquad \frac{P \xrightarrow{a} \checkmark}{P; Q \xrightarrow{a} Q} \ (seq_B)$$

where, here and in all following inference rules, P, Q and variants denote process behaviours while a denotes elements of $Act \cup Excp$.

Nondeterministic choice A central feature of LTS is the ability to express nondeterministic choices. These are specified in MODEST as choices between two or more process behaviours using the `alt` keyword. This allows us to also include the alternative of a timeout occurring instead of the acknowledgment being received in our example in Figure 3.10. The corresponding inference rule is

$$\frac{P_i \xrightarrow{a} Q \quad (i \in \{1, \ldots, n\})}{\texttt{alt}\ \{\ ::P_1\ \ldots\ ::P_n\ \} \xrightarrow{a} Q} \ (alt)$$

Cycles In order to allow the sender to retransmit a message once a timeout has occurred, it needs to be able to go back to its initial state. In MODEST, such cyclic behaviour can be achieved with the `do` keyword. To exit the cycle,

```
action snd_data;

snd_data
```

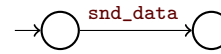


Figure 3.8: Actions in MODEST

```
action snd_data, rcv_data;

snd_data;
rcv_ack
```

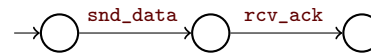


Figure 3.9: Sequential composition in MODEST

```
action snd_data, rcv_data,
       timeout;

snd_data;
alt {
:: rcv_ack
:: timeout
}
```

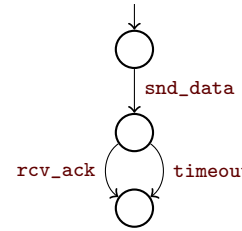


Figure 3.10: Nondeterministic choice in MODEST

```
action snd_data, rcv_data,
       timeout;

do {
:: snd_data;
   alt {
   :: rcv_ack; break
   :: timeout
   }
}
```

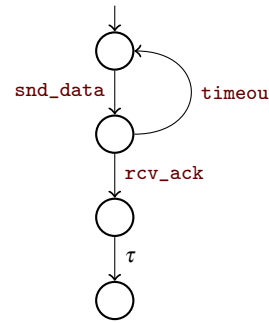


Figure 3.11: Cycles in MODEST

the special action `break` ($\flat$ in inference rules) is used. Figure 3.11 shows these new behaviours at work as part of the communication protocol's sender.

The inference rules for `do` are complicated by the fact that we need to keep track of the original behaviour to return to after each iteration. We use the auxiliary `auxdo` behaviour for this purpose:

$$\texttt{do}\ \{\ ::P_1\ \ldots\ ::P_n\ \} \stackrel{\text{def}}{=} \texttt{auxdo}\ \{\ \texttt{alt}\ \{\ ::P_1\ \ldots\ ::P_n\ \}\ \}\ \{\ \texttt{alt}\ \{\ ::P_1\ \ldots\ ::P_n\ \}\ \}$$

Its second component is where the original behaviour is preserved for later re-initialisation. Note that `do` follows the same syntax as `alt` and, as the expansion to `auxdo` shows, in fact includes an initial nondeterministic choice. The inference rules are then

$$\frac{P \xrightarrow{a} P' \qquad (a \neq \flat \wedge P' \neq \checkmark)}{\texttt{auxdo}\ \{\ P\ \}\ \{\ Q\ \} \xrightarrow{a} \texttt{auxdo}\ \{\ P'\ \}\ \{\ Q\ \}}\ (\mathit{auxdo}_A),$$

$$\frac{P \xrightarrow{a} \checkmark \qquad (a \neq \flat)}{\texttt{auxdo}\ \{\ P\ \}\ \{\ Q\ \} \xrightarrow{a} \texttt{auxdo}\ \{\ Q\ \}\ \{\ Q\ \}}\ (\mathit{auxdo}_B),$$

$$\text{and}\ \frac{P \xrightarrow{\flat} P'}{\texttt{auxdo}\ \{\ P\ \}\ \{\ Q\ \} \xrightarrow{\tau} P'}\ (\mathit{breakout})$$

where rule auxdo_A defines the semantics of performing a step within the cycle, auxdo_B ensures that we return to the next iteration when the previous iteration's the last step has been performed (restoring the original behaviour preserved in the `auxdo`'s second component), and finally *breakout* defines the consequences of executing the `break` action $\flat$.

Exceptions Many programming languages provide mechanisms to *throw* (or equivalently to *raise*) and *catch* (or equivalently to *handle*) exceptions to make dealing with rare situations that need exceptional control flow easier. Inspired by this, MODEST also has exceptions as a special kind of transition label. We use this feature in our example to raise an error by throwing exception `err` on timeout in Figure 3.12. The inference rule for throwing exceptions is

$$\frac{\mathit{excp} \in \mathit{Excp}}{\texttt{throw}(\mathit{excp}) \xrightarrow{\mathit{excp}} \texttt{abort}}\ (\mathit{throw}).$$

The immediate result of throwing an exception is that we move to the `abort` behaviour, which generates a loop labelled with the previously mentioned error action $\perp$:

$$\frac{}{\texttt{abort} \xrightarrow{\perp} \texttt{abort}}\ (\mathit{abort})$$

```
action snd_data, rcv_data,
       timeout;
exception err;

do {
:: snd_data;
   alt {
   :: rcv_ack; break
   :: timeout; throw(err)
   }
}
```

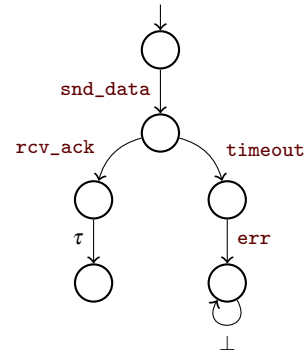


Figure 3.12: Throwing exceptions in MODEST

```
action snd_data, rcv_data,
       timeout;
exception err;

try {
   do {
   :: snd_data; alt {
      :: rcv_ack; break
      :: timeout; throw(err)
      }
   }
} catch(err) {
   stop
}
```

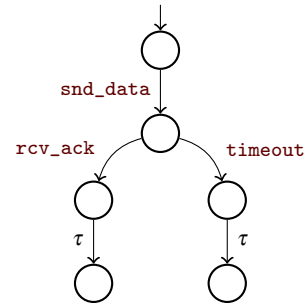


Figure 3.13: Catching exceptions in MODEST

To put exceptions to more productive use in a model, they can be wrapped within a `try` behaviour. This allows them to be caught using exception handlers specified by the `catch` keyword as shown in Figure 3.13. The inference rules for `try-catch` are the following:

$$\frac{P \xrightarrow{a} P' \quad P' \neq \checkmark \quad (a \notin \{excp_1, \ldots, excp_n\})}{Q(P) \xrightarrow{a} Q(P')} \ (try_A)$$

$$\frac{P \xrightarrow{a} \checkmark \quad (a \notin \{excp_1, \ldots, excp_n\})}{Q(P) \xrightarrow{a} \checkmark} \ (try_B)$$

$$\frac{P \xrightarrow{excp_i} P' \quad (i \in \{1, \ldots, n\})}{Q(P) \xrightarrow{\tau} P_i} \ (catch)$$

where $Q(P)$ denotes any behaviour of the form

$$Q(P) \stackrel{\text{def}}{=} \texttt{try}\ \{\ P\ \}\ \texttt{catch}\ excp_1\ \{\ P_1\ \}\ \ldots\ \texttt{catch}\ excp_n\ \{\ P_n\ \}$$

for process behaviours $P_1, \ldots, P_n$ and exceptions $excp_i \in Excp$, $i \in \{1, \ldots, n\}$. In the example of Figure 3.13, no useful exception handling is performed. The absence of any further behaviour is indicated by the `stop` keyword, which consequently has no associated inference rule.

We point out that we have thus far seen three *static operators*, namely `;`, `auxdo` and `try-catch`. In contrast to the other operators like `alt`, they appear on both sides of the transition in the conclusion of at least one of their inference rules. Intuitively, this means that they usually do not "disappear" when a transition is taken. Static operators therefore always need a dedicated inference rule like try_B that removes them when the inner behaviour terminates successfully.

MODEST for VLTS

The model shown in Figure 3.11 corresponds very well to the original LTS of the communication protocol sender as shown in Figure 3.1, save for the extra τ step. Our goal, however, is a protocol with a bounded number of retransmissions. As we saw, using variables greatly simplifies such a model. We therefore introduce variables and constructs to specify guards and assignments into MODEST at this point. From now on, the semantics of a MODEST model is thus no longer an LTS, but a VLTS whose variables are given in the declaration section. We extend the grammar for process behaviours by the following options:

$$P ::= \ldots \mid act\ \{= u_1, \ldots, u_n =\} \mid \texttt{when}(e)\ P \mid \texttt{par}\{::P_1 \ldots ::P_k\} \mid$$
$$\texttt{relabel}\{I\}\texttt{by}\{G\}\{P\} \mid \texttt{extend}\{H\}\{P\} \mid ProcName(e_1, \ldots, e_k)$$

where $act \in Act$, the $u_j \in Asgn$ form an update, $e \in Bxp$, $H \subseteq Act \setminus \{\tau\}$ is a set of observable actions, and I and G are vectors of equal length which have elements in $Act \setminus \{\flat, \perp\}$ such that all elements in I are pairwise different and not equal to τ. To keep the code small, we from now on omit the explicit action and exception declarations in the MODEST examples.

Variables, guards and assignments The syntax for variable declarations is similar to standard programming languages such as C or C#: it assigns a type and an initial value to a variable name. Our example in Figure 3.14 declares a single integer variable (i.e. a variable with domain $\mathbb{Z}$) with initial value 3. Declared variables can then be modified in assignment blocks that use the $\{= \ldots =\}$ syntax. To add a guard to an edge resulting from some process behaviour, that behaviour is prefixed with the `when` keyword. Guards and assignments are used in the example to obtain the desired behaviour of limiting the number of retransmissions to three. The inference rules for assignments and guards are

$$\frac{(a \in Act)}{a\ \{= u_1, \ldots, u_n =\} \xrightarrow{tt,a,\{u_1,\ldots,u_n\}} \checkmark}\ (assgn) \qquad \frac{P \xrightarrow{g,a,U} P'}{\texttt{when}(e)\ P \xrightarrow{g \wedge e,a,U} P'}\ (when)$$

where $u_1, \ldots, u_n \in Asgn$, the u_i are pairwise consistent and $e \in Bxp$. If the action prefix of an assignment block is τ, the MODEST syntax also allows it to be omitted.

Processes and process calls In order to reuse process behaviours, they can be encapsulated in a named *process* in the declaration section. In Figure 3.15, we do so for the sender from the previous examples (but with at most one retransmission). We also replace the `do` construct by a recursive call to the `Sender` process. (Note that this results in two edges labelled `snd_data` because the initial location is `Sender`(2), which does not occur again inside the behaviour of the process.) The inference rule for a call to a process declared as

$$\texttt{process}\ ProcName(t_1\ x_1,\ \ldots,\ t_n\ x_n)\ \{\ P\ \}$$

with types t_i and parameter names x_i is straightforward:

$$\frac{P \xrightarrow{g,a,U} P'}{ProcName(e_1, \ldots, e_n) \xrightarrow{g,a,U} P'}\ (call)$$

where $e_1, \ldots, e_n \in Sxp$ are the actual parameters. However, assignments of actual to formal parameters do not appear in this rule. This is because the parameter variables must already be set to their concrete values to determine the

edges leading out of the inner process behaviour P (which could, for example, be a guard referencing some of the parameters). The corresponding assignments must therefore occur before the location that "directly" contains the process call is entered. At this point, we informally note that this means extending all "sequential" inference rules such as *seq* or *catch* by adding the parameter assignments of the following process calls in the correct way. The complete inference rules that do so via an *assignment collecting function* can be found in Appendix A.

Parallel composition The parallel composition of process behaviours is supported by MODEST's `par` keyword, with syntax similar to `alt` and `do` to specify the parallel components. The semantics for `par` is given via an auxiliary binary parallel composition operator that keeps track of its synchronisation alphabet:

$$\texttt{par}\ \{\ ::P_1\ \ldots\ ::P_n\ \} \stackrel{\text{def}}{=} (\ldots((P_1 \parallel_{B_1} P_2) \parallel_{B_2} P_3)\ldots) \parallel_{B_{n-1}} P_n$$

where the synchronisation alphabets are computed as the shared alphabets of the process behaviours,

$$B_j = (\bigcup_{i=1}^{j} \alpha(P_i)) \cap \alpha(P_{j+1}),$$

with a function α that collects all called actions in $Act \setminus \{\tau, \flat, \perp\}$ from a process behaviour and recursively from the behaviours of the processes called within it.

The semantics of the MODEST parallel composition operator $\parallel_B$ is essentially the same as that of operator $\parallel$ for VLTS, except that the set of actions B is used instead of the full shared alphabet as for VLTS (which could include the non-synchronising labels $\flat$, $\perp$ and exceptions). Again, we refer the interested reader to Appendix A for a formal definition of function α and the semantics of $\parallel_B$.

Alphabet manipulation At this point, we would be ready to complete our example by first adding the processes for the two channels and the receiver and then specifying their parallel composition to be the overall process behaviour of the model. However, MODEST is about component reuse and concise models, so we would like to avoid including two versions of the channel that only differ in their action labels. This is indeed possible by renaming edge labels with the `relabel` keyword. Its semantics is given by the following rules:

$$\frac{P \xrightarrow{g,a,U} P' \quad (P' \neq \checkmark)}{Q(P) \xrightarrow{g,f(a),U} Q(P')}\ (try_A) \qquad \frac{P \xrightarrow{g,a,U} \checkmark}{Q(P) \xrightarrow{g,f(a),U} \checkmark}\ (try_B)$$

```
int n = 3;

do {
:: snd_data {= n = n - 1 =};
   alt {
   :: rcv_ack; break
   :: timeout; alt {
      :: when(n > 0)  tau
      :: when(n == 0) throw(err)
      }
   }
}
```

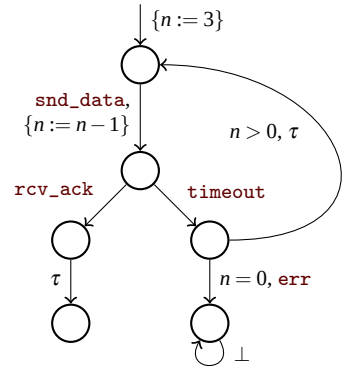


Figure 3.14: Variables, guards and assignments in MODEST

```
process Sender(int n)
{
   snd_data {= n = n - 1 =};
   alt {
   :: rcv_ack
   :: timeout; alt {
      :: when(n > 0)  Sender(n)
      :: when(n == 0) throw(err)
      }
   }
}

Sender(2)
```

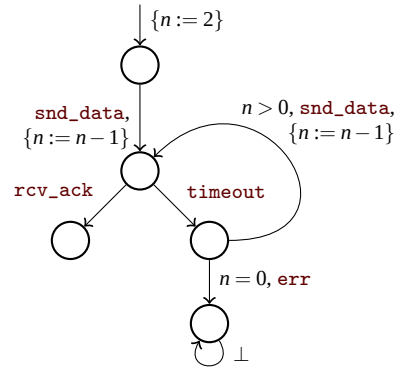


Figure 3.15: Process declarations and calls in MODEST

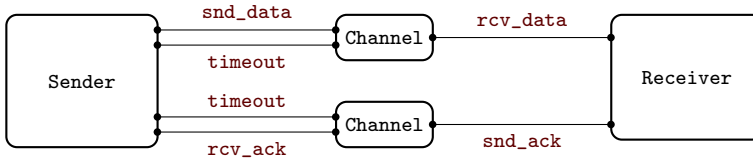


Figure 3.16: Synchronisation structure of the simple BRP model

where $Q(P)$ denotes behaviours of the form

$$Q(P) \stackrel{\text{def}}{=} \texttt{relabel}\ \{a_1,\ldots,a_n\}\ \texttt{by}\ \{b_1,\ldots,b_n\}\ \{\ P\ \}$$

for $a_i \in Act \cup Excp \setminus \{\tau, \natural, \bot\}$, $b_i \in Act \cup Excp \setminus \{\natural, \bot\}$ and

$$f(a) = \begin{cases} b_i & \text{if } a = a_i \\ a & \text{otherwise} \end{cases}$$

for $i \in \{1,\ldots,n\}$ with the restriction that actions may only be mapped to actions (including τ but not $\natural$ or $\bot$) and exceptions may only be mapped to exceptions or τ.

The renaming of actions is also taken into account in function α when computing the alphabet of a process behaviour for parallel composition. To allow a more concise syntax for relabelling actions to τ, MODEST provides the `hide` shorthand:

$$\texttt{hide}\ \{a_1,\ldots,a_n\}\ \{\ P\ \} \stackrel{\text{def}}{=} \texttt{relabel}\ \{a_1,\ldots,a_n\}\ \texttt{by}\ \{\tau,\ldots,\tau\}\ \{\ P\ \}$$

Example 8. Figure 3.17 shows the complete MODEST model of a simple communication protocol with lossy channels and an upper bound on the number of retransmissions, henceforth referred to as the *simple bounded retransmission protocol* (simple BRP). Its synchronisation structure is depicted schematically in Figure 3.16. This structure is achieved through relabelling and nested parallel composition. The latter is necessary in order to avoid synchronisation between the two `Channel` instances on action `timeout`.

MODEST also includes an `extend` keyword that allows the inclusion of actions not occurring in a process behaviour into the set of labels collected by α and thus into the synchronisation alphabets of parallel composition. Its inference

```
action snd, rcv, snd_data, rcv_data, snd_ack, rcv_ack,
       timeout, to;
bool success, failure;

property E_Succ = E<>(success);
property E_Fail = E<>(failure);

process Sender(int n)
{
   snd_data {= n = n - 1 =};
   alt {
   :: rcv_ack {= success = true =}
   :: timeout; alt {
      :: when(n > 0) Sender(n) // retry
      :: when(n == 0) {= failure = true =};
         stop // deadlock on failure
      }
   }
}

process Receiver()
{
   rcv_data; snd_ack; Receiver()
}

process Channel()
{
   snd; alt {
   :: rcv
   :: timeout // message lost
   };
   Channel()
}

par {
:: Sender(2)
:: relabel { to } by { timeout }
   {
      par {
      :: relabel { snd,      rcv }
              by { snd_data, rcv_data } Channel()
      :: relabel { snd,      rcv,     timeout }
              by { snd_ack,  rcv_ack, to } Channel()
      }
   }
:: Receiver()
}
```

Figure 3.17: The simple BRP model in MODEST

rules are simply

$$\frac{P \xrightarrow{g,a,U} P' \qquad (P' \neq \checkmark)}{\texttt{extend}\,\{\,a_1,\ldots,a_n\,\}\,\{\,P\,\} \xrightarrow{g,a,U} \texttt{extend}\,\{\,a_1,\ldots,a_n\,\}\,\{\,P'\,\}}\ (\mathit{extend}_A)$$

$$\text{and}\quad \frac{P \xrightarrow{g,a,U} \checkmark}{\texttt{extend}\,\{\,a_1,\ldots,a_n\,\}\,\{\,P\,\} \xrightarrow{g,a,U} \checkmark}\ (\mathit{extend}_B)$$

since its presence only makes a difference for α. We do not use `extend` in any examples or models in the remainder of this thesis.

3.1.4 Properties

To perform verification for LTS models, we need to introduce a formalism for the specification of properties. Such a formalism will make use of the atomic propositions and the state labelling function that are included in an LTS model. As we noted, this means that we choose to focus on state-based verification in this thesis. The alternative would have been to keep states entirely opaque and instead reason about the transition labels that occur.

In this thesis, we mainly focus on the basic class of *reachability* properties. A reachability property has the form

$$\exists \diamond \phi$$

where $\exists \diamond$ is read as "exists eventually" and ϕ is a Boolean expression over the atomic propositions of the given model, i.e. it is built according to the grammar

$$\phi ::= p \mid \mathit{true} \mid \mathit{false} \mid \neg \phi \mid \phi \wedge \phi \mid \phi \vee \phi$$

with p ranging over the atomic propositions. We call such an expression a *state formula* and denote the set of all state formulas over the atomic propositions AP by Φ_{AP} (where the index AP may be omitted if it is clear from the context). Given a concrete set V of atomic propositions, a state formula ϕ can be evaluated to *true* or *false* by replacing all occurrences in ϕ of each atomic proposition by *true* if the proposition is in V and by *false* otherwise. We write the result of such an evaluation as $\phi(V)$. If $\phi(L(s))$ holds for a state s in an LTS with labelling function L, we say that s is a ϕ-state.

Reachability properties $\exists \diamond \phi$ ask whether it is possible to encounter a ϕ-state at some point on some path from the initial state. They are thus useful for establishing that certain undesired configurations characterised by a state formula cannot occur in (the model of) a system. Likewise, if a (desirable) *invariant* of all configurations can be formulated as state formula ϕ, the negation of $[\![\exists \diamond (\neg \phi)]\!]$ tells us whether ϕ really holds in all states of the model.

The semantics of a reachability property is formally defined in terms of the reachable states of the LTS under study as follows:

Definition 25 (Semantics of LTS reachability properties)**.** Given an LTS M with $M = \langle S, A, T, s_{init}, AP, L \rangle$ as usual and a reachability property $\exists \diamond \phi$ over the atomic propositions of M, the semantics of the property is

$$[\![\exists \diamond \phi]\!]_M \stackrel{\text{def}}{=} \exists s \in \text{Reach}(M) \colon \phi(L(s)).$$

We omit the index M from the $[\![\cdot]\!]$ operator when it is clear from the context.

Example 9. For the simple MODEST BRP model presented in Example 8, the Boolean variables `success` and `failure` can be used to specify the two reachability properties $\exists \diamond$ `success` and $\exists \diamond$ `failure`. Verification of the model for these two properties will then determine whether transmission of the message can succeed or fail. The two properties are already included in the MODEST code shown in Figure 3.17 in lines 3 and 4. Properties in MODEST models are named such that they can easily be referred to, for example in a table of verification results. In this case, the two properties are named $\mathtt{E}_{\mathtt{Succ}}$ and $\mathtt{E}_{\mathtt{Fail}}$.

Safety and liveness

As mentioned in Section 1.2 previously, classic examples of functional requirements are safety and liveness properties. While a safety property is the statement that "bad things never happen", liveness requires that "good things will always eventually happen". Reachability properties are the special case of safety in which all bad "things" can be characterised by referring to any single set of states only. In general, however, the undesirable behaviours can be in the evolution of the model over multiple transitions, i.e. they can be characterised by traces. The set of (finite or infinite) traces a model must *not* have in order to satisfy some safety property is fully determined by a set of *finite bad prefixes*. Conversely, for some liveness property to be satisfied, all traces of a model must be infinite and exhibit certain patterns over and over again after *ignoring an arbitrary finite prefix*. For more details about safety and liveness properties for LTS, we refer the interested reader to e.g. [BK08].

Logics for LTS properties

An even more general way to specify properties is by using temporal logics. A formula from some temporal logic characterises the paths starting in a model's reachable states. Well-known examples are LTL [Pnu77], CTL [EC82] and CTL* [EH86]. Due to the focus of this thesis on reachability, we only give a

brief characterisation of these logics in relation to states, traces and reachability and again refer the reader to e.g. [BK08] for details:

An LTL formula represents a set of (usually infinite) traces. If this set contains all traces of some LTS, then that LTS satisfies the formula. An LTL formula thus implicitly refers to all traces starting in the initial state. In particular, this means that we cannot directly give an equivalent LTL formula for a reachability property. However, we can use an LTL verification procedure for reachability by observing that

$$\neg[\![\exists \diamond \phi]\!]_M \Leftrightarrow M \text{ satisfies LTL formula } \Box\neg\phi$$

where $\Box$ is read as "always".

In CTL, on the other hand, the quantification over paths is explicit and formulas can refer to paths that start in states other than the initial one. (It is due to this capability of referring to various states that we use paths instead of traces for CTL.) CTL formulas are constructed according to a grammar that forces an alternation between state formulas (which include the $\exists$ and $\forall$ quantifiers over the paths starting in a state) and *path formulas*, which include the $\diamond$ and $\Box$ operators. Notably, a reachability property $\exists \diamond \phi$ is a valid CTL formula with the same meaning, in which subformula $\diamond \phi$ is a path formula.

There is a common subset of LTL and CTL that includes, as we have seen, negated reachability. Still, some LTL formulas cannot be expressed in CTL and vice-versa. The temporal logic CTL* adds the quantification over paths to LTL or, equivalently, removes the strict alternation requirement of state and path formulas from CTL. The set of properties that can be expressed in CTL* is a strict superset of the sets of properties that can be expressed in either CTL or LTL alone. Reachability—positive and negative—is thus also a CTL* property.

3.1.5 Analysis

The verification of reachability properties on labelled transition systems can be performed in several very straightforward ways based on graph search algorithms. The practical challenge lies in the fact that the LTS under study are extremely large for most interesting models. As mentioned in Section 1.3, various techniques exist to reduce an LTS to an equivalent, but smaller one in order to speed up the verification procedure or make it feasible in the first place. We return to some of these techniques in more detail in Chapter 4. In this section, we instead present three verification procedures that contain the fundamental ideas used in all other analysis algorithms presented in the remainder of this thesis: classic exhaustive model checking, stateless model checking, and randomised testing. The latter is similar to the concept of statistical model checking that we introduce for DTMC in Section 3.2.3.

Input: Finite-state LTS $M = \langle S, A, T, s_{init}, AP, L \rangle$ and property $\exists \diamond \phi$
Output: $[\![\exists \diamond \phi]\!]_M$ (*true* or *false*)

```
1  agenda := { s_init }
2  visited := ∅
3  while agenda ≠ ∅ do
4  |   s := arbitrarily chosen element of agenda
5  |   agenda := agenda \ { s }
6  |   if s ∉ visited then
7  |   |   if φ(L(s)) then return true
8  |   |   visited := visited ∪ { s }
9  |   |   agenda := agenda ∪ T_*(s)
10 |   end
11 end
12 return false
```

Algorithm 1: Exhaustive model checking for LTS

In the remainder of this section, we denote the number of reachable states of the LTS under study by n and the total number of transitions of all reachable states by m. n and m can be ∞ for infinite-state LTS. We measure memory usage in terms of the number of relevant "objects", usually states, that are stored in memory at any one time point. For runtime, we assume that all operations (e.g. on sets or stacks, or to evaluate $T_*(s)$) take constant time. We also assume that the transition function is already given (usually as a compact computer program or function that can be invoked, for example an implementation of the MODEST semantics), i.e. it does not affect memory usage.

Model Checking

The core of the classic model checking approach is an exhaustive search of the model's state space. Standard graph search algorithms such as depth-first search (DFS) or breadth-first search (BFS) can be used. For a reachability property $\exists \diamond \phi$, it suffices to find a single state whose labelling satisfies ϕ in order to establish that $[\![\exists \diamond \phi]\!] = \mathit{true}$. For the converse, however, it is necessary to have checked that the labelling of *all* states does not satisfy ϕ. It is therefore necessary to keep track of the entire set of states that have already been checked, which is also required for the graph search to terminate. We thus need $n \neq \infty$ (or we could only achieve a semi-decision algorithm). Algorithm 1 is a generic

formulation of a basic model checking algorithm for reachability on LTS. Observe that it uses DFS if *agenda* is maintained as a last-in, first-out *stack*, and BFS if *agenda* is a first-in, first-out *queue*. Consequently, the memory usage and runtime of Algorithm 1 are in $O(n)$ and $O(m)$, respectively.

We mention that model checking of LTL, CTL or CTL* formulas is more involved: For LTL, the formula is usually converted to an automaton, and the state space to be explored is the product of this automaton and the LTS under study. The size of the formula automaton is exponential in the size of the LTL formula, and so is the product. The exploration of this product then also needs somewhat more complicated (but not necessarily more complex) algorithms like nested depth-first search. CTL model checking, on the other hand, can be done by recursively computing satisfying sets for subformulas. Its runtime is only linear in the size of the formula. Worst-case memory usage however, and therefore the state space explosion problem, remain unchanged whether one verifies reachability or temporal logics—be it LTL, CTL or CTL*—properties. This is one of the reasons why we focus on reachability in this thesis.

Stateless Model Checking

In case that the system under study is terminating or we are not interested in the falsification of reachability properties, *stateless model checking* [God97] is an alternative approach that trades increased analysis time for dramatically reduced memory usage. The key is that in both cases, it suffices to explore finite paths: If the system is terminating, we can model it as a finite acyclic LTS, which can only have finite paths. If on the other hand we can accept the answer *unknown* in addition to *true* for property $\exists \diamond \phi$, then it suffices to explore only finite prefixes of paths and simply return *unknown* when no state whose labelling satisfies ϕ is found on any of these prefixes.

Stateless model checking was originally introduced as a way to analyse a concurrent program implementation directly (instead of a model) and takes its name from the fact that no program state has to be captured and stored for later use. It is the extreme case of state-space caching approaches [God97], in which the currently explored path and a bounded cache of additional states is kept. The technique we describe below takes the main ideas of stateless model checking and state-space caching, but still works on an (LTS) model.

Algorithm 2 performs reachability verification for LTS in a way inspired by stateless model checking and state-space caching. The sets *agenda* and *visited* of Algorithm 1 have been replaced by a stack *schedule* that keeps track of the outgoing transitions that have already been followed on the path that is currently being explored. In this way, it exhaustively follows all paths up to length d looking for states whose labelling satisfies ϕ. Note that, if we knew that the LTS

Input: LTS $M = \langle S, A, T, s_{init}, AP, L \rangle$, property $\exists \diamond \phi$ and $d \in \mathbb{N}$
Output: $[\![\exists \diamond \phi]\!]_M$ (*true* or *unknown*)

```
1  if φ(L(s_init)) then return true
2  schedule := empty stack, schedule.push(⟨s_init, 1⟩)
3  while schedule ≠ empty stack do
4  |  ⟨s, i⟩ := schedule.pop()
5  |  succ := T_*(s) ordered according to a total order on transitions
6  |  if i ≤ |succ| ∧ |schedule| < d then
7  |  |  schedule.push(⟨s, i+1⟩)
8  |  |  s' := i-th element of succ
9  |  |  if φ(L(s')) then return true
10 |  |  schedule.push(⟨s', 1⟩)
11 |  end
12 end
13 return unknown
```

Algorithm 2: Stateless model checking for LTS

was finite and acyclic, d could be set to n. In that case, we know that the actual result should be *false* whenever *unknown* is returned. Otherwise, $d = n$ gives us what could also called *bounded model checking* [BCCZ99]. Its memory usage is very obviously in $O(d)$, but runtime strongly depends on the structure of the graph corresponding to the LTS. If we denote by $b = \max_{s \in S} |T(s)|$ the *maximum fan-out* of the LTS, we can guarantee that the runtime of stateless model checking is in $O(d \cdot b^d)$.

Algorithm 2 can be extended and improved in various ways. An obvious example is detecting cycles by checking whether the successor state s' obtained in line 9 is already part of *schedule*. This could improve runtime for reachability and would actually be necessary to detect cyclic behaviour as required for the analysis of liveness or many LTL properties. On the other hand, memory usage can be further reduced by not storing the states themselves on the stack. Instead, they would be recomputed as needed from the initial state based on the stored schedule. This would give us true stateless model checking at the cost of increased runtime and the inability to easily perform cycle detection.

Randomised Testing

We can also see stateless model checking as a form of *exhaustive testing* where prefixes of all possible paths are enumerated in a systematic fashion. This at

Input: LTS $M = \langle S, A, T, s_{init}, AP, L \rangle$, property $\exists \diamond \phi$, $d \in \mathbb{N}$ and $k \in \mathbb{N}$
Output: $[\![\exists \diamond \phi]\!]_M$ (*true* or *unknown*)

```
for i = 1 to k do
    s := s_init
    for j = 1 to d do
        if φ(L(s)) then return true
        else if T_*(s) = ∅ then break
        s := randomly chosen element of T_*(s)
    end
end
return unknown
```

Algorithm 3: Randomised testing for LTS

least guarantees that, when *unknown* is returned, state formula ϕ cannot be satisfied within d steps from the initial state. If we do not even care for this guarantee, we can use randomised testing instead, as shown in Algorithm 3: We simply explore the prefixes of length d of k random paths starting from the initial state. The algorithm is extremely simple to describe and implement, and its memory usage is clearly in $O(1)$ with runtime in $O(k \cdot d)$. In spite of its simplicity, it also has one advantage over stateless model checking in that longer paths can be explored at the cost of reduced coverage of the initial area of the state space: when we pick n much lower than the original b^d bound of stateless model checking, the newly chosen d can be much higher while still achieving the same actual runtime.

3.2 Discrete-Time Markov Chains

The other fundamental ingredient of the models we consider in this thesis are probabilistic decisions. The simplest model that includes discrete probabilistic choices is the model of discrete-time Markov chains (DTMC). From a mathematical perspective, Markov chains are a particular class of stochastic processes:

Definition 26 (Markov chain [GS01])**.** A family $X = \{X_t \mid t \in \mathbb{N}\}$ of discrete random variables that take values out of a countable set S is a *Markov chain* if it satisfies the *Markov property*, i.e.

$$\mathbb{P}(X_n = s \mid X_0 = x_0, X_1 = x_1, \ldots, X_{n-1} = x_{n-1}) = \mathbb{P}(X_n = s \mid X_{n-1} = x_{n-1})$$

for all $n \geq 1$ and all $s, x_1, \ldots, x_{n-1} \in S$.

The set S is called the *state space* of the Markov chain. The Markov property simply states that the value of the stochastic process, or equivalently the *state* of the Markov chain, at the next step only depends on its current value, but not any previous ones. Alternatively, we can say that its future only depends on the present, not on the past. Markov chains are therefore also called *memoryless*.

In the above definition, the probability of moving from one state to another, i.e. $\mathbb{P}(X_{n+1} = x_{n+1} \mid X_n = x_n)$, is determined by the current state x_n, the next state x_{n+1}, *and* the current time n. The dependence on the current time is often an undesirably concrete property that can be ruled out by considering only homogenous Markov chains, as we do in the remainder of this thesis:

Definition 27 (Homogenous Markov chain [GS01])**.** A Markov chain X with state space S is called *homogenous* if

$$\mathbb{P}(X_{n+1} = x_{n+1} \mid X_n = x_n) = \mathbb{P}(X_1 = x_1 \mid X_0 = x_0)$$

for all $n \geq 1$ and all $x_0, x_1, \ldots, x_{n-1} \in S$.

A homogenous Markov chain is thus fully characterised by its state space S and its *transition probabilities* $p_{xy} = \mathbb{P}(X_{n+1} = y \mid X_n = x)$. If we identify each element of S with a natural number, we can directly write these probabilities as a $|S| \times |S|$ *transition matrix*.

While it is useful to define Markov chains in this way to study their mathematical properties, we shall use an equivalent formulation (based on [BK08]) that is more in line with the way we already defined LTS in Section 3.1:

Definition 28 (DTMC)**.** A *discrete-time Markov chain* (DTMC), for the purpose of this thesis, is a 5-tuple $\langle S, T, s_{init}, AP, L \rangle$ where
– S is a countable set of states,
– $T \in S \to \mathrm{Dist}(S)$ is the probabilistic transition function,
– $s_{init} \in S$ is the initial state,
– AP is a set of atomic propositions, and
– $L \in S \to \mathscr{P}(AP)$ is the state labelling function.

For a given DTMC with transition function T, we also write $s \to \mu$ and $s \to s'$ instead of $T(s) = \mu$ and $T(s)(s') > 0$, respectively. We carry over the notation $T_*(s)$ from LTS and define $T_*(s) \stackrel{\text{def}}{=} \{ s' \in S \mid s \to s' \}$. Note that DTMC by definition do not have deadlock states. Given a DTMC according to Definition 28, the underlying Markov chain in the mathematical formulation of Definition 26 has state space S and transition probabilities $p_{xy} = T(x)(y)$.

Example 10. Figure 3.18 shows the graphical representation of a random walk on the set of integers $\mathbb{Z}$ where the probability of moving from one value to the

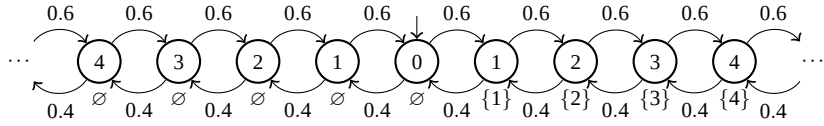


Figure 3.18: DTMC corresponding to a random walk, graphical representation

next larger value is p. It can be written as the DTMC $M_{rw} = \langle \mathbb{Z}, T, 0, \mathbb{N}^+, L \rangle$ with $T(x) = \{ x+1 \mapsto 0.6, x-1 \mapsto 0.4 \}$ for all $x \in \mathbb{Z}$ and $L(x) = \{ x \}$ if $x \in \mathbb{N}^+$ and $L(x) = \varnothing$ otherwise. In this case, the particular combination of atomic propositions and labelling function indicates that we are only interested in the positive integers (states) when it comes to verification.

Paths

While a path represented a concrete resolution of the nondeterministic choices for an LTS, it represents concrete outcomes of the probabilistic choices for a DTMC:

Definition 29 (Paths in DTMC)**.** Given a DTMC $M = \langle S, T, s_{init}, AP, L \rangle$, a (finite) *path in M from s_0 to s_n* of length $n \in \mathbb{N}$ is a finite sequence $s_0 s_1 \dots s_n$ where $s_i \in S$ for all $i \in \{0, \dots, n\}$ and $s_{i-1} \rightarrow s_i$ for all $i \in \{1, \dots, n\}$. An (infinite) *path in M starting from s_0* is an infinite sequence $s_0 s_1 s_2 \dots$ where for all $i \in \mathbb{N}$ we have that $s_i \in S$ and $s_i \rightarrow s_{i+1}$.

All the notation derived from the definition of finite and infinite paths (such as the length of a path or the set $\mathrm{Reach}(M)$ of reachable states) carries over from LTS unchanged. As DTMC have no deadlock states, all maximal paths are infinite. The trace of a path can be defined for DTMC in the same way as for LTS.

Parallel Composition

As DTMC are not closed under a concurrency semantics built on interleaving [Her02, HZ11], we have to be careful to define a parallel composition operator that, given two DTMC, actually maps to a product that is a DTMC again. A natural [HZ11] parallel composition operator that achieves this goal can be defined as follows:

Definition 30. The *parallel composition of two DTMC*

$$M_i = \langle S_i, T_i, s_{init_i}, AP_i, L_i \rangle,$$

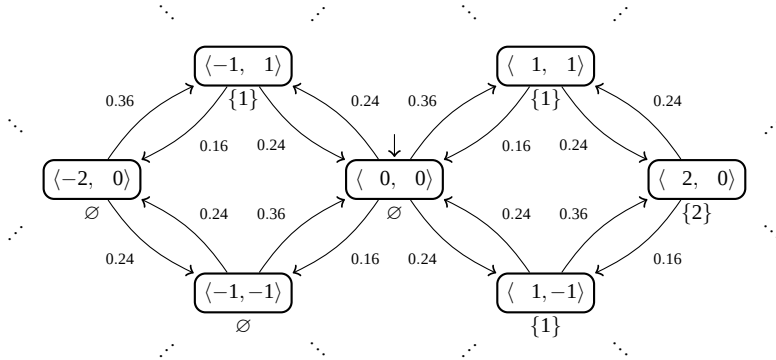


Figure 3.19: DTMC lock-step parallel composition of two random walks

$i \in \{1,2\}$, is the DTMC $M_1 \parallel M_2 = \langle S_1 \times S_2, T, \langle s_{init_1}, s_{init_2} \rangle, AP_1 \cup AP_2, L \rangle$ where
– $T \in (S_1 \times S_2) \to \text{Dist}(S_1 \times S_2)$ s.t. $T(\langle s_1, s_2 \rangle) = T_1(s_1) \otimes T_2(s_2)$ and
– $L \in (S_1 \times S_2) \to \mathscr{P}(AP_1 \cup AP_2)$ s.t. $L(\langle s_1, s_2 \rangle) = L_1(s_1) \cup L_2(s_2)$.

In contrast to LTS, there is no interleaving; both component DTMC instead proceed in a synchronous fashion. This is similar to the parallel composition of two LTS in which all transitions have the same label a, $\tau \neq a \in A_1 \cap A_2$, and is often called *fully synchronous* parallel composition with both components proceeding *in lock-step* [HZ11, Sha12]. We call this the *lock-step semantics* for parallel composition to contrast it with the interleaving semantics that we use for LTS (cf. Section 3.1.1). It avoids the introduction of nondeterminism, which would otherwise result in the parallel composition of two DTMC no longer being a DTMC itself. This also means that it makes sense to think of every transition in both the components and the composition as taking "one time unit". With an interleaving semantics as for LTS, this way of adding time into the formalism would not work out: When each of two parallel components independently performs a transition that takes one time unit, the total time to perform both transitions should also be one time unit. When interleaved, however, the two transitions are executed in sequence in the product (albeit nondeterministically in any order), which would take two time units.

Example 11. Figure 3.19 shows the parallel composition of two identical random walks $M_{rw_1} \parallel M_{rw_2}$ from Example 10. Observe how the probabilities are multiplied and states such as $\langle 0,1 \rangle$ are unreachable due to the lock-step semantics.

Variables

We finally note that DTMC can be extended to VDTMC through the addition of discrete variables similarly to how we extended LTS in Section 3.1.2. Notable differences are that

– the edge function is in $Loc \to Bxp \times \mathscr{P}(Upd \times Loc \to Axp)$, i.e. an edge leads to a function mapping pairs of an update and a target location to a *weight* expression, and
– the case that the guard of the (only) edge leaving a location evaluates to *false* should be interpreted as the existence of a self-loop in the underlying DTMC.

A function $m \in Upd \times Loc \to Axp$ that evaluates to a non-zero weight expression only on a countable range $R \subset Upd \times Loc$ denotes a *symbolic* probability distribution. Given a valuation v for the variables of the VDTMC, the corresponding concrete probability distribution is determined as

$$m^v_{conc}(\langle U,l\rangle) = \frac{[\![m(\langle U,l\rangle)]\!](v)}{\sum_{\langle U',l'\rangle \in R} [\![m(\langle U',l'\rangle)]\!](v)}.$$

We require that, for all reachable valuations v, we have

– $[\![m(\langle U,l\rangle)]\!](v) \geq 0$ for all $\langle U,l\rangle \in R$ and
– $\sum_{\langle U,l\rangle \in R} [\![m(\langle U,l\rangle)]\!](v)$ converges to a value in $\mathbb{R}^+$.

We consider it a modelling error if this is not the case, just like inconsistent assignments in parallel composition. We omit the complete definition of VDTMC here as we formally define VMDP, of which VDTMC can be seen as a special case, in Section 4.2.

3.2.1 Modelling

In order to build VDTMC models with MODEST, we first of all need to be able to specify probabilistic choices. This can be accomplished with the `palt` keyword, the probabilistic version of `alt`, which extends our grammar of process behaviours as follows:

$$P ::= \ldots \mid act\ \texttt{palt}\ \{\, :w_1 : U_1; P_1\ \ldots\ :w_k : U_k; P_k \,\}$$

where the w_j are expressions in Axp and the U_j are updates. The corresponding inference rule (*palt*) is

$$\frac{(a \in Act)}{a\ \texttt{palt}\ \{\, :w_1 : U_1; P_1 \ldots :w_n : U_n; P_1 \,\} \xrightarrow{tt,a} \{\, \langle U_1, P_1\rangle \mapsto w_1, \ldots, \langle U_n, P_n\rangle \mapsto w_n \,\}}$$

Edges now lead to symbolic probability distributions over an update and a target MODEST process behaviour. We call each of the $\langle w_i, U_i, P_i\rangle$ a *branch* of the probabilistic choice. We keep the edge labels in the inference rule as for

```
process RandomWalk(int i)
{
   sync palt {
   :6: {= i = i + 1 =}; RandomWalk(i)
   :4: {= =}; RandomWalk(i - 1)
   }
}

RandomWalk(0)
```

Figure 3.20: A random walk in MODEST

VLTS although they are not supported by VDTMC. We assume that MODEST models intended to represent VDTMC use a unique label $sync \in Act \setminus \{\tau, \flat, \perp\}$ on all edges instead. Also, care must be taken not to introduce nondeterministic choices in MODEST code intended to represent a VDTMC; this primarily means that we cannot use the `alt` keyword, and we must specify only one alternative per `do` construct.

Example 12. Figure 3.20 shows a MODEST model of the random walk DTMC presented before in Figure 3.18. The parallel composition of two `RandomWalk` processes results in a model whose semantics correspond to the DTMC shown in Figure 3.19.

Syntactically, MODEST allows the omission of *either* the assignment block following the weight *or* the process behaviour following the semicolon. The former is equivalent to an empty assignment set; this could be done in line 5 of the MODEST code of the previous example. The latter is treated as if the process behaviour were ✓ instead (which cannot be specified syntactically otherwise). Omitting the action prefix of the `palt` keyword is equivalent to the prefix τ. In the full semantics of MODEST (see Appendix A), the inference rule (*assgn*) as presented in Section 3.1.3 in fact does not exist. Process behaviours of the form $a\ \{= \ldots =\}$ are instead treated as shorthands for a probabilistic alternative that is prefixed by a and has a single branch with weight 1 and process behaviour ✓.

3.2.2 Properties

In the analysis of Markov chains, we are again primarily interested in reachability properties. For LTS, these properties had the form $\exists \diamond \phi$. The existential quantifier in this formulation, intuitively speaking, quantifies over the nondeterministic choices in the LTS. As DTMC have no nondeterminism, but

are instead based on probabilistic choices, we replace $\exists$ by a probabilistic quantifier or query operator. The result are *probabilistic reachability* properties:

$$\mathrm{P}(\diamond\,\phi) \quad \textit{(quantitative form)} \qquad \mathrm{P}(\diamond\,\phi) \sim x \quad \textit{(qualitative form)}$$

where $\sim \in \{<, \leq, >, \geq\}$ and $x \in [0,1]$. The quantitative form specifies a query for the probability of reaching a state whose labelling satisfies ϕ, while the qualitative form expresses a constraint on that probability, i.e. a requirement. A verification procedure for the quantitative form can also be used for the qualitative one by simply comparing the result to the bound x. However, specialised approaches that may perform better are available for the verification of the qualitative form. Note that both forms are quantitative properties in the sense of Section 1.2 since they refer to a quantitative modelling aspect, namely probabilities.

Example 13. For the random walk model introduced in Example 10, property $\mathrm{P}(\diamond\,5)$ asks for the probability of reaching state 5 (which is labelled 5). In the same way, property $\mathrm{P}(\diamond\,6) > 1/2$ states the requirement that the probability of reaching state 6 has to be greater than $1/2$.

To include these properties in the corresponding MODEST model of Example 12, parameter `i` of process `RandomWalk` needs to be turned into a global variable. Then, the two properties can be expressed in MODEST syntax as

```
property P_5 = P(<> i == 5);
```

and

```
property P_6 = P(<> i == 6) > 0.5;
```

giving them names $\mathtt{P}_5$ and $\mathtt{P}_6$.

In order to define the meaning of probabilistic reachability properties, we need to precisely say what "the probability of reaching a state" is. We do this by assigning probabilities to the (finite) paths that lead to this state (from the initial state of the DTMC). Intuitively, these probabilities are the product of the probabilities given by the probabilistic transition function. Formally, we need the construct of *cylinder sets* to properly define a probability measure on finite paths (since, for example, one finite path can be a prefix of infinitely many others):

Definition 31 (Cylinder set [BK08])**.** Given a DTMC M, the cylinder set of a finite path $\hat{\pi} \in \mathrm{Paths}_{\mathrm{fin}}(M)$ is defined as

$$\mathrm{Cyl}(\hat{\pi}) \stackrel{\text{def}}{=} \{\, \pi \in \mathrm{Paths}(M) \mid \hat{\pi} \text{ is a prefix of } \pi \,\}.$$

Recall that, for a DTMC, all elements of $\mathrm{Paths}(M)$ are infinite paths. Now let

$$\mathrm{Cyl}(M) \stackrel{\text{def}}{=} \{\, \mathrm{Cyl}(\hat{\pi}) \mid \hat{\pi} \in \mathrm{Paths}_{\mathrm{fin}}(M) \,\}$$

denote the set of all cylinder sets of a DTMC M. Then $\langle \mathrm{Cyl}(M), \sigma(\mathrm{Cyl}(M)) \rangle$ is a measurable space. This allows us to define a probability measure μ_M for M by assigning probabilities to cylinder sets as follows [BK08]:

Definition 32 (Probability of a cylinder set)**.** For DTMC $M = \langle S, T, s_{init}, AP, L \rangle$, μ_M is the probability measure on the measurable space $\langle \mathrm{Cyl}(M), \sigma(\mathrm{Cyl}(M)) \rangle$ that is uniquely determined by

$$\mu_M(\mathrm{Cyl}(s_0 \dots s_n)) = \prod_{0 \leq i < n} T(s_i)(s_{i+1})$$

where $s_0 = s_{init}$ by definition.

We can informally say that the measure μ_M assigns probabilities to finite paths, which is what is needed to define the semantics of a probabilistic reachability query [BK08]:

Definition 33 (Semantics of DTMC probabilistic reachability queries)**.** Given a DTMC $M = \langle S, T, s_{init}, AP, L \rangle$ and a probabilistic reachability query $\mathrm{P}(\diamond \phi)$ over the atomic propositions of M, the semantics of that property is

$$\begin{aligned} [\![\mathrm{P}(\diamond \phi)]\!]_M &\stackrel{\text{def}}{=} \mu_M \left(\cup_{\hat{\pi} \in \mathrm{Paths}_{\mathrm{fin}}(\phi, M)} \mathrm{Cyl}(\hat{\pi}) \right) \\ &= \sum_{\hat{\pi} \in \mathrm{Paths}_{\mathrm{fin}}(\phi, M)} \mu_M(\mathrm{Cyl}(\hat{\pi})) \end{aligned}$$

where

$$\mathrm{Paths}_{\mathrm{fin}}(\phi, M) = \{ s_0 \dots s_n \in \mathrm{Paths}_{\mathrm{fin}}(M) \mid \phi(L(s_n)) \wedge \forall 0 \leq i < n \colon \neg \phi(L(s_i)) \}$$

is the set of finite paths on which the last state is the only one whose labelling satisfies ϕ.

Again, we will omit M when it is clear from the context. Since $\mathrm{Paths}_{\mathrm{fin}}(\phi, M)$ is a countable set, the union of the corresponding cylinder sets is a countable union and thus measurable. $[\![\mathrm{P}(\diamond \phi)]\!]$ is therefore well-defined. The probability of the union is equal to the sum of the probabilities of the individual cylinder sets because they are pairwise disjoint by the definition of $\mathrm{Paths}_{\mathrm{fin}}(\phi, M)$. The semantics of a probabilistic reachability property in qualitative form $\mathrm{P}(\diamond \phi) \sim x$ is simply the result of the comparison of $[\![\mathrm{P}(\diamond \phi)]\!]$ with x according to $\sim$.

Temporal Logics for DTMC

We mention that more complex properties can be specified for DTMC using temporal logics. As an example that will reappear later, PCTL* [Bai98] is an adaption of CTL* to the probabilistic setting. The $\forall$ and $\exists$ quantifiers over paths of CTL* are replaced by a probabilistic quantifier $\mathrm{P}_{\sim x}$ that obtains the

probability of a set of paths characterised by a path formula and compares that probability to the bound x just as in our definition of probabilistic reachability. In this way, the property $\mathrm{P}(\diamond\phi) \sim x$ can be written in PCTL* as $\mathrm{P}_{\sim x}(\diamond\phi)$.

3.2.3 Analysis

When it comes to the verification of probabilistic reachability properties on DTMC, there are fundamentally two techniques: The first is to find the states of interest with a standard reachability analysis (as for LTS) and then explicitly compute the total probability mass of the paths that reach them. This is the exhaustive model checking approach for DTMC. The other way is to randomly generate a number of paths, determine for each whether the relevant set of states is reached, and then use statistical methods to derive an approximation of the reachability probability. We refer to the path generation step as *simulation* of the DTMC, and the complete procedure is called *statistical model checking*, or SMC for short. It is the DTMC analogue of the randomised testing approach of Section 3.1.5.

We will see that the results of exhaustive model checking are exact, or can at least be made arbitrarily close to the true probabilities, up to numerical precision. However, exhaustive model checking is only applicable to finite-state DTMC and suffers from the state-space explosion problem. SMC, on the other hand, can be used with infinite-state models and can work with just a constant amount of memory. The drawback is that its results are correct only with a certain confidence or probability, the details depending on the method used for the statistical evaluation step. When we evaluate memory usage and runtime in the remainder of this section, we use the notation and assumptions from LTS analogously.

Model Checking

The basic exhaustive model checking algorithm for DTMC reduces the computation of reachability probabilities to the problem of solving a linear equation system as shown in Algorithm 4 (based on [BK08]). The equation system in line 11 has a unique solution because of the finiteness of M and the way $S_{=0}$, $S_{=1}$ and $S_{=?}$ are chosen [BK08, Theorem 10.19]: As computed in lines 1 and 2, $S_{=0}$ contains exactly those states from which the probability of reaching a ϕ-state is zero, while $S_{=1}$ contains exactly those from which this probability is one. The preprocessing step of computing these sets can be performed based on graph searches similar to LTS model checking as in Algorithm 1.

The memory usage of a naïve implementation of this algorithm is in $\mathrm{O}(n^2)$ in order to explicitly store the matrix, which is a $|S| \times |S|$-matrix in the worst

Input: Finite-state DTMC $M = \langle S, T, s_{init}, AP, L \rangle$ and property $\mathrm{P}(\diamond \phi)$
Output: $[\![\mathrm{P}(\diamond \phi)]\!]_M$ (value in $[0,1]$)

1 $S_{=0} := \{ s \in S \mid \neg \phi(L(s)) \wedge \forall s' \in S \colon (\mathrm{Paths}(s,s') \neq \varnothing \Rightarrow \neg \phi(L(s'))) \}$
2 $S_{=1} :=$ smallest $S' \subseteq S$ s.t. $s \in S' \Leftrightarrow \phi(L(s)) \vee \forall s' \colon (s \to s' \Rightarrow s' \in S')$
3 $S_{=?} := S \setminus \{ S_{=0} \cup S_{=1} \}$
4 **if** $s_{init} \in S_{=1}$ **then return** 1
5 **else if** $s_{init} \in S_{=0}$ **then return** 0
6 **else**
7 $\quad f :=$ some bijection $\{1, \ldots, |S_{=?}|\} \to S_{=?}$
8 $\quad A :=$ matrix with entries $a_{ij} := T(f(i))(f(j))$
9 $\quad b := \langle b_1, \ldots, b_{|S_{=?}|} \rangle$ with $b_i := \sum_{s \in S_{=1}} T(f(i))(s)$
10 $\quad$ **with** $x = \langle x_1, \ldots, x_{|S_{=?}|} \rangle$**:**
11 $\quad\quad$ solve linear equation system $x = Ax + b$
12 $\quad\quad$ **return** $x_{f(s_{init})}$
13 $\quad$ **end**
14 **end**

Algorithm 4: Model checking for DTMC using linear equation systems

case. The probability matrix of typical models, however, is sparse, so memory can in practice be saved by using a more compact representation. The runtime depends on n and the method used to solve the equation system. Exact linear equation system solving has polynomial runtime, e.g. $\mathrm{O}(n^3)$ for Gaussian elimination. It does not scale to realistic model sizes in practice, but various numerical approaches that approximate a solution up to some error $\varepsilon > 0$ (in a numerically stable manner) exist and are commonly used by model checking tools [KNP10].

We can alternatively reformulate the algorithm to explicitly use a (numeric) dynamic programming technique called *value iteration*. In this approach, a value is stored for each state. That value is then iteratively improved for all states based on the values of the respective successor states. In our case, shown in Algorithm 5, that value approximates the probability of reaching a ϕ-state. Given a state s and the values of the successor states, the (new) value of s is simply the sum of the values of the successors s'_i weighted by the probability of moving from s to s'_i. In this way, the value of s in iteration k is (greater than or equal to) the probability of reaching a ϕ-state from s in no more than k steps. In Algorithm 5, these improving values are calculated in line 6. If the DTMC at hand contains cycles, the algorithm will in general not obtain the exact reachability probabilities but merely an ever-improving approximation. This is why

Input: Finite-state DTMC $M = \langle S, T, s_{init}, AP, L \rangle$, property $\mathrm{P}(\diamond\, \phi)$, $\varepsilon > 0$
Output: $[\![\mathrm{P}(\diamond\, \phi)]\!]_M$ (value in $[0,1]$)

```
1  with v ∈ S → [0,1]:
2      foreach s ∈ S do v(s) := 1 if φ(L(s)), otherwise 0
3      repeat
4          error := 0
5          foreach s ∈ { s' ∈ S | ¬φ(L(s')) } do
6              v_new := Σ_{s' ∈ support(T(s))} T(s)(s') · v(s')
7              if v_new > 0 then error := max{ error, |v_new − v(s)| / v(s) }
8              v(s) := v_new
9          end
10     until error < ε
11     return v(s_init)
12 end
```

Algorithm 5: Model checking for DTMC using value iteration

we use an additional parameter ε to specify an upper bound for the acceptable difference between subsequent values. This allows value iteration to terminate. We chose to compare ε to the *relative* difference here, which in turn is computed in line 7. It is important to note that while this termination scheme works well in practice, it does not give any guarantee on the relationship between ε, the computed probability, and the actual (exact) probability [FKNP11]. As given, the algorithm does not rely on an a priori reachability analysis. In practice, however, performing value iteration on restricted sets such as $\mathrm{Reach}(M)$, $\{ s \in S \mid \exists s' \in S\colon\ \phi(L(s')) \wedge \mathrm{Paths}(s,s') \neq \varnothing \}$ or their intersection instead of S can significantly improve performance and reduce memory usage.

The memory usage of the value iteration approach is in $\mathrm{O}(n)$ since we only store a vector of values instead of a matrix of probabilities. Its runtime strongly depends on the threshold ε in combination with the (cyclic) structure of the model. In particular, value iteration may converge slowly even for models with relatively small state spaces [SGS+13].

Statistical Model Checking

Exhaustive model checking for DTMC still suffers from state-space explosion, and takes longer than LTS model checking due to the extra numeric computations involved. Additionally, problems of numeric stability arise [WKHB08].

```
1  function simulate(M = ⟨S, T, s_init, AP, L⟩, φ, d)
2      s := s_init, seen := ∅
3      for i = 1 to d do
4          if φ(L(s)) then return true
5          else if s ∈ seen then return false
6          else if T(s) is Dirac then seen := seen ∪ {s}
7          else seen := ∅
8          s := choose a state randomly according to T(s)
9      end
10     return unknown
```

Algorithm 6: Path generation for DTMC, with cycle detection

An alternative approach is to extend the randomised testing technique of Algorithm 3 to become *statistical model checking* (SMC) for DTMC, all the while keeping its desirable properties: very low memory usage, and runtime only depending on the desired "precision". SMC consists of two largely separate steps: The exploration of individual paths through the model, and the statistical evaluation of data collected from those paths. We refer to the first step as simulation.

Finite paths for unbounded reachability As we are interested in probabilistic reachability properties of the form $\mathrm{P}(\diamond\,\phi)$ and $\mathrm{P}(\diamond\,\phi) \sim x$, we need to explore paths in a way that allows us to determine whether a ϕ-state is eventually reached or not. In contrast to LTS randomised testing, we now want to at least try to avoid giving up and returning *unknown* when no ϕ-state is seen up to a certain path length. Algorithm 6 shows a practical approach to this problem: It keeps track of the states visited since the last non-Dirac choice; when we return to such a state, we have discovered a cycle of probability-one transitions without ϕ-states. Since we cannot escape from this kind of cycle, we can then conclude that no continuation of the current path will ever reach a ϕ-state. To ensure termination for models whose non-ϕ-paths do not all end in a Dirac cycle, we still include a depth parameter d as we did in LTS randomised testing. (Observe that we could already have implemented an analogous check there, but instead accepted that it was merely a semi-decision algorithm.) The memory usage of Algorithm 6 is therefore in $\mathrm{O}(d)$; constant memory usage could be achieved by only checking for loops instead (as is done in e.g. PRISM [KNP11]).

This entire complication results from the fact that we verify unbounded reachability properties, yet we can only explore finite prefixes of paths. If instead we were interested in step-bounded reachability properties of the form $\mathrm{P}(\Diamond_{\leq k}\,\phi)$[1], Algorithm 6 could be simplified to use $d = k$, omit cycle detection, and just return *false* in line 10. However, our approach is more general since a step bound can just as well be encoded in the model. If the model is given as a VDTMC, we simply have to add a new integer variable, increment it in every assignment, and add a clause to every guard checking whether the variable does not yet exceed the step bound. This solution is a special case of a general recipe to transform models of reactive systems into a form that has all paths terminate in a Dirac cycle: We first identify a finite horizon of the system behaviour as the *simulation scenario*. Then, we make all states reached after the conclusion of the scenario loop back to themselves with probability one. As an example, consider modelling a communication protocol that continuously sends files from a sender to a receiver: We could pick the transmission attempt for a certain single file (not necessarily the first one) as the simulation scenario. In fact, the simple BRP model of Example 8 is already built in this way: It deadlocks after the first piece of data has been transmitted successfully or the sender has given up. In the remainder of this section, we shall assume that the DTMC to be analysed is such that there exists a $d \in \mathbb{N}$ for which function `simulate` of Algorithm 6 always returns *true* or *false*, and that value is used for d.

Sample mean and confidence Every run of function `simulate` (a *simulation run*) under these conditions (assuming it never aborts with *unknown*) explores a finite path $\hat{\pi}$ and returns *true* if $\hat{\pi} \in \mathrm{Paths}_{\mathrm{fin}}(\phi, M)$, otherwise *false* (which in particular means that no path in $\mathrm{Cyl}(\hat{\pi})$ contains a ϕ-state). Since the probability distributions used in line 8 of Algorithm 6 are exactly those given by the model, the probability of encountering any one of these paths in a single run is $\mu_M(\mathrm{Cyl}(\hat{\pi}))$. We let the random variable X be the result of a simulation run. If we interpret *true* as 1 and *false* as 0, then X follows the Bernoulli distribution with expected value $[\![\mathrm{P}(\Diamond\,\phi)]\!]$. This is the foundation for the statistical evaluation of simulation data [Ros06, Chapter 7]: A batch of k simulation runs corresponds to k random variables that are independent and identically distributed with expected value $p = [\![\mathrm{P}(\Diamond\,\phi)]\!]$. The average $\overline{X} = \sum_{i=1}^{k} X_i/k$ of the k random variables, the *sample mean*, is then an approximation of p. (It is in fact an unbiased estimator of that actual mean.)

The single quantity of sample mean, however, is fairly useless for verification. We also need to know how good an approximation of p it is. The key

[1] Read: "What is the probability of reaching a ϕ-state *in at most k steps* from the initial state?"

Input: DTMC $M = \langle S, T, s_{init}, AP, L \rangle$, $\mathrm{P}(\diamond \phi)$, $d \in \mathbb{N}$, two of $\{k, \varepsilon, \delta\}$
Output: $[\![\mathrm{P}(\diamond \phi)]\!]_M$ (value in $[0,1]$) and confidence $\langle k, \varepsilon, \delta \rangle$, or *unknown*

compute $\{k, \varepsilon, \delta\}$ s.t. $k \geq \ln(\frac{2}{\delta})/(2 \cdot \varepsilon^2)$ // *requires that* $k \cdot \varepsilon^2 < \ln(2)/2$
$i := 0$
for $j = 1$ **to** k **do**
    $v :=$ `simulate(`M, ϕ, d`)`
    if $v = true$ **then** $i := i + 1$
    else if $v = unknown$ **then return** *unknown*
end
return j/k and $\langle k, \varepsilon, \delta \rangle$

Algorithm 7: SMC for DTMC using the APMC method

parameter to influence the quality of approximation is k, the number of simulation runs performed. The higher k is, the more *confident* can we be that a concrete observed sampled mean $\overline{x}$ is close to p. There are various statistical methods to precisely describe this notion of confidence and determine the actual confidence for a given set of simulation results. A widely-used method is to compute a *confidence interval* [Ros06] of *width* 2ε around $\overline{x}$ with *confidence level* $100 \cdot (1-\alpha)\,\%$. Typically, k and α are specified by the user and ε is then derived from α and the collected observations of the X_i. For confidence intervals, we have that

$$\mathbb{P}(\overline{X} - \varepsilon < p < \overline{X} + \varepsilon) \approx 1 - \alpha.$$

This means the following: In the long run, $100 \cdot (1-\alpha)\,\%$ of the confidence intervals $[\overline{x} - \varepsilon, \overline{x} + \varepsilon]$ around concrete values $\overline{x}$ obtained from entirely independent sets of simulation runs will contain the actual mean p. In particular, for a single given confidence interval, there is no statement directly relating $\overline{x}$ and p. Aside from this slightly non-straightforward interpretation, basic confidence interval methods have other problems in different scenarios [AC98], too. This is why we prefer to use the *APMC method* for quantitative and *sequential testing* for qualitative probabilistic reachability properties instead. We give a summary of both methods below that is sufficient for the way we use them in the remainder of this thesis:

The APMC method The *approximate probabilistic model checking* (APMC) method was introduced together with and originally implemented in a tool of the same name [HLMP04]. The key idea is to use Chernoff-Hoeffding bounds [Hoe63] to relate the three parameters of approximation ε, confidence

level δ and number of simulation runs k in such a way that we have

$$\mathbb{P}(|\overline{X} - p| > \varepsilon) < \delta,$$

i.e. for one simulation experiment, the difference between computed and actual probability is at most ε with probability $1-\delta$. Note that in this formula, parameter k appears inside the definition of $\overline{X}$. There are at least two ways to relate the three values: APMC originally used

$$\mathbb{P}(|\overline{X} - p| > \varepsilon) < 2 \cdot e^{-\frac{k\varepsilon^2}{4}},$$

which leads to the bound $k \geq 4 \cdot \ln(2/\delta)/\varepsilon^2$ for the number of runs, whereas

$$\mathbb{P}(|\overline{X} - p| > \varepsilon) < 2 \cdot e^{-2k\varepsilon^2}$$

leads to fewer simulation runs in practice [Nim10] with the resulting bound of

$$k \geq \ln(2/\delta)/(2 \cdot \varepsilon^2) \tag{3.1}$$

Whenever two of the parameters are given, the third can thus be derived. Algorithm 7, based on [HLMP04] and [Nim10], combines this relationship and the `simulate` function of Algorithm 6 to implement the APMC method in our setting. In particular, the number of simulation runs that is necessary to achieve the desired confidence is either given, or it can be computed as the very first step.

Sequential testing for qualitative reachability In case the property to be verified is in qualitative form $\mathrm{P}(\diamond\phi) \sim x$, i.e. a Boolean decision needs to be made, we can use sequential testing: After each simulation run, we check whether we have collected enough evidence to answer *true* or *false* with the desired confidence. If that is not the case, more simulation runs are performed until we can decide the property.

The test we use is Wald's *sequential probability ratio test* (SPRT, [Wal45]). It has three parameters: the *type I error* α, the *type II error* β, and the *indifference* ε. It tests for whether hypothesis $H_0 = [\![\mathrm{P}(\diamond\phi)]\!] > x + \varepsilon$ or hypothesis $H_1 = [\![\mathrm{P}(\diamond\phi)]\!] < x - \varepsilon$ is true. The type I error specifies the probability of wrongly accepting H_0, whereas the type II error specifies the probability of wrongly accepting H_1. If the actual value of $[\![\mathrm{P}(\diamond\phi)]\!]$ lies within the indifference region $[x-\varepsilon, x+\varepsilon]$, there are no guarantees for the result. The test works by keeping track of the product of the likelihood ratios, and after every run, comparing that (newly updated) value to bounds computed based on α and β. A decision will eventually be reached with probability 1. An implementation of the SPRT-based approach for DTMC in our setting is shown in Algorithm 8, which is based on [YKNP06] and [Nim10] with the SPRT computations transferred to the logarithmic domain as in the UPPAAL SMC tool [DLL+11a].

Input: DTMC $M = \langle S, T, s_{init}, AP, L \rangle$, $\mathrm{P}(\diamond\phi) \sim x$, $d \in \mathbb{N}$, $\varepsilon, \alpha, \beta \in (0,1)$
Output: $[\![\mathrm{P}(\diamond\phi) \sim x]\!]_M$ (*true* or *false*), or *unknown*

$p_0 := \min\{x+\varepsilon, 1\}$, $p_1 := \max\{x-\varepsilon, 0\}$
$a := \log((1-\beta)/\alpha)$, $b := \log(\beta/(1-\alpha))$
$r := 0$
repeat
  $v :=$ `simulate`(M, ϕ, d)
  if $v =$ *unknown* **then return** *unknown*
  else if $v =$ *true* **then** $r := r + \log p_1 - \log p_0$
  else $r := r + \log(1-p_1) - \log(1-p_0)$
  if $r \leq b$ **then return** $\sim \in \{>, \geq\}$ // *likely* $[\![\mathrm{P}(\diamond\phi)]\!] > p_0 > x$
  else if $r \geq a$ **then return** $\sim \in \{<, \leq\}$ // *likely* $[\![\mathrm{P}(\diamond\phi)]\!] < p_1 < x$
end

Algorithm 8: Statistical model checking for DTMC using Wald's SPRT

Memory usage and runtime As both approaches presented only keep track of a fixed number of real or integral values, their memory usage is determined by that of Algorithm 6, which in turn depends on the way the cycle detection is performed: It is in $\mathrm{O}(d)$ as specified or in $\mathrm{O}(1)$ if just loop detection is used instead. The runtime entirely depends on the desired confidence. For the APMC method, we can directly see the dependence on δ and ε in Formula 3.1. For the SPRT-based approach, the difference between the actual probability p and the bound x plays an important role: as x and p get closer, the number of simulation runs needed explodes. This is shown very clearly in [YKNP06], and [Nim10] extensively compares the performance of the different statistical methods in terms of the number of simulation runs needed.

3.3 Compositionality

We have now seen the modelling formalisms of LTS and DTMC as well as their extensions with variables, VLTS and VDTMC. For all of them, there is a parallel composition operator that allows two smaller models to be combined into a single, larger one of the same type. In fact, such an operator can be specified in a natural way for all automata-based models that will be introduced in this thesis. In this section, we briefly review useful notions related to building larger models out of components using these parallel composition operators.

Open and Closed Systems

In the case of LTS, the transitions of the components in a parallel composition can be taken asynchronously unless their label is in the shared alphabet. This principle in fact applies to all models that support nondeterministic choices (i.e. all models except for DTMC). However, if we know that a model will not be used in a parallel composition, the transition labels have no influence on verification results since properties only refer to the atomic propositions of the states. We therefore distinguish between *open model* and *closed* models: We expect that an open model will be composed in parallel to some yet-unknown other component. Therefore, transition labels are important; in particular, transitions not labelled τ could be disabled in certain states in the composition (or delayed, in the timed models that will be introduced in Chapter 5). A closed model, however, will only be subject to verification procedures, but not parallel composition. No expressiveness is therefore lost if we change the labels of all transitions to τ. Obviously, an open model can at any time be transformed into a closed one, but not vice-versa:

Definition 34 (Closed models)**.** Given an automata-based model M with transition or edge labels from an alphabet A, the corresponding B-closed model $\text{closed}_B(M)$ with $B \subseteq A$ is identical to M except that all transitions or edges that have label $a \in B$ in M are instead labelled τ. We denote the closed model corresponding to M by $\text{closed}(M) \stackrel{\text{def}}{=} \text{closed}_A(M)$. We say that M is closed if all its transitions or edges have label τ.

If a model M corresponds to a MODEST model with process behaviour P, then $\text{closed}_B(M)$ corresponds to `hide` B $\{ P \}$.

Networks of Systems

The parallel composition operators we define for the automata models considered in this thesis are associative and commutative. It is therefore natural to think of the parallel composition of several components as the composition of a *set* of models instead of as a parallel composition *tree*. We call such a set of automata models a *network of automata*:

Definition 35 (Network of automata)**.** For any transition- or edge-labelled automata-based modelling formalism with an associative and commutative parallel composition operator $\|$, a *network of automata* is a set N of models $M_1, \ldots, M_n$, $n \in \mathbb{N}$. Its semantics is the closed parallel composition of the models,

$$\text{closed}(((M_1 \parallel M_2) \parallel \ldots) \parallel M_n).$$

We denote the modelling formalism "network of ..." with the prefix N; for example, networks of LTS are referred to as NLTS. We have already seen models that were automata networks: the set consisting of the four LTS given in figures 3.1 and 3.2 is an NLTS whose semantics is shown in Figure 3.4. Also, a MODEST model whose (overall) process behaviour is the parallel composition `par` $\{ ::P_1 \ldots ::P_n \}$ represents a network of automata, the component automata being the semantics of the P_i for $i \in \{1, \ldots, n\}$. In fact, the MODEST TOOLSET directly converts a parsed MODEST model into a network of automata internally. Using this internal representation for all further operations and algorithms allows it to easily support other modelling languages such as PRISM's guarded commands, whose *modules* are simply a different name for the components of an automata network.

4 Markov Decision Processes

Nondeterminism is a crucial feature for complex modelling and verification tasks: It opens the door for abstraction, it permits underspecification, and it allows compositionality and component-based modelling beyond lock-step semantics [HZ11]. The availability of probabilistic decisions enables the modelling of systems subject to an environment whose behaviour is only statistically predictable, and of systems that employ randomisation internally to achieve correctness or good performance. If we combine the two basic formalisms of labelled transition systems for nondeterministic choices and discrete-time Markov chains for probabilistic decisions, the result is a unified probabilistic-nondeterministic modelling formalism known as Markov decision processes (MDP, [Put94]). MDP are a mainstay of optimisation tasks in economics and of planning problems in artificial intelligence [KS06]. In these settings, the controllable choices are represented as nondeterministic, while an environment reacts in a probabilistically quantified manner. The usual objective is then to obtain a strategy, i.e. a way of resolving the nondeterministic choices, that maximises some goal function.

In the area of modelling and verification, MDP were originally introduced as probabilistic automata (PA, [Seg95]). Although there is a small difference in definition between MDP and PA, we treat the two as equivalent in this thesis, and in fact use the slightly more general PA-style definition. After the formal definition of MDP, including their path semantics and a parallel composition operator, we clearly state the relationship between MDP and the previous models of LTS and DTMC. We also formally describe the extended model of MDP with variables, as these are what we specify in the MODEST language. As in the previous chapter, the class of properties we focus on is probabilistic reachability. We formally define their meaning on MDP and summarise exhaustive probabilistic model checking techniques available for their analysis. In contrast to the optimisation or planning scenario described above, in the verification context we cannot consider the nondeterministic decisions as controllable (cf. the

uses of nondeterminism given in Section 1.2). Verification therefore needs to consider *all* possible resolutions, and we get an interval of probabilities instead of a single reachability probability as for DTMC. Every concrete value out of such an interval corresponds to particular resolutions of the nondeterministic choices. Our properties will thus query for or compare bounds to the maximum or minimum reachability probability.

A core contribution of this thesis is in providing sound techniques to use statistical model checking for nondeterministic models, in particular MDP. The underlying problem is that SMC, as a simulation-based approach, is limited to fully stochastic models like Markov chains: The choices available in each state encountered during simulation have to be resolved in order to proceed to a single next state. This includes the nondeterministic choices. Unless special care is taken, the result of SMC is thus merely *some* value out of the interval of reachability probabilities. This is useless for many verification scenarios. Consider checking whether the probability of reaching a certain set of bad states is below some value close to 0: The only value relevant to this property is the maximum, or worst-case, probability of reaching those states. Any other value may be unfoundedly optimistic. We present five approaches to deal with this problem, two of which have been developed by the author and collaborators and will be explained in full detail.

Origins

The partial order reduction-based approach presented in Section 4.7.1 has been developed by the author with support from Jonathan Bogdoll and Luis María Ferrer Fioriti [BFHH11]. Jonathan wrote the first implementation of the technique within the modes tool while Luis provided valuable insights into the different variants of probabilistic partial order reduction and suggested using the BEB model as a case study.

The confluence reduction-based approach of Section 4.7.2 has been developed by the author in close collaboration with Mark Timmer [HT13]. Starting from the jointly developed idea and a first algorithm sketch, Mark adapted confluence to the SMC setting (i.e. state-based verification of closed models), worked out the details of algorithms 19 and 20 and proved correctness. The author implemented the technique in the modes tool and selected and analysed the example models shown in Section 4.7.3. Lemma 4 and its proof have been newly developed for this thesis by the author with feedback from Mark.

The evaluation of both approaches that is presented in Section 4.7.3 has been performed by the author with feedback from Luis and Mark. It is based on the *case studies* and *evaluation* sections of [BFHH11] and [HT13]. The extension with caching mechanisms, which is presented in Section 4.7.4, has been

developed, implemented and evaluated by the author. Sections 4.7.1 to 4.7.4 of this thesis are also the basis of a recent journal paper [HT14b] which received additional feedback from Mark.

In Section 4.8, we review approaches to the problem of performing SMC for general MDP. Section 4.8.1 summarises a first attempt by Henriques et al. [HMZ+12] to use reinforcement learning, and then gives an overview of the more general learning-based framework recently presented by Brázdil et al. [BCC+14]. Section 4.8.2 summarises an alternative strand of work done by Legay and Sedwards [LST14] and details why the error bounds they appear to present cannot be correct without further qualification. Finally, in Section 4.8.3, we very briefly outline another idea that was based on discussions with colleagues of the author. We give an example that shows that it does not work.

4.1 Definition

A Markov decision process incorporates both nondeterministic and probabilistic choices on a discrete state space. It can be seen as the orthogonal combination of LTS and DTMC: As in the former, transitions are labelled, and several transitions can be enabled in one state. As in the latter, the target of a transition is a probability distribution over states. This means that moving from one state to the next involves the resolution of a nondeterministic choice followed by a probabilistic experiment. Formally:

Definition 36 (MDP)**.** A *Markov decision process* (MDP) is a 6-tuple

$$\langle S, A, T, s_{init}, AP, L \rangle$$

where
– S is a countable set of states,
– $A \supseteq \{\tau\}$ is the process' alphabet, a countable set of transition labels (or *actions*) that includes the *silent action* τ,
– $T \in S \to \mathscr{P}(A \times \mathrm{Dist}(S))$ is the transition function, with the restriction that $T(s)$ is countable for all $s \in S$,
– $s_{init} \in S$ is the initial state,
– AP is a set of atomic propositions, and
– $L \in S \to \mathscr{P}(AP)$ is the state labelling function.

Observe that $\mathrm{Dist}(S)$ is always an uncountable set, which is why we need the extra restriction on the transition function in order to be able to enumerate transitions. Remember that we anyway restrict to finitely branching models in this thesis, unless we say otherwise. We say that an MDP M is *finite* if it is finitely

branching and its set of states is finite. We call the pairs $\langle a, \mu \rangle \in T(s)$ the *transitions of s* and the triples $\langle s, a, \mu \rangle$ such that $\langle a, \mu \rangle \in T(s)$ the *transitions of M*. We thus overload the notation for transitions by also writing $\langle s, a, \mu \rangle \in T(s)$ for $\langle a, \mu \rangle \in T(s)$. Just like for LTS and DTMC, we also write $s \xrightarrow{a} \mu$ for the transition $\langle s, a, \mu \rangle$ and $s \xrightarrow{a} s'$ for $\exists \mu : \langle a, \mu \rangle \in T(s) \wedge \mu(s') > 0$ in the remainder of this thesis. In the same way, the notions of deadlock state, loop and cycle carry over. A nonprobabilistic (or *Dirac*) loop or cycle is one on which all probability distributions are the Dirac distribution, i.e. one target state has probability 1. However, since deadlock states would needlessly complicate almost all of the following definitions and algorithms, we assume from now on that **all MDP are deadlock-free**. Any MDP can be transformed into one that is equivalent for the purposes of this thesis and deadlock-free by simply adding a τ-labelled Dirac loop to every deadlock state.

In the traditional definition of MDP, the nondeterminism is a choice between actions and for every $a \in A$, a state therefore has at most one outgoing transition labelled a. PA allow more than one outgoing transition with the same label. Since we use transition labels only for parallel composition and disregard them during verification, it is the slightly more general definition from PA that we use here.

We also define the notion of end components, which are subsets of the states and transitions of an MDP that are strongly connected with transition probabilities greater than 0:

Definition 37 (End component)**.** In an MDP $\langle S, A, T, s_{init}, AP, L \rangle$, a n *end component* is a pair $\langle S_e, T_e \rangle$ where $S_e \subseteq S$ and $T_e \in S_e \to \mathscr{P}(A \times \mathrm{Dist}(S))$ with $T_e(s) \subseteq T(s)$ for all $s \in S_e$ such that for all $s \in S_e$ and transitions $\langle a, \mu \rangle \in T_e(s)$ we have $\mu(s') > 0 \Rightarrow s' \in S_e$ and the underlying directed graph of $\langle S_e, T_e \rangle$ is strongly connected.

Finally, we need to precisely say when a transition is visible to an outside observer of an MDP. In the case of an open system (cf. Section 3.3), this could be another MDP that runs in parallel, so in this context transition labels would be considered visible. In any case, the labelling of the current state needs to be considered visible as this is what a verification procedure works on.

Definition 38 (Visible transitions)**.** A transition $\langle s, a, \mu \rangle$ of an MDP as above is *visible* if there exists a state $s' \in \mathrm{support}(\mu)$ such that $L(s) \neq L(s')$. Additionally, if we consider the MDP as an open system, it is also visible if $a \neq \tau$.

Note that by default we consider closed systems, so unless we explicitly state otherwise, the visibility of a transition depends only on the changes of atomic proposition labels.

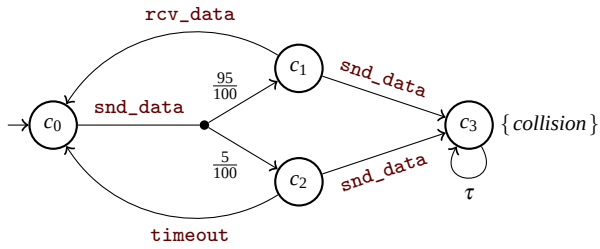


Figure 4.1: MDP for a lossy communication channel with collision detection

Example 14. To illustrate what can be expressed in MDP models, let us revisit the communication channel for data originally introduced in Example 5 for our running example of the communication protocol. The decision between losing and successfully transmitting a message was a nondeterministic one in the LTS model shown in Figure 3.2. Our new MDP model for the channel, shown in Figure 4.1, uses a probabilistic decision instead. We fix the probability of losing a message to be 0.05 in this case.

Graphically, we represent transitions in MDP as lines with an action label that lead to an intermediate node from which the branches of the probabilistic choice lead to the respective successor states. We omit the intermediate node and probability 1 for transitions that lead to a Dirac distribution.

In addition to the now probabilistically quantified message loss, we include *collision detection* in our new channel model: Whenever another attempt is made to send a message (via action `snd_data`) while the previous one is still "in transit", we move to an error state. This results in a nondeterministic choice between two outgoing transitions in states c_1 and c_2. Since we do not allow deadlocks, the error state c_3 has a (nonprobabilistic) τ-loop. There are exactly two end components: The first consists of state c_3 with its τ-loop, while the second is $\{c_0, c_1, c_2\}$ with the transitions labelled `rcv_data` and `timeout` as well as the transition labelled `snd_data` out of c_0.

Paths

The semantics of an MDP is captured by the notion of paths and traces. A path through an MDP represents a concrete resolution of both nondeterministic and probabilistic choices:

Definition 39 (Paths in MDP)**.** Given an MDP $M = \langle S, A, T, s_{init}, AP, L \rangle$, a (finite) *path in M from s_0 to s_n* of length $n \in \mathbb{N}$ is a finite sequence

$$s_0 \langle a_0, \mu_0 \rangle s_1 \langle a_1, \mu_1 \rangle s_2 \ldots \langle a_{n-1}, \mu_{n-1} \rangle s_n$$

where $s_i \in S$ for all $i \in \{0,\ldots,n\}$ and $\langle a_i,\mu_i\rangle \in T(s_i) \wedge \mu_i(s_{i+1}) > 0$ for all $i \in \{0,\ldots,n-1\}$. An (infinite) *path in M starting from* s_0 is an infinite sequence

$$s_0 \langle a_0,\mu_0\rangle s_1 \langle a_1,\mu_1\rangle s_2 \ldots$$

where for all $i \in \mathbb{N}$, we have that $s_i \in S$, $\langle a_i,\mu_i\rangle \in T(s_i)$ and $\mu_i(s_{i+1}) > 0$.

Where convenient, we identify the infinite path $s_0 \langle a_0,\mu_0\rangle s_1 \langle a_1,\mu_1\rangle \ldots$ with the sequence of transitions $\langle s_0,a_0,\mu_0\rangle \langle s_1,a_1,\mu_1\rangle \ldots$ that it corresponds to. All the notation derived from the definition of finite and infinite paths (such as the set $\mathrm{Reach}(M)$ of reachable states) carries over from LTS unchanged. As we only consider MDP without deadlock states, all maximal paths are infinite. The trace of a path can be defined for MDP in the same way as for LTS.

Schedulers and Reduction Functions

A path embodies a resolution of the nondeterministic choices for the states it contains. A state may be visited several times, and different outgoing transitions may be chosen in different visits. The resolution of nondeterminism along a path can thus be *history-dependent*, i.e. it can depend on the current state *and* on the current prefix of the path up to that state. Lifting the concept of "resolution of nondeterminism" from a single path to an entire MDP, we obtain the notion of a scheduler (based on [BK08]):

Definition 40 (Scheduler for MDP)**.** A *history-dependent scheduler* for a given MDP as usual is a function $\mathfrak{S}^* \in \mathrm{Paths}_{\mathrm{fin}}(M) \to \mathrm{Dist}(A \times \mathrm{Dist}(S))$ such that $\mathrm{support}(\mathfrak{S}^*(s_0 \ldots s)) \subseteq T(s)$ for all $s \in S$. A (memoryless) *scheduler* is a function $\mathfrak{S} \in S \to A \times \mathrm{Dist}(S)$ such that $\mathfrak{S}(s) \in T(s)$ for all $s \in S$.

Schedulers are also called *adversaries*, *policies* or *strategies*. Our definition of history-dependent schedulers is very general by allowing the nondeterminism to be resolved in a probabilistic way and taking the entire path history into account. However, we will see that memoryless schedulers are sufficient for all properties and verification procedures on MDP that we consider in the remainder of this thesis. We thus exclusively use memoryless schedulers from now on. We note that an important class of properties for which memoryless schedulers do not suffice are *step-bounded* ones, i.e. properties in which one refers to the number of transitions taken on paths up to some point. However, we do not consider such properties in this thesis.

Closely related to schedulers is the notion of reduction functions: They select a *subset* of the outgoing transitions of every state.

Definition 41 (Reduction function [HT13])**.** A *reduction function* for a given MDP $M = \langle S,A,T,s_{init},AP,L\rangle$ is a function $f \in S \to \mathscr{P}(A \times \mathrm{Dist}(S))$ such that

$f(s) \subseteq T(s)$ and $|f(s)| > 0$ for all $s \in S$. If $|f(s)| = 1$ for all states, we say that f is a *deterministic* reduction function. Given a reduction function f, the *reduced MDP for M with respect to* f is

$$\mathrm{red}(M, f) \stackrel{\text{def}}{=} \langle S_f, A, f|_{S_f}, s_{init}, AP, L|_{S_f} \rangle$$

where S_f is the smallest set such that

$$s_{init} \in S_f \wedge s' \in S_f \Rightarrow S_f \supseteq \cup_{\langle a,\mu\rangle \in f(s')} \mathrm{support}(\mu).$$

We say that $s \in S_f$ is a *reduced state* if $f(s) \neq T(s)$. All outgoing transitions of a reduced state are called *nontrivial transitions*. We say that a reduction function is *acyclic* if there are no cyclic paths in M_f starting in any state when only nontrivial transitions are considered.

A scheduler corresponds to a deterministic reduction function. We can establish the following relationship between schedulers and general reduction functions:

Definition 42 (Valid scheduler for a reduction function)**.** For a given MDP with set of states S, a scheduler $\mathfrak{S}$ is *valid* for a reduction function f if $\forall s \in S$: $\mathfrak{S}(s) \in f(s)$.

The result of removing all the transitions not selected by a scheduler from an MDP is a deterministic MDP. We will later in this section show that a deterministic MDP is indeed a DTMC when transition labels are dropped. For now, we define:

Definition 43 (Induced DTMC for a scheduler)**.** The DTMC induced by a scheduler $\mathfrak{S}$ for an MDP $M = \langle S, A, T, s_{init}, AP, L \rangle$ is

$$\mathrm{ind}(M, \mathfrak{S}) = \mathrm{red}(M, \{ s \mapsto \{ \mathfrak{S}(s) \} \}).$$

If the choices represented by a path are the same choices that a scheduler would make, then we say that the path is valid for that scheduler:

Definition 44 (Valid paths for a scheduler)**.** A given (finite or infinite) path $s_0 \langle a_0, \mu_0 \rangle s_1 \ldots$ in an MDP $\langle S, A, T, s_{init}, AP, L \rangle$ is *valid* for a scheduler $\mathfrak{S}$ if, for all i, we have $\mathfrak{S}(s_i) = \langle a_i, \mu_i \rangle$. We denote by $\mathrm{Paths}(M, \mathfrak{S}) \subseteq \mathrm{Paths}(M)$ (or $\mathrm{Paths}_{\mathrm{fin}}(M, \mathfrak{S}) \subseteq \mathrm{Paths}_{\mathrm{fin}}(M)$) the set of all maximal (or finite) paths valid for $\mathfrak{S}$.

It is easy to see that all paths in an induced DTMC correspond to valid paths for the scheduler in the MDP and vice-versa:

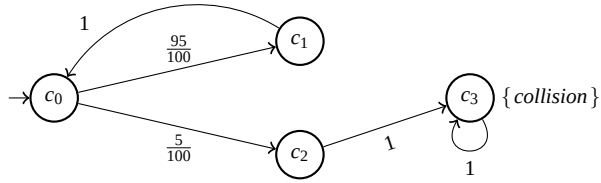


Figure 4.2: DTMC induced by a scheduler for the comm. channel MDP

Proposition 1. Given an MDP $M = \langle S, A, T, s_{init}, AP, L \rangle$ and a scheduler $\mathfrak{S}$ for M, we have that

$$\begin{aligned} &\{ s_0 \mu_0 s_1 \ldots \mid s_0 \langle a_0, \mu_0 \rangle s_1 \ldots \in \mathrm{Paths}(M, \mathfrak{S}) \} \\ = \; &\{ s_0 T(s_0) s_1 \ldots \mid s_0 s_1 \ldots \in \mathrm{Paths}(\mathrm{ind}(M, \mathfrak{S})) \} \end{aligned}$$

where T denotes the the probabilistic transition function of the respective induced Markov chains. The same holds analogously for finite paths.

Note in particular that the cylinder sets and probability distributions are preserved. We use this fact in our definition of the semantics of reachability properties for MDP in Section 4.4.

Example 15. Schedulers for the MDP model of the channel with collision detection from Example 14 can only make a nontrivial choice for states c_1 and c_2, namely between finishing the transmission and detecting a timeout. A history-dependent scheduler could, for example, choose to finish the transmission twice before selecting the collision afterwards. The memoryless schedulers we consider do not have this option. An example of a memoryless scheduler is

$$\begin{aligned} \mathfrak{S} = \{ \; & c_0 \mapsto \langle \texttt{snd\_data}, \{c_1 \mapsto 0.99, c_2 \mapsto 0.01\} \rangle, \\ & c_1 \mapsto \langle \texttt{rcv\_data}, \mathscr{D}(c_0) \rangle, c_2 \mapsto \langle \texttt{snd\_data}, \mathscr{D}(c_3) \rangle, c_3 \mapsto \langle \tau, \mathscr{D}(c_3) \rangle \; \} \end{aligned}$$

The DTMC induced by this scheduler is shown in Figure 4.2. A valid path for $\mathfrak{S}$ is

$$c_0 \langle \texttt{snd\_data}, \{c_1 \mapsto 0.99, c_2 \mapsto 0.01\} \rangle c_2 \langle \texttt{snd\_data}, \mathscr{D}(c_3) \rangle (c_3 \langle \tau, \mathscr{D}(c_3) \rangle)^\omega$$

while the path

$$(\, c_0 \langle \texttt{snd\_data}, \{c_1 \mapsto 0.99, c_2 \mapsto 0.01\} \rangle c_2 \langle \texttt{timeout}, \mathscr{D}(c_0) \rangle \,)^\omega$$

is not valid for $\mathfrak{S}$.

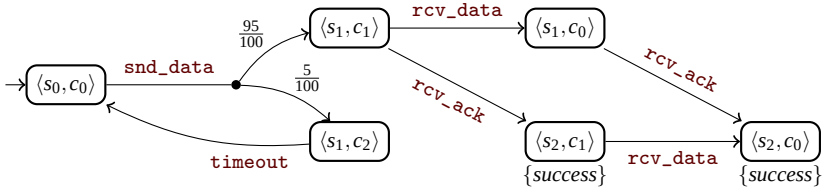


Figure 4.3: The parallel composition of sender and probabilistic channel

Parallel Composition

As MDP allow nondeterministic choices, we can define the parallel composition of two MDP using an interleaving semantics as previously done for LTS in Section 3.1.1:

Definition 45. The *parallel composition of two MDP*

$$M_i = \langle S_i, A_i, T_i, s_{init_i}, AP_i, L_i \rangle,$$

$i \in \{1,2\}$, is the MDP

$$M_1 \parallel M_2 = \langle S_1 \times S_2, A_1 \cup A_2, T, \langle s_{init_1}, s_{init_2} \rangle, AP_1 \cup AP_2, L \rangle$$

where

– $T \in (S_1 \times S_2) \to \mathscr{P}((A_1 \cup A_2) \times \mathrm{Dist}(S_1 \times S_2))$
  s.t. $\langle a, \mu \rangle \in T(\langle s_1, s_2 \rangle) \Leftrightarrow a \notin B \wedge \exists \mu_1 \colon \langle a, \mu_1 \rangle \in T_1(s_1) \wedge \mu = \mu_1 \otimes \mathscr{D}(s_2)$
  $\vee\ a \notin B \wedge \exists \mu_2 \colon \langle a, \mu_2 \rangle \in T_2(s_2) \wedge \mu = \mathscr{D}(s_1) \otimes \mu_2$
  $\vee\ a \in B \wedge \exists \mu_1, \mu_2 \colon \langle a, \mu_1 \rangle \in T_1(s_1)$
  $\wedge\ \langle a, \mu_2 \rangle \in T_2(s_2) \wedge \mu = \mu_1 \otimes \mu_2$
  with $B = (A_1 \cap A_2) \setminus \{\tau\}$, and

– $L \in (S_1 \times S_2) \to \mathscr{P}(AP_1 \cup AP_2)$ s.t. $L(\langle s_1, s_2 \rangle) = L_1(s_1) \cup L_2(s_2)$.

The only difference to the parallel composition operator for LTS is that the target of a synchronising transition is the product of the probability distributions of the individual component transitions, i.e. probabilities are multiplied. This is like in the parallel composition of DTMC, which however had to use a lock-step semantics in absence of nondeterminism. As for LTS, parallel composition can both introduce and remove nondeterministic choices to a model:

Example 16. We take the communication protocol components introduced as LTS in Example 5 and interpret them as MDP. Instead of the LTS data channel, however, we use the lossy channel with collision detection modelled as an MDP in Example 14. The parallel composition of sender and channel is

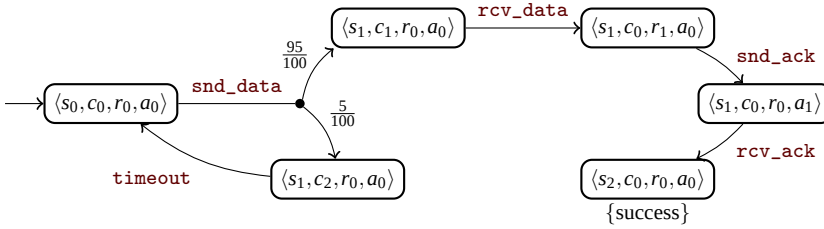


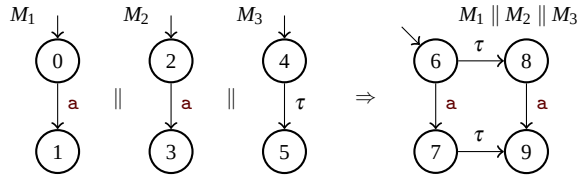
Figure 4.4: MDP for the comm. protocol with one probabilistic channel

shown in Figure 4.3. The product MDP contains nondeterminism that was not present in the components due to the interleaving of `rcv_data` and `rcv_ack`. However, the nondeterministic choice between finishing the transmission and detecting a collision in the channel has been removed because the sender never allows a collision to occur. Figure 4.4 shows the composition of sender, probabilistic channel, receiver and (nonprobabilistic) acknowledgment channel. We interpret this as a network of MDP and thus flatten the composition structure; this is why the states are simple four-tuples. Note that, as in the LTS case in Figure 3.4, the overall composition contains no nondeterminism any more.

In Section 4.7.1, we need to identify transitions that appear, in some way, to be the same. For this purpose, we shall use equivalence relations $\equiv$ on transitions and denote the equivalence class of $tr = \langle s, a, \mu \rangle$ under $\equiv$ by $[tr]_{\equiv}$. This notation can naturally be lifted to sets of transitions: The equivalence class $[Tr]$ of a set Tr of (not necessarily equivalent) transitions is the union of the equivalence classes of the individual transitions in Tr. When working with a network of MDP, one useful equivalence relation has $tr \equiv tr'$ iff transitions tr and tr' in the product MDP result from the same set of transitions $\{ tr_1, \ldots, tr_m \}$ in the component automata according to Definition 45. We denote this particular relation by $\equiv_T$. For illustration, consider the network of MDP $\{ M_1, M_2, M_3 \}$ shown in Figure 4.5. The product MDP on the right has two equivalence classes of transitions, as shown below the MDP in the figure: The transitions labelled `a` are in the same class because they both result from the synchronisation of $\langle 0, \mathtt{a}, \mathscr{D}(1) \rangle$ from M_1 and $\langle 2, \mathtt{a}, \mathscr{D}(3) \rangle$ from M_2. The two transitions labelled τ belong to the same class because they both result from transition $\langle 4, \mathtt{a}, \mathscr{D}(5) \rangle$ of M_3.

Submodels

As mentioned, MDP can be seen as the orthogonal combination of the models of LTS and DTMC. We now formally state what this means, i.e. when an



$$[\langle 6, \mathtt{a}, \mathscr{D}(7)\rangle]_{\equiv_T} = \{\, \langle 6, \mathtt{a}, \mathscr{D}(7)\rangle, \langle 8, \mathtt{a}, \mathscr{D}(9)\rangle \,\}$$
$$[\langle 7, \tau, \mathscr{D}(9)\rangle]_{\equiv_T} = \{\, \langle 6, \tau, \mathscr{D}(8)\rangle, \langle 7, \tau, \mathscr{D}(9)\rangle \,\}$$

Figure 4.5: Transition equivalence $\equiv_T$ for networks of MDP

MDP is an LTS or a DTMC, and how LTS and DTMC can be represented as equivalent MDP.

Labelled transition systems The common feature of LTS and MDP are the nondeterministic choices between the labelled transitions leading out of a state; the feature not supported by LTS are probabilistic decisions. The only MDP that can be equivalent to LTS are thus those that do not contain such decisions:

Definition 46 (Nonprobabilistic MDP)**.** In an MDP $M = \langle S, A, T, s_{init}, AP, L\rangle$, a transition $\langle a, \mu\rangle \in T(s)$ for $s \in S$ is *nonprobabilistic* if $\exists s' \colon \mu = \mathscr{D}(s')$, and probabilistic otherwise. The entire MDP M is *nonprobabilistic* if all its transitions are nonprobabilistic. Otherwise, it is probabilistic.

It is then easy to see that a nonprobabilistic MDP turns into an LTS if the Dirac distributions in the transitions are replaced by the respective states, and that an LTS is an MDP where the new transitions lead to the Dirac distributions for their original target states. Formally:

Proposition 2. A nonprobabilistic MDP $\langle S, A, T, s_{init}, AP, L\rangle$ is isomorphic to the LTS $\langle S, A, T', s_{init}, AP, L\rangle$ where $\langle a, s'\rangle \in T'(s) \Leftrightarrow \langle a, \mathscr{D}(s')\rangle \in T(s)$ for all $s \in S$. We say that a nonprobabilistic MDP *is* an LTS, and an LTS *is* a nonprobabilistic MDP.

Discrete-time Markov chains MDP inherit their probabilistic choices from DTMC. As nondeterminism is not supported by DTMC, we first state clearly what it means for an MDP to be "deterministic":

Definition 47 (Deterministic MDP)**.** Given an MDP $M = \langle S, A, T, s_{init}, AP, L\rangle$, a state $s \in S$ is *deterministic* if $|T(s)| = 1$, and nondeterministic otherwise. The

entire MDP M is *deterministic* if all its states $s \in S$ are deterministic. Otherwise, it is nondeterministic.

Note that a deterministic MDP may well be probabilistic. In any case, as we already hinted at in Definition 43, a deterministic MDP directly corresponds to a DTMC when we drop all transition labels:

Proposition 3. A deterministic MDP $\langle S, A, T, s_{init}, AP, L\rangle$ is isomorphic modulo transition labels to the DTMC $\langle S, T', s_{init}, AP, L\rangle$ where $T'(s) = \mu \Leftrightarrow \exists a \in A\colon T(s) = \{\langle a, \mu\rangle\}$ for all $s \in S$. We say that a deterministic MDP *is* a DTMC, and a DTMC *is* a deterministic MDP (for an arbitrarily chosen alphabet).

The correspondence between deterministic MDP and DTMC is not as direct as that for LTS since we need to drop transition labels when going from MDP to DTMC, and we can introduce arbitrary labels in the other direction. This in particular means that there are infinitely many MDP that are isomorphic to a single DTMC modulo transition labels. Another effect is that this correspondence is not *compositional* when it comes to the parallel composition operators: First of all, the parallel composition of two deterministic MDP is not necessarily deterministic itself. But even if it is, its corresponding DTMC in general need not be the same as the parallel composition of the two DTMC corresponding to the individual MDP. This is, of course, due to the use of a lock-step semantics in DTMC parallel composition versus an interleaving semantics for MDP. Compositionality in the above sense can be achieved in special cases, though: If all transitions in the MDP are labelled with the same action $sync \neq \tau$, then the two notions of parallel composition for MDP and DTMC coincide since all transitions in the MDP product synchronise.

4.2 Variables

Like for LTS, building MDP models becomes easier when we allow the inclusion of variables. Although variable-decorated MDP, VMDP for short, again merely combine the features of VLTS and VDTMC, we give their full definition in this section because VMDP are the basic model of the partial order reduction-based approach to SMC that we present in Section 4.7.1.

Definition

As in VLTS, the variables of a VMDP can be referred to in guards and changed in assignments. As in VDTMC, the target of an edge is a symbolic probability distribution that assigns weights to pairs of updates and target locations.

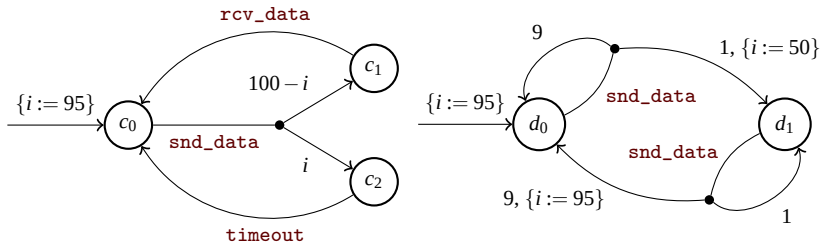


Figure 4.6: VMDP for the communication channel and an environment

Definition 48 (VMDP)**.** A *Markov decision process with variables* (VMDP) is a 7-tuple $\langle Loc, Var, A, E, l_{init}, V_{init}, VExp \rangle$ where
– Loc is a countable set of locations,
– Var is a finite set of variables with countable domains,
– $A \supseteq \{\tau\}$ is the process' alphabet,
– $E \in Loc \to \mathscr{P}(Bxp \times A \times (Upd \times Loc \to Axp))$ is the edge function, which maps each location to a set of edges, which in turn consist of a guard, a label and a symbolic probability distribution over updates and target locations,
– $l_{init} \in Loc$ is the initial location,
– $V_{init} \in \mathrm{Val}(Var)$ is the initial valuation of the variables, and
– $VExp \subseteq Bxp$ is the set of visible expressions.

For a given VMDP M with edge function E, we also write $l \xrightarrow{g,a} m$ instead of $\langle g, a, m \rangle \in E(l)$ in the remainder of this thesis. As for MDP, we may refer to an edge as $\langle l, g, a, m \rangle$ if $\langle g, a, m \rangle \in E(l)$.

Example 17. Figure 4.6 on the left shows a VMDP model for the probabilistic communication channel introduced as an MDP in Example 14. It omits the collision detection mechanism as we found out in Example 16 that the sender anyway does not cause collisions on this channel. In contrast to the MDP model, though, the message loss probability is now determined by an integer variable i instead of being fixed to 0.95. On the right of Figure 4.6 is a VMDP that models a possible environment for wireless communication: From time to time (here: whenever a message is sent, to make the next example more interesting), a disturbance may appear—such as a microwave oven being switched on—that significantly increases message losses (by modifying the value of i, which determines the channel's message loss probability), only to later disappear again. The appearance of the disturbance is a probabilistic decision, just like its later disappearance.

Semantics

The semantics of a VMDP is an MDP whose states keep track of the current location and the current values of all variables. Based on these current values, the symbolic probability distributions can be turned into concrete ones.

Definition 49 (Semantics of VMDP)**.** The *semantics of a VMDP*

$$M = \langle Loc, Var, A, E, l_{init}, V_{init}, VExp \rangle$$

is the MDP

$$[\![M]\!] = \langle Loc \times Val, A, T, \langle l_{init}, V_{init} \rangle, VExp, L \rangle$$

where

– $T \in Loc \times Val \to \mathscr{P}(A \times \mathrm{Dist}(Loc \times Val))$ s.t.

$$\begin{aligned} \langle a, \mu \rangle \in T(\langle l, v \rangle) \Leftrightarrow \exists \langle g, a, m \rangle \in E(l) \colon [\![g]\!](v) \wedge \mu = m^v_{conc} \\ \vee\, a = \tau \wedge \mu = \mathscr{D}(l) \wedge \nexists \langle g, a, m \rangle \in E(l) \colon [\![g]\!](v) \end{aligned} \tag{4.1}$$

where m^v_{conc} is the concrete probability distribution corresponding to m for valuation v as defined in Section 3.2, and

– $L \in Loc \times Val \to \mathscr{P}(VExp)$ s.t.

$$\forall \langle l, v \rangle \in Loc \times Val \colon L(\langle l, v \rangle) = \{\, e \in VExp \mid [\![e]\!](v) \,\}.$$

Observe that the second line of Equation 4.1 adds a loop to a state in the MDP in case no edge is enabled. This explicitly ensures that the result is deadlock-free. The same issues regarding the conversion from symbolic to concrete probability distributions arise as for VDTMC: VMDP models are considered invalid, i.e. a modelling error, if for any reachable valuation in the semantics, a single weight would evaluate to a negative number or the sum of all weights of a distribution would not converge or would evaluate to zero.

Parallel Composition

The definition of a parallel composition operator for VMDP is straightforward; the symbolic probability distributions are combined by simply creating multiplication expressions:

Definition 50 (Parallel composition of VMDP)**.** The *parallel composition of two consistent VMDP* $M_i = \langle Loc_i, Var_i, A_i, E_i, l_{init_i}, V_{init_i}, VExp_i \rangle$, $i \in \{1,2\}$, is the VMDP

$$\begin{aligned} M_1 \parallel M_2 = \langle Loc_1 \times Loc_2, Var_1 \cup Var_2, A_1 \cup A_2, E, \\ \langle l_{init_1}, l_{init_2} \rangle, V_{init_1} \cup V_{init_2}, VExp_1 \cup VExp_2 \rangle \end{aligned}$$

with $E \in (Loc_1 \times Loc_2) \to \mathscr{P}(Bxp \times A_1 \cup A_2 \times (Upd \times (Loc_1 \times Loc_2) \to Axp))$
s.t. $\langle g, a, m\rangle \in E(\langle l_1, l_2\rangle) \Leftrightarrow$
$$\begin{aligned} & a \notin B \wedge \exists m_1 : \\ & \quad \langle g, a, m_1\rangle \in E_1(l_1) \wedge m = m_1 \times \{\langle l_2, \varnothing\rangle \mapsto 1\} \\ \vee\ & a \notin B \wedge \exists m_2 : \\ & \quad \langle g, a, m_2\rangle \in E_2(l_2) \wedge m = \{\langle l_1, \varnothing\rangle \mapsto 1\} \times m_2 \\ \vee\ & a \in B \wedge \exists g_1, g_2, m_1, m_2 : \\ & \quad \langle g_1, a, m_1\rangle \in E_1(l_1) \wedge \langle g_2, a, m_2\rangle \in E_2(l_2) \\ & \quad \wedge (g = g_1 \wedge g_2) \wedge (m = m_1 \times m_2) \end{aligned}$$

with $B = (A_1 \cap A_2) \setminus \{\tau\}$ and the product of two consistent symbolic probability distributions $m_i \in Upd \times Loc_i \to Axp$, $i \in \{1,2\}$, defined as

$$\begin{aligned} & m_1 \times m_2 \in Upd \times (Loc_1 \times Loc_2) \to Axp \text{ s.t.} \\ & (m_1 \times m_2)(\langle U_1 \cup U_2, \langle l_1, l_2\rangle\rangle) = m_1(U_1, l_1) \cdot m_2(U_2, l_2). \end{aligned}$$

Again, just as for VLTS, the two components need to be consistent in order to avoid conflicting assignments. The definition of consistency that we use is similar to the one we gave for LTS in Definition 24, and is also a sufficient condition only:

Definition 51 (Consistent VMDP)**.** Two VMDP M_i as in the previous definition are consistent if their initial valuations are consistent and, for all $l_1 \in Loc_1$, $l_2 \in Loc_2$ and $a \in A_1 \cap A_2 \setminus \{\tau\}$, we have that

$$U_1 \in f_1(l_1, a) \wedge U_2 \in f_2(l_2, a) \Rightarrow U_1 \text{ and } U_2 \text{ are consistent}$$

where

$$f_i(l, a) \stackrel{\text{def}}{=} \{U \mid \langle g, a, m\rangle \in E_i(l) \wedge \exists l' \in Loc_i : m(U, l') \neq 0\}$$

is the set of all assignments that have non-constant-zero weight on the edges from l labelled with a.

Also just like for LTS, we allow shared variables, which leads to the recurrence of the effect that the application of parallel composition and MDP semantics is not commutative in general (cf. Theorem 1).

Example 18. The parallel composition of the communication channel and environment VMDP shown in the previous example can be seen in Figure 4.7. It would have been more realistic to have the edges of the environment model labelled τ instead of `snd_data` since, for example, a microwave usually does not switch on or off in reaction to some independent wireless communication. The resulting parallel composition, however, would not have showed the way the product of symbolic probability distributions is built. Drawing that more realistic composition is instead left as an exercise to the reader.

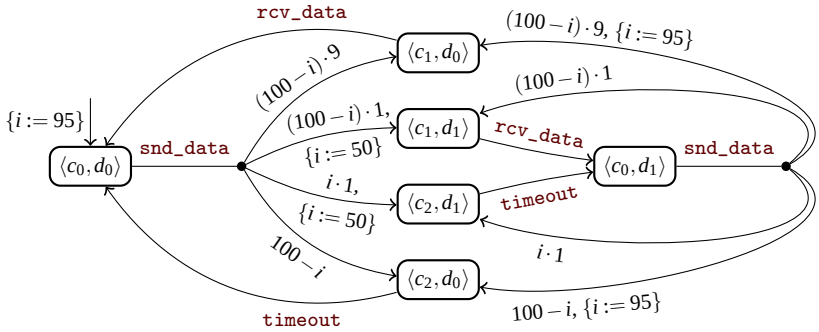


Figure 4.7: The VMDP parallel composition of the channel and its environment

As an analogue to $\equiv_T$ for MDP parallel composition, a useful equivalence relation $\equiv$ over the transitions of the MDP semantics of a network of VMDP is the one that identifies those transitions that result from the same (set of) *edges* in the component VMDP. We denote this relation by $\equiv_E$.

4.3 Modelling

To model VMDP in MODEST, all we need in addition to the facilities for modelling VLTS is a way to specify a probabilistic choice. For this purpose, we introduced the `palt` keyword in Section 3.2.1. Although we used it to model VDTMC, the (*palt*) inference rule at that point was already given in a form that is suitable for modelling VMDP. In fact, we can now freely use any actions we declare, mix in nonprobabilistic choices via `alt` and `do`, or in short: use all MODEST keywords for VLTS without restrictions. The VMDP feature of nondeterministic choices followed by probabilistic ones can be used by nesting `palt`s inside an `alt` or `do`. MODEST's parallel composition via `par` also turns out to precisely match the VMDP parallel composition we just introduced in Definition 50.

Example 19. Figure 4.8 contains MODEST code whose semantics[1] is the MDP for the communication channel with probabilistic message loss introduced in Example 14. If we replace the `Channel` process of the simple BRP model of Example 8 by this new probabilistic channel, we obtain a model that we will refer to as the *simple probabilistic BRP*.

[1] To be precise: *"the semantics of whose semantics"*, since the semantics of MODEST is now a VMDP.

```
process Channel()
{
   snd palt {
   :95: alt { :: rcv     :: snd {= collision = true =}; stop }
   : 5: alt { :: timeout :: snd {= collision = true =}; stop }
   };
   Channel()
}

Channel()
```

Figure 4.8: The lossy comm. channel with collision detection in MODEST

4.4 Properties

Reachability properties for MDP are probabilistic and thus come in a quantitative and a qualitative form as for DTMC. However, the actual probability of reaching a certain set of states depends on how the nondeterministic choices are resolved. When ranging over all possible resolutions, we obtain an interval $\subseteq [0,1]$ of possible probabilities. For verification, it is the interval's extremal values that we are interested in. In quantitative form, we can thus ask for the maximum or minimum probability of reaching a set of target states (resp. for the infimum and supremum if these values do not exist in general). In qualitative form, we perform a *worst-case* analysis: upper bounds are compared to maximum, lower bounds to minimum probabilities. This ensures that the bound is satisfied no matter how the nondeterminism is resolved. Syntactically, we write probabilistic reachability properties for MDP as follows:

$\mathrm{P}_{\max}(\diamond \phi)$ –and– $\mathrm{P}_{\min}(\diamond \phi)$ (quantitative form) $\quad \mathrm{P}(\diamond \phi) \sim x$ (qualitative form)

where $\sim \in \{<, \leq, >, \geq\}$ and $x \in [0,1]$.

Example 20. Interesting properties for the simple probabilistic BRP model from the previous example are
– $\mathrm{P}(\diamond \mathit{collision}) \leq 0$ "the probability of a collision on the data channel is zero",
– $\mathrm{P}_{\min}(\diamond \mathit{success})$ "what is the worst-case probability of success?", and
– $\mathrm{P}_{\max}(\diamond \mathit{failure})$ "what is the worst-case probability that the sender gives up?"
Observe how, for the properties in quantitative form, the meaning of "worst-case" w.r.t. maximum and minimum probabilities depends on whether we observe a "bad" or a "good" event. In MODEST, we can express these three properties as

```
– property P_NoCollision = P(<> collision) <= 0;
– property P_Success = Pmin(<> success);
– property P_Failure = Pmax(<> failure);
```

To define the semantics of probabilistic reachability properties for MDP, we simply formalise the meaning of "ranging over all possible resolutions of non-determinism" via schedulers:

Definition 52 (Semantics of MDP probabilistic reachability properties)**.** For an MDP $\langle S, A, T, s_{init}, AP, L\rangle$, the semantics of a probabilistic reachability query $\mathrm{P}_{\max}(\diamond\phi)$ or $\mathrm{P}_{\min}(\diamond\phi)$ is defined as

$$\begin{aligned}[\![\mathrm{P}_{\max}(\diamond\phi)]\!]_M &\stackrel{\text{def}}{=} \sup_{\mathfrak{S}^*}[\![\mathrm{P}(\diamond\phi)]\!]_{\mathrm{ind}(M,\mathfrak{S}^*)} \\ &= \max_{\mathfrak{S}}[\![\mathrm{P}(\diamond\phi)]\!]_{\mathrm{ind}(M,\mathfrak{S})}\end{aligned} \qquad \begin{aligned}[\![\mathrm{P}_{\min}(\diamond\phi)]\!]_M &\stackrel{\text{def}}{=} \inf_{\mathfrak{S}^*}[\![\mathrm{P}(\diamond\phi)]\!]_{\mathrm{ind}(M,\mathfrak{S}^*)} \\ &= \min_{\mathfrak{S}}[\![\mathrm{P}(\diamond\phi)]\!]_{\mathrm{ind}(M,\mathfrak{S})}\end{aligned}$$

The semantics of a probabilistic reachability property in qualitative form is defined as follows:

$$[\![\mathrm{P}(\diamond\phi) \sim x]\!]_M \stackrel{\text{def}}{=} \begin{cases} [\![\mathrm{P}_{\max}(\diamond\phi)]\!]_M \sim x & \text{if } \sim \in \{<, \leq\} \\ [\![\mathrm{P}_{\min}(\diamond\phi)]\!]_M \sim x & \text{if } \sim \in \{>, \geq\} \end{cases}$$

As usual, we omit the subscript M when it is clear from the context. A proof of the facts that 1) a scheduler exists that maximises/minimises the reachability probability and that 2) this scheduler is memoryless can be found in, for example, [BK08] as the proof of lemmas 10.102 and 10.113.

Temporal Logics for MDP

The temporal logic PCTL* that we mentioned in Section 3.2.2 for DTMC is equally applicable to MDP. The interpretation of its $\mathrm{P}_{\sim x}$ operator is a comparison with maximum or minimum probability similar to how we defined the semantics of the qualitative form of probabilistic reachability above. LTL (see Section 3.1.4) formulas can also be interpreted probabilistically for MDP: An LTL formula f represents a set of traces. The probability of f can then, intuitively speaking, be defined via the probabilities of the paths that correspond to the traces in a given MDP. What is important for this thesis, in particular for the correctness of the techniques presented in sections 4.7.1 and 4.7.2, is that probabilistic reachability can be expressed via probabilities of LTL formulas as well as in PCTL*:

– A query $\mathrm{P}_{\max}(\diamond\phi)$ is equivalent to asking for the maximum probability of the LTL formula $\diamond\phi$ (and analogously for minimum queries).
– Properties of the form $\mathrm{P}(\diamond\phi) \sim x$ are equivalent to PCTL* formulas of the form $\mathrm{P}_{\sim x}(\diamond\phi)$.

4.5 Model Checking

In this section, we summarize the standard existing algorithms to perform exhaustive model checking of probabilistic reachability properties for finite-state MDP. The first two, which use linear programming and value iteration, are similar to the algorithms we presented for DTMC in Section 3.2.3. The third approach, policy iteration, is based on the enumeration of schedulers and thus has no DTMC counterpart. Again, we use the notions and assumptions from LTS analogously here when describing memory usage and runtime. In particular, the MDPs considered have n states, maximum fan-out $b = \max_{s \in S} |T(s)|$, and we are able to evaluate/store the transition function in constant time/memory.

Using Linear Programming

Exact reachability probabilities for *DTMC* can be computed by solving a linear equation system. The equations capture the relation between the probability values of a state and those of its successors: a state's reachability probability is the weighted sum of the probabilities of its successors according to the probabilistic transition function. For *MDP*, this relationship is more complex due to the additional nondeterministic choice between probability distributions over successors. Depending on whether we are interested in maximum or minimum reachability probabilities, we need to determine, for all states at once, the maximum or minimum of the weighted sums over all the successor distributions. This can be formulated as a linear programming (LP) problem as shown in Algorithms 9 and 12, which are based on [BK08, FKNP11]. Let us focus on Algorithm 9 for the moment, which computes maximum reachability probabilities. It builds the LP problem based on sets of states that reach a target state with maximum probability one ($S^{\max}_{=1}$) or zero ($S^{\max}_{=0}$). As for DTMC, these sets can be obtained with standard graph algorithms inside functions `Smax0` and `Smax1`, the details of which we omit here. In particular, no numeric computations are necessary. The variables x_s of the LP problem represent, for each state s, the (maximum) probability of reaching a ϕ-state. Line 7 produces one constraint for every transition of every relevant state. By using the greater-or-equal operator here and asking for maximal values in line 3, we achieve the global maximisation of the reachability probability.

Minimum reachability probabilities can be computed with LP as shown in Algorithm 12. In the LP problem itself, we see that maximisation has been replaced by minimisation, and the probability values x_s are required to be less than or equal than the weighted sum over the successors. However, the graph-based computations necessary to obtain the sets of probability-one and probability-zero states are also different and cannot directly be expressed

Input: Finite MDP $M = \langle S, A, T, s_{init}, AP, L \rangle$ and property $\mathrm{P}_{\max}(\diamond \phi)$
Output: $[\![\mathrm{P}_{\max}(\diamond \phi)]\!]_M$ (value in $[0,1]$)

$S^{\max}_{=1} :=$ `Smax1(`M, ϕ`)`, e.g. $\{ s \in S \mid \phi(L(s)) \}$
$S^{\max}_{=0} :=$ `Smax0(`M, ϕ`)`,
    e.g. $\{ s \in S \setminus S^{\max}_{=1} \mid \forall s' \colon (\mathrm{Paths}(s,s') \neq \varnothing \Rightarrow \neg\phi(L(s'))) \}$
$lp :=$ **maximise** $\sum_{s \in S} x_s$ **subject to**
    $x_s = 1$ for all $s \in S^{\max}_{=1}$
    $x_s = 0$ for all $s \in S^{\max}_{=0}$
    $0 < x_s < 1$ for all $s \notin S^{\min}_{=1} \cup S^{\min}_{=0}$
    $x_s \geq \sum_{s' \in S} \mu(s') \cdot x_{s'}$ for all $s \notin S^{\max}_{=1} \cup S^{\max}_{=0}$ and $\langle a, \mu \rangle \in T(s)$
end
solve the linear program lp and **return** $x_{s_{init}}$

Algorithm 9: MDP max. reachability checking with linear programming

function `Smin0(`$M = \langle S, T, s_{init}, AP, L \rangle$, ϕ`)`
    $R := \{ s \in S \mid \phi(L(s)) \}$
    repeat
        $R' := R$
        $R := R' \cup \{ s \in S \mid \forall \langle a, \mu \rangle \in T(s) \colon \exists s' \in R \colon \mu(s') > 0 \}$
    until $R = R'$
    return $S \setminus R$

Algorithm 10: Computing the set of states $S^{\min}_{=0}$ for an MDP [FKNP11]

function `Smin1(`$M = \langle S, T, s_{init}, AP, L \rangle$, $S^{\min}_{=0}$`)`
    $R := S \setminus S^{\min}_{=0}$
    repeat
        $R' := R$
        $R := R' \setminus \{ s \in R' \mid \exists \langle a, \mu \rangle \in T(s) \colon \exists s' \in S \setminus R' \colon \mu(s') > 0 \}$
    until $R = R'$
    return R

Algorithm 11: Computing the set of states $S^{\min}_{=1}$ for an MDP [FKNP11]

with our established notions like states reachable via finite paths. We thus give the (non-numeric) fixpoint computations to obtain these sets in Algorithms 10 and 11.

There is a wide range of methods to solve LP problems. The complexity of solving an LP problem is polynomial in the number of variables. The worst-case runtime of algorithms 9 and 12 with an asymptotically optimal LP solution algorithm is consequently a polynomial in n. In practice, the state spaces of realistic and relevant MDP models are too large to be analysed with LP methods [FKNP11]. Memory usage depends on the way the LP problem is represented and solved; what we see in our algorithms is that the number of constraints we generate is in $O(n \cdot b)$. Clearly, exhaustive model checking for MDP suffers from the state space explosion problem again (which affects the value and policy iteration approaches presented below just as well as this LP-based one).

Value Iteration

For DTMC models, the practical limitations of the exact linear equation system-based technique can be alleviated by using numeric dynamic programming in the form of value iteration. As mentioned, state space explosion is still the principal problem, but value iteration can be used on much larger state spaces before it becomes a limitation in practice. The value iteration algorithm for DTMC (Algorithm 5) only needs small changes to be applicable to MDP. The result is shown as Algorithm 13, which is the "Gauss-Seidel" variant of [FKNP11]. The main difference is that we need to compute the maximum over all transitions in addition to the weighted sum according to the probability distributions in line 8. To model-check minimum instead of maximum reachability queries, we would simply replace every occurrence of min by max, except for the one in line 9. Memory usage is in $O(n)$ as for DTMC, and runtime is dependent on ε and the structure of the model, too.

Policy Iteration

A new approach for MDP is to use *policy iteration*, also called *Howard's algorithm* for its discoverer Ronald A. Howard. *Policy* is an alternative name for what we called a *scheduler* in Definition 40. The idea of the algorithm, which is shown as Algorithm 14 [FKNP11], is to

1. start with some arbitrary (memoryless) scheduler (line 1),
2. use one of the DTMC model checking techniques to obtain the reachability probabilities for the Markov chain induced by the current scheduler (line 4),

Input: Finite MDP $M = \langle S, A, T, s_{init}, AP, L \rangle$ and property $\mathrm{P}_{\min}(\diamond \phi)$
Output: $[\![\mathrm{P}_{\min}(\diamond \phi)]\!]_M$ (value in $[0,1]$)

$S^{\min}_{=0} :=$ `Smin0(`M, ϕ`)`
$S^{\min}_{=1} :=$ `Smin1(`M, $S^{\min}_{=0}$`)`
$lp :=$ **minimise** $\sum_{s \in S} x_s$ **subject to**
    $x_s = 1$ for all $s \in S^{\min}_{=1}$
    $x_s = 0$ for all $s \in S^{\min}_{=0}$
    $0 < x_s < 1$ for all $s \notin S^{\min}_{=1} \cup S^{\min}_{=0}$
    $x_s \leq \sum_{s' \in S} \mu(s') \cdot x_{s'}$ for all $s \notin S^{\min}_{=1} \cup S^{\min}_{=0}$ and $\langle a, \mu \rangle \in T(s)$
end
solve the linear program lp and **return** $x_{s_{init}}$

Algorithm 12: MDP min. reachability checking with linear programming

Input: Finite MDP $M = \langle S, T, s_{init}, AP, L \rangle$, property $\mathrm{P}_{\max}(\diamond \phi)$ and $\varepsilon > 0$
Output: $[\![\mathrm{P}_{\max}(\diamond \phi)]\!]_M$ (value in $[0,1]$)

$S^{\max}_{=0} :=$ `Smax0(`M, ϕ`)`
$S^{\max}_{=1} :=$ `Smax1(`M, ϕ`)`
with $v \in S \to [0,1]$**:**
    foreach $s \in S$ **do** $v(s) := 1$ if $s \in S^{\max}_{=1}$, otherwise 0
    repeat
        $error := 0$
        foreach $s \in S \setminus (S^{\max}_{=1} \cup S^{\max}_{=0})$ **do**
            $v_{new} := \max \{ \sum_{s' \in \mathrm{support}(\mu)} \mu(s') \cdot v(s') \mid \langle a, \mu \rangle \in T(s) \}$
            if $v_{new} > 0$ **then** $error := \max\{ error, |v_{new} - v(s)| / v(s) \}$
            $v(s) := v_{new}$
        end
    until $error < \varepsilon$
    return $v(s_{init})$
end

Algorithm 13: MDP model checking with value iteration [FKNP11]

Input: Finite MDP $M = \langle S, A, T, s_{init}, AP, L \rangle$, property $\mathrm{P}_{\max}(\diamond \phi)$
Output: $[\![\mathrm{P}_{\max}(\diamond \phi)]\!]_M$ (value in $[0,1]$)

1 $\mathfrak{S} :=$ arbitrary scheduler for M
2 **repeat**
3 $\quad \mathfrak{S}' := \mathfrak{S}$
4 $\quad$ compute $p_s := [\![\mathrm{P}(\diamond \phi)]\!]$ in $\mathrm{ind}(M, \mathfrak{S})$ with initial state s for all $s \in S$
5 $\quad$ **foreach** $s \in S$ **do** $\mathfrak{S}(s) := \arg_{\langle a,\mu \rangle \in T(s)} \max \left\{ \sum_{s' \in \mathrm{support}(\mu)} \mu(s') \cdot p_{s'} \right\}$
6 **until** $\mathfrak{S} = \mathfrak{S}'$
7 **return** $p_{s_{init}}$

Algorithm 14: MDP model checking with policy iteration [FKNP11]

3. improve the scheduler by changing its decisions where it does not yet go to the successors with the highest probabilities (line 5), and
4. repeat until the scheduler cannot be improved any more.

As shown, Algorithm 14 computes maximum probabilities. To compute minimum probabilities instead, the max operation in line 5 would need to be replaced by min. Note also that the computation in line 4 can be performed for all states in one go using only a slightly modified version of any of the techniques presented for DTMC model checking in Section 3.2.3.

An upper bound on the runtime of policy iteration is the number of different schedulers, which is exponential. However, the performance of policy iteration in practice is competitive and comparable to that of value iteration [FKNP11]. Memory usage is obviously in $\mathrm{O}(n)$.

Other Approaches and Implementations

A prominent and widely-used tool that implements these standard techniques for exhaustive MDP model checking is PRISM [KNP11]. There are also other approaches, implemented in different tools, in addition to the three described so far. To name a few examples:

– The PASS tool [HHWZ10b] uses probabilistic counterexample-guided abstraction refinement [HWZ08] to combat the state space explosion problem. Instead of performing its analysis on the full state space of the given MDP, it extracts predicates from the high-level description of the process (i.e. a VMDP) to compute a more abstract model. It exploits the bounds given as part of qualitative form of probabilistic properties to improve the abstraction until the probability in that abstraction is lower or higher than the bound. This approach works very well whenever a coarse abstraction is sufficient to

satisfy the bound, but takes very long if many refinement steps are needed. By representing an infinite number of variable valuations in a finite number of predicates, PASS can deal with some infinite-state models.
– Another tool that can deal with infinite-state MDPs is INFAMY [HHWZ09]. Originally developed for continuous-time Markov chains, it has been extended to also work for MDP models. The key idea is to explore only the part of the model's state space, similar to a number of BFS layers around the initial state, that is sufficient for the bounds of the given property or the acceptable error in answering a query.
– Finally, an early implementation that used refinement and very specific state space reduction techniques [DJJL01, DJJL02] was the RAPTURE tool.

4.6 Statistical Model Checking

As mentioned at the beginning of this chapter, using statistical model checking to analyse probabilistic reachability properties on MDP models is problematic: If we are to perform simulation as we did for DTMC in Algorithm 6, we not only have to conduct a probabilistic experiment in each state, but also resolve the nondeterminism between the outgoing transitions first. That is, we have to mimic a scheduler's decisions. These scheduling choices determine which probability out of the interval between maximum and minimum we actually observe in SMC. As the only relevant values in verification are the actual maximum and minimum probabilities, we would need to be able to use the corresponding extremal schedulers to obtain useful results. These schedulers are not known upfront, however.

In the remainder of this chapter, we present several approaches to tackle this problem. First, in this section, we adapt the DTMC simulation algorithm to the MDP setting and show that naïve scheduling choices cannot lead to trustworthy, useful results. We then give a detailed presentation of two approaches in Section 4.7 that make use of the fact that a particular nondeterministic choice may not make a difference for the property at hand. In that case, the nondeterminism is spurious, and any resolution leads to the same probability. Provided all nondeterministic choices in an MDP are spurious for some property, its maximum and minimum reachability probability coincide and simulation results can be relied upon. The first approach (Section 4.7.1) is to use techniques adapted from *partial order reduction* to detect, on-the-fly during simulation, if the nondeterministic choice just encountered in a state is spurious and simulation can continue. However, the partial order-based check is conservative: It may not be able to identify all spurious choices as such. It works

on networks of VMDP and can in particular only prove interleavings, i.e. nondeterminism resulting from the interleaving semantics of parallel composition, as spurious. This limitation can be overcome by using the notion of *confluence* instead of partial order reduction (Section 4.7.2). The confluence-based check applies directly to the concrete state space of a model, i.e. to MDP. It is not limited to spurious interleavings, but it still is conservative. In contrast to the notion of partial order reduction we used previously, confluence can only deal with choices between nonprobabilistic transitions. Thus, the two approaches turn out to be incomparable w.r.t. the classes of MDP they are applicable to.

The goal of the two techniques outlined above, both based on state space reduction methods, is to prove on-the-fly during simulation that all nondeterministic choices encountered are spurious. This guarantees that the chosen resolution leads to the maximum and minimum probability. It indeed means that the minimum and maximum probabilities of reaching the target states are the same, i.e. the interval of probabilities is a singleton. This is, of course, not the case in general for arbitrary MDP models. Three approaches to perform SMC for general nondeterministic MDP have been developed by others, and we briefly present them for completeness and comparison in Section 4.8.

Henriques et al. [HMZ+12] first proposed the use of reinforcement learning, a technique from artificial intelligence, to actually learn the resolutions of nondeterminism (by memoryless schedulers) that *maximise* probabilities for a given bounded LTL property. While this allows SMC for models with arbitrary nondeterministic choices (not only spurious ones), scheduling decisions need to be stored for every *explored* state. Memory usage can thus be as in traditional model checking, but is highly dependent on the structure of the model and the learning process. However, several problems in their algorithm w.r.t. convergence and correctness have recently been described [LST14]. Similar learning-based methods have been picked up again by Brázdil et al. [BCC+14]. They propose two techniques that require different amounts of information about the model, but provide clear error bounds. Memory usage can again be as high as in model checking but depends on the model structure. We summarise these two learning-based approaches in Section 4.8.1. Our approaches based on confluence and POR have the same theoretical memory usage bound as the learning-based ones, but use comparatively little memory in practice. They do not introduce any additional overapproximation and thus have no influence on the usual error bounds of SMC.

Legay and Sedwards recently developed a second technique [LST14]. It is based on randomly generating a (large) number of schedulers, for each of which

```
1  function simulate(M = ⟨S, A, T, s_init, AP, L⟩, ℜ, φ, d)
2      s := s_init, seen := ∅
3      for i = 1 to d do
4          if φ(L(s)) then return true
5          else if s ∈ seen then return false
6          μ := ℜ(s)
7          ν := choose a transition ⟨a, ν⟩ randomly according to μ
8          if μ and ν are Dirac then seen := seen ∪ {s} else seen := ∅
9          s := choose a state s randomly according to ν
10     end
11     return unknown
```

Algorithm 15: Path gen. for an MDP and a resolver, with cycle detection

a standard SMC analysis is performed. To achieve the necessary memory efficiency, they propose an innovative O(1) encoding of a subset of all (memoryless or history-dependent) schedulers. However, their method cannot guarantee that the optimal schedulers are contained in the encodable subset, and cannot provide an error bound. We give a more detailed overview of this technique in Section 4.8.2, before concluding this chapter with an overall summary in Section 4.9.

4.6.1 Resolving Nondeterminism

In order to simulate an MDP, i.e. to generate paths through it, the nondeterministic choices need to be resolved. Adapting the DTMC simulation algorithm to MDP thus results in Algorithm 15. It takes as an additional parameter a *resolver* $\mathfrak{R}$, i.e. a function in $S \to \mathrm{Dist}(A \times \mathrm{Dist}(S))$ such that, for all states s of the MDP at hand, we have $\langle a, \mu \rangle \in \mathrm{support}(\mathfrak{R}(s)) \Rightarrow \langle a, \mu \rangle \in T(s)$. We can say that a resolver is a memoryless but probabilistic scheduler. If we burden the user with the task of specifying a resolver, SMC for MDP is easy: we can apply Algorithms 7 and 8 by merely changing the `simulate` function they use to the new one of Algorithm 15.

Many simulation tools, including e.g. the simulation engine that is part of PRISM, in fact implicitly use a specific built-in resolver so users do not even need to bother specifying one. On the other hand, this means that users are not able to do so if they wanted to, either. The implicit resolver that is typically used simply makes a uniformly distributed choice between the available transitions:

$$\mathfrak{R}_{\mathrm{Uni}} \stackrel{\mathrm{def}}{=} \{ s \mapsto \mathscr{U}(T(s)) \mid s \in S \}$$

However, one can think of other generic resolvers. For example, a total order on the actions (i.e. priorities) can be specified by the user, with the corresponding resolver making a uniform choice only between the available transitions with the highest-priority label. A special case of this appears when we consider MDP that model the passage of a unit of physical time with a dedicated `tick` action: If we assign the lowest priority to `tick`, we schedule the other transitions *as soon as possible*; if we assign the highest priority to `tick`, we schedule the other transitions *as late as possible*. We will revisit these *ASAP* and *ALAP* schedulers when we investigate real-time models in Chapter 5.

Unfortunately, performing SMC with some implicit scheduler as described above is not *sound*: While a probabilistic reachability property asks for *the* minimum or maximum probability of reaching a set of target states, using an implicit scheduler merely results in *some* probability in the interval between minimum and maximum.

Definition 53. An SMC procedure for MDP is *sound* if, given any MDP M and probabilistic reachability query $\mathrm{P}_{\max}(\diamond\, \phi)$ or $\mathrm{P}_{\min}(\diamond\, \phi)$, it returns a sample mean p_{smc} and a *useful* confidence statement relating p_{smc} to $[\![\mathrm{P}_{\max}(\diamond\, \phi)]\!]$ or $[\![\mathrm{P}_{\min}(\diamond\, \phi)]\!]$, respectively. Likewise, for qualitative probabilistic reachability properties, the reported decision must be connected to the correct (but unknown) result with a useful confidence statement.

Observe that we informally require a "useful" confidence statement. This is in order to remain abstract w.r.t. the concrete statistical method used. We consider e.g. confidence intervals with small ε and α or an APMC confidence $\langle k, \varepsilon, \delta \rangle$ with small ε and δ useful. In contrast, merely reporting some probability between minimum and maximum means that the potential error can be arbitrarily close to 1. This may still be of use in some applications, but is not a useful statement for verification.

In fact, doing so can be very misleading in verification where, after all, nondeterminism is assumed to be uncontrollable. The actual implementation of a system that we model with an MDP will need to actually resolve the decisions that were modelled as nondeterministic in some way[2], but there is no guarantee that this resolution matches the resolver we used for SMC. In particular, if the SMC result happens to be very optimistic (e.g. by not considering some few adverse environments or unfortunate implementations), the implicit-resolver approach can lead to unfounded conclusions that may jeopardise the safety of the actual system whose study the model was built for. Although nondeterminism can model the total absence of knowledge about a certain choice and a

[2] Assuming that reality is deterministic (and possibly probabilistic).

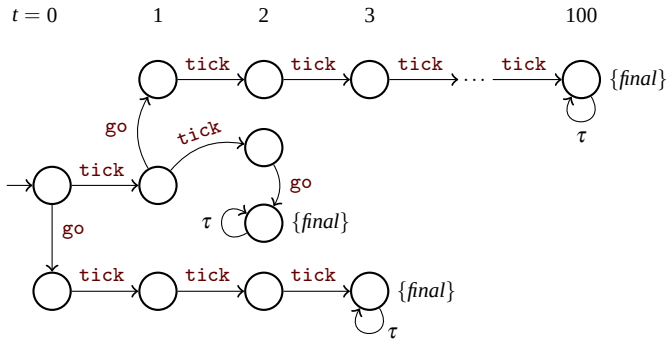


Figure 4.9: An anomalous discrete-timed system

uniformly random resolution seems to make sense because it is the maximum-entropy resolution, it still carries the risk of catastrophic behaviours being balanced by excellent ones in SMC, leading to an acceptable average result.

Example 21. Figure 4.9 shows a nonprobabilistic MDP that models a discrete-timed system with a special `tick` action as described above. It contains two nondeterministic choices between the action go and letting time pass. Let the property of interest be one of performance, namely whether a state labelled with atomic proposition *final* can be reached with probability at least 0.5 in time at most t, i.e. by taking at most t transitions labelled `tick`. When we encode that number in the atomic propositions as well, we need to verify properties $R_i = \mathrm{P}_{\max}(\Diamond \mathit{final} \wedge i) \geq 0.5$ for $i \in \{0, \ldots, 100\}$ (since we can easily see for this model that the maximum number of `tick`-labelled transitions that can be taken on any path is 100). The maximum and minimum i for which $[\![R_i]\!]$ is true are then the maximum and minimum time needed to reach a *final*-state with probability ≥ 0.5.

The results we would obtain via exhaustive model-checking are a minimum (best-case) time of 2 ticks and a maximum (worst-case) time of 100 ticks. For an SMC analysis, the nondeterminism needs to be resolved. Using $\mathfrak{R}_{\mathrm{Uni}}$, the result would be around 27 ticks. Note that this is quite far from the actual worst-case behaviour. In particular, by adding more "fast" or "slow" alternatives to the model, we can arbitrarily change the SMC result. Even worse, a very small change to the model can make quite a big difference: If the go-labelled transition to the upper branch were available in the initial state instead of after one `tick`, the "uniform" result would be 35 ticks.

Knowing that this is a timed model, we could try the ASAP and ALAP resolvers. Intuitively, we might expect to obtain the best-case behaviour for ASAP and the worst-case behaviour for ALAP. Unfortunately, the results run counter to this intuition: ASAP yields a time of 3 ticks, ALAP leads to the best-case result of 2 ticks, and the worst case of 100 ticks is completely ignored. This is because the example exhibits a timing anomaly: It is best to wait as long as possible before taking go in order to obtain the minimum time. For this toy example, the anomaly can easily be seen by inspecting the model, but similar effects (not limited to timing-related properties) may of course occur in complex models where they cannot so easily be spotted.

While problems with the credibility of simulation results have been observed before [AY06], most state-of-the-art simulation and SMC tools still implicitly resolve nondeterminism, typically using $\mathfrak{R}_{\text{Uni}}$. We argue that using some resolution method under-the-hood in an SMC tool—without warning the user of the possible consequences—is dangerous. As a consequence, the modes tool that provides SMC for MODEST models within the MODEST TOOLSET (cf. Section 1.4, page 25) in its default configuration aborts with an error when a nondeterministic choice is encountered. While it is possible to select between different built-in resolvers to have modes simulate such models anyway, including $\mathfrak{R}_{\text{Uni}}$, this requires deliberate action on part of the user.

4.7 SMC for Spuriously Nondeterministic MDP

We now present two first approaches to perform SMC for MDP in a sound way. They exploit the fact that resolving a nondeterministic choice in one way or another does not necessarily make a difference for the property at hand. In such a case, the choice is spurious, and any resolution leads to the same probability. When all nondeterministic choices in an MDP are spurious for some reachability property, then the maximum and minimum probability coincide and simulation results can be relied upon. Consider the following example:

Example 22. In the BRP models we considered so far, sender and receiver exchanged messages over dedicated channels. However, communication protocols often have to transfer messages in a broadcast fashion over shared media where a *collision* results if two senders transmit at the same time. In such an event, receivers are unable to extract any useful data out of the ensuing distortion. In Figure 4.10, we show VMDP modelling the sending of a message in such a scenario[3]. Processes H_i^a represent the senders, or *hosts*, which communicate with two alternative models M_{var} and M_{sync} for the shared medium that

[3]We omit the τ-loops that need to be added to deadlock states for brevity from now on.

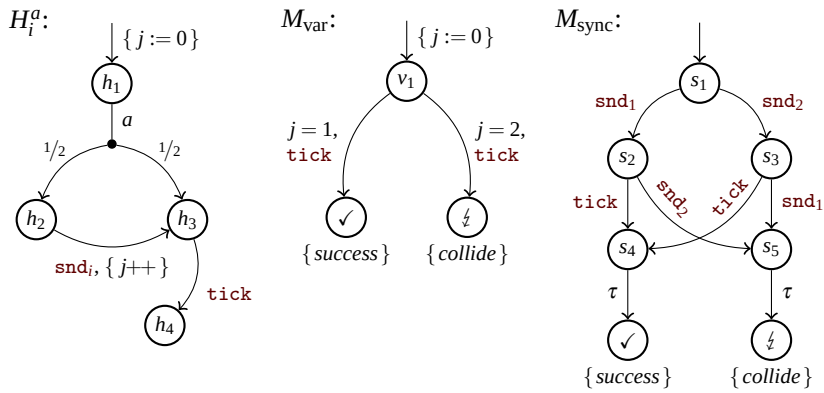


Figure 4.10: VMDP modelling hosts (H) that send on a shared medium (M)

observes whether a message is transmitted successfully or a collision occurs. Communication with M_{var} is by synchronisation on $\mathtt{tick}$ and via shared variable j, while communication with M_{sync} is purely by synchronisation. The full models we are interested in are the networks $N_v^a = \{ H_1^a, H_2^a, M_v \}$ for all four combinations of $a \in \{ \mathtt{act}, \tau \}$ and $v \in \{ \text{var}, \text{sync} \}$.

Seen on their own, the host VMDP as well as their MDP semantics are deterministic. M_{var} looks nondeterministic as a VMDP (as v_1 has two outgoing edges) but its semantics is a deterministic MDP (since the guards of the edges are disjoint), while M_{sync} and its semantics are nondeterministic (because it has no variables and thus already is an MDP). If we consider the MDP semantics of the four networks, we can with moderate effort see that they all contain at least one nondeterministic state (namely when both hosts happen to be in location h_2 and thus at least the two transitions labelled $\mathtt{snd}_1$ and $\mathtt{snd}_2$ are enabled) and possibly more. Still, we have that $\mathrm{P}_{\min}(\diamond \mathit{success}) = \mathrm{P}_{\max}(\diamond \mathit{success}) = 0.5$ and $\mathrm{P}_{\min}(\diamond \mathit{collide}) = \mathrm{P}_{\max}(\diamond \mathit{collide}) = 0.25$. For the given atomic propositions, all the nondeterministic choices are thus spurious. As for the problem highlighted by Figure 4.9 previously, this is relatively easy to see for these small models, but will usually be anything but obvious for larger, more complex and realistic networks of MDP.

Reduced Deterministic MDP

In order to perform SMC for spuriously nondeterministic MDP as those presented in the previous example, it suffices to supply a resolver to the path generation procedure of Algorithm 15 that corresponds to a deterministic reduction

function and that preserves minimum and maximum reachability probabilities. Formally, we want to use a function f such that

$$\begin{aligned} & f \text{ is a deterministic reduction function} \\ \wedge\ & [\![\mathrm{P}_{\max}(\diamond\phi)]\!]_M = [\![\mathrm{P}_{\max}(\diamond\phi)]\!]_{\mathrm{red}(M,f)} \\ \wedge\ & [\![\mathrm{P}_{\min}(\diamond\phi)]\!]_M = [\![\mathrm{P}_{\min}(\diamond\phi)]\!]_{\mathrm{red}(M,f)} \end{aligned} \tag{4.2}$$

Observe that the existence of such a reduction function for a given MDP and property indeed means that the minimum and the maximum probability are the same:

Lemma 1. Given an MDP M and a state formula ϕ over its atomic propositions, we have that

$$\exists\, f \text{ satisfying Condition 4.2} \Rightarrow [\![\mathrm{P}_{\max}(\diamond\phi)]\!]_M = [\![\mathrm{P}_{\min}(\diamond\phi)]\!]_M.$$

Proof. Because f is deterministic, $\mathrm{red}(M,f)$ is a DTMC. Therefore, we have

$$[\![\mathrm{P}_{\max}(\diamond\phi)]\!]_{\mathrm{red}(M,f)} = [\![\mathrm{P}_{\min}(\diamond\phi)]\!]_{\mathrm{red}(M,f)}$$

and by Condition 4.2, it follows that $[\![\mathrm{P}_{\max}(\diamond\phi)]\!]_M = [\![\mathrm{P}_{\min}(\diamond\phi)]\!]_M$. □

The existence of a reduction function satisfying Condition 4.2 consequently means that all nondeterministic choices in the MDP can be resolved in such a way that SMC computes the actual minimum and maximum probability (which are necessarily the same). Moreover, it means that *no matter how* we resolve the nondeterminism, we obtain the correct probabilities:

Theorem 2. Given an MDP M and a state formula ϕ over its atomic propositions, we have that

$$\begin{aligned} & \exists\, f \text{ satisfying Condition 4.2} \\ \Rightarrow\ & \forall \text{reduction functions } f' \colon [\![\mathrm{P}_{\max}(\diamond\phi)]\!]_M = [\![\mathrm{P}_{\max}(\diamond\phi)]\!]_{\mathrm{red}(M,f')} \\ & \qquad \wedge\ [\![\mathrm{P}_{\min}(\diamond\phi)]\!]_M = [\![\mathrm{P}_{\min}(\diamond\phi)]\!]_{\mathrm{red}(M,f')}. \end{aligned}$$

Proof. We show the contraposition

$$\begin{aligned} & \forall \text{reduction functions } f \colon f \text{ does not satisfy Condition 4.2} \\ \Leftarrow\ & \exists \text{reduction function } f' \colon [\![\mathrm{P}_{\max}(\diamond\phi)]\!]_M \neq [\![\mathrm{P}_{\max}(\diamond\phi)]\!]_{\mathrm{red}(M,f')} \\ & \qquad \vee\ [\![\mathrm{P}_{\min}(\diamond\phi)]\!]_M \neq [\![\mathrm{P}_{\min}(\diamond\phi)]\!]_{\mathrm{red}(M,f')}. \end{aligned}$$

We observe that, for reduction functions f',

$$\begin{aligned} & \exists\, f' \colon [\![\mathrm{P}_{\max}(\diamond\phi)]\!]_M \neq [\![\mathrm{P}_{\max}(\diamond\phi)]\!]_{\mathrm{red}(M,f')} \\ \Rightarrow\ & \exists\, f', \mathfrak{S}_1, \mathfrak{S}_2 \colon [\![\mathrm{P}(\diamond\phi)]\!]_{\mathrm{ind}(M,\mathfrak{S}_1)} \neq [\![\mathrm{P}(\diamond\phi)]\!]_{\mathrm{ind}(\mathrm{red}(M,f'),\mathfrak{S}_2)} \end{aligned}$$

if we apply the definition of the semantics of probabilistic reachability properties for MDP. Since a scheduler for $\mathrm{red}(M, f')$ is also a scheduler for M, it follows that

$$\exists \mathfrak{S}_1, \mathfrak{S}_2 \colon [\![\mathrm{P}(\diamond \phi)]\!]_{\mathrm{ind}(M,\mathfrak{S}_1)} \neq [\![\mathrm{P}(\diamond \phi)]\!]_{\mathrm{ind}(M,\mathfrak{S}_2)}$$

which, again using the probabilistic reachability semantics for MDP, implies

$$[\![\mathrm{P}_{\max}(\diamond \phi)]\!]_M \neq [\![\mathrm{P}_{\min}(\diamond \phi)]\!]_M$$

which is in contradiction to Lemma 1. Thus no reduction function can satisfy Condition 4.2. The same argument works analogously if we start with the $\mathrm{P}_{\min}(\cdot)$ case instead of $\mathrm{P}_{\max}(\cdot)$. □

With small modifications to the proof, we could also show the same result for resolvers instead of reduction functions. This means that we could use $\mathfrak{R}_{\mathrm{Uni}}$ and obtain correct results, too—that is, *if we know* that a reduction function satisfying Condition 4.2 exists for the given model and property. Unfortunately, attempting to find such a function in an "offline" manner before we start simulation, i.e. for all states at once, would negate the core advantage of SMC over exhaustive model checking: its constant or low memory usage.

Preservation of Probabilistic Reachability

The reduction functions we are looking for need to satisfy two relatively separate requirements according to Condition 4.2: they need to be deterministic, and they need to preserve maximum and minimum reachability probabilities. The former is a simple property that appears easy to check or ensure by construction. However, it is not so obvious what kind of criteria would guarantee the latter.

In exhaustive model checking, equivalence relations that divide the state space into partitions of states with "equivalent" behaviour have been studied extensively: They allow the replacement of large state spaces with smaller quotients under such a relation and thus help alleviate the state space explosion problem. We aim to build upon this research to construct our reduction functions. As we are interested in the verification of probabilistic reachability properties, we could potentially use any equivalence relation that preserves those properties:

Definition 54 (Preservation of probabilistic reachability)**.** An equivalence relation $\sim$ over MDP *preserves probabilistic reachability* if, for all pairs $\langle M_1, M_2 \rangle$ of MDP with atomic propositions AP, we have that

$$M_1 \sim M_2 \Rightarrow \forall \phi \in \Phi_{AP} \colon [\![\mathrm{P}_{\max}(\diamond \phi)]\!]_{M_1} = [\![\mathrm{P}_{\max}(\diamond \phi)]\!]_{M_2}$$

and also

$$M_1 \sim M_2 \Rightarrow \forall \phi \in \Phi_{AP} \colon [\![\mathrm{P}_{\min}(\diamond \phi)]\!]_{M_1} = [\![\mathrm{P}_{\min}(\diamond \phi)]\!]_{M_2}.$$

In case this statement is correct even if we replace the implications by equivalences, we say that $\sim$ *characterises* probabilistic reachability. On the other hand, if only one of the two conditions holds, we say that $\sim$ preserves maximum or minimum probabilistic reachability, respectively.

Candidates for $\sim$ would be appropriate variants of trace equivalence, simulation or bisimulation relations. In fact, it turns out that there are two well-known techniques to reduce the size of MDP in exhaustive model checking that appear promising: *partial order reduction* [BDG06, God96, Pel94, Val90] and *confluence reduction* [BvdP02, TSvdP11, TvdPS13]. Both provide an algorithm to obtain a reduction function such that the original and the reduced model are equivalent according to relations that preserve probabilistic reachability. Both algorithms can be performed *on-the-fly* while exploring the state space [GvdP00, Pel96], which avoids having to store the entire (and possibly too large) state space in memory at any time. Instead, the reduced model is generated directly. We therefore study in sections 4.7.1 and 4.7.2 whether the two algorithms can be adapted to compute a reduction function on-the-fly during simulation with little extra memory usage.

Partial Exploration during Simulation

If we compute the reduction function on-the-fly, however, we only compute it for a subset of the reachable states, namely those visited during the simulation runs of the current SMC analysis. We are thus unable to check whether the supposedly simple first requirement of Condition 4.2, determinism, actually holds for all states of the model, or at least (and sufficiently) for all states in the reduced state space.

Yet, requiring determinism for all states is more restrictive than necessary. Instead of Condition 4.2, let us require that the reduction function f computed on-the-fly during the calls to function `simulate` in one concrete SMC analysis satisfies

f is a reduction function s.t.

$$\begin{aligned} s \in \Pi \Rightarrow |f(s)| = 1 \wedge s \notin \Pi \Rightarrow f(s) = T(s) \\ \wedge\, [\![\mathrm{P}_{\max}(\diamond \phi)]\!]_M = [\![\mathrm{P}_{\max}(\diamond \phi)]\!]_{\mathrm{red}(M,f)} \\ \wedge\, [\![\mathrm{P}_{\min}(\diamond \phi)]\!]_M = [\![\mathrm{P}_{\min}(\diamond \phi)]\!]_{\mathrm{red}(M,f)} \end{aligned} \tag{4.3}$$

where T is the transition function of M and Π is the set of paths explored during the simulation runs and we abuse notation to write $s \in \Pi$ in place of

$s \in \{ s' \in S \mid \exists \ldots s' \ldots \in \Pi \}$. This means that the function still has to preserve probabilistic reachability, and it still must be deterministic on the states we visit during simulation, but on all other states, it is now required to perform no reduction instead. As before, if we compute f such that $M \sim \mathrm{red}(M, f)$ is guaranteed for a relation $\sim$ that preserves probabilistic reachability, we already know that the second line of Condition 4.3 is satisfied.

Although Lemma 1 and Theorem 2 do not hold for such a reduction function in general, let us now show why it still leads to a sound SMC analysis in the sense of Definition 53. Recall that, for every MDP M and state formula ϕ, there are schedulers $\mathfrak{S}_{\max}$ and $\mathfrak{S}_{\min}$ that maximise resp. minimise the probability of reaching a ϕ-state, i.e.

$$[\![\mathrm{P}_{\max}(\diamond \phi)]\!]_M = [\![\mathrm{P}(\diamond \phi)]\!]_{\mathrm{ind}(M, \mathfrak{S}_{\max})} \quad \text{and} \quad [\![\mathrm{P}_{\min}(\diamond \phi)]\!]_M = [\![\mathrm{P}(\diamond \phi)]\!]_{\mathrm{ind}(M, \mathfrak{S}_{\min})}.$$

If f is a reduction function that satisfies Condition 4.3, then there is at least one such maximising (minimising) scheduler $\mathfrak{S}_{\max}$ ($\mathfrak{S}_{\min}$) that is valid for f. For the states where f is deterministic, this is the case due to the first (second) part of the second line of Condition 4.3. For this scheduler, we therefore also have

$$[\![\mathrm{P}_{\max}(\diamond \phi)]\!]_{\mathrm{red}(M,f)} = [\![\mathrm{P}(\diamond \phi)]\!]_{\mathrm{ind}(M, \mathfrak{S}_{\max})}$$

(and the corresponding statement for $\mathfrak{S}_{\min}$). When exploring the set of paths Π, by following f, the simulation runs also follow both $\mathfrak{S}_{\max}$ and $\mathfrak{S}_{\min}$ (due to determinism of f on Π and the schedulers being valid for f). The resulting sample mean is thus the same as if we had performed the simulation runs on either of the DTMC $\mathrm{ind}(M, \mathfrak{S}_{\max})$ or $\mathrm{ind}(M, \mathfrak{S}_{\min})$. In consequence, whatever statement connecting the sample mean and the actual result we obtain from the ensuing statistical analysis (such as those in algorithms 7 and 8) is correct. In particular, we do not need to modify the reported confidence to account for nondeterministic choices that we did not encounter on the paths in Π: We already did simulate the correct "maximal" respectively "minimal" DTMC.

We can now adapt the simulation function of Algorithm 15 to use instead of a resolver $\mathfrak{R}$ a procedure $\mathfrak{A}$ that acts as a function in $S \to (A \times \mathrm{Dist}(S)) \cup \{\bot\}$ (i.e. it deterministically selects a transition or $\bot$ whereas a resolver returns a distribution over transitions) and on-the-fly computes the output of a reduction function satisfying Condition 4.3. It returns a transition to follow during simulation if the current state is deterministic or if it can show the nondeterminism to be spurious. Otherwise, it returns $\bot$, which causes both the current simulation run as well as the SMC analysis to be aborted. In particular, for the underlying reduction function f to satisfy Condition 4.3, $\mathfrak{A}$ must be implemented in a deterministic manner, i.e. it must always return the same transition for the same state. When the SMC analysis terminates successfully (i.e. $\mathfrak{A}$ has never returned $\bot$), $\mathfrak{A}$ will have determined f to singleton sets for the states visited, and

```
1  function simulate(M = ⟨S, A, T, s_init, AP, L⟩, 𝔄, φ, d)
2      s := s_init, seen := ∅
3      for i = 1 to d do
4          if φ(L(s)) then return true
5          else if s ∈ seen then return false
6          tr := 𝔄(s)
7          if tr = ⊥ then return unknown
8          ⟨a, μ⟩ := tr
9          if μ is Dirac then seen := seen ∪ {s} else seen := ∅
10         s := choose a state randomly according to μ
11     end
12     return unknown
```

Algorithm 16: Simulation with an algorithm computing a reduction function

we complete f to map all other states s to $T(s)$ for the correctness argument. We show the adapted simulation function as Algorithm 16.

It now remains to find out whether there are such procedures $\mathfrak{A}$ that are both efficient (i.e. they do not destroy the advantage of SMC in memory usage and they do not excessively increase runtime) and effective (i.e. they never return $\perp$ for practically relevant models). It turns out that at least two approaches work well: using a check inspired by partial order reduction techniques, which we present in Section 4.7.1, and checking for confluent transitions as we show in Section 4.7.2. We investigate their efficiency and effectiveness using three case studies in Section 4.7.3.

4.7.1 Using Partial Order Reduction

For exhaustive model checking of networks of VLTS, an efficient method to deal with models containing spurious nondeterminism resulting from the interleaving of parallel processes already exists, namely partial order reduction (POR, [God96, Pel94, Val90]). It reduces such models to smaller ones containing only those paths of the interleavings necessary to not affect the end result. POR was first generalised to the probabilistic domain preserving linear time properties, including the probabilities of LTL formulas without the *next* operator X [BGC04, DN04], with a later extension to preserve branching time properties without next, i.e. PCTL$^*_{\setminus \mathrm{X}}$ [BDG06]. In the remainder of this section, we first recall how POR for MDP works and then detail how to harvest it to compute a reduction function satisfying Condition 4.3 on-the-fly during

A0 For all states $s \in S$, $\mathsf{ample}(s) \subseteq T(s)$.
A1 If $s \in S_f$ and $\mathsf{ample}(s) \neq T(s)$, then no transition in $\mathsf{ample}(s)$ is visible.
A2 For every path $(tr_1 = \langle s, a, \mu \rangle), \ldots, tr_n, tr, tr_{n+1}, \ldots$ in M where $s \in S_f$ and tr is dependent on some transition in $\mathsf{ample}(s)$, there exists $i \in \{1, \ldots, n\}$ such that $tr_i \in [\mathsf{ample}(s)]_{\equiv}$.
A3 In each end component $\langle S_e, T_e \rangle$ of M_f, there exists a state $s \in S_e$ that is fully expanded, i.e. $\mathsf{ample}(s) = T(s)$.
A4 If $\mathsf{ample}(s) \neq T(s)$, then $|\mathsf{ample}(s)| = 1$.

Table 4.1: Conditions for the ample sets

simulation. The relation $\sim$ between the original and the reduced model guaranteed by this approach is stutter equivalence, which preserves the probabilities of $\mathrm{LTL}_{\setminus \mathrm{X}}$ properties [BDG06].

Partial Order Reduction for MDP

The aim of partial order techniques for exhaustive model checking is to avoid building the full state space corresponding to a model. Instead, a smaller state space is constructed and analysed where the spurious nondeterministic choices resulting from the interleaving of independent transitions are resolved. The reduced system is not necessarily deterministic, but smaller, which increases the performance and reduces the memory demands of model checking (if the reduction procedure is less expensive than analysing the full model right away).

Based on the *ample set method* [Pel94] for nonprobabilistic systems, partial order reduction has been generalised to the MDP setting for both linear-time [BGC04, DN04] and branching-time properties [BDG06][4]. A method based on stubborn sets [Val90] has been developed later, too [HKQ11]. Our approach is based on ample sets. The essence is to identify an ample set of transitions $\mathsf{ample}(s)$ for every state $s \in S$ of the MDP M, yielding the reduction function

$$f = f_{\mathsf{ample}} = \{\, s \mapsto \{\, \langle a, \mu \rangle \mid s \xrightarrow{a} \mu \in \mathsf{ample}(s) \,\} \,\},$$

such that conditions A0-A4 of Table 4.1 are satisfied (where S_f denotes the state space of $M_f = \mathrm{red}(M, f)$, cf. Definition 41).

For partial order reduction, the notion of *(in)dependent* transitions[5] (see rule A2) is crucial. Intuitively, the order in which two independent transitions

[4] We mostly cite [BDG06] in the remainder of this section as it nicely summarises the linear-time approaches as well.
[5] By abuse of language, we use the word "transition" when we actually mean "equivalence class of transitions under $\equiv$".

are executed is irrelevant in the sense that they do not disable each other (forward stability) and that executing them in a different order from a given state still leads to the same states with the same probabilities (commutativity). Formally:

Definition 55. Two equivalence classes $[tr'_1]_{\equiv} \neq [tr'_2]_{\equiv}$ of transitions of an MDP $M = \langle S, A, T, s_{init}, AP, L\rangle$ are independent if and only if for all states $s \in S$ with $tr_1, tr_2 \in T(s)$, $tr_1 = \langle s, a_1, \mu_1\rangle \in [tr'_1]_{\equiv}$, $tr_2 = \langle s, a_2, \mu_2\rangle \in [tr'_2]_{\equiv}$,

I1 $s' \in \text{support}(\mu_1) \Rightarrow tr_2 \in [T(s')]_{\equiv}$ and vice-versa *(forward stability)*, and then also

I2 $\forall s' \in S\colon \sum_{s'' \in S} \mu_1(s'') \cdot \mu_2^{s''}(s') = \sum_{s'' \in S} \mu_2(s'') \cdot \mu_1^{s''}(s')$ *(commutativity)*

where $\mu_i^{s''}$ is the single element of $\{\, \mu \mid \langle s'', a_i, \mu\rangle \in T(s'') \cap [tr_i]_{\equiv} \,\}$.

Checking dependence by testing these conditions on all pairs of transitions for all states of an MDP is impractical. Partial order reduction is thus typically applied to the MDP semantics of networks of VMDP where sufficient and easy-to-check conditions on the symbolic level can be used. In that setting, $\equiv_E$ is used for the equivalence relation $\equiv$. Then, two transitions tr_1 and tr_2 in the MDP correspond to two edges $e_i = \langle l_i, g_i, a_i, m_i\rangle$ on the level of the parallel composition semantics of the network of VMDP. Each of these edges in turn is the result of one or more individual edges in the component VMDP. We can thus associate with each transition tr_i a (possibly synchronised) edge e_i and a (possibly singleton) set of component VMDP. The following are then an example of such sufficient and easy-to-check symbolic-level conditions:

J1 The sets of VMDP that tr_1 and tr_2 originate from are disjoint, and

J2 for all valuations v, $m_1(\langle U_1, l'\rangle) \neq 0 \wedge m_2(\langle U_2, l'\rangle) \neq 0$ implies that

$$[\![g_2]\!](v) \Rightarrow [\![g_2]\!]([\![A_1]\!](v))) \wedge [\![A_1]\!]([\![A_2]\!](v)) = [\![A_2]\!]([\![A_1]\!](v)$$

and vice-versa.

J1 ensures that the only way for the two transitions to influence each other is via global variables, and J2 makes sure that this does not actually happen, i.e. each transition modifies variables only in ways that do not *disable* the other's guard and the assignments are commutative. This check can be implemented on a syntactic level for the guards and the expressions occurring in assignments.

Using the ample set method with conditions A0-A4 and I1-I2 or J1-J2 gives the following result:

Theorem 3 ([BDG06])**.** If an MDP M is reduced to an MDP $\text{red}(M, f_{\mathsf{ample}})$ using the ample set method as described above, then $M \sim \text{red}(M, f_{\mathsf{ample}})$ where $\sim$ is stutter equivalence.

A0 For all states $s \in S$, $\mathsf{ample}(s) \subseteq T(s)$.
A1 If $s \in S_f$ and $\mathsf{ample}(s) \neq T(s)$, then no transition in $\mathsf{ample}(s)$ is visible.
A2' Every path in M starting in s has a finite acyclic prefix $\langle tr_1, \ldots, tr_n \rangle$ of length at most k_{max} (i.e. $n \leq k_{max}$) such that $tr_n \in [\mathsf{ample}(s)]_{\equiv}$ and for all $i \in \{1, \ldots, n-1\}$, tr_i is independent of all transitions in $\mathsf{ample}(s)$.
A3' If more than l states have been explored, one of the last l states was fully expanded.
A4 If $\mathsf{ample}(s) \neq T(s)$, then $|\mathsf{ample}(s)| = 1$.

Table 4.2: On-the-fly conditions for every state s encountered during simulation

Stutter equivalence preserves the probabilities of $\mathrm{LTL}_{\setminus \mathrm{X}}$ properties and thus probabilistic reachability. For simulation, we are not particularly interested in smaller state spaces, but we can use partial order reduction to distinguish between spurious and actual nondeterminism.

On-the-fly Partial Order Checking

We can use partial order reduction on-the-fly during simulation to find out whether nondeterminism is spurious: For any state with more than one outgoing transition, we simply check whether a singleton set of transitions exists that is an ample set according to conditions A0 through A4. This check can be used as parameter $\mathfrak{A}$ to Algorithm 16: If a singleton ample set exists, we return its transition. If all valid ample sets contain more than one transition, we cannot conclude that the nondeterminism between them is spurious, and $\perp$ is returned to abort simulation and SMC. To make the algorithm deterministic in case there is more than one singleton ample set, we assume a global order on transitions and return the first set according to this order.

Algorithm The ample set construction relies on conditions A0 through A4, but looking at their formulation in Table 4.1, conditions A2 and A3 cannot be checked on-the-fly without possibly exploring and storing lots of states—potentially the entire MDP. To bound this number of states and ensure termination for infinite-state systems, we instead use the conditions shown in Table 4.2, which are parametric in k_{max} and l. Condition A2 is replaced by A2', which bounds the lookahead inherent to A2 to paths of length at most k_{max}. Notably, choosing $k_{max} = 0$ is equivalent to not checking for spuriousness at all but aborting on the first nondeterministic choice. Instead of checking for end components as in Condition A3, we use A3' that replaces the notion of an end component with the notion of a sequence of at least l states.

```
1  function simulate(M = ⟨S, A, T, s_init, AP, L⟩, 𝔄, φ, d, l)
2      s := s_init, seen := empty stack, l_current := 0
3      for i = 1 to d do
4          if φ(L(s)) then return true
5          else if s ∈ seen then
6              len_cycle := 1
7              while seen.pop() ≠ s do len_cycle := len_cycle + 1
8              if len_cycle ≤ l_current then return unknown else return false
9          end
10         if |T(s)| = 1 then l_current := 0, tr := the single element of T(s)
11         else if l_current + 1 = l then return unknown
12         else l_current := l_current + 1, tr := 𝔄(s)
13         if tr = ⊥ then return unknown
14         ⟨a, μ⟩ := tr
15         if μ is Dirac then seen.push(s) else seen := empty stack
16         s := choose a state randomly according to μ
17     end
18     return unknown
```

Algorithm 17: Simulation with reduction function and cycle condition check

We first modify Algorithm 16 to include the cycle check of Condition A3'. The result is shown as Algorithm 17. A new variable $l_{current}$ keeps track of the number of transitions taken since the last fully expanded state. It is reset when such a state is encountered in line 10, and otherwise incremented in line 12. When $l_{current}$ would reach the bound of Condition A3', given as parameter l, simulation aborts in line 11. While this is so far straightforward and guarantees that Condition A3' holds when `simulate` returns *true*, the case of returning *false*, which relies on cycle detection, needs special care: We need to make sure that the detected cycle also contains at least one fully expanded state. For this purpose, we compute the length of the cycle and compare it to $l_{current}$ in lines 6 to 8. Finally, whenever a nondeterministic state is encountered, we call the procedure $\mathfrak{A}$ to check whether the nondeterminism is spurious in line 12.

In order to complete our partial order-based simulation procedure, procedures for checking conditions A1 and A2' are needed. This can be done by using the `resolvePOR` function of Algorithm 18 in place of $\mathfrak{A}$. `resolvePOR` simply iterates over the outgoing transitions of the nondeterministic state and returns the first one that constitutes a valid singleton ample set according to conditions A1 and A2'. Checking that these two conditions hold for a candidate ample set

is the job of function `chkAmpleSet`. It first compares the labelling of s with that of each successor to make sure that Condition A1 holds. If that is the case, `chkPaths` is called to verify A2'. `chkPaths` takes four parameters: The state s from which to check all outgoing paths, the single transition tr_{ample} in the potential ample set, a reference to a set *seen* of states already visited during this particular POR check, and a natural number *steps* counting the transitions taken from the initial nondeterministic state. The function follows all paths starting in s in the MDP recursively (i.e. via depth-first search) until it finds a transition that is either equivalent to or dependent on the one in the candidate ample set (lines 16 and 17). If the former happens before the latter, the path satisfies the condition of A2'. On the other hand, if a dependent transition occurs before an "ample" one, the current path is a counterexample to the requirements on all paths of A2' and $\{tr_{ample}\}$ is not a valid ample set. All other transitions are neither equivalent to tr_{ample} nor dependent on it, so we recurse in line 20 with an incremented *step* counter. If *step* reaches the bound k_{max} before an ample or dependent transition is found (line 14), a counterexample to A2' (though not necessarily to A2) has been found, too. Finally, `chkPaths` ignores cycles of independent transitions (line 12), which is what the set *seen* is used for. This means that indeed, only acyclic prefixes of length up to k_{max} are considered.

Function `chkPaths` uses two additional helper methods that we do not show in further detail: `equivalent` and `dependent`. The former returns *true* if and only if its two parameters are equivalent transitions according to $\equiv_E$. If the latter returns *false*, then its two parameters are independent transitions. `equivalent` necessarily needs to go back to the network of VMDP that the MDP at hand originates from in order to be able to reason about $\equiv_E$. This is also the case in typical implementations of `dependent` that use conditions J1 and J2 (which includes our implementation in modes).

Correctness We can now state the correctness of the on-the-fly partial order check as described above:

Lemma 2 (Correctness of SMC with on-the-fly POR check)**.** If an SMC analysis terminates and does not return *unknown*
– using function `simulate` of Algorithm 17 to explore the set of paths Π
– together with function `resolvePOR` of Algorithm 18 in place of $\mathfrak{A}$,
then the function $f = f_{POR} \cup \{ s \mapsto T(s) \mid s \notin \Pi \}$ satisfies Condition 4.3, where f_{POR} maps a state $s \in \Pi$ to $T(s)$ if it is deterministic and to the result of the call to `resolvePOR` otherwise.

Proof. By construction and because `resolvePOR` is deterministic, f is a reduction function that satisfies $s \in \Pi \Rightarrow |f(s)| = 1 \wedge s \notin \Pi \Rightarrow f(s) = T(s)$. It

```
1  function resolvePOR(s)
2      foreach tr ∈ T(s) in fixed global order do
3          if chkAmpleSet({ tr }) then return tr
4      end
5      return ⊥

6  function chkAmpleSet({ s -a-> μ })
7      foreach s' ∈ support(μ) do
8          if L(s) ≠ L(s') then return false
9      end
10     return chkPaths(s, s -a-> μ, { s }, 0)

11 function chkPaths(s, tr_ample, ref seen, steps)
12     if s ∈ seen then return true        // cyclic path w/o dependent trans.
13     seen := seen ∪ { s }
14     if steps ≥ k_max then return false        // lookahead bound exceeded
15     foreach tr = s -a-> μ ∈ T(s) do
16         if equivalent(tr, tr_ample) then continue    // ample tr. reached
17         if dependent(tr, tr_ample) then return false  // dependent trans.
18         foreach s' ∈ support(μ) do
19             if ¬ chkPaths(s', tr_ample, ref seen, steps + 1) then
20                 return false
21             end
22         end
23     end
24     return true                         // all paths satisfy Condition A2'
```

Algorithm 18: On-the-fly partial order check

remains to show that minimum and maximum probabilistic reachability probabilities for any state formula over the atomic propositions are the same for the original and the reduced MDP. From Theorem 3, we know that this is the case if f maps every state to a valid ample set according to conditions A0 through A4. Note that $T(s)$ is always a valid ample set, so this is already satisfied for the states $s \notin \Pi$. If conditions A2' and A3' hold, then so do A2 and A3. All the conditions of Table 4.2 are indeed guaranteed for the states $s \in \Pi$:

A0 is satisfied by construction.

A1 is checked for nondeterministic states by `chkAmpleSet`, and does not apply to deterministic states.

A2' is ensured by `chkPaths` as described previously.

A3' is checked via $l_{current}$ in the modified `simulate` function of Algorithm 17. In case *false* is returned by `simulate`, i.e. a cycle is reached, correctness of the check can be seen directly. In case *true* is returned, $\phi(L(s))$ has *just* become true, so the previous transition was visible. By Condition A1, this means that the very previous state was fully expanded.

A4 is satisfied by construction for the states visited because we only select singleton ample sets, and by definition for all other states since we assume no reduction for those, i.e. $\mathsf{ample}(s) = T(s)$. □

Runtime and Memory Usage

The runtime and memory usage of SMC with on-the-fly POR check depends directly on the amount of lookahead necessary in the function `chkPaths`. If k_{max}, which imposes a bound on this lookahead, needs to be increased to detect all spurious nondeterminism as such, performance in terms of runtime and memory demand will degrade. Note, though, that it is not the actual user-chosen value of k_{max} that is relevant for the performance penalty, but what we denote by k: the smallest value of k_{max} necessary for Condition A2' to succeed in the model at hand. If a larger value is chosen for k_{max}, A2' still only causes paths of length k to be explored[6]. The value of l has no performance impact.

More precisely, the memory usage of this approach is bounded by $b \cdot k$ where b is the maximum fan-out of the MDP. We will see that small k tend to suffice in practice, and the actual extra memory usage stays very low. Regarding runtime, exploring parts of the state space that are not part of the current path (up to b^k states per invocation of A2') induces a performance penalty. In addition, the algorithm may recompute information that was partially computed beforehand for a predecessor state, but not stored. The magnitude of this penalty is highly dependent on the structure of the model. In practice, however, we expect small values for k, which limits the penalty, and this is evidenced in our case studies (see Section 4.7.3).

The on-the-fly approach naturally works for infinite-state systems, both in terms of control and data. In particular, the kind of behaviour that Condition A3 is designed to detect—the case of a certain choice being continuously available, but also continuously discarded—can, in an infinite system, also come in via infinite-state "end components". Since A3' replaces the notion of end components by the notion of sufficiently long sequences of states, this is no problem.

[6] Our implementation in modes therefore uses large default values for k_{max} and l so the user usually need not worry about these parameters. If SMC aborts, the cause and its location is reported, including how it was detected, which may be that k_{max} or l was exceeded.

Applicability and Limitations

Although partial order reduction has led to drastic state space reductions for some models in exhaustive model checking, it is only an approximation: Whenever transitions are removed, they are indeed spurious nondeterministic alternatives, but not all spurious choices may be detected as such. In particular, when using feasibly checkable independence conditions like J1 and J2, only spurious interleavings can be reduced. These restrictions directly carry over to our use of POR for statistical model checking. Worse yet, while not being able to reduce a certain single choice during exhaustive model checking leads to the same verification results at the cost of higher memory usage and runtime, the same would make an SMC analysis abort. More important than any performance consideration is therefore whether the approach is applicable at all to realistic models. We investigate this question in detail using a set of case studies in Section 4.7.3 and content ourselves with a look at the shared medium communication example introduced earlier for now:

Example 23. We already saw that all nondeterminism in the different networks of VMDP modelling the sending of a message over a shared medium presented in Example 22 is spurious. However, for which of them would the on-the-fly POR check work?

First, it clearly cannot work for any of the $N^{\cdot}_{\text{sync}}$ networks that contain the M_{sync} process: The nondeterministic choice between $\texttt{snd}_1$ and $\texttt{snd}_2$ that occurs when both hosts want to send the message is internal to M_{sync} and not a spurious interleaving. The transitions labelled $\texttt{snd}_1$ and $\texttt{snd}_2$ would thus be marked as dependent by the function `dependent`, since they do not satisfy Condition J1.

On the other hand, the nondeterministic choices in both N^{τ}_{var} and $N^{\texttt{act}}_{\text{var}}$ pose no problem. Let us use N^{τ}_{var} for illustration: its concrete MDP semantics, which all SMC methods except for the two checks `equivalent` and `dependent` work on, is shown in Figure 4.11. The nondeterministic states are the initial state $s_{init} = \langle h_1, h_1, v_1, 0\rangle$ (composed of the initial states of the three component VMDP plus the current value of j), the two symmetric states $\langle h_2, h_1, v_1, 0\rangle$ and $\langle h_1, h_2, v_1, 0\rangle$, and finally $\langle h_2, h_2, v_1, 0\rangle$. For brevity, we write h_{ij} for state $\langle h_i, h_j, v_1, 0\rangle$. The model contains no cycles and all paths have length at most 5, so the cycle condition A3' is no problem for e.g. $l = 5$.

Let us focus on the initial state for this example. The nondeterministic choice here is between the initial τ-labelled edges of the two hosts. Let $\{ tr_1^{\tau} = s_{init} \xrightarrow{\tau} \{ h_{21} \mapsto 0.5, h_{31} \mapsto 0.5 \} \}$ be the candidate ample set selected first by `resolvePOR`, i.e. it contains the initial τ-labelled transition of the first host. tr_1^{τ} is obviously invisible as only the transitions labelled `tick` lead to changes in state labelling. Thus `chkAmpleSet` calls `chkPaths(`s_{init}`,` tr_1^{τ}`, {`s_{init}`}, 0)` to

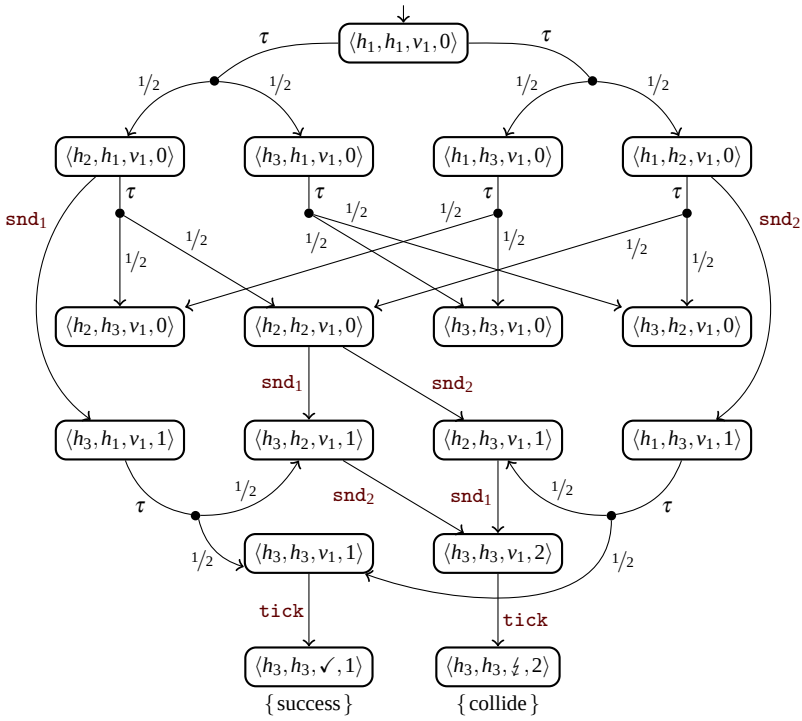


Figure 4.11: MDP semantics of the network of VMDP N^{τ}_{var}

verify Condition A2'. It is trivially satisfied for all paths starting with tr_1^τ itself because that *is* the single transition in the ample set. For the paths starting with $tr_2^\tau = s_{init} \xrightarrow{\tau} \{h_{12} \mapsto 0.5, h_{13} \mapsto 0.5\}$, i.e. the case that the second host performs its initial τ first, `chkPaths` returns *true* for successor state h_{13}: It has only one outgoing transition, namely the initial τ of the first host, which is thus equivalent to the ample set transition tr_1^τ. In successor state h_{12}, however, we have another nondeterministic choice. The τ-labelled alternative is again equivalent to tr_1^τ, but the transition labelled `snd2` is neither equivalent to nor dependent on the ample set (it modifies global variable *i*, but that has no influence on tr_1^τ). We thus need another recursive call to `chkPaths`. In the following state $\langle h_1, h_3, v_1, 1\rangle$, we can return *true* as the only outgoing transition is finally the first host's τ that is in $[tr_1^\tau]_{\equiv_E}$.

The choices in the other nondeterministic states can similarly be resolved successfully. In state h_{22}, the choice is between two sending transitions which

consequently both modify global variable *i*, but their assignments just increment *i* and are thus commutative.

To summarise our observations: For large enough *k* and *l*, this POR-based approach will allow us to use SMC for networks of VMDP where the nondeterminism introduced by the parallel composition is spurious. Nevertheless, if conditions J1 and J2 are used to check for independence instead of I1 and I2, nondeterministic choices *internal to the component VMDP*, if present, must already be removed while obtaining the MDP semantics (i.e. by synchronization via actions or shared variables). While avoiding internal nondeterminism is manageable during the modelling phase, parallel composition and the nondeterminism it creates naturally occur in models of distributed or component-based systems. We thus expect this approach to be readily applicable in practice: the modeller needs to take care to specify deterministic components while the nondeterminism from interleaving can usually be handled by the POR check—as long as it is spurious, of course. Usually, a non-spurious interleaving represents a race condition in the model and thus, if the model is useful, in the underlying system. Race conditions are undesirable artifacts of concurrency in most cases, so aborting simulation and alerting the user to the potential presence of a race condition as done in this approach to SMC appears a useful course of action, and is clearly more desirable than hiding the potential error by using an implicit resolver instead.

4.7.2 Using Confluence Reduction

An alternative to POR in exhaustive model checking is confluence reduction. It, too, was originally defined for LTS [BvdP02, GvdP00] and has been generalised to probabilistic systems [HT14a, TSvdP11, TvdPS13]. The goal is the same: generating smaller, but equivalent, state spaces. In the probabilistic generalisation, confluence reduction preserves $\mathrm{PCTL}^*_{\setminus \mathrm{X}}$, i.e. branching-time properties. However, as we will see, the way confluence is defined is very different from the ample set conditions of POR.

Confluence reduction has recently been shown theoretically to be more powerful than the variant of partial order reduction that preserves branching-time properties [HT14a]. We have already pointed out that it is absolutely vital for the search for a valid singleton subset to succeed when a nondeterministic choice is encountered during simulation: One choice that cannot be resolved means that the entire analysis fails and SMC cannot safely be applied to the given model at all. Therefore, any additional reduction power is highly welcome. Although we used the more liberal variant of POR that preserves only linear-time properties in the previous approach and its relation to confluence

is unknown, this theoretical difference is still a clear motivation to investigate whether confluence reduction can be used for SMC in place of POR as described in the previous section.

Furthermore, in practice, confluence reduction is easily implemented on the MDP level, i.e. the concrete state space alone, without any need to go back to the symbolic/syntactic VMDP level for an independence check based on conditions like J1 and J2. It thus allows even spurious nondeterminism that is internal to components to be ignored during simulation, lifting the restriction to spurious interleavings of the POR implementation.

Confluence Reduction for MDP

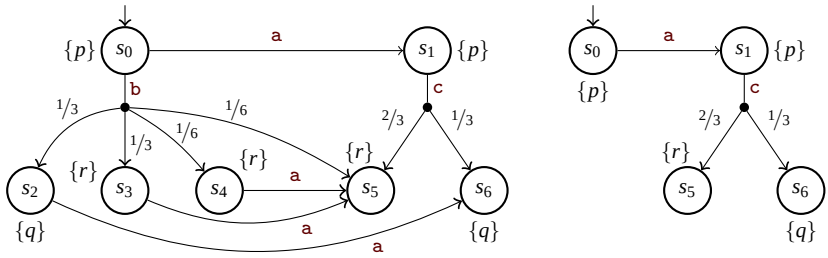
Confluence reduction is based on commutativity of invisible transitions. It works by denoting a subset of the invisible transitions of an MDP as *confluent*. Basically, this means that they do not change the observable behaviour; everything that is possible before a confluent transition is still possible afterwards. Therefore, they can be given *priority*, omitting all their neighbouring transitions.

Previous work defined conditions for a set of transitions to be confluent. In the nonprobabilistic action-based setting, several variants were introduced, ranging from ultra weak confluence to strong confluence [Blo01]. They are all given diagrammatically, and define in which way two outgoing transitions from the same state have to be able to join again. Basically, for a transition $s \xrightarrow{\tau} t$ to be confluent, every transition $s \xrightarrow{a} u$ has to be mimicked by a transition $t \xrightarrow{a} v$ such that u and v are bisimilar. This is ensured by requiring a confluent transition from u to v.

To extend confluence to the probabilistic action-based setting, a similar approach has been taken [TSvdP11]. For a transition $s \xrightarrow{\tau} \mathscr{D}(t)$ to be confluent, every transition $s \xrightarrow{a} \mu$ has to be mimicked by a transition $t \xrightarrow{a} \nu$ such that μ and ν are equivalent; as usual in probabilistic model checking, this means that they should assign the same probability to each *equivalence class* of the state space in the bisimulation quotient. Bisimulation is again ensured using confluent transitions.

In this thesis, we are dealing with a state-based context: only the atomic propositions that are assigned to each state are of interest. Therefore, we base our definition of confluence on the state-based probabilistic notions given in [HT14a]. It is still parameterised in the way that distributions have to be connected by confluent transitions, denoted by $\mu \rightsquigarrow_{\mathscr{T}} \nu$:

Definition 56 (Equivalence up to $\mathscr{T}$-steps)**.** Let $M = \langle S, A, T, s_{init}, AP, L \rangle$ be an MDP, $\mathscr{T}$ a set of nonprobabilistic transitions of M, and $\mu, \nu \in \mathrm{Dist}(S)$ two



(a) Original system (b) Reduced system

Figure 4.12: An MDP to demonstrate confluence reduction

probability distributions. Let R be the smallest equivalence relation containing the set

$$R' = \{\, \langle s,t \rangle \mid s \in \text{support}(\mu) \wedge t \in \text{support}(\nu) \wedge \exists a \in A\colon s \xrightarrow{a} \mathscr{D}(t) \in \mathscr{T} \,\}.$$

Then, μ and ν are *equivalent up to $\mathscr{T}$-steps*, denoted by $\mu \rightsquigarrow_{\mathscr{T}} \nu$, if $\langle \mu, \nu \rangle \in R$.

While this definition is abstract w.r.t. the set of transitions $\mathscr{T}$, we only use $\mathscr{T}$ for sets of confluent transitions. Confluent transitions are used to detect equivalent states. Hence, when $\mathscr{T}$ is a set of confluent transitions, the definition means that two distributions are equivalent if they assign the same probabilities to sets of states that are connected by confluent transitions. For ease of detection, we only take into account confluent transitions from the support of μ to the support of ν. In principle, larger equivalence classes could be used when also considering transitions in the other direction and chains of confluent transitions. However, for efficiency reasons we chose not to be so liberal.

Example 24. As an example of Definition 56, consider Figure 4.12a. Let $\mathscr{T}$ be the set consisting of all a-labelled transitions. Note that these transitions indeed are all nonprobabilistic. We denote by μ the probability distribution associated with the b-transition from s_0, and by ν the one associated with the c-transition from s_1. We find $R' = \{\, \langle s_2, s_6 \rangle, \langle s_3, s_5 \rangle, \langle s_4, s_5 \rangle \,\}$, and so

$$R = Id \cup \{\, \langle s_2, s_6 \rangle, \langle s_6, s_2 \rangle, \langle s_3, s_4 \rangle, \langle s_4, s_3 \rangle, \langle s_3, s_5 \rangle, \langle s_5, s_3 \rangle, \langle s_4, s_5 \rangle, \langle s_5, s_4 \rangle \,\}.$$

Hence, R partitions the state space into $\{\, s_0 \,\}$, $\{\, s_1 \,\}$, $\{\, s_2, s_6 \,\}$ and $\{\, s_3, s_4, s_5 \,\}$. For probabilities we have $\mu(\{\, s_0 \,\}) = \nu(\{\, s_0 \,\}) = 0$, $\mu(\{\, s_1 \,\}) = \nu(\{\, s_1 \,\}) = 0$, $\mu(\{\, s_2, s_6 \,\}) = \nu(\{\, s_2, s_6 \,\}) = \frac{1}{3}$ and $\mu(\{\, s_3, s_4, s_5 \,\}) = \nu(\{\, s_3, s_4, s_5 \,\}) = \frac{2}{3}$. Consequently, $\langle \mu, \nu \rangle \in R$ and thus $\mu \rightsquigarrow_{\mathscr{T}} \nu$.

We can now formally define the notion of a set of confluent transitions, which is at the core of the confluence reduction technique:

Definition 57 (Probabilistic confluence)**.** Let $M = \langle S, A, T, s_{init}, AP, L\rangle$ be an MDP. Then a subset $\mathscr{T}$ of the transitions of M is *probabilistically confluent* if it only contains invisible nonprobabilistic transitions, and

$$\forall s \xrightarrow{a} \mathscr{D}(t) \in \mathscr{T} : \forall s \xrightarrow{b} \mu \in T(s) : \mu = \mathscr{D}(t) \vee \exists t \xrightarrow{c} \nu \in T(t) : \mu \rightsquigarrow_{\mathscr{T}} \nu$$

Additionally, if $s \xrightarrow{b} \mu \in \mathscr{T}$, then so should $t \xrightarrow{c} \nu$ be. A transition is *probabilistically confluent* if there exists a probabilistically confluent set that contains it.

Example 25. Let $\mathscr{T}$ be the set of all a-labelled transitions of the MDP shown in Figure 4.12a as introduced in the previous example. $\mathscr{T}$ is a valid confluent set according to Definition 57. First, all its transitions are indeed invisible and nonprobabilistic. Second, for the a-transitions from s_2, s_3 and s_4, nothing interesting has to be checked. After all, from their source states there are no other outgoing transitions, and every transition satisfies the condition $\mu = \mathscr{D}(t) \vee \exists t \xrightarrow{c} \nu \in T(t) : \mu \rightsquigarrow_{\mathscr{T}} \nu$ for itself due to the clause $\mu = \mathscr{D}(t)$. For $s_0 \xrightarrow{a} \mathscr{D}(s_1)$, we do need to check if the condition holds for $s_0 \xrightarrow{b} \mu$. There is a mimicking transition $s_1 \xrightarrow{c} \nu$, and as we saw above $\mu \rightsquigarrow_{\mathscr{T}} \nu$ as required.

We now define confluence reduction functions. Such a function always chooses to either fully explore a state, or only explore one outgoing confluent transition.

Definition 58 (Confluence reduction)**.** Given an MDP $M = \langle S, A, T, s_{init}, AP, L\rangle$, a reduction function f is a *confluence reduction function for M* if there exists some confluent set $\mathscr{T}$ of transitions of M for which, for every $s \in S$ such that $f(s) \neq T(s)$, it holds that $f(s) = \{\langle a, \mathscr{D}(t)\rangle\}$ for some $a \in A$ and $t \in S$ such that $s \xrightarrow{a} \mathscr{D}(t) \in \mathscr{T}$. In such a case, we also say that f is a *confluence reduction function under* $\mathscr{T}$.

Confluent transitions might be retaken indefinitely, thereby ignoring the presence of other actions. This is the well-known *ignoring problem* [EP10]. In the ample set method of POR, it is dealt with by the cycle condition. We can just as easily deal with it in the context of confluence reduction by requiring the reduction function to be acyclic. Acyclicity can be checked during SMC in the same way as was done for POR in the previous section: always check whether in the last l steps at least one state was fully expanded (i.e. the state already had only one outgoing transition).

Example 26. In the system of Figure 4.12a, we already saw that the set of all a-labelled transitions is a valid confluent set. Based on this set, we can define

the reduction function f given by $f(s_0) = \{\langle \mathtt{a}, \mathscr{D}(s_1)\rangle\}$ and $f(s) = T(s)$ for every other state s. That way, the reduced system is given by Figure 4.12b. Note that the two models indeed share the same properties, such as that the (minimum and maximum) probability of eventually observing r is $\frac{2}{3}$.

Confluence reduction preserves $\mathrm{PCTL}^*_{\setminus \mathrm{X}}$, and hence basically all interesting quantitative properties, including $\mathrm{LTL}_{\setminus \mathrm{X}}$, which was preserved by partial order reduction as presented in the previous section, and of course probabilistic reachability.

Theorem 4. Let M be an MDP, $\mathscr{T}$ a confluent set of its transitions and f an acyclic confluence reduction function under $\mathscr{T}$. Then, M and $\mathrm{red}(M, f)$ satisfy the same $\mathrm{PCTL}^*_{\setminus \mathrm{X}}$ formulas.

Proof. By minor adjustment of the proofs of Corollary 26 of [HT14a], which precisely corresponds to this theorem, as discussed below. □

Compared to [HT14a], our definition of equivalence up to $\mathscr{T}$-steps (Definition 56) is slightly more liberal. In [HT14a], the number of states in the support of μ was required to be at least as large as the number of states in the support of ν, since no nondeterministic choice between equally-labelled transitions was allowed. Since we do allow this, we take the more liberal approach of just requiring the probability distributions to assign the same probabilities to the same classes of states with respect to confluent connectivity. The correctness arguments are not influenced by this, as the reasoning that confluent transitions connect bisimilar states does not break down if these support sets are potentially more distinct.

Our Definition 57 is also more liberal in two aspects. First, not necessarily $b = c$. In [HT14a], this was needed to preserve probabilistic visible bisimulation. Equivalent systems according to that notion preserve state-based as well as action-based properties. However, in our setting the actions are only for synchronisation of parallel components, and have no purpose anymore in the final model: we only need to consider closed systems. Therefore, we can just as well rename all actions to a single one. Then, if a transition is mimicked, the action would be the same by construction. We thus simply chose to omit the required accordance of action names altogether.

Second, we only require confluent transitions to be invisible and nonprobabilistic themselves. In [HT14a], all transitions with the same label had to be so as well (for a fairer comparison with POR). Here, this is not an option, since during simulation we only know part of the state space. However, it is also not needed for correctness, as a local argument about mimicking behaviour until some joining point can clearly never be broken by transitions after this point.

With these differences in mind, let us now describe in detail how to construct a proof of Theorem 4 based on the proof of the corresponding Corollary 26 as presented in [HT14a][7]. That corollary is based on Theorem 25 of that paper, which states that $M \sim_{\text{pvb}} \text{red}(M, f)$. Although those results were for MDPs where each state can have only one outgoing transition for each action label, this property is not used in any of the proofs. Hence, the results apply just as well for our type of MDPs. Additionally, we allow countable state spaces, while in [HT14a] finiteness was assumed. However, as we only consider finite subparts of an MDP during simulation, this also does not matter. More importantly, the results are for the old definitions of confluent sets and equivalence of distributions. Hence, we discuss to what extent the results still hold for our adapted definitions.

We first discuss the influence of our new definition of equivalence of distributions (Definition 56). It appears that this change does not influence the correctness of the old results in any way. To see why, note that the definition of equivalence is only used in [HT14a] in Lemma 22, Lemma 24 and Theorem 25. In Lemma 22 and Theorem 25, the definition of equivalence is used to show that $\mathscr{D}(s) \rightsquigarrow_{\mathscr{T}} \mathscr{D}(t)$ implies that either $s = t$ or there is a transition from s to t in $\mathscr{T}$. This also still directly follows from our Definition 56. After all, $\langle \mathscr{D}(s), \mathscr{D}(t) \rangle \in R$ holds only if s and t are in the same equivalence class of R. This is indeed only the case if either $s = t$ or if there is a $\mathscr{T}$-transition from s to t (since support(μ) and support(ν) are singletons, no transitivity can be involved). In Lemma 24 it is shown that $\mu \rightsquigarrow_{\mathscr{T}} \nu$ implies $\langle \mu, \nu \rangle \in R$ for the set R that relates all states that can join while only following confluent transitions. Since that set R can easily be seen to be a superset of the set R from Definition 56 if $\mathscr{T}$ is a confluent set (using Lemma 22 of [HT14a]), the result still holds by Proposition 5.2.1.5 from [Sto02].

The second change we made was to use a more liberal version of the notion of confluent sets (Definition 57). Although technically probabilistic visible bisimulation is not preserved anymore under this new definition, the bisimulation notion could be altered to also just require invisible transitions instead of invisible actions, and also allow transitions to be mimicked by transitions with a different action. As discussed above, this would not change anything to the fact that $\text{PCTL}^*_{\setminus \text{X}}$ properties are preserved.

Hence, the old proofs from [HT14a] can be used practically unchanged to show that our new definitions preserve the adjusted variant of probabilistic visible bisimulation, and thus that indeed a reduced system based on confluence satisfies the same $\text{PCTL}^*_{\setminus \text{X}}$ formulas as the original system.

[7] [HT14a] was already submitted for publication at the time of writing of [HT13], which is where this proof description was first published.

Finally, let us mention notable differences between confluence reduction as defined here and POR [BDG06]. Confluence also allows mimicking by differently-labelled transitions, commutativity in triangles instead of diamonds, and local instead of global independence [HT14a]. Additionally, its coinductive definition is well-suited for on-the-fly detection, as we show in the next subsection. However, since confluence preserves branching-time properties, it cannot reduce when an interleaving involves a probabilistic transition, a scenario that can be handled by the original linear-time notions of probabilistic POR defined in [BGC04, DN04] and used in our POR-based simulation approach.

On-the-fly Confluence Checking

The first approach to detect non-probabilistic confluence worked directly on concrete state spaces to reduce them modulo branching bisimulation [GvdP00]. Although the complexity was linear in the size of the state space, the method was not very useful: it required the complete unreduced state space to be available, which might already be too large to generate. Therefore, two directions of improvements were pursued.

The first idea was to detect confluence on higher-level process-algebraic system descriptions [Blo01, BvdP02]. Using this information from the symbolic level, the reduced state space could be generated directly without first constructing any part of the original one. More recently, this technique was generalised to the probabilistic setting [TSvdP11]. The other direction was to use the ideas from [GvdP00] to on-the-fly detect non-probabilistic weak or strong confluence [MW12, PLM03] during state space generation. These techniques are based on Boolean equation systems and have not yet been generalised to the probabilistic setting.

We present a novel on-the-fly method, shown as algorithms 19 and 20, that works on concrete probabilistic state spaces and does not require the unreduced state space, making it perfectly applicable during simulation for statistical model checking of MDP models.

Algorithm The functions that implement our approach are presented in algorithms 19 and 20. Given $s \xrightarrow{a} \lambda$, the call `chkConfluence(`$s \xrightarrow{a} \lambda$`, ...)` tells us whether or not this transition is confluent. We first discuss this function, and then `chkEquivalence` on which it relies (which determines whether or not two distributions are equivalent up to confluent steps). These functions do not yet fully take into account the fact that confluent transitions have to be mimicked by confluent transitions. Therefore, we have an additional function called

```
1  function resolveConfluence(s)
2      foreach tr ∈ T(s) in fixed global order do
3          𝒯 := ∅, C := ∅
4          if chkConfluence(tr, ref 𝒯, ref C) then
5              if chkConfluentMimicking(𝒯, C) then return tr
6          end
7      end
8      return ⊥

9  function chkConfluence(s -a-> λ, ref 𝒯, ref C)
10     if s -a-> λ ∈ 𝒯 then return true
11     if ∄t: λ = 𝒟(t) ∨ L(s) ≠ L(t) then return false
12     𝒯_old := 𝒯, C_old := C, 𝒯 := 𝒯 ∪ { s -a-> 𝒟(t) }
13     foreach tr_sμ = s -b-> μ ∈ T(s) do
14         if μ = 𝒟(t) then continue
15         foreach tr_tν = t -c-> ν ∈ T(t) do
16             if ¬ chkEquivalence(μ, ν, ref 𝒯, ref C) then continue
17             if tr_sμ ∉ 𝒯 ∨ chkConfluence(tr_tν, ref 𝒯, ref C) then
18                 C := C ∪ { ⟨tr_sμ, tr_tν⟩ }
19                 continue outer loop        // found a matching transition
20             end
21         end
22         𝒯 := 𝒯_old, C := C_old    // restore sets: s -a-> 𝒟(t) is not confluent
23         return false
24     end
25     return true

26 function chkEquivalence(μ, ν, ref 𝒯, ref C)
27     Q := { {p} | p ∈ support(μ) ∪ support(ν) }        // initial partitions
28     foreach u -d-> 𝒟(ν) such that u ∈ support(μ), ν ∈ support(ν) do
29         if chkConfluence(u -d-> 𝒟(ν), ref 𝒯, ref C) then
30             Q := { q ∈ Q | u ∉ q ∧ ν ∉ q } ∪ { ∪_{q∈Q: u∈q∨ν∈q} q }
31         end
32     end
33     return ∀q ∈ Q: μ(q) = ν(q)    // check that the probabilities match
```

Algorithm 19: Detecting confluence on a concrete state space (1)

```
1 function chkConfluentMimicking(𝒯, C)
2     foreach ⟨s -b-> μ, t -c-> ν⟩ ∈ C do
3         if s -b-> μ ∈ 𝒯 ∧ t -c-> ν ∉ 𝒯 then
4             if ¬chkConfluence(t -c-> ν, ref 𝒯, ref C) then return false
5             return chkConfluentMimicking(𝒯, C)          // restart
6         end
7     end
8     return true
```

Algorithm 20: Detecting confluence on a concrete state space (2)

`chkConfluentMimicking` that is called after `chkConfluence` terminates to see if indeed no violations of this condition occur.

`chkConfluence` first checks if the transition $s \xrightarrow{a} \lambda$ was already detected to be confluent before (line 10). If it was not, we check whether the transition is nonprobabilistic, i.e. $\lambda = \mathscr{D}(t)$ for some state t, and invisible (line 11). Then, it is added to the global set $\mathscr{T}$ of confluent transitions (line 12). To determine whether this is valid, the loop beginning in line 13 checks if indeed all outgoing transitions from s commute with $s \xrightarrow{a} \mathscr{D}(t)$. If so, we return *true* (line 25) and keep the transition in $\mathscr{T}$. Otherwise, all transitions that were added to $\mathscr{T}$ during these checks are removed again and we return *false* (lines 22 and 23). Note that it would not be sufficient to only remove $s \xrightarrow{a} \mathscr{D}(t)$ from $\mathscr{T}$, since during the loop some transitions might have been detected to be confluent (and hence added to $\mathscr{T}$) based on the fact that $s \xrightarrow{a} \mathscr{D}(t)$ was in $\mathscr{T}$. As $s \xrightarrow{a} \mathscr{D}(t)$ turned out not to be confluent, we can also not be sure anymore whether these other transitions are indeed confluent.

The loop to check whether all outgoing transitions commute with s follows directly from the definition of confluent sets, which requires for every $s \xrightarrow{b} \mu$ that either $\mu = \mathscr{D}(t)$, or that there exists a transition $t \xrightarrow{c} \nu$ such that $\mu \rightsquigarrow_{\mathscr{T}} \nu$, where $t \xrightarrow{c} \nu$ has to be in $\mathscr{T}$ if $s \xrightarrow{b} \mu$ is. Indeed, if $\mu = \mathscr{D}(t)$ we immediately continue to the next transition (line 14; this includes the case that $s \xrightarrow{b} \mu = s \xrightarrow{a} \mathscr{D}(t)$). Otherwise, we range over all transitions $t \xrightarrow{c} \nu$ to see if there is one such that $\mu \rightsquigarrow_{\mathscr{T}} \nu$. For this, we use the function `chkEquivalence` in line 16, which is described below. Also, if $s \xrightarrow{b} \mu \in \mathscr{T}$, we have to check that also $t \xrightarrow{c} \nu \in \mathscr{T}$. We do this by checking it for confluence, which immediately returns if it is already in $\mathscr{T}$, and otherwise tries to add it.

If indeed we find a mimicking transition, we continue (line 19). If $s \xrightarrow{b} \mu$ cannot be mimicked, confluence of $s \xrightarrow{a} \mathscr{D}(t)$ cannot be established. Hence, we reset $\mathscr{T}$ as discussed above, and return *false*. If this did not happen for any of

the outgoing transitions of s, then $s \xrightarrow{a} \mathscr{D}(t)$ is indeed confluent and we return *true*.

`chkEquivalence` checks whether $\mu \rightsquigarrow_{\mathscr{T}} \nu$. Since $\mathscr{T}$ is constructed on-the-fly, during this check some of the transitions from the support of μ might not have been detected to be confluent yet, even though they are. Therefore, instead of checking for connecting transitions that are already in $\mathscr{T}$, we try to add possible connecting transitions to $\mathscr{T}$ using a call back to `chkConfluence` (line 29). The two functions are thus mutually recursive.

In accordance with Definition 56, we first determine the smallest equivalence relation that relates states from the support of μ to states from the support of ν in case there is a confluent transition connecting them. We do so by constructing a set of equivalence classes Q, i.e. a partitioning of the state space according to this equivalence relation. We start with the smallest possible equivalence relation in line 27, in which each equivalence class is a singleton. Then, for each confluent transition $u \xrightarrow{d} \mathscr{D}(v)$, with $u \in \text{support}(\mu)$ and $v \in \text{support}(\nu)$, we merge the equivalence classes containing u and v (line 30). Finally, we can easily compute the probability of reaching each equivalence class of Q by either μ or ν. If all of these probabilities coincide, indeed $\langle \mu, \nu \rangle \in R$ and we return *true*; otherwise, we return *false* (line 33).

`chkConfluentMimicking` is called after the main call to `chkConfluence` designated a transition to be confluent, to verify that $\mathscr{T}$ satisfies the requirement that confluent transitions are mimicked by confluent transitions. After all, when a mimicking transition for some transition $s \xrightarrow{b} \mu$ was found, it might have been the case that $s \xrightarrow{b} \mu$ was not yet in $\mathscr{T}$ while in the end it is. Hence, `chkConfluence` keeps track of the mimicking transitions in a global set C. If a transition $s \xrightarrow{a} \mathscr{D}(t)$ is shown to be confluent, all pairs $\langle s \xrightarrow{b} \mu, t \xrightarrow{c} \nu \rangle$ of other outgoing transitions from s and the transitions that were found to mimic them from t are added to C. This happens in line 18 in `chkConfluence`. If $s \xrightarrow{a} \mathscr{D}(t)$ turns out not to be confluent after all, the mimicking transitions that were found in the process are removed again (line 22).

Based on the set of pairs C, `chkConfluentMimicking` ranges over all its elements $\langle s \xrightarrow{b} \mu, t \xrightarrow{c} \nu \rangle$, checking if one violates the requirement. If no such pair is found, we return *true*. Otherwise, the current set $\mathscr{T}$ is not valid yet. However, it could be the case that $t \xrightarrow{c} \nu$ is not in $\mathscr{T}$, while it is confluent (but since $s \xrightarrow{b} \mu$ was not in $\mathscr{T}$ at the moment the pair was added to C, this was not checked earlier). Therefore, we still try to denote $t \xrightarrow{c} \nu$ as confluent. If we fail, we return *false* (line 4). Otherwise, in line 5, we check again for confluent mimicking using the new sets $\mathscr{T}$ and C.

Correctness We will now show that, together, the functions `chkConfluence` and `chkConfluentMimicking` correctly identify confluent transitions. For this proof, we first need the following lemma:

Lemma 3. Given two distributions μ and ν, an initial set of confluent set transitions $\mathscr{T}$, and some set of pairs of transitions C,

$$\texttt{chkEquivalence}(\mu,\, \nu, \text{ref } \mathscr{T}, \text{ref } C) \Rightarrow \mu \rightsquigarrow_{\mathscr{T}} \nu$$

where the rightmost occurrence of $\mathscr{T}$ refers to the updated state of the set of confluent transitions upon termination of the call to `chkEquivalence`.

Proof ([HT13]). First of all, note that $\mathscr{T}$ only *grows* during `chkEquivalence`: each call to `chkConfluence` might add transitions to it or leave it unchanged.

Assume that `chkEquivalence`$(\mu,\, \nu,\, \ldots)$ yields *true*. Hence, we have $\mu(q) = \nu(q)$ for every $q \in Q$, using the set Q after the loop. Note that Q is a partitioning of the set $\text{support}(\mu) \cup \text{support}(\nu)$, since initially it contains all singletons, and the loop only merges some of its elements. Now let

$$Q' = Q \cup \{\, \{\, q \,\} \mid q \notin \text{support}(\mu) \cup \text{support}(\nu) \,\}$$

be a partitioning of the complete set of states S. We also have $\mu(q) = \nu(q)$ for every $q \in Q'$, as both μ and ν assign probability 0 to all newly added classes. Let Q'' be the equivalence relation associated with Q', i.e. $\langle s,t \rangle \in Q''$ if and only if there is a set $q' \in Q'$ such that $s,t \in q'$. Since the function returns *true*, by definition we have $\langle \mu, \nu \rangle \in Q''$.

It remains to show that $Q'' \subseteq R$; by Proposition 5.2.1.5 of [Sto02], then indeed $\langle \mu, \nu \rangle \in R$. Recall that R is the smallest equivalence relation containing the set

$$R' = \{\, \langle s,t \rangle \mid s \in \text{support}(\mu), t \in \text{support}(\nu), \exists a \in A\colon s \xrightarrow{a} t \in \mathscr{T} \,\}$$

where we chose $\mathscr{T}$ to be the set at termination of `chkEquivalence`. Hence, $\langle s,t \rangle \in R$ if and only if $s = t$ or there are states $s_0, s_1, \ldots, s_n$ such that $s_0 = s$, $s_n = t$ and either $\langle s_i, s_{i+1} \rangle \in R'$ or $\langle s_{i+1}, s_i \rangle \in R'$ for every $0 \leq i < n$.

So, let $\langle s,t \rangle \in Q''$. We show that also $\langle s,t \rangle \in R$. If $s = t$, this is immediate, so assume that $s \neq t$. By construction, there is a set $q' \in Q$ such that $s,t \in q'$. For s and t to be in the same set, some merges must have taken place in the loop.

If $s \in \text{support}(\mu)$, $s_1 \in \text{support}(\nu)$ and $s \xrightarrow{a} s_1 \in \mathscr{T}$ (at some point, so since $\mathscr{T}$ only grows also at the end), then $\{\, s \,\}$ and $\{\, s_1 \,\}$ are merged. Hence, this corresponds to $\langle s, s_1 \rangle \in R'$. Alternatively, the same merge also happens if $s \in \text{support}(\nu)$, $s_1 \in \text{support}(\mu)$ and $s_1 \xrightarrow{a} s \in \mathscr{T}$, hence, $\langle s_1, s \rangle \in R'$. The set $\{\, s, s_1 \,\}$ can grow further in the same way, until it at some point contains t. This procedure corresponds exactly to the requirement that $\langle s,t \rangle \in R$.

(In this proof we used $s \xrightarrow{a} \mu \in \mathscr{T}$ and `chkConfluence(`$s \xrightarrow{a} \mu$, ref $\mathscr{T}$, ...`)` interchangeably; after all, if `chkConfluence(`$s \xrightarrow{a} \mu$, ref $\mathscr{T}$, ...`)` returns *true* then indeed also $s \xrightarrow{a} \mu \in \mathscr{T}$, and if $s \xrightarrow{a} \mu \in \mathscr{T}$ then `chkConfluence(`$s \xrightarrow{a} \mu$, ref $\mathscr{T}$, ...`)` returns *true*.) □

We can now state and prove the correctness of the combination of the functions `chkConfluence` and `chkConfluentMimicking`:

Theorem 5 (Correct detection of confluent transitions)**.** Given a transition $p \xrightarrow{l} \lambda$ and using initially empty sets $\mathscr{T}$ and C, if

`chkConfluence(`$p \xrightarrow{l} \lambda$, ref $\mathscr{T}$, ref C`)`

returns *true* as well as subsequently `chkConfluentMimicking(`$\mathscr{T}$, C`)`, this together implies that $p \xrightarrow{l} \lambda$ is probabilistically confluent.

Proof ([HT13]). `chkConfluence` immediately checks whether $p \xrightarrow{l} \lambda$ is deterministic, i.e. whether it is of the form $p \xrightarrow{l} \mathscr{D}(q)$. Then, we need to show that there exists a confluent set of transitions containing $p \xrightarrow{l} \mathscr{D}(q)$. We show that, upon termination of the algorithm and returning *true*, the set $\mathscr{T}$ fulfills this condition. Clearly, $p \xrightarrow{l} \mathscr{D}(q) \in \mathscr{T}$, since it is always added immediately at the beginning of `chkConfluence` (except in case that *false* is returned due to it being visible), and only removed before returning *false*. Since we assumed that *true* is returned, indeed $p \xrightarrow{l} \mathscr{D}(q) \in \mathscr{T}$.

To show that $\mathscr{T}$ is a confluent set, let $s \xrightarrow{a} \mathscr{D}(t) \in \mathscr{T}$ be an arbitrary element. Note that indeed any element of $\mathscr{T}$ is nonprobabilistic, since for probabilistic transitions *false* is returned right at the beginning of `chkConfluence` before any modification of $\mathscr{T}$ is done. We have to prove that $s \xrightarrow{a} \mathscr{D}(t)$ is invisible and that, for every $s \xrightarrow{b} \mu$ we have either $\mu = \mathscr{D}(t)$ or there exists a transition $t \xrightarrow{c} \nu$ such that $\mu \rightsquigarrow_{\mathscr{T}} \nu$. Also, we need to show that $t \xrightarrow{c} \nu$ is in $\mathscr{T}$ if $s \xrightarrow{b} \mu$ is. We postpone this last part to the end.

Since $s \xrightarrow{a} \mathscr{D}(t) \in \mathscr{T}$, `chkConfluence(`$s \xrightarrow{a} \mathscr{D}(t)$, ref $\mathscr{T}$, ...`)` must have been called at some point, $s \xrightarrow{a} \mathscr{D}(t)$ was added to $\mathscr{T}$ and subsequently not removed. This implies that $L(s) = L(t)$ (and hence indeed the transition is invisible) and that the algorithm terminated with the final **return** *true* statement. Hence, the outermost **foreach** loop never reached the end of its body, but was always cut short before by a **continue** statement. So, for each $s \xrightarrow{b} \mu$ it holds that either $\mu = \mathscr{D}(t)$ or there exists a transition $t \xrightarrow{c} \nu$ for which the second **foreach** loop reached its **continue** statement. In the second case, the call `chkEquivalence(`μ, ν, ref $\mathscr{T}$, ...`)` yielded *true*, and by Lemma 3, this implies that $\mu \rightsquigarrow_{\mathscr{T}} \nu$ was true at the end of each iteration of the loop. Since $\mathscr{T}$ can only grow during the loop, and also afterwards no transitions are removed

from $\mathscr{T}$ anymore (because otherwise $p \xrightarrow{l} \mathscr{D}(q)$ would have been removed too), the set $\mathscr{T}$ at the end of the algorithm is a superset of the set $\mathscr{T}$ at the moment that $\mu \rightsquigarrow_{\mathscr{T}} \nu$ was established. Hence, we also have $\mu \rightsquigarrow_{\mathscr{T}} \nu$ for the final $\mathscr{T}$ (based on Proposition 5.2.1.5 of [Sto02]), as required.

Finally, we show that if $s \xrightarrow{b} \mu$ is mimicked by $t \xrightarrow{c} \nu$ and $s \xrightarrow{b} \mu \in \mathscr{T}$, then so is $t \xrightarrow{c} \nu$. This follows from `chkConfluentMimicking`. After all, each transition and its mimicking transition that are found are added to C in the body of `chkConfluence`. Only when $\mathscr{T}$ is reset also C is, since the mimickings that were found in that call are then clearly not relevant anymore. At the end, `chkConfluentMimicking` checks all of the mimicking pairs. If one fails the test, the function checks to see if it can still add $t \xrightarrow{c} \nu$ to $\mathscr{T}$ to make things right. Since we assumed that it returns *true*, no irreparable violation was found, and indeed all confluent transitions are mimicked by confluent transitions. □

Note that the converse of this theorem does not always hold. To see why, consider the situation that `chkConfluentMimicking` fails because a transition $s \xrightarrow{b} \mu$ was mimicked by a transition $t \xrightarrow{c} \nu$ that is not confluent, and $s \xrightarrow{b} \mu$ was added to $\mathscr{T}$ later on. Although we then abort, there might have been another transition $t \xrightarrow{d} \rho$ that could also have been used to mimic $s \xrightarrow{b} \mu$ and that *is* confluent. We chose not to consider this due to the additional overhead of the implementation. Additionally, this situation did not occur in any of the case studies we considered so far.

We can now use Theorem 5 to show that the on-the-fly confluence check implemented by `chkConfluence` and `chkConfluentMimicking` can be used for a trustworthy SMC analysis of MDP:

Lemma 4 (Correctness of SMC with on-the-fly confluence check)**.** If an SMC analysis terminates and does not return *unknown*
– using function `simulate` of Algorithm 17 to explore the set of paths Π
– together with function `resolveConfluence` of Algorithm 19 in place of $\mathfrak{A}$,
then the function $f = f_{confl} \cup \{ s \mapsto T(s) \mid s \notin \Pi \}$ satisfies Condition 4.3, where f_{confl} maps a state $s \in \Pi$ to $T(s)$ if it is deterministic and to the result of the call to `resolveConfluence` otherwise.

Proof. By construction and because `resolveConfluence` is deterministic, f is a reduction function satisfying $s \in \Pi \Rightarrow |f(s)| = 1 \wedge s \notin \Pi \Rightarrow f(s) = T(s)$. It remains to show that minimum and maximum probabilistic reachability probabilities for any state formula over the atomic propositions are the same for the original and the reduced MDP.

The way it is constructed, we can see f as the combination of individual reduction functions $f_{s_i} = \{ s_i \mapsto tr_i^{confl} \} \cup \{ s' \mapsto T(s') \mid s' \notin \Pi \}$ for $i \in \{1, \ldots, n\}$

and $\{s_1,\ldots,s_n\} = \{s \in \Pi\}$ where tr_i^{confl} is the result of the call to the function `resolveConfluence` for s_i. For each of these f_{s_i}, we know that it is acyclic (otherwise s_i would be mapped to a self-loop and the cycle check of Algorithm 17 would have aborted simulation) and a confluence reduction function for M (by Theorem 5). However, f_{s_j} is not necessarily a confluence reduction function for $\mathrm{red}(M, f_{s_i})$, $i \neq j$: `resolveConfluence(`s_j`)` performed its checks on M and not on $\mathrm{red}(M, f_{s_i})$. However, each f_{s_i} gives priority to one transition between two branching bisimilar states (see [HT14a, Tim13]). We denote branching bisimulation by $\sim$ here; it is a relation that preserves probabilistic reachability. Therefore, if $R_\sim$ is the coarsest concrete bisimulation relation for M under $\sim$ and $M_\sim$ is an MDP such that $\langle M, M_\sim\rangle \in R_\sim$, then also $\langle M_\sim, \mathrm{red}(M_\sim, f_{s_i})\rangle \in R_\sim$. By transitivity, we have $\langle M, \mathrm{red}(M_\sim, f_{s_i})\rangle \in R_\sim$, too. In consequence, $M \sim M_f$ since $M_f = \mathrm{red}(\ldots \mathrm{red}(\mathrm{red}(M, f_{s_1}), f_{s_2})\ldots, f_{s_n})$. □

Remark. One may want to prove preservation of probabilistic reachability directly for f based on theorems 4 and 5. However, this does not work out: It would have to be shown that f is itself an acyclic confluence reduction function under a confluent set of transitions $\mathscr{T}$. The acyclicity of the entire function f is also guaranteed by the modified `simulate` function of Algorithm 17. It would remain to show following Definition 58 that $\mathscr{T}_\cup = \{tr \in f(s) \mid s \in \Pi\}$ is a confluent set of transitions. While we know from Theorem 5 that each transition $s \xrightarrow{a} \mathscr{D}(t) \in \mathscr{T}_\cup$ is probabilistically confluent, this only means that there exists a confluent set that contains it, namely the set $\mathscr{T}$ computed by `chkConfluence` and `chkConfluentMimicking` for s. Unfortunately, neither is $\mathscr{T}_\cup$ in general the union of these individual sets, nor is the union of two confluent sets necessarily a confluent set again [Tim13, Chapter 6].

Runtime and Memory Usage

Exactly as for the POR-based approach, the runtime and memory usage of SMC with on-the-fly confluence check depends on the amount of lookahead that is necessary in function `chkConfluence`. Although we have not included this in Algorithm 19 as shown, it is in practice parameterised by a lookahead bound k_{max} to enforce termination, too, with the same characteristics as in the POR-based approach. Any differences in runtime and memory usage we see between the two approaches, which look at in Section 4.7.3, should thus come only from a more optimised or computationally simpler implementation. There are no fundamental differences in performance characteristics to be expected.

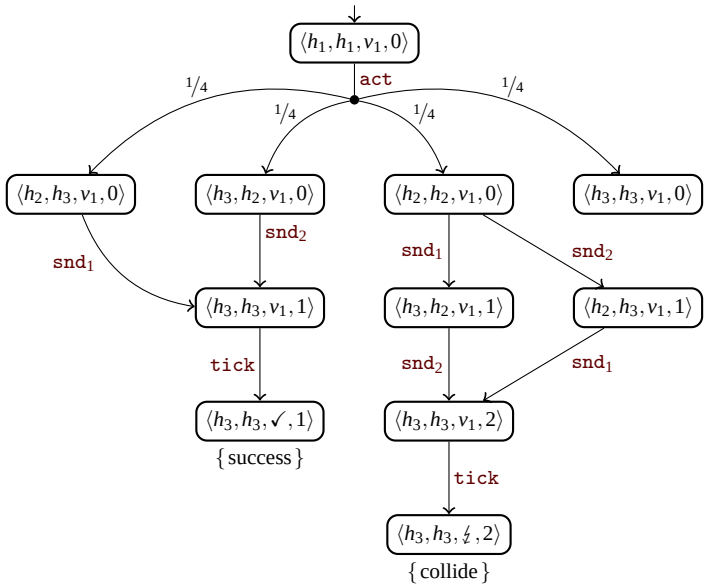


Figure 4.13: MDP semantics of the network of VMDP $N_{\text{var}}^{\texttt{act}}$

Applicability and Limitations

Confluence, too, is only a safe approximation when it comes to the detection of spurious choices. Although the confluence check does not need to resort to information from the syntactic/symbolic VMDP level like POR with conditions J1 and J2, and in particular is not limited to detecting spurious interleavings only, we see a few limitations right in the definition of confluence. The most significant one is that only nonprobabilistic transitions can be confluent. Let us again look at the shared medium communication setting of Example 22 to get an impression of what this may mean in practice.

Example 27. We saw in Example 23 that the POR-based approach works for the networks N_{var}^{τ} and $N_{\text{var}}^{\texttt{act}}$, but cannot work for any instance of N_{sync} due to the nondeterministic choice inside process M_{sync}. This, however, is no problem for confluence reduction, and SMC with the on-the-fly confluence check can handle $N_{\text{sync}}^{\texttt{act}}$ without problems. On the other hand, nonprobabilistic transitions cannot be confluent. Therefore, simulation with the new approach will abort for N_{sync}^{τ} as well as for N_{var}^{τ}. What about $N_{\text{var}}^{\texttt{act}}$? We can consider this a

approach	alg.	N^{τ}_{var}	$N^{\texttt{act}}_{\text{var}}$	N^{τ}_{sync}	$N^{\texttt{act}}_{\text{sync}}$
POR	18	✓	✓	–	–
confluence	19	–	✓	–	✓

Table 4.3: Applicability of POR- and confluence-based SMC to Example 22

version of N^{τ}_{var} that has been specifically fixed to make it amenable to the confluence check: the interleaving of the probabilistic decision has been replaced by a synchronisation. All probabilistic reachability properties are unaffected by this change, but both confluence- and POR-based simulation work with this model. For comparison with Figure 4.11, we show the MDP semantics of $N^{\texttt{act}}_{\text{var}}$ in Figure 4.13. The only nondeterministic choice that remains in this case is which host sends first in state $\langle h_2, h_2, v_1, 0\rangle$. It is easy to see that both available transitions are confluent and indeed are identified as such by Algorithm 19. Table 4.3 summarises the applicability of the two simulation approaches to all four variants of the shared medium communication example.

In summary, the new confluence-based approach for the first time allows simulation of models with spurious nondeterministic choices that are internal to one component. However, as confluence preserves branching time properties, it cannot handle nondeterminism between probabilistic choices. Although this can often be avoided, for example by transforming the example models N^{τ}_{var} and N^{τ}_{sync} into $N^{\texttt{act}}_{\text{var}}$ and $N^{\texttt{act}}_{\text{sync}}$, respectively (i.e. by replacing interleaved probabilistic transitions by synchronised ones through a change of transition labels—a technique that we will also use for some of the case studies in Section 4.7.3), for some models POR might work while confluence does not. Hence, neither of the techniques subsumes the other, and it is best to combine them: If one cannot be used to resolve a nondeterministic choice, the simulation algorithm can try to apply the other. Implementing this combination is trivial and yields a technique that handles the union of what confluence and POR can deal with.

4.7.3 Evaluation

The modes tool provides SMC for models specified in the MODEST language and other input formalisms (cf. Section 1.4). It implements both approaches presented so far to perform SMC for MDP with spurious nondeterministic choices: the POR-based and the confluence-based check. In this section, we apply modes and its implementation of the two approaches to three examples to evaluate their applicability and performance impact on practical examples beyond the tiny models of Example 22. The case studies were selected so as to

allow us to clearly identify the approaches' strengths and limitations. For each, we (1) give an overview of the model, (2) discuss, if POR or confluence fails, why it does and which, if any, modifications were needed to apply it, and (3) evaluate memory use and runtime.

The performance results are summarised in Table 4.4. For the runtime assessment, we compare to simulation with uniformly-distributed probabilistic resolution of nondeterminism. Although such an assumption cannot lead to trustworthy results in general (but is silently implemented in many tools), it is a good *baseline* to judge the *overhead* of POR and confluence checking. We generated 10 000 runs per model instance to compute probabilities p_{smc} for case-specific properties. Using the statistical evaluation of Algorithm 7, this guarantees the following probabilistic error bound: $\mathbb{P}(|p - p_{\text{smc}}| > 0.015) < 0.022$, where p is the actual probability of the property under consideration. That is, the probability of p_{smc} deviating more than 0.015 from the actual probability is at most 0.022.

We measure memory usage in terms of the maximum number of extra states kept in memory at any time during POR or confluence checking, denoted by s. The average number of states per check is listed as s_{avg}. We also report the maximum number of "lookahead" steps necessary in the POR and confluence checks as k, as well as the average length t of a simulation trace and the average number c of nontrivial checks, i.e. of nondeterministic choices encountered, per trace.

To get a sense for the size of the models considered, we also attempt model checking using mcpta , which uses PRISM for the core analysis. Due to modelling restrictions of PRISM, we use mcsta , an explicit-state model checker that is also part of the MODEST TOOLSET and that uses the same core state space exploration engine that modes relies on, for our first example. (mcsta will reappear with more detailed explanations in Chapter 6.) We report the probability p for the model-specific properties as computed by mcpta/mcsta where possible, and otherwise list "$\sim p_{\text{smc}}$" in that column of Table 4.4 instead. In all cases, p and p_{smc} match within the expected confidence. Note that we do not intend to perform a rigorous comparison of SMC and traditional model checking here and instead refer the interested reader to dedicated comparison studies such as [YKNP06]. Unless otherwise noted, all measurements used a 1.7 GHz Intel Core i5-3317U system with 4 GB of RAM running 64-bit Windows 8.1.

Binary Exponential Backoff

We first study a model of the IEEE 802.3 Binary Exponential Backoff (BEB) protocol for a set of network hosts, adapted from the PRISM model of [GDF09].

model	params	$\mathfrak{R}_{\text{Uni}}$ time	with **partial order reduction** time	k	s	s_{avg}	c	t	with **confluence** time	k	s	s_{avg}	c	t	model checking states	time	p
BEB (tau) $\langle K,N,H\rangle$	$\langle 4,3,3\rangle$	0 s	7 s	3	28	5.9	6.5	22.4	–	–	–	–	–	–	4 660	0 s	0.917
	$\langle 8,7,4\rangle$	0 s	59 s	4	736	12.7	11.0	32.3	–	–	–	–	–	–	20 186 888	114 s	0.999
	$\langle 16,15,5\rangle$	1 s	338 s	5	12 400	30.0	16.4	41.4	–	–	–	–	–	–	– memout –		∼1.00
	$\langle 16,15,6\rangle$	1 s	1 374 s	6	83 536	89.5	22.3	51.3	–	–	–	–	–	–	– memout –		∼1.00
BEB (sync) $\langle K,N,H\rangle$	$\langle 4,3,3\rangle$	0 s	1 s	3	4	2.8	3.3	11.8	1 s	3	7	4.6	3.3	11.8	2 319	0 s	0.917
	$\langle 8,7,4\rangle$	0 s	4 s	4	8	3.9	5.6	16.9	3 s	4	15	6.8	5.6	16.7	10 105 805	59 s	0.999
	$\langle 16,15,5\rangle$	1 s	11 s	5	16	5.5	8.2	21.4	8 s	5	31	10.0	8.2	21.2	– memout –		∼1.00
	$\langle 16,15,6\rangle$	1 s	26 s	6	32	8.0	11.1	25.9	23 s	6	63	15.1	11.1	25.8	– memout –		∼1.00
	$\langle 4,3,5\rangle$	0 s	10 s	5	16	5.5	8.2	19.5	7 s	5	31	10.1	8.1	19.3	185 043	1 s	0.670
	$\langle 5,4,5\rangle$	0 s	10 s	5	16	5.4	8.4	21.2	7 s	5	31	9.8	8.5	21.3	1 903 921	11 s	0.879
	$\langle 6,5,5\rangle$	0 s	10 s	5	16	5.4	8.5	21.8	7 s	5	31	9.8	8.5	21.9	15 237 260	133 s	0.974
	$\langle 4,3,6\rangle$	1 s	25 s	6	32	8.2	10.9	22.9	22 s	6	63	15.4	10.9	22.8	1 659 897	10 s	0.544
	$\langle 5,4,6\rangle$	1 s	25 s	6	32	7.9	11.6	25.9	22 s	6	63	14.7	11.7	26.3	– memout –		∼0.79
	$\langle 6,5,6\rangle$	1 s	25 s	6	32	7.9	11.6	26.7	22 s	6	63	14.8	11.6	26.8	– memout –		∼0.94
	$\langle 4,3,7\rangle$	1 s	58 s	7	64	12.7	13.8	26.4	72 s	7	127	24.1	13.8	26.5	15 111 003	129 s	0.432
	$\langle 5,4,7\rangle$	1 s	59 s	7	64	11.8	15.1	30.8	73 s	7	127	22.6	15.1	30.9	– memout –		∼0.68
	$\langle 6,5,7\rangle$	1 s	59 s	7	64	11.7	15.3	32.6	73 s	7	127	22.6	15.2	32.1	– memout –		∼0.89
CSMA/CD $\langle RF, BC_{max}\rangle$	$\langle 2,1\rangle$	1 s	–	–	–	–	–	–	4 s	4	46	16.3	5.0	16.3	15 283	11 s	1
	$\langle 1,1\rangle$	1 s	–	–	–	–	–	–	4 s	4	46	16.3	5.0	16.3	30 256	49 s	1
	$\langle 2,2\rangle$	1 s	–	–	–	–	–	–	8 s	4	150	25.0	4.9	16.0	98 533	52 s	1
	$\langle 1,2\rangle$	1 s	–	–	–	–	–	–	8 s	4	150	25.0	4.9	16.0	194 818	208 s	1
dining cryptographers N	3	0 s	–	–	–	–	–	–	1 s	4	9	6.5	4.0	8.0	609	1 s	1
	4	0 s	–	–	–	–	–	–	4 s	6	25	13.7	6.0	10.0	3 841	2 s	1
	5	0 s	–	–	–	–	–	–	17 s	8	67	29.0	8.0	12.0	23 809	7 s	1
	6	0 s	–	–	–	–	–	–	92 s	10	177	62.8	10.0	14.0	144 705	26 s	1
	7	0 s	–	–	–	–	–	–	– timeout –						864 257	80 s	1

Table 4.4: SMC with POR or confluence: runtime overhead and comparison

It gives rise to a network of VMDP. The model is determined by K, the maximum backoff counter value, i.e. the maximum number of slots to wait in the exponential backoff procedure, by N, the number of times each host tries to seize the channel, and by H, the number of hosts. The number of component VMDP is $H+1$: the hosts plus a global timer process. The probability we compute is that of some host eventually getting access to the shared medium. Our simulation scenario ends when a host seizes that channel. The MODEST model is shown in Figure 4.14. Note that it makes use of the `if` shorthand, which is syntactic sugar for an `alt` with two alternatives that are guarded by the `if`'s condition and its negation (for the `else` branch), respectively. The underlying network of VMDP is similar to N^{τ}_{var} of Example 22: there is an interleaved probabilistic choice (implicitly: the assignment of DISCRETEUNIFORM to `wt` is expanded by modes to a `palt` construct), and variable `cr` keeps track of how many hosts currently try to send, with value 2 indicating a collision. In contrast to the simple model of N^{τ}_{var}, the hosts react to a collision by starting an exponential backoff procedure, trying to send again N times before giving up.

First of all, we observe in Table 4.4 in the rows labelled "BEB (tau)" that model checking with mcsta aborts due to lack of memory for the two larger instances $\langle 16,15,5\rangle$ and $\langle 16,15,6\rangle$. We also attempted to perform model checking using PRISM in default configuration on a 64-bit Linux system with 120 GB of RAM, but this failed due to memory usage for the two large instances, too.

Simulation, on the other hand, works for all instances. In all cases the nondeterminism is identified as spurious by the POR-based method for small values of k_{max} and l. The runtime values for uniform resolution show that these models are extremely simple to simulate, while the on-the-fly partial order reduction approach induces a significant time overhead. We see a clear dependence between the number of components and the value of k, with $k = H$. In line with our expectations concerning the performance impact from Section 4.7.1, the increase in memory usage (not shown in the table because it is not possible to obtain precise and useful measurements for a garbage-collected implementation like modes) is more moderate. Although 83 536 states have to be kept in memory during at least one of the POR checks for the largest instance $\langle 16,15,6\rangle$, this is just a tiny fraction of the whole state space considering that the small instance $\langle 8,7,4\rangle$ already has more than 20 million states. It is also obvious from the relation between s and s_{avg} that a POR check with such a large amount of on-the-fly state space exploration is a relatively rare occurrence.

Because this model contains nondeterministic choices between probabilistic transitions, any attempt to perform SMC with the confluence check immediately aborts. However, just like we transformed N^{τ}_{var} into $N^{\texttt{act}}_{\text{var}}$ without

```
action tick, tack, tock;

const int K = 4; // maximum value for backoff
const int N = 3; // number of tries before giving up
const int H = 3; // number of hosts

int(0..2) cr; // #hosts that want to seize the line (0, 1, many)
bool line_seized, gave_up;

property LineSeized = Pmax(<> line_seized);

process Clock() {
   tick; tack; tau {= cr = 0 =}; tock; Clock()
}

process Host() {
   int(0..N) na;     // nr_attempts 0..N
   int(0..K) ev = 2; // exp_val 0..K
   int(0..K) wt;     // slots_to_wait 0..K
   do {
   :: if(wt > 0)
      {
         tick {= wt-- =} // wait this slot
      }
      else
      {
         tau {= cr = min(2, cr+1) =}; // attempt to seize line
         tick;
         if(cr == 1)
         {
            // someone managed to seize the line
            tau {= line_seized = true =}; stop
         }
         else if(na >= N)
         {
            // maximum number of attempts exceeded
            tau {= gave_up = true =}; stop
         }
         else
         {
            // backoff
            tau {= wt = DiscreteUniform(0, max(0, ev - 1)),
                   ev = min(2 * ev, K), na++ =}
         }
      };
      tack; tock
   }
}

par { :: Clock() :: Host() :: Host() :: Host() }
```

Figure 4.14: MODEST model of the BEB case study (interleaved prob. choice)

affecting reachability probabilities in Example 27, we can transform the MODEST code of the BEB model to make the probabilistic choices synchronise on action `tick`. The result is shown in Figure 4.15, while the performance numbers are in the rows labelled "BEB (sync)" in Table 4.4. Where model checking with mcsta is possible, we see that the number of states of this synchronised model is roughly of the same order of magnitude as that of the original.

Simulation now works fine using either of the two approaches. The runtime overhead necessary to get trustworthy results by enabling either confluence or POR is now much lower. In particular, the amount of states that need to be explored during the POR and confluence checks is very low compared to the original model. It thus appears that the probabilistic interleavings actually caused most of the work for the POR check. This is very similar to N^{τ}_{var} into $N^{\texttt{act}}_{\text{var}}$: just compare their concrete state spaces as shown in figures 4.11 and 4.13.

In addition to the four instances that we studied with the unsynchronised model, we now also look at $\langle 4,3,H\rangle$, $\langle 5,4,H\rangle$ and $\langle 6,5,H\rangle$ for $H \in \{5,6,7\}$. This allows us to more systematically investigate how scaling the different model parameters affects runtime. We first of all see that state space explosion occurs no matter whether we scale the counters K and N or the number of hosts H, and model checking fails for the larger instances. While simulation does work for all instances, there is a clear pattern in the runtime and memory overhead of using the POR or confluence check: It grows significantly with H, but is almost invariant under increases to K and N. H determines the number of parallel components in the model and thus the potential amount of interleaving. Both the POR- and the confluence-based approach thus work for models of arbitrary state space size, but they are sensitive to the size of the interleavings.

Although the confluence-based approach is somewhat faster than POR for $H \leq 6$ and somewhat slower for $H = 7$, the differences are not very large and could probably be reduced by further optimising both implementations. Most importantly, the memory overhead compared to uniform resolution of nondeterminism is in all cases negligible, and one of the central advantages of SMC over traditional model checking is thus retained.

IEEE 802.3 CSMA/CD

As a second example, we take the MODEST model of the Ethernet (IEEE 802.3) CSMA/CD approach that was introduced in [HH09]. It consists of two identical stations attempting to send data at the same time, with collision detection and a randomised backoff procedure that tries to avoid collisions for subsequent retransmissions. We consider the probability that both stations eventually manage to send their data without collision. The model is a network of probabilistic

```
action tick, tack;

const int K = 4; // maximum value for backoff
const int N = 3; // number of tries before giving up
const int H = 3; // number of hosts

int(0..2) cr; // #hosts that want to seize the line (0, 1, many)
bool line_seized, gave_up;

property LineSeized = Pmax(<> line_seized);

process Clock() {
   tick; tack {= cr = 0 =}; Clock()
}

process Host() {
   int(0..N) na;     // nr_attempts 0..N
   int(0..K) ev = 2; // exp_val 0..K
   int(0..K) wt;     // slots_to_wait 0..K
   do {
   :: if(wt > 0)
      {
         tick {= wt-- =} // wait this slot
      }
      else
      {
         tau {= cr = min(2, cr+1) =}; // try to seize the line
         if(cr == 1)
         {
            // someone managed to seize the line
            tick {= line_seized = true =}; stop
         }
         else if(na >= N)
         {
            // maximum number of attempts exceeded
            tick {= gave_up = true =}; stop
         }
         else
         {
            // backoff
            tick {= wt = DiscreteUniform(0, max(0, ev - 1)),
                    ev = min(2 * ev, K), na++ =}
         }
      };
      tack
   }
}

par { :: Clock() :: Host() :: Host() :: Host() }
```

Figure 4.15: MODEST model of the BEB case study (sync. prob. choice)

timed automata (PTA), but delays are fixed and deterministic, making it equivalent to a network of VMDP (with real variables for clocks, updated on edges that explicitly represent the delays; modes does this transformation automatically and on-the-fly). We will investigate this model of *time-deterministic PTA* in more detail in Section 5.6.1. The model has two parameters: a time reduction factor RF (i.e. delays of t time units with $RF = 1$ correspond to delays of $\frac{t}{2}$ time units with $RF = 2$), and the maximum value used in the exponential backoff part of the protocol, BC_{max}. The MODEST code can be found on the MODEST TOOLSET's website at `www.modestchecker.net`.

We chose to look into this model because it is similar to the BEB case study: both model a shared medium access protocol that uses an exponential backoff procedure. Yet there are two main differences—apart from one being an untimed, the other a timed model—that justify a separate investigation: the CSMA/CD model focuses on just two hosts, and it explicitly models the shared medium with a dedicated process that uses synchronisation to detect collisions. In this way, it is very similar to N^{τ}_{sync} of Example 22.

Unfortunately, but not unexpectedly, modes immediately reports nondeterminism that cannot be discarded as spurious when using the confluence-based check. Inspection of the reported lines in the model quickly shows a nondeterministic choice between two probabilistic transitions as the culprit again. Fortunately, this problem can easily be eliminated in the same way as for the BEB model and the $N^{\tau}_{\cdot}$ examples: with an additional synchronisation. This appears to be a recurring issue, yet the relevant model code could quite clearly be identified as a modelling artifact without semantic impact in both examples where it appeared so far. SMC on the modified model then leads to $p_{\text{smc}} = 1.0$, which is the correct result.

The POR-based approach also fails for the unmodified model: Initially, both stations send at the same time, the order (of the interleaving in zero time) being determined nondeterministically. In the process representing the shared medium, this must be an *internal* nondeterministic choice. This is exactly the same problem that prevented POR from working for the $N^{\cdot}_{\text{sync}}$ examples. In contrast to the problem for confluence, this cannot be fixed so easily.

In terms of runtime, the confluence checks incur a moderate overhead for this example, lower than for the BEB models. However, we also see that the paths being explored in the confluence checks (value k) are shorter. Performance appears to quite directly depend on k, which stays low in this case. Again, we observed no significant increase in memory usage compared to uniform resolution. Compared to model checking with PRISM, SMC even with the confluence checking overhead appears highly competitive here, and it in particular

does not depend on the timing scale (performance is independent of model parameter RF).

Dining Cryptographers

As a last and very different example, we consider the classical dining cryptographers problem [Cha88]: N cryptographers use a protocol that has them toss coins and communicate the outcome with some of their neighbours at a restaurant table in order to find out whether their master or one of them just paid the bill, without revealing the payer's identity in the latter case. We model this problem as the parallel composition of N instances of a `Cryptographer` process that communicate via synchronisation on shared actions, and consider as properties the probabilities of (a) protocol termination and (b) correctness of the result.

The MODEST code for the model is shown in Figure 4.16 for the case of $N = 3$. It is a network of VMDP whose semantics is a nondeterministic MDP. In particular, the order of the synchronisations between the cryptographer processes is not specified, and could conceivably be relevant. It turns out that all nondeterminism can be discarded as spurious by the confluence-based approach though, allowing the application of SMC to this model.

The POR-based approach does not work: Although the nondeterministic ordering of synchronisations between non-neighbouring cryptographers is due to interleaving, the choice of which neighbour to communicate with first for a given cryptographer process is a nondeterministic choice *within* that process. At its core, this is yet again the same problem as with the $N^{\cdot}_{\text{sync}}$ networks of Example 22.

Concerning performance, we see that runtime increases significantly with the number of cryptographers N. In fact, for $N = 7$, we aborted simulation after 30 minutes and consider this a timeout. An increase is expected, since the number of steps until independent paths from nondeterministic choices join again (k) depends directly on N. It is so drastic due to the sheer amount of branching that is present in this model. At the same time, the model is extremely symmetric and can thus be handled easily with a symbolic model checker like PRISM.

Summary

All in all, these three case studies show that SMC using an on-the-fly POR or confluence check is effective and efficient. The memory overhead is negligible, and one of the central advantages of SMC over exhaustive model checking is thus retained. In terms of runtime, we see two models where the confluence

```
action flip, end, show_heads, show_tails, see_heads, see_tails;
action show_heads1, show_tails1, show_heads2, show_tails2,
       show_heads3, show_tails3;

const int N = 3;

int(0..N) pay, nagree; bool done;

property Terminate = Pmax(<> done);
property Correct = Pmax(<> done && (nagree%2 == N%2 && pay == 0
                                 || nagree%2 != N%2 && pay != 0));

process Cryptographer(int(1..N) id) {
   bool heads, agree;

   process Show() {
      alt {
      :: when( heads) show_heads
      :: when(!heads) show_tails
      }
   }

   process See() {
      alt {
      :: see_heads {= agree = heads =}
      :: see_tails {= agree = !heads =}
      }
   }

   flip palt { :1: {= heads = true =} :1: {= heads = false =} };
   par { :: Show() :: See() };
   tau {= nagree += ((agree == (pay != id)) ? 1 : 0) =}; end
}

process Payment() {
   // Choose a scenario probabilistically
   flip {= pay = DiscreteUniform(0, N) =}; end {= done = true =}
}

par {
:: Payment()
:: relabel { show_heads,  show_tails,  see_heads,   see_tails   }
   by      { show_heads1, show_tails1, show_heads3, show_tails3 }
   Cryptographer(1)
:: relabel { show_heads,  show_tails,  see_heads,   see_tails   }
   by      { show_heads2, show_tails2, show_heads1, show_tails1 }
   Cryptographer(2)
:: relabel { show_heads,  show_tails,  see_heads,   see_tails   }
   by      { show_heads3, show_tails3, show_heads2, show_tails2 }
   Cryptographer(3)
}
```

Figure 4.16: MODEST model of the dining cryptographers case study ($N = 3$)

and, when applicable, the POR-based approach induce a moderate overhead, and one model where runtime explodes. This reinforces our previously stated expectation that performance will be extremely dependent on the structure of the model under study. When comparing confluence and POR, we see that confluence struggles with probabilistic interleavings, yet we were able to overcome this limitation by modifying the models in both cases. On the other hand, SMC is only possible for two of the three examples with the confluence check due to POR's restriction to spurious interleavings from parallel composition. The different reduction power of confluence is thus relevant and useful, but neither of the techniques subsumes the other. As mentioned, in practice, it would probably be best to combine them: if one cannot be used to resolve a nondeterministic choice, the SMC algorithm can still try to apply the other. Implementing this combination is trivial and yields a technique that handles the union of what confluence and POR can deal with.

4.7.4 Caching the Reduction Function

A key problem that leads to long runtimes of simulation with on-the-fly POR or confluence checks for some models is that the same information may have to be computed over and over again: once a nondeterministic choice has been proven to be spurious, simulation continues to the next state and this spuriousness result is forgotten. We can potentially avoid this problem by caching the results of the calls to `resolvePOR` or `resolveConfluence`, i.e. additionally storing a mapping from states to (ample or confluent) transitions.

In the worst case, all states are visited on the set of paths Π and they are all (spuriously) nondeterministic. Then, memory usage would be as in exhaustive model checking: the entire state space would be stored. Still, we have seen in the previous section that the fraction of states that are nondeterministic is usually low: c/t in Table 4.4, which is per simulation run and not for the entire model, is below 1 for all models, and significantly so (at least less than $1/2$) for all but the dining cryptographers case. For the latter, however, exhaustive model checking is feasible, so memory usage is unproblematic in any case.

Even if we look at this tradeoff from a high-level point of view, SMC with on-the-fly POR or confluence check can be seen as performing simulation with a certain amount of embedded on-the-fly exhaustive model checking. It is thus a "hybrid" procedure already. Building in more aspects otherwise typical for exhaustive model checking—i.e. storing more states—thus makes it somewhat "less SMC" and more exhaustive, but is no fundamental change of character.

model	params	POR time	**POR cached** time	**POR cached** states	confl. time	**confl. cached** time	**confl. cached** states	state space states
BEB (tau) $\langle K,N,H\rangle$	$\langle 4,3,3\rangle$	7 s	0 s	248	–	–	–	4 660
	$\langle 8,7,4\rangle$	59 s	9 s	5 304	–	–	–	20 186 888
	$\langle 16,15,5\rangle$	338 s	247 s	21 000	–	–	–	– memout –
	$\langle 16,15,6\rangle$	1 374 s	1 096 s	52 594	–	–	–	– memout –
BEB (sync) $\langle K,N,H\rangle$	$\langle 4,3,3\rangle$	1 s	0 s	118	1 s	0 s	119	2 319
	$\langle 8,7,4\rangle$	3 s	0 s	2 474	2 s	0 s	2 435	10 105 805
	$\langle 16,15,5\rangle$	7 s	1 s	9541	5 s	1 s	9 553	– memout –
	$\langle 16,15,6\rangle$	16 s	2 s	24 045	13 s	2 s	24 123	– memout –
CSMA/CD $\langle RF, BC_{max}\rangle$	$\langle 2,1\rangle$	–	–	–	4 s	1 s	49	(15 283)
	$\langle 1,1\rangle$	–	–	–	4 s	1 s	49	(30 256)
	$\langle 2,2\rangle$	–	–	–	8 s	1 s	262	(98 533)
	$\langle 1,2\rangle$	–	–	–	8 s	1 s	262	(194 818)
dining cryptographers N	3	–	–	–	1 s	0 s	128	609
	4	–	–	–	4 s	0 s	480	3 841
	5	–	–	–	17 s	1 s	1 536	23 809
	6	–	–	–	92 s	13 s	4 480	144 705
	7	–	–	–	– t/o –	– timeout –		864 257

Table 4.5: SMC with POR or confluence: caching the reduction function

Evaluation We have implemented a caching wrapper around the POR and confluence checks in modes and applied it to the case studies presented in the previous section. The results are shown in Table 4.5, where the state numbers reported for the cached variants are the numbers of states for which a POR or confluence result has been stored at the end of the SMC analysis. They consequently provide an indication of the additional memory usage due to the caching. The setting is otherwise the same as in the previous section (10 000 runs, same machine, etc.).

We see that caching leads to speedups in all cases, most of them significant, and that the number of states cached is always small in comparison to the full state spaces[8]. However, it does not seem to drastically improve performance for the larger instances of the non-synchronising BEB model. This is likely because, due to the size of the state spaces and the nature of the probabilistic branching, any particular state is rarely visited several times in 10 000 runs. In line with this effect is the observation (by looking at modes' progress indicator) that, on all models and instances, the first simulation runs take relatively long, but from some point on the remaining ones need almost no time anymore. This point is where most of the relevant choices have been proven spurious. The

[8] For the CSMA/CD models, the state space sizes reported are for the digital clocks semantics of the PTA created by mcpta and thus not directly comparable to the state spaces that modes explores.

number of runs after which it is reached depends on the model and parameters, but the sudden change in simulation speed is clearly visible in all cases (except for the largest unsynchronised BEB instance).

Still, even caching does not allow us to perform SMC for the dining cryptographers case with $N = 7$ within acceptable time. Here, even the first dozen runs take extremely long and do not yet provide enough cached results for any observable speedup.

In summary, caching the results of the POR or confluence checks appears to be a very good way to make the approaches scale to a higher number of simulation runs and thus to more precise and accurate verification results, yet models where SMC was previously unfeasible due to excessive runtime—probably caused by huge underlying state spaces and extreme branching—remain unfeasible.

As an example of scaling accuracy and precision with declining runtime impact by caching, we also performed SMC with 10 times as many simulation runs (i.e. 100 000 instead of 10 000) for two of the unsynchronised BEB model instances. For $\langle 16, 15, 5 \rangle$, this takes 1272 s, i.e. just around 5 times as long; for $\langle 8, 7, 4 \rangle$ the time is 27 s, i.e. an increase by a factor of 3 only. Again using the statistical evaluation of Algorithm 7, this for example means that the probability of the SMC result deviating more than 0.005 from the actual probability is now at most 0.014, whereas we previously had 0.015 deviation with probability 0.022—yet we only invested 3 to 5 times the simulation runtime.

4.8 SMC for General MDP

Spuriously nondeterministic MDP, to which the approaches presented in the previous section can be applied, are but a specific subclass of MDP. Although we argued that the existence of non-spurious interleavings (i.e. of race conditions) typically indicates an error in itself, nondeterminism (in particular in all its other forms) is often a desired modelling feature. Finding ways to apply SMC-like techniques that preserve as much of the memory advantage as possible to general MDP has thus recently become an active research topic. In this section, we summarise the two approaches we are currently aware of that are relevant to our setting of probabilistic reachability verification: the use of learning algorithms to try to learn the maximising scheduler for a property (Section 4.8.1), and sampling over probabilistic choices *and* schedulers with a memory-efficient representation of the latter (Section 4.8.2). Finally, we show that the direct application of stateless model checking techniques (cf. Section 3.1.5) to exhaustively explore the nondeterministic choices while resolving the probabilistic ones does not work (Section 4.8.3).

4.8.1 Using Learning Algorithms

Where nondeterministic choices in an MDP are not spurious, the maximum and minimum probabilities of reaching a state satisfying a certain state formula ϕ are not the same. However, schedulers exist that lead to precisely those probabilities. In exhaustive model checking using e.g. value or policy iteration, they are (implicitly or explicitly) found by a fixpoint operation over the whole state space.

Using Reinforcement Learning

Henriques et al. [HMZ+12] instead propose the use of reinforcement learning [SB98, CFHM07], an artificial intelligence technique, to improve an arbitrary candidate probabilistic scheduler to make its decisions closer and closer to a maximising one. The algorithm works with probabilistic schedulers during the learning and improvement phase in order to explore more of the state space instead of completely ruling out certain transitions early on.

The improvement of schedulers is based on a measure of how "good" a transition is in reaching a target state. This is approximated by performing a number of simulation traces and observing which decisions actually do lead to target states. Subsequently, the scheduler is updated such that good transitions are chosen with a higher probability (but not with probability 1). After a number of these improvement steps, a nonprobabilistic scheduler is computed by selecting the most likely decisions of the probabilistic candidate in each state, and a standard SMC analysis is performed using this scheduler to resolve the nondeterministic choices. The entire procedure is summarised in Algorithm 21.

First of all, we observe that the procedure is only specified for the qualitative form of probabilistic reachability properties, $\mathrm{P}(\diamond\phi) \leq x$ (or equivalently with bound $< x$). This is because the scheduler that is being approximated is the one that maximises the reachability probability. When a scheduler $\mathfrak{S}$ has been found that leads a standard SMC analysis (line 8), e.g. using the SPRT of Algorithm 8, to conclude that the property is *false*, we know—with the usual error bounds of the SMC algorithm used—that this scheduler is a counterexample to the bound x. However, the entire process does not give any guarantees about the optimality of that final scheduler. This means that we simply do not know at all how far the probability of reaching a ϕ-state in $\mathrm{ind}(M, \mathfrak{S})$ is from the actual maximum probability. This is why it is convenient to primarily consider the qualitative form, and why the algorithm can only return *probably true* (a.k.a. *unknown*) when a sufficiently large number t of "learned" schedulers does not lead to a violation of the bound.

Input: MDP M, property $\mathrm{P}(\diamond\phi) \leq x$, d, t, L
Output: $[\![\mathrm{P}(\diamond\phi) \leq x]\!]_M$ (*false* or *probably true*)

```
1  for i = 1 to t do
2      𝔖 := 𝔑_Uni
3      for j = 1 to L do
4          f := evaluate transitions in ind(M, 𝔖)
5          𝔖 := update probabilities in 𝔖 based on f
6      end
7      𝔖 := determinise 𝔖
8      if SMC for M and 𝔖 returns false then return false
9  end
10 return probably true
```

Algorithm 21: SMC for general MDP with learning [HMZ+12]

Pitfalls A significant problem that is not taken into account in [HMZ+12], pointed out by Legay and Sedwards [LST14], is that the SMC analysis, using a test like the SPRT that may sometimes give the wrong answer, is performed several times until it returns *false* (or we give up after t tries). This leads to error accumulation: While a single invocation of SMC in line 8 incorrectly returns *false* with a certain probability, the probability of the entire Algorithm 21 incorrectly returning *false* is higher. In fact, if the actual probability of reaching a ϕ-state is greater than zero, we are guaranteed to get the result *false* if we just iterate the outer loop of the algorithm often enough.

Other oversights include that this technique learns and improves memoryless schedulers, but it is specified for the verification of step-bounded properties. For these, the assumption of Definition 52 that memoryless schedulers are sufficient to maximise or minimise the reachability probability does not hold. An easy way to fix this problem would be to analyse unbounded properties instead, using cycle detection as outlined in Section 3.2.3 and used by all the `simulate` functions presented so far in this thesis.

Performance The memory usage of this approach clearly depends on how many scheduling decisions need to be stored by the computed schedulers. For each iteration of the outer loop of Algorithm 21, the amount of memory used is thus given by the number of nondeterministic states encountered during the simulation runs performed for scheduler evaluation in line 4. In the worst case, this is the number of states n of the MDP under study. However, as we have already seen for the POR- and confluence-based approaches in Section 4.7.3

and as Henriques et al. also show for other examples, this number can be much lower, but is in general highly dependent on the structure of the model. In terms of worst-case runtime, which occurs when the property is actually *true* (or t is large enough to incorrectly make the algorithm answer *false*), the parameters t and L are decisive as we need to perform up to $t \cdot L$ standard SMC analyses.

Learning Framework with BRTDP and DQL

In a different attempt to make use of learning techniques for the analysis of MDP models, Brázdil et al. [BCC+14] recently proposed a "general framework" to apply different learning algorithms to this problem. They define a learning algorithm as one that iteratively generates paths via simulation and updates upper and lower bounds U and L for a *value function*

$$V \in S \times (A \times \mathrm{Dist}(S)) \to [0,1]$$

over state-transition pairs defined as $V(s,\langle a,\mu\rangle) = \sum_{s' \in S} \mu(s') \cdot V(s')$. V maps each state s' to its *value*, which is the maximum probability of reaching a ϕ-state from s'. The algorithm terminates when $|U(s_{init},tr) - L(s_{init},tr)| < \varepsilon$ for all outgoing transitions tr of the initial state s_{init}. The *precision* ε is given as an argument to the algorithm.

While the idea superficially appears similar to the reinforcement learning approach presented before, and in particular focuses on maximum reachability probabilities $[\![\mathrm{P}_{\max}(\diamond\,\phi)]\!]$ again, there are crucial differences. For one, there is no separation between a learning and an evaluation phase; instead, path generation and the improvement of the approximations occur iteratively in alternation, starting from safe bounds. This avoids the problem of repeated invocations causing error accumulation. Furthermore, both a lower and an upper bound are computed. Memory usage, however, remains similar in that the current approximations U and L need to be stored for every state-transition pair that is visited during the simulation runs. It is thus again highly dependent on the structure of the model. The same applies to runtime: The number of iterations performed is simply the number of iterations needed until the difference between U and L in the initial state is below ε. This, too, obviously depends on the model at hand.

The framework is instantiated in [BCC+14] using two concrete learning algorithms, *bounded real-time dynamic programming* (BRTDP) and *delayed Q-learning* (DQL), that require different a priori information about the analysed model's state space and that deliver different confidence statements. We omit the details of how these algorithms work, which in particular involves some technical complexity to correctly handle MDP with multiple end components, and just summarise their requirements and the resulting error bounds:

Complete and limited information The BRTDP algorithm requires what the authors of [BCC+14] call "complete information" and which is mostly equivalent to the assumptions we make in this thesis: that the transition function may be given as a program, but can be evaluated for any state at will, thereby giving access to the complete information about a state's outgoing transitions and their target probability distributions. In addition, however, algorithms working in the complete information setting are also assumed to have access to global information about the MDP at hand, such as the number of states or the maximum number of outgoing transitions over all states. DQL, on the other hand, can work with "limited information", which means that evaluating the transition function still gives the set of outgoing transitions, but the target probability distributions are opaque and can only be sampled. In addition, *upper bounds* on the numbers of states and outgoing transitions are assumed to be known, as is a lower bound > 0 for the smallest probability of any branch in the entire MDP. In summary, this means that both learning algorithms need slightly more information than what we assume to be available for SMC. Still, global bounds like those required for DQL could be computed from e.g. the MODEST code of an MDP model.

Error bounds As mentioned, the proposed learning framework does not explicitly compute the number of iterations (and thus of simulation runs) necessary to achieve the desired precision, but instead iterates as long as necessary. On termination, we get two values $u = \max_{tr \in T(s_{init})} U(s_{init}, tr)$ and $l = \max_{tr \in T(s_{init})} L(s_{init}, tr)$ that are possibly an upper resp. a lower bound for $[\![\mathrm{P}_{\max}(\diamond \phi)]\!]$, with $|u - l| < \varepsilon$. For BRTDP, we have the guarantee that the algorithm almost surely terminates with u and l indeed being correct bounds. For DQL, the corresponding statement is similar to what the APMC method gives us (cf. Section 3.2.3): The algorithm takes an extra parameter δ, and with probability at least $1 - \delta$, terminates with u and l being correct bounds.

The bits of global information about the MDP at hand that we mentioned above are required at various places inside the two algorithms to achieve correctness and convergence; for example, the state space size is necessary to compute a valid "delay" for use inside DQL.

Exhaustive or statistical? In summary, the two learning algorithms presented for this "framework" make for sound learning-based approaches to the analysis of general MDP. However, their memory footprint and especially the fact that they require some global information about the MDP at hand (which also means that they cannot be applied to infinite-state models) mean that they depart from the spirit of SMC in some way. It may appear more natural to view

Input: MDP M, state formula ϕ, number of schedulers a, d, ε, δ
Output: $\langle \hat{p}_{\min}, \hat{p}_{\max} \rangle$ (two real numbers)

$N := \lceil -\ln(1 - \sqrt[M]{1-\delta})/(2\varepsilon^2) \rceil$
$\hat{p}_{\min} := 1$, $\hat{p}_{\max} := 0$
for $i = 1$ to M **do**
  $\mathfrak{S}_i :=$ randomly selected scheduler for M
  $truecount := 0$
  for $j = 1$ to N **do**
    $res :=$ `simulate`$(\text{ind}(M, \mathfrak{S}_i), d)$
    if $res = unknown$ **then return** *unknown*
    else if $res = true$ **then** $truecount = truecount + 1$
  end
  $\hat{p}_i := truecount/N$
  if $\hat{p}_{\max} < \hat{p}_i$ **then** $\hat{p}_{\max} := \hat{p}_i$
  if $\hat{p}_{\min} > \hat{p}_i$ **then** $\hat{p}_{\min} := \hat{p}_i$
end
return $\langle \hat{p}_{\min}, \hat{p}_{\max} \rangle$

Algorithm 22: SMC for MDP with scheduler sampling (based on [LST14])

them as techniques that significantly improve exhaustive model checking by reducing the number of states that have to be explored. In fact, the experimental section of [BCC+14] focuses on the BRTDP technique in comparison to the exhaustive model checking implementation of PRISM, where indeed significant speedups can be observed.

4.8.2 Sampling Schedulers with PRNG and Hashing

Instead of attempting to obtain an optimal scheduler by successive improvement, Legay and Sedwards [LST14] suggest to not only sample paths under some scheduler, but to also sample from the scheduler space. This is inspired by the Kearns algorithm [KMN02] that uses local sampling in order to on-the-fly determine an ε-optimal scheduler decision during simulation. However, that algorithm is used to verify *discounted* reward-based properties, and it exploits the discounting by only doing a bounded lookahead. For the verification of probabilistic reachability properties, we need to sample over the set of global schedulers for the entire state space. An outline of the corresponding approach is given as Algorithm 22.

Avoiding memory explosion for schedulers A naïve implementation would have to explicitly store the sampled schedulers, i.e. store one decision per (nondeterministic) state of the entire MDP, and thus have (worst-case) memory usage just like exhaustive model checking. The core contribution of Legay and Sedwards is to use pseudo-random number generators (PRNG) to represent schedulers instead. A PRNG is a deterministic function that, given an initial integer *seed*, produces a sequence of numbers that appears, for all practical purposes, to be random. Still, given the same seed, the same sequence is produced every time. Algorithm 22 can be modified to use PRNG for schedulers as follows [LST14]:

1. Before the start of the outer **For** loop, initialise PRNG $\mathscr{U}_{\text{seed}}$ with a random seed.
2. Replace line 4 by assigning to $\mathfrak{s}$ the next value obtained from $\mathscr{U}_{\text{seed}}$. $\mathfrak{s}$ is thus a (pseudo-)random number that represents a scheduler $\mathfrak{S}_{\mathfrak{s}}$.
3. $\mathfrak{S}_{\mathfrak{s}}$, which is used in place of $\mathfrak{S}$ in line 7, is given by

$$\mathfrak{S}_{\mathfrak{s}}(s) = \text{firstsample}(\mathscr{U}_{\text{nondet}}, \mathfrak{s} \otimes \text{hash}(s)) \mod |T(s)|$$

   if we assume some bijection between numbers in $\{0,\ldots,|T(s)|-1\}$ and transitions in $T(s)$. Here, $\text{firstsample}(\mathscr{U},u)$ initialises the PRNG $\mathscr{U}$ with seed u and returns its first value, $\text{hash} \in S \to \mathbb{Z}$ is a *hash function* that maps each state to an integer in such a way that collisions, i.e. two different states being mapped to the same value, are unlikely, and finally $\otimes$ is some operator to combine two integers, e.g. concatenation of their binary representation.

In this way, a scheduler is represented by a single integer instead of requiring memory in the size of the entire state space. However, in a typical machine implementation, a bounded domain is used for the integer values. In fact, this is necessary in order to achieve constant memory usage; otherwise, the size of the representation of the integer values would just grow with the state space. In consequence, there is a) only a bounded number of seeds and b) a nonzero probability for hash function collisions on a large enough state space. This means that the set of schedulers that can be represented by PRNG in this way is a strict subset of the set of all (memoryless) schedulers. The hope is that it is sufficiently large to contain (a good approximation of) the optimal schedulers.

No error bounds Algorithm 22 uses a statistical evaluation of the simulation results based on the APMC method as in Algorithm 7, but modified (line 1) to account for error accumulation, including the fact that in the long run, the same scheduler will be used multiple times. An alternative that uses the SPRT for properties in qualitative form with error accumulation taken into account also exists [LST14, Algorithm 4].

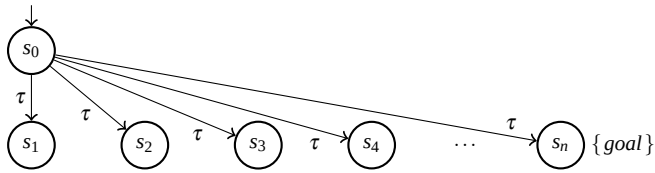


Figure 4.17: Counterexample for error bounds with scheduler sampling

Unfortunately, the resulting confidence, for example the APMC-like statement that $\mathbb{P}(|p - p_{\text{smc}}|) > \varepsilon) < 1 - \delta$, is only valid for the sampled set of schedulers. In particular, it cannot be valid for the MDP at hand in the sense of Definition 53, i.e. it cannot connect the result $\langle \hat{p}_{\min}, \hat{p}_{\max} \rangle$ to the actual probabilities $\langle [\![\mathrm{P}_{\min}(\diamond \phi)]\!]_M, [\![\mathrm{P}_{\max}(\diamond \phi)]\!]_M \rangle$ in a useful way. First of all, it may well be that no close-to-extremal scheduler is part of the encoding via PRNG and hash functions. However, the issue remains even when we assume "ideal" scheduler sampling. The underlying problem is the following: Sampling paths faithfully reproduces their probabilities (in the absence of nondeterminism), i.e. the probability to select a path via sampling is its path probability μ_M (cf. Definition 32). Schedulers, however, have no probability; they are either (near-)optimal or they are not. Taking the minimum/maximum of the probabilities computed by an arbitrary finite set of schedulers thus does not tell us anything about how close these values are to the actual probabilities.

As an example, consider the MDP M_n shown in Figure 4.17, which has $n+1$ states. We are interested in $[\![\mathrm{P}_{\max}(\diamond\, goal)]\!]$, i.e. in the maximum probability of reaching the single state s_n. This probability is obviously 1. There are n different schedulers, and exactly one of them realises that maximum probability. If we now fix parameters a, ε and δ for Algorithm 22, it returns the correct result of $\langle 0, 1 \rangle$ for M_n with probability $1 - (1 - 1/n)^a$, and $\langle 0, 0 \rangle$ otherwise (assuming uniform ideal scheduler sampling). Clearly there is no connection to ε and δ; in fact, we can select n such that $\mathbb{P}(|p - p_{\text{smc}}|) > \varepsilon) = (1 - 1/n)^a > 1 - \delta$ for any a, $\varepsilon < 1$ and $\delta < 1$. This should come as no surprise: M_n is nonprobabilistic, so for a given scheduler, every simulation run is the same. Since M_n is a tree, running Algorithm 22 on it therefore has the same effect as performing randomised testing.

The only potential way we see to solve this problem and arrive at proper error bounds is to take global information about the MDP into account as well, such as the size of the state space or the maximum number of outgoing transitions per state—just like it is done in the BRTDP and DQL techniques presented in the previous section.

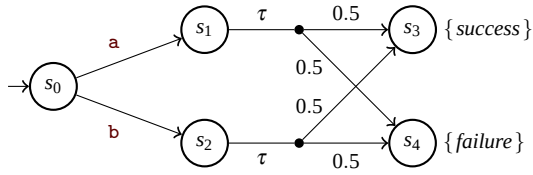


Figure 4.18: Counterexample for an exhaustive simulation approach for MDP

Performance Memory usage of this approach, using PRNG and state hashes, is the same as for SMC of DTMC: It is constant, or depends only on the way that cycle detection is performed. Runtime, however, is higher: First, the formula to compute the number of runs necessary has been changed from Algorithm 7 to Algorithm 22 (and similarly for the SPRT-based variant), and second, this number of runs needs to be performed for every one of the M sample schedulers. For example, to achieve a confidence of $\mathbb{P}(|p - p_{\text{smc}}|) > 0.015) < 0.022$ valid for a set of $M = 1000$ schedulers, 23808 simulation runs per scheduler, i.e. a total of 23808000 runs, are necessary. Achieving the same confidence for a DTMC (which additionally makes it independent of any under-the-hood process like scheduler selection) requires only about 10000 runs.

4.8.3 Failed Idea: Exhaustive SMC

Where the previous approach combines random experiments over schedulers with random path generation, an alternative could be to combine an exhaustive search over the nondeterministic choices with the randomised resolution of probabilistic decisions. Such a technique would bring together elements of stateless model checking (cf. Section 3.1.5) with the Markov chain SMC approach. Consider the MDP shown in Figure 4.18 as an example: When generating paths in the `simulate` function, we encounter the nondeterministic choice in the initial state s_0. At this point, instead of performing a confluence or partial order check or invoking a resolver, we would branch off two new simulation runs: One for the choice of action `a` and thus starting in successor state s_1, one for action `b` from successor state s_2. In case these new runs encounter more nondeterminism, new branches would again be started. The notion of a path that is generated during simulation is thus replaced by the exploration of a "simulation tree". Just like stateless exhaustive model checking, this approach would face problems of scalability for models with nondeterministic cycles, but could be expected to work well for MDPs with few but non-spurious nondeterministic choices.

Unfortunately, it does not work in the first place: How should the results of the two runs that were started from state s_0 be combined, i.e. what is the interpretation of a simulation tree w.r.t. the property at hand? In this example, four outcomes are possible: both runs end in a *success*-state, only the first or the second run does, or both runs end in *failure*. Note that each of these outcomes is observed with the same probability of $1/4$. Let us assume that the property of interest are the maximum and minimum probabilities of reaching a *success*-state. It is easy to see that the actual maximum and minimum probabilities are both $1/2$.

An obvious, but wrong, interpretation of the outcomes would be to count trees whose leaves are *all success*-states toward the minimum and trees that contain at least *one success*-state toward the maximum probability. On the long run, i.e. for many generated simulation trees, each possible outcome as described above corresponds to about $1/4$ of the trees. One of the four outcomes has all *success* leaves, while three have at least one *failure* leaf. We would thus compute a minimum probability of $1/4$ and a maximum probability of $3/4$, which would be incorrect.

We argue that there is no correct way of interpreting these simulation trees unless either all leaves happen to satisfy the state formula, or all leaves are on a loop not satisfying the formula. In these cases, the MDP was spuriously nondeterministic in the first place. Intuitively, selecting certain leaves from any other tree is equivalent to resolving the nondeterministic choices with a priori knowledge about how probabilistic choices that occur later will be resolved. Doing so is making a decision about the present with knowledge about the future, which no valid scheduler can do.

4.9 Summary and Discussion

In this chapter, we have described the model of Markov decision processes (MDP), which arise as the orthogonal combination of LTS and DTMC. As such, they support both nondeterministic and probabilistic choices. We have shown in detail how MDP can be extended with discrete variables to give rise to the symbolic model of VMDP. We then described how to specify such models with MODEST, and defined the class of probabilistic reachability properties to perform verification of MDP. After a review of the existing exhaustive model checking approaches, we turned to the problem of performing SMC for MDP, which is an area of very active research. We presented five different approaches that attempt to tackle this problem, with a particular focus on the two that are geared towards *spuriously* nondeterministic models.

approach	nondeterminism	probabilities	memory	error bounds
$\mathfrak{R}_{\text{Uni}}$	any	some	$s = 1$	none
POR [4.7.1]	spurious interleavings	max = min	$s \ll n$	unchanged
confluence [4.7.2]	spurious nonprobabilistic	max = min	$s \ll n$	unchanged
learning [HMZ+12]	any	max only	$s < n$	incorrect
learning [BCC+14]	any	max only	$s < n$	yes
schedulers [LST14]	any	max and min	$s = 1$	none

Table 4.6: SMC approaches for nondeterministic MDP models (with n states)

We need to point out again that in MDP as we defined them, it is possible to have two transitions with the same action label that originate in the same state. This is not allowed in the classical definition of MDP [Put94], and strictly speaking, the model we introduced is that of probabilistic automata (PA) as originally defined by Segala [Seg95]. However, for the purposes of this thesis, the difference between the two models is but a technicality since we use transition labels only for synchronisation in parallel composition. Any (closed network of) PA can thus be transformed into a "classical" MDP by appropriately relabelling the problematic transitions.

The contribution of this chapter lies in methods to perform SMC for MDP, in spite of the presence of nondeterministic choices in the model. Any such method has to be measured w.r.t. at least four criteria, namely 1) whether it can deal with any possible MDP or just a subclass of models, usually restricted in the way nondeterministic choices can be used; 2) what kinds of properties it can deliver sound results for, in particular whether it allows only maximum, only minimum, or both extremal probabilities to be computed; 3) how it impacts memory usage, which was constant for SMC of DTMC models; and finally 4) what kind of error bounds on the computed probabilities it delivers. We give a condensed overview of the five approaches presented in this chapter in Table 4.6 (where we measure memory usage as usual in terms of the maximal number s of states stored at any time).

The first method that we presented, which is to simply resolve the nondeterministic choices with a resolver such as $\mathfrak{R}_{\text{Uni}}$, needs no more memory than SMC for DTMC, is very fast, and cannot lead to useful or trustworthy results: it computes merely some probability out of the range between maximum and minimum. Yet, it is still used in many tools without warning the user of these consequences.

We then investigated the restricted, but often useful case of spuriously nondeterministic MDP. In such models, nondeterministic choices may be present, but they do not affect the properties that are being verified. As a consequence, the maximum and the minimum probability values for these properties coincide. The two approaches we presented to handle this subclass of MDP are core contributions of this thesis. Both are based on existing state space reduction techniques that were designed to eliminate (some) spurious nondeterministic choices to speed up exhaustive model checking, and that could be implemented in an on-the-fly manner during state space exploration. Due to the latter, we were able to adapt them to the path exploration setting of SMC. The first approach is based on linear-time partial order reduction (POR), while the second uses a check based on confluence reduction for branching-time properties.

Both techniques are overapproximations, i.e. they cannot detect all spurious choices as such, and their detection strengths are incomparable: The POR-based check is defined for networks of VMDP and takes advantage of information only available at this symbolic level, but is thus restricted to spurious interleavings, i.e. it cannot mark nondeterministic choices internal to component (V)MDP as spurious. This is due to the need for an efficient implementation of the check for (in)dependence of transitions. The confluence-based method, on the other hand, is specified on the level of the concrete state space of an MDP (which could be the semantics of a closed network of VMDP) and thus handles "internal" nondeterministic choices naturally. However, it is limited to prioritising nonprobabilistic transitions only. This restriction stems from the fact that confluence reduction also preserves branching-time properties; the branching-time version of POR has an extra condition with the same effect compared to the linear-time version we used.

In this thesis, we have limited ourselves to using the two reduction techniques with as few modifications as possible. Although they can trivially be combined sequentially during SMC, we expect that the two limitations described above can also be overcome when some changes are made to the underlying techniques: With a local independence check performed on the level of the concrete state space, the restriction of POR to interleavings could likely be removed. A notion of local independence has indeed already been defined by Hansen and Timmer in their comparison of partial order and confluence reduction [HT14a]. However, just as they show that the two reduction techniques are in fact very, very similar (although they work in the branching-time setting only), we would expect large parts of the resulting implementation for SMC to be the same. We thus consider our use of the original "standard" implementation of the independence check to be more interesting from a scientific viewpoint, especially when it comes to the performance comparison. In order

to overcome the restriction of confluence to nonprobabilistic transitions in confluent sets, the entire notion would have to be adapted to handle linear-time properties (which was not its original design goal). We see this as potential future work that in particular likely needs entirely new correctness proofs. It is important to point out that in the end, with the limitations overcome for both techniques, we would still (likely) not arrive at full trace equivalence or bisimulation checks, so the resulting approaches would still be overapproximations, that should however also offer better performance as a tradeoff.

The latter is important since we have seen in our evaluation of the two approaches' performance that the increase in runtime can be problematic for certain models. However, SMC has always been a matter of trading increased runtime (at least when a certain level of precision is desired) for reduced memory usage. While memory usage does not remain constant by adding the on-the-fly checks, it does remain very low. In combination with the fact that the error bounds for the computed probabilities are the same as in the DTMC case, i.e. in particular that it is not necessary to perform more simulation runs to arrive at the same level of confidence, the two approaches do transfer the core characteristics of SMC from DTMC to MDP rather well.

We also outlined three approaches developed by others that apply to general MDP. Out of these, the reinforcement learning technique [HMZ+12] has already been found to be flawed in such a way that it cannot lead to trustworthy results, and the method by Legay and Sedwards [LST14] that is based on PRNG and hashing does not deliver useful results in the sense that it currently does not provide any error bounds. As they stand, these two methods are thus not sound according to our Definition 53. However, the PRNG-based method is likely of use in practice since it delivers an increased subjective, informal "confidence" that the results are somewhat close to the desired extremal probabilities. The new learning-based methods using BRTDP or DQL introduced by Brázdil et al. [BCC+14] do provide (new) error bounds, they work for any MDP, and they deliver both maximum and minimum probabilities. However, in particular in the way they make use of memory, they appear to be closer to exhaustive than to statistical model checking in spirit.

It thus appears that, at the time of writing this thesis, the partial order- and confluence-based techniques, albeit restricted to spurious nondeterminism, are still the only ones for MDP that are clearly *statistical* model checking (in that they do not need persistent memory for non-local information or checks) and that at the same time provide sound results (in the sense of Definition 53).

Finally, we mention that there is also work by Lassaigne and Peyronnet on handling MDP in a simulation setting that focuses on planning problems

and infinite-state models [LP12]. As we are interested in the verification of reachability properties, we did not consider their approach in further detail.

5 Probabilistic Timed Automata

Communication protocols must work in an environment where the transmission of data takes a certain time, and timeouts are the method of choice to detect message loss. At the same time, the protocol itself should cause minimal additional delays. Electronic control systems, such as the fly-by-wire system of a modern airliner, must react to inputs quickly. For electronic trading systems, the delay between the placement and the execution of an order has an immediate monetary impact; at the same time, such a trading system must be able to keep up with a large number of orders within short time periods.

These examples show that time-related aspects of systems can play an important role for their correctness and performance. We thus need *models* that include a notion of time. We must be able to specify the time that certain actions take, and we must be able to observe the progress of time as well as the crossing of time bounds. Timed models need to be accompanied by timed *properties*: Can a process possibly terminate within a certain time limit? Will all tasks certainly finish within that time? What is the probability of at least one job not being ready at time t? Is the expected time until the message is handed to the upper protocol layers no larger than 1.5 times the wire transmission time?

Timed automata [AD94, HNSY94], or TA for short, are a widely used model for real-time systems. They extend labelled transition systems with *clock variables* (or simply *clocks*): real-valued variables that synchronously increase over time with rate 1. As in VLTS, clocks can be referred to in guards in order to make sure that a certain transition can only be taken before a time bound or after a certain delay. In assignments, they can be reset to zero, which allows the subsequent measurement of a time interval. Additionally, states can be equipped with an invariant to restrict the passage of time and enforce the execution of transitions within a specified amount of time. Timed automata are consequently suitable to capture the timing-related aspects we outlined above.

As an extension of LTS, they also allow nondeterministic choices, which are a key verification feature as we saw. However, these choices as taken over

from LTS are discrete ones: They represent decisions between certain transitions. To model uncertainty or abstraction with respect to the timed aspects of a model, we also need nondeterminism over time. Fortunately, this is already included in the TA model: all conditions on clocks can be comparisons—they need not necessarily be equality tests. We can thus, for example, use an invariant to specify that an event must happen *before* t time units (e.g. seconds) have passed, while the guard of the transition representing the event makes sure that it can happen only *after* the passage of t' time units with $t' < t$. In this way, we model a nondeterministic choice over the precise point in time at which the event happens: it can occur after any delay in the interval $[t', t]$.

Since TA were originally proposed in the early 1990s, a lot of research has been done in the area of model checking algorithms for TA, including efficient data structures and clever optimisation and approximation methods. An early implementation was the model-checking tool KRONOS [BDM+98]. Today, the modelling and analysis of TA w.r.t. real-time properties is supported by a number of tools such as RED/REDLIB [Wan06] and, most prominently, UPPAAL [BDL04].

Timed automata, however, are a nonprobabilistic model. This means that randomised algorithms or random outside influences cannot be combined with time, and the two example properties mentioned above that refer to "the probability" and "the expected time" cannot be checked. Fortunately, the way time is added to LTS to obtain TA is largely orthogonal to the way probabilities are added to obtain MDP. If we combine the two, we obtain probabilistic timed automata (PTA, [KNSS02]). We can thus see PTA as TA with probabilistic choices, or alternatively MDP with clock variables.

A family of systems that PTA as a modelling formalism are well-suited for is network protocols. As we have seen in the previous chapter, many of them employ randomisation internally to e.g. break symmetries, and they are often subject to outside disturbances that can be modelled probabilistically. At the same time, they are inherently timed: The transmission of messages, be it over wires or wirelessly and even though it occurs at near the speed of light, takes a certain amount of time. This has practical consequences, as can be seen in e.g. the maximal cable lengths allowed by different versions of the Ethernet standards. Additionally, signal processing and other tasks incur delays that need to be considered. As mentioned, timeouts can subsequently be used to improve performance or, typically, to detect (probabilistic!) message loss. Timed verification for these PTA models is necessary: The precise values used for timeouts can determine whether a protocol works correctly or not, and various time-related measures such as expected or maximum total transmission times need to be studied to evaluate its performance.

In this chapter, we first introduce the model of probabilistic timed automata (Section 5.1), including its semantics, a parallel composition operator, and its relation to the submodels of TA and MDP. We then present a variant of TA and consequently of PTA that uses *deadlines* [BS00] to impose limitations on the passage of time (Section 5.2) in place of or in addition to the location invariants of standard TA. The model originally used by the MODEST language is based on a variant of TA with deadlines, and we show a way to translate (most) of these to standard TA with invariants. After we have shown how to build PTA models in MODEST (Section 5.3), we introduce the set of properties that we focus on for the verification of PTA (Section 5.4) and summarise the existing exhaustive model checking approaches (Section 5.5). Finally, we investigate a number of ways to perform statistical model checking for PTA (Section 5.6).

Origins

The semantics for PTA presented in Section 5.1 is adapted from that given for STA in [Har10], which in turn is based on the original concrete semantics of the MODEST language given in [BDHK06], and includes updates from [HHH14].

The comparison between invariants and deadlines as used in MODEST plus the translation presented in Section 5.2 are based on work done by the author and originally published in [Har10].

The formal statements for probabilistic timed reachability as well as the definition of reward-based properties for PTA given in Section 5.4 are adapted from [HHH14] (where they are defined for stochastic timed automata).

Section 5.5 is based on the original overview of PTA model checking approaches that was compiled by the author for his Master's thesis and the paper that originally introduced mcpta [HH09]. It has been updated and extended with details from the overview paper on PTA model checking by Normal et al. [NPS13].

Section 5.6.3 summarises work done by Alexandre David et al. [DLL+11b, DLL+11a] and an investigation of this work performed by Markus Hoffmann as part of his Bachelor's thesis [Hof13], which was supervised by the author.

In Section 5.7, we study an example that has been developed by the author for his Master's thesis and that was first mentioned in the corresponding paper on the mcpta tool [HH09]. The comparison of analyses using mctau, mcpta as well as modes was performed by the author and first appeared in the tool demonstration paper on mctau [BDHH12]. We additionally include analysis results using the mcsta tool, which we describe in more detail in the next chapter, here.

5.1 Definition

PTA deal with time through *clock variables*, often simply called *clocks*. Their domain is the set of nonnegative real numbers $\mathbb{R}_0^+$. They all advance synchronously at the same rate over time, which we will formally see in the definition of the semantics of PTA later. For now, we need some specialised notation for clock variables, valuations and expressions before we can define the model of PTA.

Clocks, valuations and clock constraints Given a set of clock variables $\mathscr{C}$, the valuation $\mathbf{0}_{\mathscr{C}} \in \mathrm{Val}(\mathscr{C})$, or simply $\mathbf{0}$ when $\mathscr{C}$ is clear from the context, assigns zero to every clock $c \in \mathscr{C}$. If $v \in \mathrm{Val}(\mathscr{C})$ and $t \in \mathbb{R}_0^+$, then we denote by $v+t$ the valuation where all clock variables have been incremented by t. (If v is a general valuation, then only the clock variables are incremented and all other variables remain unchanged.) *Clock constraints* over a set of clocks $\mathscr{C}$ are expressions of the form

$$\mathscr{CC}(\mathscr{C}) ::= \mathit{true} \mid \mathit{false} \mid \mathscr{CC} \wedge \mathscr{CC} \mid \mathscr{CC} \vee \mathscr{CC} \mid c \sim x \mid c_1 - c_2 \sim x$$

where $\sim \in \{>, \geq, <, \leq, =, \neq\}$, $c, c_1, c_2 \in \mathscr{C}$ and $x \in \mathbb{R}$. As usual, we just write $\mathscr{CC}$ for the set of clock constraints when $\mathscr{C}$ is clear from the context. If all $x \in \mathbb{R}$ are actually integers, we have an *integer* clock constraint. The last case, where two clock variables are compared, is called a *diagonal*, and a clock constraint that does not contain diagonals is *diagonal-free*. A clock constraint that is built using only the last two cases is *simple*; if it does not contain any disjunctions (fourth case), it is *basic*, and if it contains neither disjunctions nor conjunctions, it is *atomic*. The set of simple clock constraints is denoted $\mathscr{CC}^S$, that of basic ones $\mathscr{CC}^B$. If all comparison operators $\sim$ used in a clock constraint are in $\{\geq, \leq, =\}$, it is a *closed* clock constraint. We treat clock constraints as expressions in *Bxp* and thus reuse the corresponding notation, e.g. writing $[\![e]\!](v)$ for $v \in \mathit{Val}$ and $e \in \mathscr{CC}$ to denote the value of e evaluated in v.

We can now formally define probabilistic timed automata:

Definition 59 (PTA)**.** A *probabilistic timed automaton* (PTA) is an 8-tuple

$$\langle \mathit{Loc}, \mathscr{C}, A, E, l_{\mathit{init}}, \mathit{Inv}, \mathit{AP}, L \rangle$$

where
– *Loc* is a countable set of locations,
– $\mathscr{C}$ is a finite set of clock variables,
– $A \supseteq \{\tau\}$ is the automaton's countable alphabet,

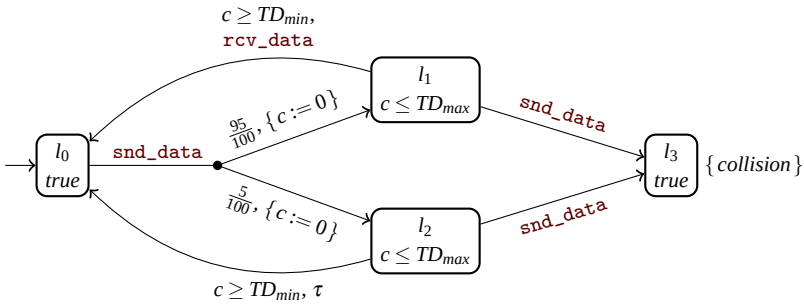


Figure 5.1: PTA model of a lossy comm. channel with collision detection

– $E \in Loc \to \mathscr{P}(\mathscr{CC} \times A \times \mathrm{Dist}(\mathscr{P}(\mathscr{C}) \times Loc))$ is the automaton's edge function, which maps each location to a set of edges, which in turn consist of a guard, a label and a probability distribution over sets of clocks to reset to zero and target locations,
– $l_{init} \in Loc$ is the initial location,
– $Inv \in Loc \to \mathscr{CC}$ is the invariant function, which maps each location to a clock constraint that allows time to pass as long as it evaluates to *true*,
– AP is a set of atomic propositions, and
– $L \in Loc \to \mathscr{P}(AP)$ is the location labelling function.

The usual notation that we already used for previous models, such as also using arrows to denote edges, can be applied to PTA analogously. Let us now illustrate the capabilities of PTA by extending the communication protocol component models from previous examples.

Example 28. In Example 14 in the previous chapter, we introduced an MDP that models a lossy communication channel with collision detection. A modelling artifact caused by the use of an untimed formalism was that message loss was explicitly signalled to the sender via a `timeout` action, for there was no other way to make it observable. Now, with PTA, we can use clocks to model transmission delays and the detection of timeouts in a more realistic way.

Figure 5.1 shows the updated model for the communication channel. The guards are shown on the transitions before the action label, the clock resets are represented by standard assignments that set the affected clocks to zero, and the invariants are given inside the locations, below the location name. Instead of `timeout`, the label of the edge that is taken when a message is lost is now simply τ. However, a new clock c has been introduced that is used to make the transmission of a message take between TD_{min} and TD_{max} time units, modelling

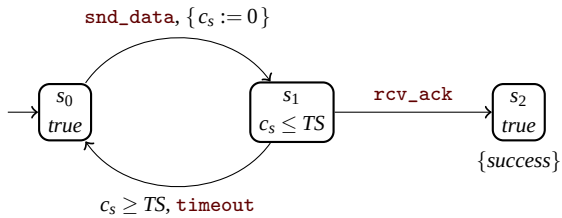


Figure 5.2: PTA model of the simple sender with timeout detection

a nondeterministic transmission delay. Guards on the edges going back to the initial location from l_1 and l_2 ensure the lower bound, while the invariants of those locations are used to obtain the desired upper bound. If, for example, this abstract channel was implemented using some variant of Ethernet, we could calculate minimum and maximum transmission delay from the allowed cable lengths and transmission rates. In that case, we would use the nondeterministic choice of delay to capture the fact that we do not know the precise length of cable that will actually be used, but want the protocol to be correct no matter how long it is.

Now that the channel no longer reports message losses explicitly, we need to modify the model of the sender, originally given in Example 2, to observe a timeout. The corresponding PTA is shown in Figure 5.2. In location s_1, we now wait exactly *TS* time units before concluding that the message or its acknowledgment has been lost. If the acknowledgment arrives during that time, we go to the *success*-state s_2 as before. Observe that our use of guards and invariants leads to a deterministic delay for the edge labelled `timeout` (which is no longer synchronising). However, as the guard of the edge labelled `rcv_ack` is *true* (and thus omitted), there is a nondeterministic choice between `rcv_ack` and `timeout` in case the acknowledgment is available exactly *TS* time units after the message was sent. Note that this cannot easily be avoided: If we were to change the invariant of s_1 to $c_s < TS$, then the guard of the `timeout`-labelled edge could never become *true*. We could well set the guard of the `rcv_ack`-labelled edge to $c_s < TS$ to give timeouts priority over the receipt of an acknowledgment for this corner case—but why should acknowledgments always be discarded in that situation?

Semantics

So far, we have relied on an intuitive understanding of the timing behaviour of a PTA that is induced by its clocks, guards and invariants. As this is clearly

not sufficient for formal modelling and verification, we now give a precise semantics for PTA. Since they are a real-time model where clocks take values in $\mathbb{R}_0^+$, the semantics of a PTA is an uncountably infinite object: a timed probabilistic transition system.

Definition 60 (TPTS)**.** A *timed probabilistic transition system* (TPTS) is a 7-tuple

$$\langle S, \Sigma_S, A, T, s_{init}, AP, L \rangle$$

where
– S is a (usually uncountable) set of states with an associated σ-algebra Σ_S,
– $A = \mathbb{R}^+ \uplus A'$ is the system's (uncountable) alphabet that can be partitioned into delays in $\mathbb{R}^+$ and normal actions in $A' \supseteq \{\tau\}$,
– $T \in S \to \mathscr{P}(A \times \text{Prob}(S, \Sigma_S))$ is the transition function, which is explicitly allowed to map a state to an uncountable set of transitions,
– $s_{init} \in S$ is the initial state,
– AP is a set of atomic propositions, and
– $L \in S \to \mathscr{P}(AP)$ is the state labelling function
where, for every delay-labelled transition $\langle s, x, \mu \rangle$, $x \in \mathbb{R}^+$, the following three conditions hold:
– $\exists s' \in S\colon \mu = \mathscr{D}(s')$,
– *time determinism*: $\langle x, \mu' \rangle \in T(s) \Rightarrow \mu = \mu'$, and
– *additivity*: $\exists x' \in \mathbb{R}^+\colon \langle x + x', \mathscr{D}(s') \rangle \in T(s)$
$\quad \Leftrightarrow \exists s'' \in S\colon \langle x, \mathscr{D}(s'') \rangle \in T(s) \wedge \langle x', \mathscr{D}(s') \rangle \in T(s'')$.

TPTS are a special case of the model of nondeterministic labelled Markov processes [Wol12]. They can also be seen as uncountably infinite-state, uncountably-branching MDP. Most MDP notions, such as paths and traces, can thus directly be transferred to TPTS. The semantics of PTA in terms of TPTS can now be defined as follows:

Definition 61 (Semantics of PTA)**.** The semantics of a PTA

$$M = \langle Loc, \mathscr{C}, A, E, l_{init}, Inv, AP, L \rangle$$

is the TPTS

$$[\![M]\!] = \langle Loc \times \text{Val}(\mathscr{C}), \mathscr{P}(Loc) \otimes \bigotimes_{c \in \mathscr{C}} \mathscr{B}(\mathbb{R}_0^+), \mathbb{R}^+ \uplus A, T_M, \langle l_{init}, \mathbf{0}_{\mathscr{C}} \rangle, AP, L_M \rangle$$

where $L_M(\langle l, v \rangle) = L(l) \cup \{ e \in AP \cap \mathscr{C}^S \mid [\![e]\!](v) \}$ and T_M is the smallest function such that the following two inference rules are satisfied:

$$\frac{l \xrightarrow{g,a}_E \mu \quad [\![g]\!](v)}{\langle l, v \rangle \xrightarrow{a}_{T_M} \mu_M^v} \ (\textit{jump}) \qquad \frac{t \in \mathbb{R}^+ \quad \forall t' \leq t\colon [\![Inv(l)]\!](v + t')}{\langle l, v \rangle \xrightarrow{t}_{T_M} \mathscr{D}(\langle l, v + t \rangle)} \ (\textit{delay})$$

where, for $l' \in Loc$ and measurable T, we have the probability measure μ_M^v for the discrete distribution μ defined by

$$\mu_M^v(T) = \begin{cases} \mu(l', X) & \text{if } \langle l', v[X \mapsto 0] \rangle \in T \\ 0 & \text{otherwise} \end{cases}$$

again using the correspondence between valuations and $|\mathscr{C}|$-tuples.

The premise of rule *delay* is also called the *time progress condition* for PTA; it ensures that time can pass if and only if the current location's invariant is satisfied. Observe that states $\langle l, v \rangle$ where $[\![Inv(l)]\!](v)$ does not hold can be reachable in the semantics of a PTA: An invariant prevents time from progressing any further, but there is nothing that prevents a transition according to rule *jump* from going into a state where the invariant is violated. We thus use a *weak invariant semantics*, whereas TA are usually (e.g. when using UPPAAL) specified with a *strong* invariant semantics: a transition that would lead into a state where the invariant is violated simply does not exist. The strong semantics thus constitutes a "one-step lookahead" to prevent violated invariants. Such a semantics would be problematic for PTA, where edges lead into probability distributions: If an invariant is violated in one but not all of the target states, should the entire transition be dropped? Or should the "remaining" probabilities be renormalised? To avoid this problem, one usually only considers well-formed PTA:

Definition 62 (Well-formed PTA)**.** A PTA $M = \langle Loc, \mathscr{C}, A, E, l_{init}, Inv, AP, L \rangle$ is *well-formed* if for all $l \in Loc$ and $v \in \mathrm{Val}(\mathscr{C})$, we have $[\![Inv(l_{init})]\!](\mathbf{0}_{\mathscr{C}})$, and for all $\langle g, a, \mu \rangle \in E(l)$ and $\langle X, l' \rangle \in \mathrm{support}(\mu)$, we have

$$[\![Inv(l)]\!](v) \wedge [\![g]\!](v) \Rightarrow [\![Inv(l')]\!](v[X \mapsto 0]).$$

In a well-formed PTA, the invariant of a location is thus implied by the guards of all incoming edges. As we use a weak invariant semantics in this thesis, we do not need to restrict to well-formed PTA.

Independently of well-formedness, the semantics of a PTA can contain deadlock and *timelock* states. A deadlock state is one that has no outgoing transitions according to inference rule *jump*, while a timelock state has no outgoing transitions at all. Note that deadlock is thus a prerequisite for a timelock state. In a timelock state, time simply stops—a very unrealistic behaviour. Timelocks therefore typically indicate a modelling error and should be fixed before actual verification is started.

Parallel Composition

As PTA are part symbolic (they use expressions for guards and invariants) and part explicit (using concrete instead of symbolic probability distributions), the

definition of the parallel composition operator for PTA combines aspects we know from the corresponding operators for both MDP and VMDP:

Definition 63 (Parallel composition of PTA)**.** The *parallel composition of two PTA* $M_i = \langle Loc_i, \mathscr{C}_i, A_i, E_i, l_{init_i}, Inv_i, AP_i, L_i \rangle$, $i \in \{1,2\}$, is the PTA

$$M_1 \parallel M_2 = \langle Loc_1 \times Loc_2, \mathscr{C}_1 \cup \mathscr{C}_2, A_1 \cup A_2, E, \langle l_{init_1}, l_{init_2} \rangle, Inv, AP_1 \cup AP_2, L \rangle$$

where

$$E \in (Loc_1 \times Loc_2) \to \mathscr{P}(\mathscr{CC} \times (A_1 \cup A_2) \times \mathrm{Dist}(\mathscr{P}(\mathscr{C}) \times (Loc_1 \times Loc_2)))$$

is such that $\langle g, a, \mu \rangle \in E(\langle l_1, l_2 \rangle)$

$$\begin{aligned} &\Leftrightarrow a \notin B \wedge \exists \mu_1 \colon \langle g, a, \mu_1 \rangle \in E_1(l_1) \wedge \mu = \mu_1 \times \mathscr{D}(\langle \varnothing, l_2 \rangle) \\ &\vee\ a \notin B \wedge \exists \mu_2 \colon \langle g, a, \mu_2 \rangle \in E_2(l_2) \wedge \mu = \mathscr{D}(\langle \varnothing, l_1 \rangle) \times \mu_2 \\ &\vee\ a \in B \wedge \exists g_1, g_2, \mu_1, \mu_2 \colon \\ &\qquad \langle g_1, a, \mu_1 \rangle \in E_1(l_1) \wedge \langle g_2, a, \mu_2 \rangle \in E_2(l_2) \\ &\qquad \wedge\ (g = g_1 \wedge g_2) \wedge (\mu = \mu_1 \times \mu_2) \end{aligned}$$

with $B = (A_1 \cap A_2) \setminus \{\tau\}$, the new invariant function *Inv* is defined by

$$Inv(\langle l_1, l_2 \rangle) = Inv(l_1) \wedge Inv(l_2)$$

and the location labelling function L maps to the union of the labels of the original states:

$$L(\langle l_1, l_2 \rangle) = L(l_1) \cup L(l_2).$$

Observe that we allow global clock variables, but since clocks can only be reset to zero, there is no need to require any kind of consistency of the component PTA.

Example 29. The parallel composition of the channel and sender PTA from the previous example is shown in Figure 5.3. In locations $\langle l_1, s_1 \rangle$ and $\langle l_2, s_1 \rangle$, we see that the invariant function now assigns the conjunction of the component invariants. The automaton is also very different from what we saw in Figure 4.3 for the corresponding MDP example. The difference is due to the fact that there is no synchronisation on the `timeout` action anymore; instead, the timeout to detect message loss is explicitly modelled with clocks, guards and invariants. However, whether the sender detects a timeout even in case that the message is not lost depends on the value of *TS* in relation to the transmission delays of this channel and the one for acknowledgments. Therefore, the parallel composition contains edges for all possibilities. These edges for *premature timeouts* will even be present if we were to look at the overall network of PTA semantics including receiver and acknowledgment channel. Only in the TPTS semantics of that overall parallel composition will they not be reachable if *TS* is chosen correctly. Finding out whether this is the case is clearly a verification task.

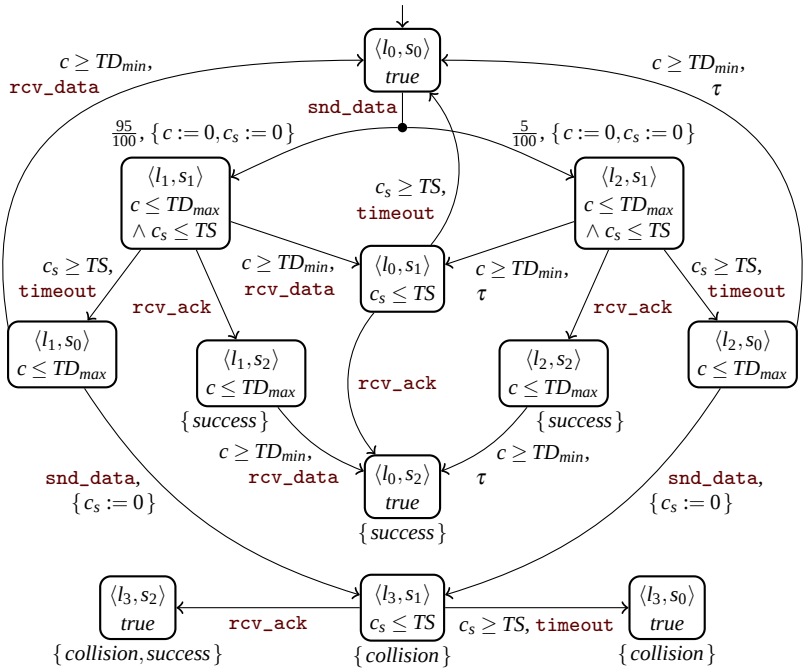


Figure 5.3: PTA for the parallel composition of channel and sender

Submodels

As mentioned, PTA can be seen as timed MDP or alternatively as TA with probabilistic choices. Formally, these relationships can be captured as follows:

Markov decision processes Clearly, as MDP do not include real-time features, a PTA can only be an MDP when it does not use clock variables, or at least does not use them in any nontrivial way. At the same time, there is a significant difference between the semantics of a state with some outgoing transitions in an MDP and a similar location in a PTA: In the MDP, we assume that at some point, one of the transitions will be taken. Whether this happens in the PTA depends on the location's invariant: Only if that evaluates to *false* at some point does an edge need to be taken. Otherwise, time can also pass up to infinity while all of the available (and enabled) edges are ignored. Whether this happens is a nondeterministic choice. While obtaining an equivalent PTA

for a given MDP is straightforward, the opposite direction is thus not so clear: For a PTA to be an MDP, we could

– require the set of clocks to be empty and the invariant function to map all locations to invariant *false* (taking advantage of the weak invariant semantics), or
– allow clock variables, but require the invariant function to map all locations to an invariant of the form $c \leq 0 \wedge e$, where c is some clock (that could be different in each state) and e is a clock constraint, or
– allow clock variables and invariants of the forms *true*, *false* and $c \leq 0 \wedge e$ as above and require all guards to be *true* or *false*; in this case, the state in the MDP corresponding to a location with invariant *true* would have a τ-labelled loop to represent the (unbounded) passage of time.

Of course, these are just three examples; one can surely find many more classes of PTA that have a corresponding equivalent MDP. For now, we settle for the simplest definition that will be useful later on.

Definition 64 (Untimed PTA)**.** A given PTA $\langle Loc, \mathscr{C}, A, E, l_{init}, Inv, AP, L\rangle$ is *untimed* if $\mathscr{C} = \varnothing$ and for all $l \in Loc$, we have $Inv(l) = false$ and $\langle g, a, \mu\rangle \in E(l) \Rightarrow g = true$.

This restricted class of PTA can then be shown to be equivalent to MDP:

Proposition 4. An untimed PTA $\langle Loc, \varnothing, A, E, l_{init}, Inv, AP, L\rangle$ is isomorphic to the MDP $\langle Loc, A, T, l_{init}, AP, L\rangle$ where $\langle a, \mu\rangle \in T(l) \Leftrightarrow \langle true, a, \mu_{\varnothing}\rangle \in E(l)$ for all $l \in Loc$ where $\mu_{\varnothing}(\langle C, l\rangle) = \mu(l)$ if $C = \varnothing$ and 0 otherwise. Additionally, the semantics of this PTA is isomorphic to the PTA itself. We say that an untimed PTA *is* an MDP, and an MDP *is* an untimed PTA.

The isomorphism between untimed PTA and their semantics is because all invariants are *false*, thus the premise of inference rule *delay* of Definition 61 is never fulfilled.

Timed automata If, on the other hand, we place no restrictions on clocks and clock constraints but require all probability distributions of the edge function to be Dirac, we obtain nonprobabilistic PTA.

Definition 65 (Nonprobabilistic PTA)**.** A PTA $\langle Loc, \mathscr{C}, A, E, l_{init}, Inv, AP, L\rangle$ is nonprobabilistic if for all $l \in Loc$, we have that

$$\langle g, a, \mu\rangle \in E(l) \Rightarrow \exists l' \in Loc, C \in \mathscr{P}(\mathscr{C}) \colon \mu = \mathscr{D}(\langle C, l'\rangle).$$

In order to establish a relationship between (nonprobabilistic) PTA and timed automata, we first need to define what a TA actually is.

Definition 66 (TA)**.** A *timed automaton* (TA) is an 8-tuple

$$\langle Loc, \mathscr{C}, A, E, l_{init}, Inv, AP, L \rangle$$

where
– Loc is a countable set of locations,
– $\mathscr{C}$ is a finite set of clock variables,
– $A \supseteq \{\tau\}$ is the automaton's alphabet,
– $E \in Loc \to \mathscr{P}(\mathscr{CC} \times A \times \mathscr{P}(\mathscr{C}) \times Loc)$ is the edge function, which maps each location to a set of edges, which in turn consist of a guard, a label, a set of clocks to reset and a target location,
– $l_{init} \in Loc$ is the initial location,
– $Inv \in Loc \to \mathscr{CC}$ is the invariant function,
– AP is a set of atomic propositions, and
– $L \in Loc \to \mathscr{P}(AP)$ is the location labelling function.

Observe that this definition is identical to the one for PTA except for the edge function, which does not include probability distributions. The semantics of TA is given in terms of *timed transition systems* (TTS), which are similarly just the nonprobabilistic version of TPTS. As the formal definitions of TTS and the semantics of TA would be repetitions of most of definitions 60 and 61, just like the definition of TA repeats most of the definition of PTA, we omit them.

Proposition 5. A nonprobabilistic PTA $\langle Loc, \mathscr{C}, A, E, l_{init}, Inv, AP, L \rangle$ is isomorphic to the TA $\langle Loc, \mathscr{C}, A, E', l_{init}, Inv, AP, L \rangle$ where

$$\langle g, a, C, l' \rangle \in E'(l) \Leftrightarrow \langle g, a, \mathscr{D}(C, l') \rangle \in E(l)$$

for all $l \in Loc$. Additionally, the semantics of the PTA and the TA are isomorphic. We say that a nonprobabilistic PTA *is* a TA, and a TA *is* a nonprobabilistic PTA.

Example 30. The PTA modelling the simple communication protocol sender as shown in Figure 5.2 is a TA.

While we can directly model nonprobabilistic PTA, it is also useful to consider the automaton that results from the replacement of the probabilistic choices of a given PTA by nondeterministic ones:

Definition 67 (Nonprobabilistic overapproximation of PTA)**.** The *nonprobabilistic overapproximation* of a PTA $\langle Loc, \mathscr{C}, A, E, l_{init}, Inv, AP, L \rangle$ is the nonprobabilistic PTA $\langle Loc, \mathscr{C}, A, E', l_{init}, Inv, AP, L \rangle$ where, for all $l, l' \in Loc$ and $C \in \mathscr{P}(\mathscr{C})$, we have that

$$\langle g, a, \mu \rangle \in E(l) \wedge \langle C, l' \rangle \in \text{support}(\mu) \Leftrightarrow \langle g, a, \mathscr{D}(C, l') \rangle \in E'(l).$$

Within the MODEST TOOLSET, the mctau tool that connects to UPPAAL for exhaustive model checking of TA can generate and check the nonprobabilistic overapproximation of a PTA. In this way, it is able to successfully verify properties that query for a probability that is 0 or 1, or use a bound of 0 or 1 [BDHH12], for certain models.

Countable TPTS as MDP We mentioned earlier that TPTS can be seen as uncountably infinite-state, uncountably branching MDP. If we turn the restriction around, we can say that a TPTS with a countable set of states and a countable set of transitions actually *is* an MDP:

Proposition 6. If $M = \langle S, \Sigma_S, A, T, s_{init}, AP, L \rangle$ is a finitely branching TPTS where S is a countable set and for all $s \xrightarrow{a} \mu$ we have that the probability measure μ is equivalent to a probability distribution μ', then M is isomorphic to the MDP $\langle S, A_M, T_M, s_{init}, AP, L \rangle$ where T_M behaves exactly like T except that it maps to μ' where T maps to μ, and $A_M = \{ a \in A \mid \exists s \in S, \mu \in \mathrm{Dist}(S) \colon \langle a, \mu \rangle \in T(s) \}$. We say that a TPTS as above *is* an MDP, and an MDP *is* a TPTS.

Observe that the MDP's alphabet A_M is countable by definition because S is countable and M is finitely branching.

Variables

The extension of PTA with discrete variables to obtain the model of VPTA follows the same basic recipe as the extension of MDP to VMDP. We thus omit the details here and merely point out that

– guards and invariants are replaced by expressions that follow the grammar for clock constraints with an additional case that allows the inclusion of clock-free Boolean expression in *Bxp*,
– the concrete probability distributions on edges are replaced by symbolic ones, and
– instead of sets of clocks to reset, general updates are used, with the restriction that clocks can only be assigned to 0.

A parallel composition operator for consistent VPTA can then be defined in the straightforward way, too.

5.2 Deadlines

In standard PTA as defined in the previous section, the passage of time is constrained by location invariants: clock constraints associated to locations that allow time to pass as long as they are satisfied. However, invariants may easily

lead to undesired timelocks, in particular in combination with parallel composition, and they cannot be used to represent certain forms of synchronisation [Góm09].

In timed automata with deadlines [BS00], expressions associated to the edges control the passage of time. That makes it possible to avoid the problems hinted at above and, for example, build models that are timelock-free by construction. Several different ways to compose the clock constraints in a parallel composition are possible. Of particular note is that deadlines allow *as soon as possible* (ASAP) synchronisation of edges, which is useful when two edges in different components need to synchronise because their actions are part of the synchronisation alphabet, but they have different guards that enable them after different points in time. ASAP synchronisation allows time to progress until both edges' guards are enabled, i.e. one edge waits for the other, without introducing timelocks.

Example 31. Let us take another look at the PTA modelling BRP sender and receiver introduced in Example 28 and their parallel composition as shown in Example 29. When both automata are in their respective initial locations, data can be sent via action `snd_data`. However, there is no invariant to enforce that the corresponding edge is actually taken, so we can also wait indefinitely. We could try to avoid this problem by assigning invariant *false* to location l_0. This, however, causes problems in the parallel composition: The invariants of all states of the form $\langle l_0, \cdot \rangle$ would also become *false*, even if they do not have an outgoing edge labelled `snd_data`. One of these cases is $\langle l_0, s_1 \rangle$, where `rcv_ack` would thus effectively get priority over `timeout`. This is clearly an undesirable effect, so we cannot use an invariant in this way to avoid the problem of infinite waiting. What we really would like is to specify that data is sent *as soon as possible*, i.e. no time is spent waiting whenever the sender wants to perform `snd_data` *and* the channel is ready to synchronise on it. Product locations where `snd_data` is not an option should not be affected. This can be achieved with deadlines.

PTA as defined in the previous section exclusively use invariants to control the passage of time. If we wanted to use deadlines instead, we would modify the PTA model of Definition 59 as follows: In addition to its guard, every edge carries a deadline as a second clock constraint, and the invariant function is dropped from the tuple defining the PTA. We use the letter d to refer to deadlines in the same way that we refer to guards as g. In the semantics (Definition 61), the premise of inference rule *delay* is replaced by the time progress condition for deadlines [BDHK06], which states that $t > 0$ time units can pass

in state $\langle l, v\rangle$ if

$$\forall t' < t : [\![\neg \bigvee_{\langle g,d,a,\mu\rangle \in E(l)} d]\!](v + t')$$

holds. Contrast this to the time progress condition for PTA with invariants, which requires that

$$\forall t' \leq t : [\![Inv(l)]\!](v + t').$$

A fine point here is the use of $<$ instead of $\leq$ in the first quantification for deadlines. This has some interesting consequences, for example that the deadlines $c > x$ and $c \geq x$ are equivalent, and affects the expressivity of deadlines in general.

In parallel composition (Definition 63), the invariant of a product location is defined as $Inv(l_1) \wedge Inv(l_2)$, i.e. using the logical conjunction operator, which is the only sensible choice. In contrast, a significant advantage of deadlines is the flexibility to use any of a number of different operators for this purpose: The parallel composition of two PTA with deadlines is largely analogous to Definition 63, except that the invariant function is left out and the new edge function is defined by

$$\begin{aligned}\langle g,d,a,\mu\rangle \in E(\langle l_1,l_2\rangle) \Leftrightarrow\ & a \notin B \wedge \langle g,d,a,\mu_1\rangle \in E_1(l_1) \wedge \mu = \mu_1 \times \mathscr{D}(\langle \varnothing, l_2\rangle) \\ & \vee\ a \notin B \wedge \langle g,d,a,\mu_2\rangle \in E_2(l_2) \wedge \mu = \mathscr{D}(\langle \varnothing, l_1\rangle) \times \mu_2 \\ & \vee\ a \in B \\ & \quad \wedge \langle g_1,d_1,a,\mu_1\rangle \in E_1(l_1) \wedge \langle g_2,d_2,a,\mu_2\rangle \in E_2(l_2) \\ & \quad \wedge (g = g_1 \wedge g_2) \wedge (d = d_1 \otimes d_2) \wedge (\mu = \mu_1 \times \mu_2)\end{aligned} \tag{5.1}$$

where $\otimes$ is a binary operator on clock constraints. A number of useful operators for $\otimes$ have been proposed [BS00], in particular in combination with variations of the operator used for guards. In the remainder of this thesis, we focus only on $\otimes \in \{\wedge, \vee\}$, i.e. logical conjunction and disjunction, since these are the operators available in the MODEST language.

In MODEST, it is the edge label that decides which of the two is used for a synchronising edge: If it is a *patient* action, conjunction is used; if the action is *impatient*, disjunction. Consequently, the alphabet of a PTA with deadlines that is the semantics of a MODEST model contains distinguished subsets of patient actions *PAct* and impatient actions *IAct* with $PAct \cap IAct = \varnothing$.

Observe that deadlines and invariants are orthogonal features, so we can also define the model of PTA with deadlines *and* invariants by just *extending* Definition 59 with deadlines on the edges, without dropping its invariant-related parts. The same can be done for parallel composition, and the semantics of PTA with deadlines and invariants uses the conjunction of the two time progress conditions as its premise for inference rule *delay*. While MODEST was

deadline	$c > x$	$c \geq x$	$c < x$	$c \leq x$	$c_1 - c_2 \sim x$	$c \neq x$	$c = x$	$c \leq x \wedge c \geq x$
invariant	$c \leq x$	$c \leq x$	$c \geq x$	$c > x$	$c_1 - c_2 \widetilde{\sim} x$	$c = x$	–	$\lightning$

Table 5.1: Converting between deadlines and invariants

originally specified with deadlines only [BDHK06], the language in its current form supports both deadlines and invariants [HHHK13].

Example 32. We can now solve the problem presented in the previous example: We use deadlines to make sender and channel synchronise on `snd_data` *as soon as possible*. We thus obtain models that are PTA with deadlines and invariants. There are two ways to do this:

a) We make `snd_data` an impatient action, set the deadline of the corresponding edge in the sender to *true*, and set all other edges' deadlines (including the `snd_data` edges in the channel) to the default value *false*.
b) We make `snd_data` a patient action, set the deadlines of all edges labelled `snd_data` in both automata to *true*, and set all other edges' deadlines to *false*.

While both ways work equally well, using an impatient action in the sender and not changing the channel appears slightly more natural: It more directly captures the intuition of the sender being the active part that initiates the sending while the channel just passively waits for input from other components.

Expressivity

Due to the use of $<$ in the time progress condition for deadlines, certain deadlines cannot be expressed as invariants and vice-versa. Table 5.1 summarises the differences: We already saw that the deadlines $c > x$ and $c \geq x$ are equivalent. A location l with a single outgoing edge with such a deadline can be equivalently represented by $Inv(l) = c \leq x$. However, the invariant $c < x$ cannot be represented by any deadline, because deadlines cannot prevent the time point $c = x$ from being reached from below due to the use of $<$. For $c < x$, $c \leq x$ and all instances of $c_1 \sim c_2$, the translation between deadline and invariant is straightforward and simply based on the differences in the time progress conditions.

$c = x$ and $c \neq x$ are interesting cases: The deadline $c \neq x$ is equivalent to the invariant $c = x$, but both the deadline $c = x$ and the invariant $c \neq x$ do not have a corresponding equivalent. The reason for the latter is the same as for the invariant $c < x$, while the deadline $c = x$ is a curious case: Starting at a point in time where $c \leq x$, the deadline $c = x$ allows time to progress just until $c = x$ is

reached, but no further. Starting at $c > x$, time progress is not constrained. This behaviour cannot be represented by an invariant: In order to prevent time from passing when $c = x$, there must be an $\varepsilon > 0$ such that the open interval $(x, x+\varepsilon)$ is not included in the invariant, but this would prevent time from passing when already $c > x$, for example when $c = x + \frac{1}{2}\varepsilon$.

The first seven columns of Table 5.1 define a function $Conv \in \mathscr{CC} \to \mathscr{CC}$ to convert atomic clock constraints from deadlines to equivalent invariants (*true* and *false* are simply negated, and $\backsim$ denotes the "opposite" of the corresponding operator $\sim$, e.g. $<$ for $\geq$). This function can be lifted to general clock constraints by applying it to the atomic subexpressions and flipping the Boolean operators ($\wedge \mapsto \vee, \vee \mapsto \wedge$). However, this lifting fails if the deadline $c = x$ is encoded in an indirect way as shown in the table's last column:

$$Conv(c \leq x \wedge c \geq x) = c > x \vee c \leq x = true,$$

but the deadline $c \leq x \wedge c \geq x$ cannot be represented as an invariant.

From Invariants to Deadlines

A PTA with invariants that can all be expressed as deadlines can be transformed into an automaton with deadlines by simply adding a permanently disabled loop $l \xrightarrow{ff, \neg Inv(l), \tau} \mathscr{D}(\langle \varnothing, l \rangle)$ to every location l and setting the deadline of all other edges to *false*. This makes use of the fact that disabled guards do not influence the effect of deadlines. If a strong invariant semantics were desired, the guards of all of the other, original edges from locations l would need to be set to the conjunction $g \wedge \bigwedge_{\langle C, l' \rangle \in \text{support}(\mu)} Inv(l')$ to prevent any edge from being taken when the invariant is violated, where g is the edge's guard and μ its probability distribution in the original PTA with invariants. This transformation is also correct for a network of automata when performed for each of the components independently. A more detailed explanation and a correctness proof can be found in [BDHK06, Section VI].

From Deadlines to Invariants

For a single PTA whose deadlines are all (and, to avoid the "obfuscation" problem with *Conv* shown above, in combination) expressible as invariants, the transformation into a PTA with weak invariants is straightforward and follows from the time progress conditions: Set $Inv(l) = \bigwedge_{\langle g,d,a,\mu \rangle \in E(l)} Conv(d)$ and keep everything else as-is. Due to the flexible handling of deadlines in parallel composition with patient and impatient actions, however, transforming a network of PTA with deadlines into a network of PTA with invariants is more complicated.

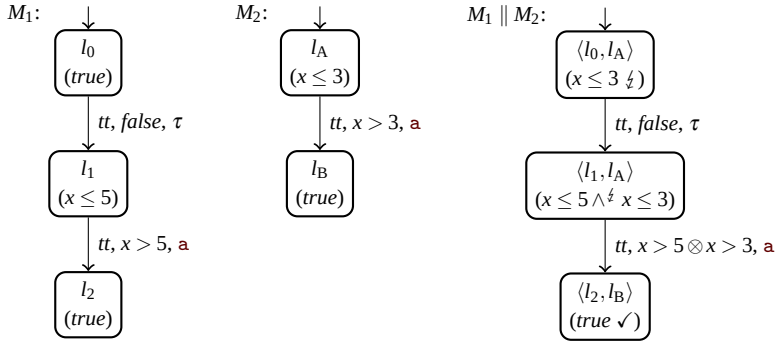


Figure 5.4: Converting deadlines to invariants compositionally (failed attempt)

Example 33. Figure 5.4 shows two PTA with deadlines M_1 and M_2 and their translation into invariants according to *Conv* (in brackets inside the locations). If we just perform a componentwise transformation of deadlines to invariants as described above, the resulting parallel composition is wrong, as shown on the rightmost automaton: The deadline $x > 3$ should not have an effect on location $\langle l_0, l_A \rangle$, and if a is a patient action (i.e. $\otimes = \wedge$), then the invariant of location $\langle l_1, l_A \rangle$ is also incorrect—it should be a disjunction, but invariants only support conjunction in parallel composition.

We thus need to take the context in terms of automata in parallel compositions (and, for MODEST, also in terms of relabellings) into account to transform deadlines for edges labelled a into invariants: We need to make sure that they only apply when an edge labelled a is actually available, i.e. when all automata that must synchronise on a can do so (to solve the problem of location $\langle l_0, l_A \rangle$), and that we use the correct operators to compose deadlines (to solve the problem of location $\langle l_1, l_A \rangle$).

In order to achieve this without flattening a network of PTA into a single PTA (with invariants) using the parallel composition semantics, we compute a global invariant $gi(\hat{e})$ for an expression $\hat{e}$ describing the network. In addition to the parallel composition $M_1 \parallel M_2$ of two PTA M_1 and M_2, we also allow in such an expression the relabelling of the edge labels of a PTA M according to a function $f \in A \to A$, written as $f(M)$. We call $\hat{e}$ a *process-algebraic expression*, and denote its PTA semantics by $[\![\hat{e}]\!]$. The relabelling operation corresponds to MODEST's `relabel` construct. We then add a single-location PTA GI with invariant $gi(\hat{e})$ to the network, while all other locations' invariants are set to *true*.

Definition 68 (Deadlines to invariants)**.** We compute the global invariant recursively over the process-algebraic expression for a network of PTA as follows, with $\text{shared}(a) \stackrel{\text{def}}{=} a \in A(e_1) \cap A(e_2) \setminus \{\tau\}$:

$$gi(e) = \bigwedge_{a \in A(e)} gi_a(e)$$

$$gi_a(e_1 \parallel e_2) = \begin{cases} En_a(e_1) \wedge En_a(e_2) \Rightarrow gi_a(e_1) \mathbin{\overline{\otimes}} gi_a(e_2) & \text{if shared}(a) \\ gi_a(e_1) \wedge gi_a(e_2) & \text{otherwise} \end{cases}$$

$$gi_a(f(e)) = \bigwedge_{b:\, f(b)=a} gi_b(e)$$

$$gi_a(M) = \bigwedge_{l \in Loc_M} \left(P_l^M \Rightarrow \bigwedge_{\langle g,d,a,\mu \rangle \in E_M(l)} Conv(d) \right)$$

where e, e_1 and e_2 are process-algebraic expressions, M refers to PTA, $A(e)$ is the union of the alphabets of the PTA in the (sub-)expression e, P_l^M is determined as $P_l^M \Leftrightarrow (M \text{ is in location } l)$, $\overline{\otimes}$ is $\vee$ if a is a patient action and $\wedge$ otherwise, and $En_a(e)$ is a predicate that characterises the locations in which an edge labelled a is available in the automaton represented by e:

$$En_a(e_1 \parallel e_2) = \begin{cases} En_a(e_1) \wedge En_a(e_2) & \text{if shared}(a) \\ En_a(e_1) \vee En_a(e_2) & \text{otherwise} \end{cases}$$

$$En_a(f(e)) = \bigvee_{b:\, f(b)=a} En_b(e)$$

$$En_a(M) = P_a^M$$

where P_a^M is *true* iff the PTA M is in a location with an outgoing edge labelled a.

Theorem 6. For a process-algebraic expression e, we have that $[\![e]\!]$ (with deadlines) and $[\![GI \parallel e]\!]$ (with invariants computed as described in the definition above) are isomorphic.

Proof. Because GI has a single location l_g without any edges and the synchronisation alphabet is empty, $l \mapsto \langle l_g, l \rangle$ is a bijection between the locations of the parallel composition and relabelling semantics of e and $GI \parallel e$ and the two PTA are isomorphic (up to deadlines and invariants). In the following, we denote the PTA that makes up the aforementioned semantics of a process algebraic expression e by $[e]$.

It remains to show (indicated by $\stackrel{?}{\Leftrightarrow}$) that the time progress conditions for all l and $\langle l_g, l \rangle$ are equivalent, i.e. that for all $l \in Loc_{[e]}$, we have

$$\forall t' < t : [\![\neg \bigvee_{\langle g,d,a,\mu \rangle \in E_{[e]}(l)} d]\!](v+t') \stackrel{?}{\Leftrightarrow} \forall t' \leq t : [\![Inv_{[e]}(\langle l_g, l \rangle)]\!](v+t')$$

$$\Leftrightarrow \quad \forall t' < t : [\![\bigwedge_{\langle g,d,a,\mu \rangle \in E_{[e]}(l)} \neg d]\!](v+t') \stackrel{?}{\Leftrightarrow} \forall t' \leq t : [\![\bigwedge_{a \in A(e)} gi_a(e)]\!](v+t')$$

$$\Leftrightarrow \forall t' \leq t : [\![\bigwedge_{\langle g,d,a,\mu \rangle \in E_{[e]}(l)} Conv(d)]\!](v+t') \stackrel{?}{\Leftrightarrow} \forall t' \leq t : [\![\bigwedge_{a \in A(e)} gi_a(e)]\!](v+t')$$

$$\Leftarrow \quad \bigwedge_{\langle g,d,a,\mu \rangle \in E_{[e]}(l)} Conv(d) \stackrel{?}{\Leftrightarrow} \bigwedge_{a \in A(e)} gi_a(e)$$

The last equivalence is in particular satisfied if we have that

$$\forall l \in Loc_{[e]},\, a \in A(e):\ \bigwedge_{\langle g,d,a,\mu\rangle \in E_{[e]}(l)} Conv(d) \Leftrightarrow gi_a(e).$$

We will now show this by induction over the structure of e, using $*$ to mark steps where the induction hypothesis was applied:

1. Case $e = M$:

$$\begin{aligned} gi_a(e) &= gi_a(M) \\ &= \bigwedge_{l' \in Loc_M} \left(P^M_{l'} \Rightarrow \bigwedge_{\langle g,d,a,\mu\rangle \in E_M(l')} Conv(d) \right) \\ &\Leftrightarrow \bigwedge_{\langle g,d,a,\mu\rangle \in E_M(l)} Conv(d) && \text{(def. of } P^M_l\text{)} \\ &= \bigwedge_{\langle g,d,a,\mu\rangle \in E_{[e]}(l)} Conv(d) \end{aligned}$$

2. Case $e = f(e')$:

$$\begin{aligned} gi_a(e) &= gi_a(f(e')) \\ &= \bigwedge_{b:\ f(b)=a} gi_b(e') \\ &\overset{*}{\Leftrightarrow} \bigwedge_{b:\ f(b)=a} \bigwedge_{\langle g,d,b,\mu\rangle \in E_{[e']}(l)} Conv(d) \\ &\Leftrightarrow \bigwedge_{\langle g,d,a,\mu\rangle \in E_{[f(e')]}(l)} Conv(d) \\ &= \bigwedge_{\langle g,d,a,\mu\rangle \in E_{[e]}(l)} Conv(d) \end{aligned}$$

3. Case $e = e_1 \parallel e_2$:
   Let $l = \langle l_1, l_2 \rangle$, i.e. the l_i are the component locations of the $[e_i]$ that make up l. We consider three subcases:
   (a) Case $a \notin A(e_1) \cap A(e_2) \setminus \{\tau\}$:

$$\begin{aligned} gi_a(e) &= gi_a(e_1 \parallel e_2) \\ &= gi_a(e_1) \wedge gi_a(e_2) \\ &\overset{*}{\Leftrightarrow} \bigwedge_{\langle g_1,d_1,a,\mu_1\rangle \in E_{[e_1]}(l_1)} Conv(d_1) \wedge \bigwedge_{\langle g_2,d_2,a,\mu_2\rangle \in E_{[e_2]}(l_2)} Conv(d_2) \\ &\Leftrightarrow \bigwedge_{\langle g_i,d_i,a,\mu_i \times \mathscr{D}(\langle \varnothing, l_{3-i}\rangle)\rangle \in E_{[e_1 \parallel e_2]}(\langle l_1,l_2\rangle)} Conv(d_i) \\ &\Leftrightarrow \bigwedge_{\langle g,d,a,\mu\rangle \in E_{[e]}(l)} Conv(d) \end{aligned}$$

   (b) Case $a \in A(e_1) \cap A(e_2) \setminus \{\tau\}$, and there is no edge labelled a from at least one of l_1 or l_2:

$$\begin{aligned} gi_a(e) &= gi_a(e_1 \parallel e_2) \\ &= En_a(e_1) \wedge En_a(e_2) \Rightarrow gi_a(e_1) \mathbin{\overline{\otimes}} gi_a(e_2) \\ &\Leftrightarrow false \Rightarrow gi_a(e_1) \mathbin{\overline{\otimes}} gi_a(e_2) \\ &\Leftrightarrow true \\ &\Leftrightarrow \bigwedge_{\langle g,d,a,\mu\rangle \in E_{[e]}(l)} Conv(d) && (\nexists\, \langle g,d,a,\mu\rangle \in E_{[e]}(l)) \end{aligned}$$

(c) Case $a \in A(e_1) \cap A(e_2) \setminus \{\tau\}$, and there are edges labelled a from both l_1 and l_2:

$$\begin{aligned}
gi_a(e) &= gi_a(e_1 \parallel e_2) \\
&= En_a(e_1) \wedge En_a(e_2) \Rightarrow gi_a(e_1) \mathbin{\overline{\otimes}} gi_a(e_2) \\
&\Leftrightarrow gi_a(e_1) \mathbin{\overline{\otimes}} gi_a(e_2) \\
&\overset{*}{\Leftrightarrow} \textstyle\bigwedge_{\langle g_1,d_1,a,\mu_1\rangle \in E_{[e_1]}(l_1)} Conv(d_1) \mathbin{\overline{\otimes}} \bigwedge_{\langle g_2,d_2,a,\mu_2\rangle \in E_{[e_2]}(l_2)} Conv(d_2) \\
&\Leftrightarrow \textstyle\bigwedge_{\langle g_1 \wedge g_2, d_1 \otimes d_2, a, \mu_1 \times \mu_2\rangle \in E_{[e_1 \parallel e_2]}(\langle l_1,l_2\rangle)} Conv(d_1) \mathbin{\overline{\otimes}} Conv(d_2) \\
&\Leftrightarrow \textstyle\bigwedge_{\langle g_1 \wedge g_2, d_1 \otimes d_2, a, \mu_1 \cdot \mu_2\rangle \in E_{[e_1 \parallel e_2]}(\langle l_1,l_2\rangle)} Conv(d_1 \otimes d_2) \\
&\Leftrightarrow \textstyle\bigwedge_{\langle g,d,a,\mu\rangle \in E_{[e]}(l)} Conv(d)
\end{aligned}$$

□

Example 34. For the PTA in Example 33, assuming that a is patient, the global invariant computed according to Definition 68 is

$$P_{\mathtt{a}}^{M_1} \wedge P_{\mathtt{a}}^{M_2} \Rightarrow (P_1^{M_1} \Rightarrow x \leq 5) \vee (P_A^{M_2} \Rightarrow x \leq 3).$$

Since $P_{\mathtt{a}}^{M_1} \Leftrightarrow P_1^{M_1}$ and $P_{\mathtt{a}}^{M_2} \Leftrightarrow P_A^{M_2}$, it is equivalent to what we intuitively expect:

$$P_1^{A_1} \wedge P_A^{A_2} \Rightarrow x \leq 5 \vee x \leq 3$$

5.3 Modelling

MODEST allows the specification of models that represent networks of VPTA with deadlines and invariants. Clock variables can be introduced into a model by simply declaring a variable of type `clock`. Variables declared in this way can only be set to zero in assignments. To enable or disable edges over time, clock constraints can simply be used in the expressions for guards specified with `when`. In order to control the passage of time itself, MODEST provides support for two new keywords that allow the addition of deadlines and invariants: `urgent` and `invariant`, extending the grammar for process behaviours as follows:

$$P ::= \ldots \mid \texttt{urgent}(e)\ P \mid \texttt{invariant}(e)\ P \mid \texttt{invariant}(e)\ \{P\}$$

where $e \in Bxp$ with subexpressions involving clocks conforming to the syntax for clock constraints.

Deadlines The `urgent` keyword to associate a deadline with an edge follows the same pattern as the `when` keyword for guards. Its inference rule (with edge

labels as introduced in the previous section now consisting of a guard, a deadline and an action label) is

$$\frac{P \xrightarrow{g,d,a} \mathscr{W}}{\texttt{urgent}(e)\ P \xrightarrow{g,d\vee e,a} \mathscr{W}}\ (\textit{urgent})$$

where $e \in \mathit{Bxp}$ and $\mathscr{W}$ denotes a symbolic probability distributions over updates and target MODEST process behaviours (i.e. a function in $\mathit{Upd} \times P \to \mathit{Axp}$). Note the use of $\vee$ to combine the new deadline with an existing one: The edge is now urgent, i.e. time cannot pass any more, if either the new or the existing deadline (or both) are satisfied. It would not be intuitive to use $\wedge$ here instead because this would mean that the "addition" of a deadline leads to "less" urgency.

As mentioned, actions in MODEST can be patient or impatient. When declaring an action, that declaration is simply prefixed with either `patient` or `impatient`. If omitted, it is impatient by default. MODEST's parallel composition then deals with the composition of deadlines using $\otimes \in \{\wedge, \vee\}$ for patient or impatient actions just like the parallel composition operator for PTA with deadlines (cf. Equation (5.1)).

Invariants Location invariants can be specified by the `invariant` keyword. For modelling convenience, it comes in two forms: as a dynamic operator that "disappears" when an edge is taken (like the `when` or `urgent` constructs), and as a static operator that remains in effect until the contained process behaviour terminates (like `try-catch`). Syntactically, the difference is that the former looks like a guard or deadline, while the latter uses curly brackets around the process behaviour that it applies to. The inference rule for the dynamic operator case is very simple:

$$\frac{P \xrightarrow{g,d,a} \mathscr{W}}{\texttt{invariant}(e)\ P \xrightarrow{g,d,a} \mathscr{W}}\ (\textit{inv})$$

For the static operator case, some effort is needed to preserve the operation in the process behaviours in the support of the symbolic probability distribution. The inference rule thus is as follows, with $Q(P) = \texttt{invariant}(e)\ \{P\}$:

$$\frac{P \xrightarrow{g,d,a} \mathscr{W}}{Q(P) \xrightarrow{g,d,a} \mathscr{W} \circ \mathbf{M}_{\text{inv}}^{-1}}\ (\textit{sinv}) \quad \text{where } \mathbf{M}_{\text{inv}}(\langle U, P'\rangle) \stackrel{\text{def}}{=} \begin{cases} \langle U, Q(P')\rangle & \text{if } P' \neq \checkmark \\ \langle U, P'\rangle & \text{if } P' = \checkmark \end{cases}$$

The inference rules for the two variants of the `invariant` construct ignore the actual invariant expression $e \in \mathit{Bxp}$ because it does not become part of the

$Inv(P) = e \wedge Inv(Q)$	if $P =$ `invariant`$(e)\ Q$ or $P =$ `invariant`$(e)\ \{\, Q \,\}$
$Inv(P) = tt$	if $P = act$, $P = act$ `palt` $\{\, :w_1 \colon U_1 ; P_1 \ \ldots\ :w_k \colon U_k ; P_k \,\}$, $P =$ `stop`, $P =$ `abort`, $P =$ `break` or $P =$ `throw`$(excp)$
$Inv(P) = Inv(P_1)$	if $P = P_1 ; P_2$ or $P =$ `auxdo` $\{\, P_1 \,\}\ \{\, P_2 \,\}$
$Inv(P) = \bigwedge_{i=1}^{k} Inv(P_i)$	if $P =$ `alt` $\{\, :: P_1 \ldots :: P_k \,\}$, $P =$ `do` $\{\, :: P_1 \ldots :: P_k \,\}$ or $P =$ `par` $\{\, :: P_1 \ldots :: P_k \,\}$
$Inv(P) = Inv(Q)$	if $P =$ `when`$(e)\ Q$, $P =$ `urgent`$(e)\ Q$, $P =$ `relabel` $\{\, I \,\}$ `by` $\{\, G \,\}\ Q$, $P =$ `extend` $\{\, H \,\}\ Q$, $P =$ `try` $\{Q\}$ `catch` $excp_1\ \{\, P_1 \,\} \ldots$ `catch` $excp_k\ \{\, P_k \,\}$ or $P = ProcName(e_1, \ldots, e_k)$ and $ProcName$ is declared as `process` $ProcName(t_1\, x_1, \ldots, t_n\, x_k)\ \{\, Q \,\}$

Table 5.2: The invariant function for MODEST process behaviours

edges, but is instead preserved as part of the function *Inv* that maps each location to an invariant. Remember that the locations of the VPTA semantics of a MODEST model are the MODEST process behaviours. The definition of *Inv* for process behaviours, and thus for the VPTA semantics of MODEST, is given in Table 5.2. This is where the invariant expressions e are picked up to become part of the automaton.

Example 35. We can now model the timed channel and sender processes in MODEST that we have built as PTA in previous examples. Figure 5.5 shows the MODEST code that directly represents the channel PTA as shown in Figure 5.1 (modulo renaming of `snd_data` to `snd`). In Figure 5.6, we show MODEST code that combines the sender with timeout detection introduced in Figure 5.2 with the bounded retransmission approach of the simple BRP sender from Example 8. Observe that we use both deadlines and invariants, and that we opted for making `snd_data` an impatient action that is urgent in the `Sender` process. In MODEST, we can also simply write `urgent` as a shorthand for `urgent`$(true)$. In the model of the sender, we have also made the edge that sets `failure` to *true* urgent, with deadline `n` $== 0$. This is to make sure that the assignment cannot be delayed indefinitely, but the progress of time is only restricted if the assignment is actually available—thus guard and deadline are the same. Since the affected edge is (implicitly) labelled τ, it cannot synchronise in parallel composition, so we could have used `invariant`$(\texttt{n} \neq 0)$ with the same effect. If we now take the simple probabilistic BRP model of Example 19, replace its

```
process Channel()
{
   clock c;

   snd palt {
   :95: {= c = 0 =};
        invariant(c <= TD_MAX) alt {
        :: when(c >= TD_MIN) rcv
        :: snd {= collision = true =}; stop
        }
   : 5: {= c = 0 =};
        invariant(c <= TD_MAX) alt {
        :: when(c >= TD_MIN) tau
        :: snd {= collision = true =}; stop
        }
   };
   Channel()
}
```

Figure 5.5: The lossy comm. channel with transmission delay in MODEST

```
impatient action snd_data;

process Sender(int n)
{
   clock c;

   urgent(true) snd_data {= n = n - 1 =};
   invariant(c <= TS) alt {
   :: rcv_ack {= success = true =}
   :: when(c >= TS) timeout;
      alt {
      :: when(n > 0) Sender(n) // retry
      :: when(n == 0) urgent(n == 0) {= failure = true =};
         stop // deadlock on failure
      }
   }
}
```

Figure 5.6: The simple BRP sender with timeout detection in MODEST

channel and sender processes by the ones introduced here (thereby also relabelling `snd` back to `snd_data` where appropriate), and add deadline *true* to the `snd_ack` in the receiver, then we get what we call the *simple probabilistic-timed BRP*.

5.4 Properties

For PTA, we consider two classes of properties: those that refer to the probability of reaching a certain set of states (extending the reachability properties of the previous chapters by timed aspects), and those that refer to the expected value of certain random variables given by so-called rewards.

5.4.1 Reachability

To specify properties to verify on PTA models, we can use standard probabilistic reachability properties as we know them from MDP. To include timing requirements, they can be extended to probabilistic timed reachability properties, which include the popular special case of probabilistic time-bounded reachability.

Probabilistic timed reachability Probabilistic reachability properties in the form that we already used for MDP are directly applicable to PTA as well:

$\mathrm{P}_{\max}(\diamond\phi)$ –and– $\mathrm{P}_{\min}(\diamond\phi)$ (quantitative form) $\quad \mathrm{P}(\diamond\phi) \sim x$ (qualitative form)

In order to express timed requirements, such as the probability of eventually reaching a certain state within t time units, simple clock constraints can be included in the set of atomic propositions AP of a PTA. Although they cannot sensibly be used by its labelling function L (since there is no way to know whether a clock constraint is satisfied in a certain location), they are treated specially by the PTA semantics of Definition 61 to become part of the state labelling in the underlying TPTS whenever they are satisfied for the current valuation of the clock variables. Allowing clock constraints in state formulas leads to the class of *probabilistic timed reachability* properties. This new class in particular includes probabilistic time-bounded reachability: A property with time bound TB can be specified by adding a clock c_B to the PTA that is never reset and including the clause $c_B \leq TB$ in the property. In MODEST, such a clock is available inside properties with the name `time` and does not need to be declared explicitly.

The semantics of probabilistic timed reachability properties for PTA is simply defined as their semantics on the underlying TPTS. In order to give

a formal definition, we first need to slightly update our notion of schedulers, originally introduced in Definition 40 for MDP, to the continuous-probability setting of TPTS:

Definition 69 (Scheduler)**.** For a TPTS $M = \langle S, A, T, s_{init}, AP, L \rangle$, a *scheduler* is a function $\mathfrak{S}\colon \text{Paths}_{\text{fin}}(M) \to \text{Prob}(A \times \text{Prob}(S))$ s.t. for each $\pi \in \text{Paths}_{\text{fin}}(M)$ we have $\mathfrak{S}(\pi)(A \times \text{Prob}(S) \setminus T(\text{last}(\pi))) = 0$.

A scheduler assigns probabilities to sets of enabled action-distribution pairs depending on the history seen so far. The schedulers we consider here are thus more powerful than the memoryless schedulers that were sufficient for MDP previously. However, we will show later in this chapter when it comes to model checking that memoryless schedulers (for an MDP abstraction of a PTA) actually do suffice for the properties we consider. This may come as a surprise since probabilistic timed reachability properties can reference clock constraints, and can thus include *time* bounds. However, this is fundamentally different from the *step* bounds for which memoryless schedulers did not not suffice for MDP: The timing information used in a time-bounded probabilistic timed reachability property is part of the TPTS' state space (within the valuations of the clock variables), whereas the mentioned step bounds for MDP were not. If we encoded step bounds through an extra variable that is incremented on every edge in a VMDP and referred to in the property, then they would become part of the state space and memoryless schedulers would suffice, too.

A scheduler resolves the nondeterminism in a TPTS so as to obtain probability measures, allowing to derive according stochastic processes: A scheduler $\mathfrak{S}$ induces the stochastic processes $X_M^{\mathfrak{S}}(\cdot) \in \mathbb{N} \to S$ of the current state of the TPTS M after a number of transitions have been taken (where always $X_M^{\mathfrak{S}}(0) = s_{init}$) and $Y_M^{\mathfrak{S}}(\cdot) \in \mathbb{N} \to A \times \text{Prob}(S)$ of the transition chosen by the scheduler in the current state.

We already saw that PTA can have timelocks, i.e. situations where time stops, and we pointed out that these represent an unrealistic situation and should thus be seen as modelling errors. Similarly, *Zeno* behaviour should be ruled out: paths in TPTS that contain infinitely many delay transitions, but where the sum of the delays is finite. For example, where a delay of one time unit is possible, a Zeno path could delay for $1/2$, then $1/4$, $1/8$ and so on. When defining the semantics of PTA properties in terms of the probabilities of sets of paths in the underlying TPTS, one thus needs to be careful to only consider *time-divergent* paths, i.e. paths where the sum of the delays is infinity. On the level of schedulers, this means that only those must be used that almost surely admit but time-divergent paths:

Definition 70 (Time-divergent scheduler)**.** A scheduler $\mathfrak{S}$ as in Definition 69 is *time-divergent* if $\mathbb{P}(\sum_{i=0}^{\infty} f(Y_M^{\mathfrak{S}}(i)) = \infty) = 1$ for $f(s \xrightarrow{a} \mu) = a$ if $a \in \mathbb{R}^+$ and $f(s \xrightarrow{a} \mu) = 0$ otherwise. We denote the set of all time-divergent schedulers of M by $\mathfrak{S}(M)$.

The semantics of probabilistic timed reachability properties can then be defined on the TPTS semantics in the usual way using time-divergent schedulers to get minimal/maximal values, that is infima/suprema over all $\mathfrak{S} \in \mathfrak{S}_M$ as for MDP in Section 4.4, and then using measurable sets of paths and the cylinder construction like we have done for DTMC in Section 3.2.2. If the set of ϕ-states for a given property is B, then the *reachability probability* induced by $\mathfrak{S}$ is defined as $\mathbb{P}(\exists i \geq 0 \colon X_M^{\mathfrak{S}}(i) \in B)$, i.e. the measure of paths with a state in B.

Temporal logics for PTA Various temporal logics have also been extended with timed aspects. For timed automata, this includes TCTL, the timed variant of standard CTL. For PTA, TCTL can be combined with PCTL to obtain PTCTL [KNSS02]. As before, we focus on reachability instead in this thesis; probabilistic timed reachability can be expressed in PTCTL.

5.4.2 Rewards

To reason about average-case behaviour, it is useful to be able to refer to *expected values*: The expected time until a set of states is reached, or the expected number of messages lost within the first t time units, and so on. These are examples of the general class of *expected accumulated reward* properties. To specify them formally, we first add to a given PTA a *reward*, which can be seen as a real-valued variable that is only available to external observers (i.e. it can be used in properties, but not be read inside the model in guards, invariants etc.). A reward advances at a certain rate in locations and can be increased when taking an edge:

Definition 71 (Reward)**.** Given a PTA $M = \langle Loc, \mathscr{C}, A, E, l_{init}, Inv, AP, L \rangle$, a *reward* is a pair $r = \langle Rew_{Loc}, Rew_E \rangle \in (Loc \to \mathbb{R}) \times (E \to \mathbb{R})$ of a function Rew_{Loc} that assigns *rate rewards* to the locations of M and a function Rew_E that assigns *edge rewards* to its edges.

A particular reward that is useful in many cases is the *global time* given by

$$r_{time} = \langle \{ \langle l, 1 \rangle \mid l \in Loc \}, \{ \langle e, 0 \rangle \mid e \in E \} \rangle$$

for any PTA with locations Loc and edges E. It intuitively corresponds to a clock variable like `time` that is never reset and only used inside properties. In

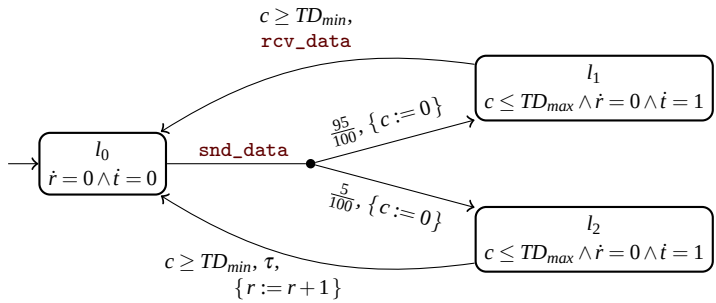


Figure 5.7: PTA model of a lossy communication channel with rewards

fact, in MODEST, `time` doubles as a clock variable (as mentioned previously) and a reward (as described here).

Example 36. Figure 5.7 shows a variant of the lossy communication channel introduced in Example 28 without collision detection, but with two rewards: r assigns an edge reward of 1 to the edge leading back to the initial location when a message has been lost. Its rate reward is 0 in all locations. r thus counts the number of message losses. Reward t, on the other hand, has edge reward 0 for all edges, but its rate reward is 1 in l_1 and l_2. It thus keeps track of the time spent sending a message (both successfully and unsuccessfully). In the remainder of this thesis, we will omit edge and rate rewards of 0 when drawing automata.

The semantics of a reward is a *reward structure* for the TPTS semantics of the corresponding PTA:

Definition 72 (Reward structure)**.** A *reward structure* for a TPTS with transition function T is a function $\mathrm{rew}\colon T \to \mathbb{R}_0^+$ assigning a nonnegative reward to each of the transitions.

Definition 73 (Semantics of a reward)**.** Let $r = \langle Rew_{Loc}, Rew_E \rangle$ be a reward for a PTA $M = \langle Loc, \mathscr{C}, A, E, l_{init}, Inv, AP, L \rangle$ and let the transition function of $[\![M]\!]$ be T_M. Then the semantics of r is the reward structure $[\![r]\!]\colon T_M \to \mathbb{R}_0^+$ such that

$$[\![r]\!](tr = \langle\langle l, v\rangle, a, \mu\rangle) = \begin{cases} a \cdot Rew_{Loc}(l) & \text{if } a \in \mathbb{R}^+ \\ Rew_E(jump^{-1}(tr)) & \text{if } a \in A \end{cases}$$

where $jump^{-1}(tr)$ is the edge in M that induced the transition tr according to the inference rule *jump*.

For transitions labelled with time actions $t \in \mathbb{R}_0^+$, we thus assign a reward of t times the location reward rate according to Rew_{Loc}. For A-labelled transitions, the reward value is as defined by Rew_E for the original PTA edge. Using rewards, we can now define a second class of properties for PTA:

Expected-reward Properties

An expected-reward property has the following syntax:

$$\mathrm{X}_{\max}(r \mid \phi) \text{ –and– } \mathrm{X}_{\min}(r \mid \phi) \qquad \text{(quantitative form)}$$
$$\mathrm{X}(r \mid \phi) \sim x \qquad \text{(qualitative form)}$$

where r is a reward and ϕ is a state formula that may include clock constraints. Intuitively, the semantics of $\mathrm{X}_{\max}(r \mid \phi)$ / $\mathrm{X}_{\min}(r \mid \phi)$ is the maximum/minimum expected sum of reward values accumulated until a ϕ-state is reached for the first time. As before, "maximum" or "minimum" refers to the resolution of nondeterminism, while "expected" tells us to compute the expected value (or mean) over the probabilistic choices.

Formally, the semantics of $\mathrm{X}_{\max}(r \mid \phi)$ resp. $\mathrm{X}_{\min}(r \mid \phi)$ is the supremum resp. infimum over all time-divergent schedulers $\mathfrak{S}$ of the *expected accumulated reward*. If the set of ϕ-states is B, the expected accumulated reward is

$$\mathbb{E}(\sum_{i=0}^{j} [\![r]\!](Y_M^{\mathfrak{S}}(i))) \quad \text{where } j = \min\{\, i \mid X_M^{\mathfrak{S}}(i) \in B \,\}$$

if $\mathbb{P}(\exists i \geq 0 \colon X_M^{\mathfrak{S}}(i) \in B) = 1$, and ∞ otherwise. It is thus the expected reward accumulated along paths provided B is reached eventually; otherwise the value is infinity. This means that, if there is a scheduler that does not almost surely reach a ϕ-state, then $\mathrm{X}_{\max}(\cdot \mid \phi) = \infty$; if all schedulers do not almost surely reach a ϕ-state, then also $\mathrm{X}_{\min}(\cdot \mid \phi) = \infty$.

In MODEST, rewards are declared as variables of type `reward`. Such a reward variable `r` must not be used in guards, invariants, deadlines etc. or on the right-hand side of any assignment except

– in assignments of the form $\mathtt{r} := \mathtt{r} + e$ to specify an edge reward of $e \in Axp$ and
– in invariant expressions $e_1 \wedge \mathtt{der}(\mathtt{r}) = e_2$ with $e_1 \in Bxp$ and $e_2 \in Axp$ to specify a rate reward of e_2 for the location that the invariant applies to.

If a reward variable is not referenced in a location's invariant (in a branch's update), it implicitly has rate reward (transition reward) 0 for that location (branch).

The `X`/`Xmax`/`Xmin` keywords can then be used to write expected-reward properties:

```
property E_Success = Xmax(time | success);
```

asks for the expected time until the transmission is completed successfully in our probabilistic timed simple BRP model based on Example 35, using the implicitly-declared global clock `time` that can also be used as a reward variable to refer to the reward r_{time}.

5.5 Model Checking

The semantics of a probabilistic timed automaton is an object with uncountably infinitely many states and transitions. The same is true for the submodel of timed automata. However, model checking techniques have been developed for TA, and efficient implementations like UPPAAL exist. This is possible because there is an equivalent, but finite model—an LTS in the case of TA, an MDP in the case of PTA—for every automaton that uses only integer clock constraints. In practice, PTA and TA are therefore always assumed to **only use integer clock constraints**, and we also do so from now on. With this assumption, there are several known techniques to perform exhaustive model checking for PTA: the region graph construction, forwards and backwards reachability, the use of digital clocks, and the use of stochastic games. In this section, we summarise these approaches, focussing on details only for the region graph since that is the key foundation to obtain a finite semantics for PTA that is amenable to model checking.

Region Graph

A finite model that is equivalent to a given PTA is the *region graph* [KNSS02], which is the straightforward probabilistic extension of the concept of the same name for TA [AD94].

Definition Intuitively, the region graph is the quotient of the TPTS semantics of a given PTA under the equivalence relation that groups those states that satisfy the same (integer) clock constraints. We now formally define this equivalence relation, which allows the reduction of the uncountable state space to a finite set of *regions*. We then show how to map the uncountably many timed transitions to a finite set of representatives, and finally give the formal definition of the region graph of a PTA. The following definitions are for a given PTA $M = \langle Loc, \mathscr{C}, A, E, l_{init}, Inv, AP, L \rangle$ and based on [NPS13, Section 5.1].

Definition 74 (Clock equivalence)**.** Let $i \in \mathbb{N}$. Two valuations v_1 and v_2 over the clock variables $\mathscr{C}$ are *clock equivalent* up to i if and only if
– for all $c \in \mathscr{C}$, either $v_1(c) > i \wedge v_2(c) > i$ or $v_1(c) =_{\mathbb{Z}} v_2(c)$, and

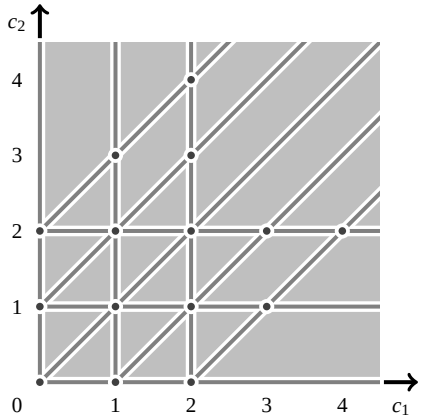


Figure 5.8: Clock equivalence classes for two clocks and maximum constant 2

– for all $c_1, c_2 \in \mathscr{C}$, either $v_1(c_1) - v_1(c_2) > i \wedge v_2(c_1) - v_2(c_2) > i$ or $v_1(c_1) - v_1(c_2) =_{\mathbb{Z}} v_2(c_1) - v_2(c_2)$

where $x_1 =_{\mathbb{Z}} x_2$ for $x_1, x_2 \in \mathbb{R}_0^+$ iff $\lfloor x_1 \rfloor = \lfloor x_2 \rfloor \wedge (\lfloor x_1 \rfloor - x_1 = 0 \Leftrightarrow \lfloor x_2 \rfloor - x_2 = 0)$, i.e. the "integer parts" of the two values are the same and they are either both in $\mathbb{Z}$ or both in $\mathbb{R} \setminus \mathbb{Z}$. The equivalence classes are denoted $[v_1]^i_{\mathscr{C}}$, $[v_2]^i_{\mathscr{C}}$ etc. We omit i if it is clear from the context.

Example 37. Figure 5.8 graphically shows a view of the clock equivalence classes for two clocks c_1, c_2 and $i = 2$. Each grey area, line and point is a clock equivalence class.

The states of the region graph will be pairs of locations and clock equivalence classes. These are called *regions*:

Definition 75 (Region)**.** A pair $\langle l, [v]_{\mathscr{C}} \rangle$ of a location $l \in Loc$ and a clock equivalence class $[v]^{i_{max}}_{\mathscr{C}}$ for a valuation v is called a *region*, where i_{max}, the maximum clock constant, is the maximum integer that occurs in any clock constraint of the PTA M (i.e. in guards, invariants and atomic propositions). The set of regions of M is denoted Reg_M, or simply Reg if M is clear from the context.

Observe that the set of regions of a PTA with a finite number of locations is finite, but its size is exponential in the number of clocks [NPS13]:

$$|Reg_M| \leq |Loc| \cdot (2 \cdot i_{max} + 2)^{(|\mathscr{C}|+1)^2}$$

We now define the time successor of a region, which will later allow us to specify the equivalent of *delay* transitions for the region graph.

Definition 76 (Time successor)**.** The *time successor* of a region $\langle l, \alpha \rangle$ is the region $\langle l, \beta \rangle$ where either
– $\beta = \alpha$ if $\forall v \in \alpha, t \in \mathbb{R}_0^+ : v + t \in \alpha$, or
– β is the unique clock equivalence class s.t. $\beta \neq \alpha$ and

$$\exists v \in \alpha, t \in \mathbb{R}_0^+ : v + t \in \beta \wedge \forall t' \in [0,t] : v + t' \in \alpha \cup \beta \wedge [\![Inv(l)]\!](v + t') \quad (5.2)$$

Otherwise, $\langle l, \alpha \rangle$ has no time successor.

For PTA with deadlines, Equation (5.2) can easily be modified to check the time progress condition for deadlines instead of the one for invariants.

Example 38. Let us assume that the clock equivalence classes shown in the previous example represent the regions for some location l with $Inv(l) = true$. Then the time successor of each region can be found by following the line with gradient 1 from any point inside the region until it hits a new region. If any such line never leaves a region, then that region is its own time successor.

We now have all the necessary ingredients to formally define what the region graph of a PTA is:

Definition 77 (Region graph)**.** The *region graph* of M is the MDP

$$RG(M) = \langle Reg_M, A, T_M, \langle l_{init}, [\mathbf{0}_{\mathscr{C}}] \rangle, AP, L_M \rangle$$

where $L_M(\langle l, \alpha \rangle) = L(l) \cup \bigcup_{v \in \alpha} \{ e \in AP \cap \mathscr{C}^S \mid [\![e]\!](v) \}$ and T_M is the smallest function such that the following two inference rules are satisfied:

$$\frac{l \xrightarrow{g,a}_T \mu \quad \exists v \in \alpha : [\![g]\!](v) \wedge \mu^* = \mu_M^{v,\alpha}}{\langle l, \alpha \rangle \xrightarrow{a}_{T_M} \mu^*} \ (jump_{Reg})$$

$$\frac{\beta \text{ is the time successor of } \alpha}{\langle l, \alpha \rangle \xrightarrow{\tau}_{T_M} \mathscr{D}(\langle l, \beta \rangle)} \ (delay_{Reg})$$

where

$$\mu_M^{v,\alpha}(\langle l', \beta \rangle) = \begin{cases} \mu(l', X) & \text{if } v' = v[X \mapsto 0] \text{ and } \beta = [v']_{\mathscr{C}} \\ 0 & \text{otherwise.} \end{cases}$$

Observe the overall similarity to Definition 61: We mostly just replaced valuations by regions and hid the time progress condition inside the definition of the time successor.

Example 39. The region graph for the simple channel PTA of Figure 5.1 with $TD_{min} = 1$ and $TD_{max} = 2$ is shown in Figure 5.9. In this graphical representation, we describe a clock equivalence class with a predicate that characterises the valuations it contains.

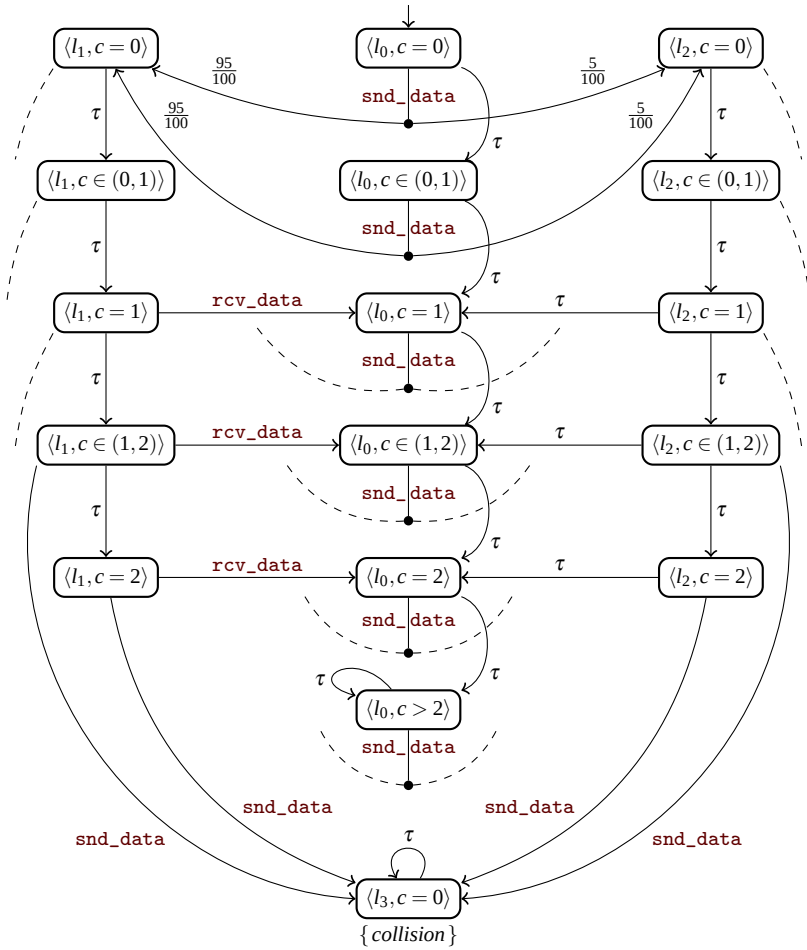


Figure 5.9: Region graph of the lossy communication channel PTA

Model checking The region graph is equivalent to the original PTA in the sense that the semantics of properties specified in the temporal logic PTCTL on the PTA and on the region graph lead to the same results as long as the PTA is structurally divergent. The latter is a syntactic property that ensures that all schedulers for the PTA's semantics are time-divergent [NPS13]. Otherwise, only maximum probabilities are preserved. However, there is an algorithm that can be used to lift the restriction to structurally divergent PTA even for minimum probabilities [Spr09].

To check probabilistic timed reachability, we can thus construct the region graph for the PTA and use MDP model checking techniques on it to verify maximum probabilistic (timed) reachability properties, but not expected-time reachability. The *boundary region graph* can be used for expected-time properties [NPS13]. However, as the size of the region graph is exponential in the size of the PTA, model checking in this way has exponential runtime and memory usage in general. It is considered not to be useful in practice.

Forwards Reachability

The region graph gets very large even for small models. Yet, as we were able to see in Example 39, the behaviour of many regions may actually be the same. This observation has already been made for TA and led to the development of *zone*-based approaches. A zone is a convex combination of regions:

Definition 78 (Zone)**.** Given a set of clocks $\mathscr{C}$, a *zone* Z is a set of valuations described by a basic (integer) clock constraint. Given an integer $i > 0$, the set of zones is denoted $\mathscr{ZZ}(\mathscr{C}, i)$,

$$\mathscr{ZZ}(\mathscr{C}, i) \stackrel{\text{def}}{=} \{ \{ v \in \mathrm{Val}(\mathscr{C}) \mid [\![e]\!](v) \} \mid e \in \mathscr{CC}^B(\mathscr{C}) \wedge \text{all constants in } e \text{ are} \leq i \}.$$

The set of zones of a PTA M is $\mathscr{ZZ}(M) \stackrel{\text{def}}{=} \mathscr{ZZ}(\mathscr{C}, i_{max})$ where $\mathscr{C}$ is the set of clocks of M and i_{max} is the maximum integer that occurs in any clock constraint of M. As usual, we omit $\mathscr{C}$ if it is clear from the context.

In the forwards reachability approach to model-check PTA [KNSS02, DKN04], a finite MDP representing the PTA is constructed by following edges and delay transitions from the initial state. However, instead of representing the valuations for the clocks as regions, zones are used. Since a single zone can represent the union of a large number of regions, this can result in significantly smaller MDP in practice. We omit the details of this construction, which can be found in [KNSS02], here in favour of an example:

Example 40. The zone MDP for the simple channel PTA of Figure 5.1 is shown in Figure 5.10, again for $TD_{min} = 1$ and $TD_{max} = 2$. Compare to the region graph shown in the previous example.

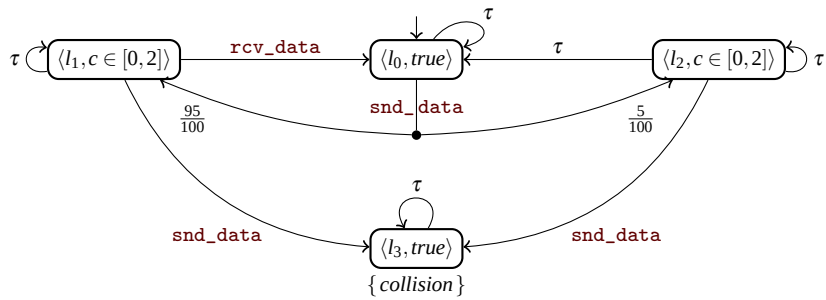


Figure 5.10: Zone MDP for the lossy communication channel PTA

As for the region graph, standard model checking techniques can then be used on the resulting zone-based MDP to verify the properties at hand. However, this approach only results in correct *upper bounds* on maximum probabilistic timed reachability properties in quantitative form. Consequently, it is only applicable to properties in qualitative form if the comparison operator used is $\leq$ or $<$, and can only answer *true* or *unknown* in that case. On the other hand, although the zone-based MDP may be the region graph in the worst case, it is usually significantly smaller, making the forwards reachability approach applicable to realistic models.

Backwards Reachability

The efficient representation of clock valuations with zones can be used in a backwards exploration of the PTA, too. In this approach [KNSW07], instead of computing time and edge *successors* starting from the *initial state* as in the previous two techniques, time and edge *predecessors* are computed from the *target states* of the reachability property, i.e. the ϕ-states. The result is, again, a finite MDP, and it allows the verification of the full logic PTCTL. However, some operations in this approach lead to non-convex "zones", so a more complicated data structure needs to be used (such as sets of zones). This has an impact on runtime. In the end, though, the resulting MDP are usually very small, and the approach is practically useable.

Digital Clocks

Another alternative that relies on the reduction of a PTA to a sufficiently equivalent MDP is the digital clocks approach. It works by replacing the real-valued

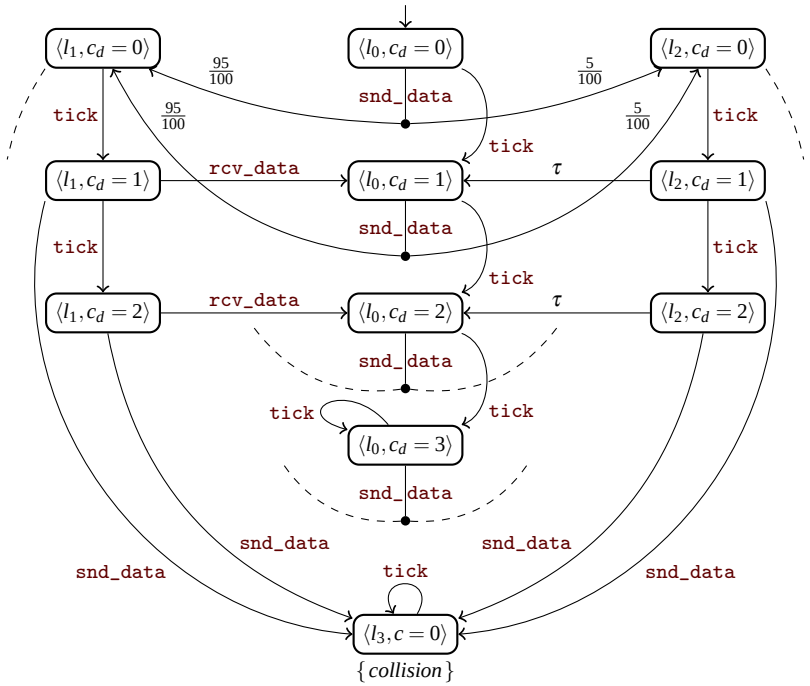


Figure 5.11: Digital clocks MDP of the lossy communication channel PTA

clock variables of the PTA by integer variables that are bounded by the maximum constant the relevant clock is compared to in any clock constraint plus one. When we then apply the PTA semantics (Definition 61) using delays of 1 only, the result is an MDP that is usually larger than the zone-based one, but smaller than the region graph.

Example 41. The digital clocks MDP for the simple channel PTA of Figure 5.1 with $TD_{min} = 1$ and $TD_{max} = 2$ is shown in Figure 5.11. Delay transitions are labelled `tick` here. Compare to the region graph and zone MDP of the previous two examples.

However, some information is lost when reals are replaced by integers, namely the difference between open and closed clock constraints, e.g. between $c > 1$ and $c \geq 1$. The digital clocks method thus only works correctly for PTA in

approach	prob	exp	logic	results	tool support
region graph [KNSS02]	(✓)	–	(✓)	exact	–
forwards reachability [KNSS02]	max	–	–	upper bounds	(unavailable)
backwards reachability [KNSW07]	✓	–	✓	exact	(unavailable)
digital clocks [KNPS06]	✓	✓	–	exact (closed PTA only)	mcpta [HH09], PRISM
stochastic games [KNP09]	✓	–	–	exact	(PRISM)

Table 5.3: Model checking approaches for PTA [HH09, NPS13]

which all clock constraints are closed and diagonal-free[1]. In that case, however, the results of verifying probabilistic timed reachability as well as expected-reward properties on the MDP are valid for the original PTA. This is currently the only (practical) approach that allows expected-reward properties to be analysed.

Stochastic Games

The last approach to model checking (diagonal-free) PTA that we mention uses abstraction refinement based on stochastic games [KNP09]. It also relies on a forward exploration of the zone MDP for the given PTA, but then uses it to construct a stochastic game abstraction. In this game, one player controls the nondeterminism that was originally in the MDP, while the other controls the nondeterminism that was introduced by the abstraction. This abstraction is then refined until the probabilities for the probabilistic timed reachability properties at hand are obtained exactly. Again, the final game obtained upon termination of the refinement process can be the entire region graph in the worst case, but is usually much smaller. Indeed, this approach is currently known to be the most competitive in terms of runtime in practice.

Summary

Table 5.3 gives a concise overview of the approaches we presented in this section. Columns “prob”, “exp” and “logics” refer to whether the techniques can

[1]The support of diagonals is possible with extra bookkeeping information about the relationships between clocks when the “open regions” beyond the maximum clock constants are reached, but this is typically not implemented.

handle probabilistic timed reachability, expected-reward properties, and the full temporal logic PTCTL, respectively. We see that there is no method that can verify both PTCTL and expected-reward properties. In fact, the latter can only be checked using the digital clocks approach. While the region graph (and boundary region graph) construction works for PTCTL (and expected-reward properties), its main purpose is to show the decidability of PTA model checking because an implementation is not considered practical.

Naturally, in practice, tool support is a crucial issue, too. Of the five approaches presented, only two have been implemented in publicly available, stable model checking tools: mcpta, developed by the author and part of the MODEST TOOLSET, was the first tool to allow fully automatic model checking of PTA [HH09]. It uses the digital clocks technique and additional optimisations to obtain an MDP that is subsequently handed to PRISM for the probabilistic, but untimed, analysis. The PRISM language has meanwhile been extended to cover PTA as well [KNP11], and the digital clocks and stochastic game-based approaches have been implemented. However, only a severely restricted subset of the PRISM language can be used for the latter (which, for example, does not allow the use of global variables), making the construction of realistic models cumbersome. To the author's knowledge, tools supporting the forwards and backwards reachability techniques never left a prototypical stage and have not been released to the public. An implementation using the region graph has not been attempted because it can only work for tiny PTA models.

5.6 Statistical Model Checking

We have seen in the previous chapter that it is difficult to devise a statistical model checking method that achieves both correct results and almost-constant memory usage for a model with nondeterministic choices. Probabilistic timed automata contain continuous nondeterminism in delays in addition to the already challenging nondeterministic choices of MDP. This appears to make statistical model checking for PTA an even harder problem than for MDP.

In this section, we review a number of approaches to apply SMC to PTA models. First, we consider the model of PTA without continuous nondeterminism, i.e. time-deterministic PTA, in Section 5.6.1. Then, similar to how we studied ways to resolve nondeterminism in MDP in Section 4.6.1, we look at *scheduling* options for nondeterministic delays in Section 5.6.2. Both approaches entail removing the continuous nondeterminism, either by restriction of the model or by introducing a time scheduler, and thus allow us to apply the SMC techniques we presented for MDP in the previous chapter. A

very particular implementation of a scheduling policy for PTA is part of UPPAAL SMC [DLL+11b]. We explain the motivation and ideas behind this policy, as well as some surprising effects, in Section 5.6.3. Finally, we give an overview of possible ways to perform SMC for general nondeterministic PTA in Section 5.6.4, which all ultimately map back to the problem of SMC for nondeterministic MDP that we studied in the previous chapter.

5.6.1 Time-Deterministic PTA

The main feature of the PTA model is the ability to represent nondeterministic delays, i.e. continuous nondeterministic choices over time. If all delays in a PTA have fixed durations, we call it a time-deterministic PTA:

Definition 79 (TDPTA)**.** A PTA M with TPTS semantics

$$[\![M]\!] = \langle Loc \times Val, \Sigma_{Loc\times Val}, \mathbb{R}^+ \uplus A, T_M, \langle l_{init}, \mathbf{0}\rangle, AP, L_M\rangle$$

is *time-deterministic*, or equivalently a *time-deterministic probabilistic timed automaton* (TDPTA), if AP is countable and for all reachable states $\langle l, v\rangle$ of $[\![M]\!]$, we have that if there is an action-labelled transition $\langle a, \mu\rangle \in T_M(\langle l, v\rangle)$, then there is no delay-labelled transition in $T_M(\langle l, v\rangle)$.

This means that, in a TDPTA, an edge must be taken as soon as it becomes enabled. This is usually achieved by combining guards of the form $c \geq x$ for a clock c and $x \in \mathbb{R}_0^+$ with the corresponding invariant $c \leq x$. For TDPTA, we do not need to require clock constraints to be integer, and thus allow general clock constraints whenever we explicitly deal with a TDPTA.

Time-deterministic Semantics

When a modeller creates a TDPTA model, they use a subset of the features of PTA, namely clocks and clock constraints, to include real-time aspects in their model in a natural and concise way. However, the same model could directly be specified as an equivalent (though arguably less natural or concise) MDP by dropping all "intermediate" delay transitions from the semantics of the TDPTA:

Definition 80 (Time-deterministic semantics)**.** Given a TDPTA M and its semantics $[\![M]\!]$ as in Definition 79, we define the *time-deterministic semantics* of M to be the TPTS

$$\mathrm{tdet}([\![M]\!]) = \langle S, \Sigma_S, \mathbb{R}^+ \uplus A, T_M^{\mathrm{det}}, \langle l_{init}, \mathbf{0}\rangle, AP, L_M\rangle$$

where $\langle l, v\rangle \xrightarrow{x}_{T_M^{\mathrm{det}}} \mu$ if $\langle l, v\rangle \xrightarrow{x}_{T_M} \mu$ and either $x \in A$ (it is an action-labelled transition) or $x \in \mathbb{R}^+ \wedge \mathit{maxDelay}(\langle l, v\rangle) = x$ (it represents many delay-labelled

transitions), the function *maxDelay* is defined by

$$\mathit{maxDelay}(\langle l,v\rangle) = \min\{\, t \in \mathbb{R}^+ \mid L(\langle l,v\rangle) \neq L(\langle l,v+t\rangle) \\ \vee \exists\, a \in A, \mu' \in \mathrm{Prob}(S) \colon \langle l,v+t\rangle \xrightarrow{a}_{T_M} \mu' \,\}$$

if such a minimal t exists and $\mathit{maxDelay}(\langle l,v\rangle) = \bot$ (assuming $\bot \notin A$) otherwise, and the subset of states $S \subseteq \mathit{Loc} \times \mathit{Val} \cup \mathit{Loc} \times \{\infty\}$ contains exactly those that are reachable from $\langle l_{\mathit{init}}, \mathbf{0}\rangle$ according to T_M^{det} and the MDP definition of paths (which we can use because all probability measures can also been seen as distributions due to Definitions 59 and 61).

The representative delay chosen by the function *maxDelay* is the shortest one that leads to either an observable change (i.e. a change in state labelling) or to a state where an action-labelled transition is enabled, i.e. where no further delay is possible (by the definition of TDPTA).

Observe that $\mathrm{tdet}(\llbracket M \rrbracket)$ is an MDP (in the sense of Proposition 6): It is finitely branching because the definition of T_M^{det} selects a single representative delay-labelled transition whenever time can pass. S is countable because the system is finitely branching and there are at most countably many states between two action-labelled transitions. The latter is because the set of atomic propositions of $\llbracket M \rrbracket$ was required to be countable and thus contains at most countably many clock constraints.

Theorem 7. For a TDPTA M, the semantics (i.e. the value) of any probabilistic timed reachability or expected-reward query is the same for $\llbracket M \rrbracket$ and $\mathrm{tdet}(\llbracket M \rrbracket)$.

Proof. We show that for any time-divergent scheduler $\mathfrak{S} \in \mathfrak{S}(\llbracket M \rrbracket)$, there exists a corresponding time-divergent scheduler $\mathfrak{S}_{\mathrm{det}} \in \mathfrak{S}(\mathrm{tdet}(\llbracket M \rrbracket))$ that results in the same reachability probabilities and expected accumulated rewards and vice-versa:

Consider $\mathfrak{S}$ as given and $\pi \in \mathrm{Paths}_{\mathrm{fin}}(\mathrm{tdet}(\llbracket M \rrbracket))$ with $\mathrm{last}(\pi) = \langle l,v\rangle$. Following Definition 79, we distinguish two cases:

a) There is an action-labelled transition $\langle a,\mu\rangle \in T_M^{\mathrm{det}}(\langle l,v\rangle)$. Then there is no delay-labelled transition out of $\langle l,v\rangle$, neither in $\llbracket M \rrbracket$ nor in $\mathrm{tdet}(\llbracket M \rrbracket)$, but all action-labelled transitions are present in both. We thus set $\mathfrak{S}_{\mathrm{det}}(\pi) = \mathfrak{S}(\pi)$.
b) There is a delay-labelled transition $\langle t, \mathscr{D}(\langle l,v+t\rangle)\rangle \in T_M^{\mathrm{det}}(\langle l,v\rangle)$. Then there is no action-labelled transition out of $\langle l,v\rangle$, neither in $\llbracket M \rrbracket$ nor in $\mathrm{tdet}(\llbracket M \rrbracket)$, so $\mathfrak{S}$ chooses some delay. We set

$$\mathfrak{S}_{\mathrm{det}}(\pi) = \mathscr{D}(\langle \mathit{maxDelay}(\langle l,v\rangle), \mathscr{D}(\langle l,v+t\rangle)\rangle).$$

In the other direction, if $\mathfrak{S}_{\mathrm{det}}$ is given and we consider a path $\pi \in \mathrm{Paths}_{\mathrm{fin}}(\llbracket M \rrbracket)$ with $\mathrm{last}(\pi) = \langle l,v\rangle$ that only contains delays chosen by *maxDelay*, i.e. π is also in $\mathrm{Paths}_{\mathrm{fin}}(\mathrm{tdet}(\llbracket M \rrbracket))$, the cases are:

a) There is an action-labelled transition $\langle a, \mu \rangle \in T_M(\langle l, v \rangle)$. Then we again let the schedulers make the same choice, i.e. $\mathfrak{S}(\pi) = \mathfrak{S}_{\text{det}}(\pi)$.
b) There is a delay-labelled transition $\langle t, \mathscr{D}(\langle l, v+t \rangle) \rangle \in T_M(\langle l, v \rangle)$. If $\langle l, v \rangle$ is also a state of $\text{tdet}([\![M]\!])$, then we set $\mathfrak{S}(\pi) = \mathscr{D}(\langle t', \mathscr{D}(\langle l, v+t' \rangle) \rangle)$ where $t' = \mathit{maxDelay}(\langle l, v \rangle)$.

Otherwise, if $\pi \notin \text{Paths}_{\text{fin}}(\text{tdet}([\![M]\!]))$, then $\mathfrak{S}(\pi)$ can be anything (as long as $\mathfrak{S}$ remains a valid time-divergent scheduler) because that path is not possible in $\text{ind}([\![M]\!], \mathfrak{S})$ in the first place.

When $\mathfrak{S}_{\text{det}}$ is given, we have $\text{ind}(\text{tdet}([\![M]\!]), \mathfrak{S}_{\text{det}}) = \text{ind}([\![M]\!], \mathfrak{S})$ by construction. When $\mathfrak{S}$ is given, the only difference between the two induced systems is that $\text{ind}([\![M]\!], \mathfrak{S})$ may make complex probabilistic sequences of delay-labelled transitions where there is only a single such transition with delay according to *maxDelay* in $\text{ind}(\text{tdet}([\![M]\!]), \mathfrak{S}_{\text{det}})$. However, due to time determinism, time additivity and $\mathfrak{S}$ being time-divergent, we see the same intermediate changes in state labelling with probability one in both, and we reach the exact same next state with an action-labelled transition with probability one in both, too. Consequently, all reachability probabilities for probabilistic timed reachability queries must be the same for both schedulers. Due to time additivity, rate rewards are also preserved, and thus the values of expected-reward queries. □

As a consequence of all of the above, we could have used TDPTA in our definition of the class of PTA that actually are an MDP towards the end of Section 5.1. However, the conditions of Definition 79 are requirements on the level of the semantics of a PTA, while we were looking for "syntactic" requirements that would allow us to identify a PTA as an MDP without having to consider its semantics.

SMC for TDPTA

We have seen above that the semantics of a TDPTA is actually an MDP that satisfies the same properties (including expected-reward ones if we convert rate rewards on the TPTS into transition rewards in the MDP by calculating the multiplication of Definition 73 during the transformation). In order to perform an SMC analysis of a TDPTA, we can thus just perform an SMC analysis of its MDP semantics using any of the methods we previously described in Section 4.6. In particular, discrete nondeterministic choices are allowed in TDPTA and are preserved in its MDP semantics—the problems of applying SMC to such a model thus remain, but all solutions that we develop for MDP can be reused, too. To keep memory efficiency, the MDP semantics would not be constructed explicitly and stored in memory, but computed on-the-fly, the main task

TD_{min}	TD_{max}	TS	P_Collision	P_Success	P_Failure	P_Success3	E_Time
2	2	5	0	0.9913	0.0087	0.9042	4.4875

Table 5.4: SMC analysis results for the simple probabilistic-timed BRP

being an implementation of the *maxDelay* function, which is however straightforward.

Example 42. Let us now perform an SMC analysis of the simple probabilistic-timed BRP model of Example 35. When we set $TD_{min} = TD_{max}$, it turns out to be a TDPTA whose MDP semantics is deterministic, too. We use $TD_{min} = TD_{max} = 2$ and $TS = 5$ for the sender timeout, and check the following properties:

```
– property P_Collision = Pmax(<> collision);
– property P_Success = Pmin(<> success);
– property P_Failure = Pmax(<> failure);
– property P_Success3 = Pmin(<> success && time <= 4);
– property E_Time = Xmax(time | success || failure);
```

Because the entire model is now deterministic, we can soundly simulate the MDP semantics as if it were a DTMC and use the APMC method with $\varepsilon = 0.01$ and $\delta = 0.95$, so we need to generate $k = 18450$ simulation runs. In the end, we observe the means shown in Table 5.4.

5.6.2 Scheduling in Time

In the case where the PTA at hand is not in fact a TDPTA, but we want to perform an SMC analysis, we have to resolve the nondeterministic selections of delays. This is in addition to the need to resolve the (discrete) nondeterministic choices between different enabled edges, which can be done similar to how we described it for MDP in Section 4.6.1: A resolver $\mathfrak{R}$ for a PTA gets the current location and the current valuation of the clocks and returns a distribution over edges. It is thus a function in $Loc \times Val \to \mathrm{Dist}(\mathscr{C} \times A \times \mathrm{Dist}(\mathscr{P}(\mathscr{C}) \times Loc))$ such that

$$\langle g, a, \mu \rangle \in \mathrm{support}(\mathfrak{R}(\langle l, v \rangle)) \Rightarrow \langle g, a, \mu \rangle \in E(l) \wedge [\![g]\!](v).$$

In order to resolve delays, we may use a time scheduler $\mathfrak{S}$ which selects one value out of the delays allowed by the invariants for the current valuation. The selection may be randomised, and it is usually performed in a way such that the guard of at least one edge is satisfied or time diverges after the chosen delay. Similar to the resolver, the random selection is achieved by returning a

probability measure. A time scheduler $\mathfrak{S}$ is thus a function in

$$(Loc \times Val) \times \mathscr{B}(\mathbb{R}_0^+) \times \mathscr{B}(\mathbb{R}_0^+) \to \text{Prob}(\mathbb{R}_0^+ \cup \{\infty\})$$

such that if $\mathfrak{S}(\langle\langle l, v\rangle, I_i, I_g\rangle) = \mu$ and $J \in \mathscr{B}(\mathbb{R}_0^+)$, then

$$I_i \cap J = \varnothing \Rightarrow \mu(J) = 0$$

where I_i will be the interval $[0,x]$ or $[0,y)$ with $x \in \mathbb{R}_0^+$ and $y \in \mathbb{R}^+ \cup \{\infty\}$ of delays allowed by the invariant (which cannot be empty due to the weak invariant semantics) and I_g will be the measurable set of delays that result in at least one edge being enabled. A time scheduler thus resolves continuous nondeterminism over time in a stochastic manner. It may include ∞ in its selection when I_i is $[0,\infty)$ to indicate that time should immediately diverge without any more edges being taken.

Algorithm 23 shows the adaptation of the `simulate` function of Algorithm 15 to handle PTA when given a time scheduler $\mathfrak{S}$. The "reduction" of a PTA w.r.t. a time scheduler results in an object that belongs to a model class more general than PTA due to the continuous probability measures used to select delays, namely that of time-deterministic stochastic timed automata (TDSTA), which we present in some more detail in the next chapter (Section 6.6). What Algorithm 23 does is in effect to generate and explore the underlying TDSTA on-the-fly. In particular, the delays that are possible without at any point violating the current location's invariant are computed in line 6. In line 7, the delays that lead to enabled edges are computed. Then the scheduler is invoked on these sets of delays in line 9, and subsequently a particular delay is sampled from the resulting measure. The loop starting in line 11 performs the selected delay, but makes sure that all intermediate changes in state labelling are observed. In order for the min operation in line 12 to be valid, we have to restrict to PTA that use only closed clock constraints in their set of atomic propositions. By using $\infty - x \stackrel{\text{def}}{=} \infty$ and assuming $\infty > x$ for all $x \in \mathbb{R}$, we get the correct behaviour in case the time scheduler decides to let time diverge: We observe all state labellings that are reachable by just letting time pass before we can conclude that ϕ cannot be satisfied any more and we return *false*. Algorithm 23 is very close to the actual implementation of PTA simulation in the modes tool of the MODEST TOOLSET, which allows the user to select from a set of predefined generic time schedulers (and resolvers), but also handles e.g. deadlines, which we do not cover here.

There are various "natural" or "interesting" time schedulers. We already hinted at some of them in Example 21, where we also saw that their use, just like the use of resolvers, cannot lead to sound SMC results. Let us nevertheless formally define some time schedulers that are popular in practice:

function `simulate`$(M = \langle Loc, \mathscr{C}, A, E, l_{init}, Inv, AP, L\rangle, \mathfrak{R}, \mathfrak{S}, \phi, d)$
  $\langle l, v\rangle := \langle l_{init}, \mathbf{0}\rangle$, $seen := \varnothing$, $i := 1$
  while $i \leq d$ **do**
    if $\phi(L_M(\langle l, v\rangle)$ **then return** *true*
    else if $\langle l, v\rangle \in seen$ **then return** *false*
    $I'_i := \{ t \in \mathbb{R}_0^+ \mid [\![Inv(l)]\!](v+t) \wedge \nexists t' \in \mathbb{R}_0^+ : t' < t \wedge \neg [\![Inv(l)]\!](v+t') \}$
    $I_g := \{ t \in \mathbb{R}_0^+ \mid \exists\, l \xrightarrow{g,a} \mu : [\![g]\!](v+t) \}$
    if $I'_i \cup \{0\} = \{0\} \wedge I_g \cap \{0\} = \varnothing$ **then return** *false* // *timelock*
    $\mu_{\mathfrak{s}} := \mathfrak{S}(\langle \langle l, v\rangle, I'_i \cup \{0\}, I_g\rangle)$ // $\cup\{0\}$ *for weak invariants*
    $t_{\mathfrak{s}} :=$ choose a delay $t_{\mathfrak{s}}$ randomly according to $\mu_{\mathfrak{s}}$
    repeat
      $t := \min(\{t_{\mathfrak{s}}\} \cup \{t' \in \mathbb{R}^+ \mid L_M(\langle l, v\rangle) \neq L_M(\langle l, v+t'\rangle)\})$
      if $t = \infty$ **then return** *false*
      $v := v+t$, $t_{\mathfrak{s}} := t_{\mathfrak{s}} - t$, $i := i+1$
      if $\mu_{\mathfrak{s}}$ is Dirac **then** $seen := seen \cup \{\langle l, v\rangle\}$ **else** $seen := \varnothing$
      if $\phi(L_M(\langle l, v\rangle)$ **then return** *true*
      else if $\langle l, v\rangle \in seen$ **then return** *false*
      else if $i > d$ **then return** *unknown*
    until $t = 0$
    if $\exists\, l \xrightarrow{g,a} \mu : [\![g]\!](v)$ **then**
      $\mu_{\mathfrak{R}} := \mathfrak{R}(\langle l, v\rangle)$
      $\nu :=$ choose an edge $\langle g, a, \nu\rangle$ randomly according to $\mu_{\mathfrak{R}}$
      if $\mu_{\mathfrak{R}}$ and ν are Dirac **then**
        $seen := seen \cup \{\langle l, v\rangle\}$
      else
        $seen := \varnothing$
      end
      $\langle X, l'\rangle :=$ choose reset and location randomly according to ν
      $\langle l, v\rangle := \langle l', v[X \mapsto 0]\rangle$, $i := i+1$
    end
  end
  return *unknown*

Algorithm 23: Path generation for a PTA, a resolver, and a time scheduler

– **ASAP**, the *as soon as possible* scheduler, always selects the minimum value in $I_i \cap I_g$, i.e.

$$\mathfrak{S}_{\text{ASAP}}(\langle s, I_i, I_g\rangle) = \begin{cases} \mathscr{D}(\min(I_i \cap I_g)) & \text{if } I_i \cap I_g \neq \varnothing \\ \mathscr{D}(\min(I_i)) & \text{otherwise.} \end{cases}$$

In order for the minima to always exist, care must be taken during modelling to use closed clock constraints for the relevant lower bounds.

– **ALAP**, the *as late as possible* scheduler, always selects the maximum value in $I_i \cap I_g$ or lets time diverge, i.e.

$$\mathfrak{S}_{\text{ALAP}}(\langle s, I_i, I_g\rangle) = \begin{cases} \mathscr{D}(\max(I_i \cap I_g)) & \text{if } I_i \cap I_g \neq \varnothing \\ \mathscr{D}(\max(I_i)) & \text{otherwise} \end{cases}$$

where $\max([x,\infty)) \stackrel{\text{def}}{=} \max((x,\infty)) \stackrel{\text{def}}{=} \infty$ for all $x \in \mathbb{R}$. Again, in order for the maximum values to always exist, care must be taken during modelling to use closed clock constraints for the relevant upper bounds.

– **INV**, the *invariant* scheduler, is a variant of ALAP: it always waits as long as the invariant allows without taking into account whether an edge is then enabled or not, i.e.

$$\mathfrak{S}_{\text{INV}}(\langle s, I_i, I_g\rangle) = \mathscr{D}(\max(I_i))$$

assuming, as usual, that the model is such that the maximum exists.

– **St**, the *stochastic* scheduler, uses the probability measure that gives maximum entropy for the possible set of delays at hand. For bounded delays, it thus uses the continuous uniform distribution, while the exponential distribution with some given rate λ is used for unbounded delays. Formally:

$$\mathfrak{S}_{\text{St}}(\langle s, I_i, I_g\rangle) = \begin{cases} \textsc{Uni}(\inf(I_i \cap I_g), \sup(I_i \cap I_g)) & \text{if } \sup(I_i \cap I_g) \in \mathbb{R} \\ \inf(I_i \cap I_g) + \textsc{Exp}(\lambda) & \text{otherwise} \end{cases}$$

with $\sup([x,\infty)) \stackrel{\text{def}}{=} \sup((x,\infty)) \stackrel{\text{def}}{=} \infty \notin \mathbb{R}$ for all $x \in \mathbb{R}$ and under the assumptions that I_g, like I_i, is a single interval and $I_i \cap I_g \neq \varnothing$. The first assumption is only to achieve a more readable presentation; it can easily be removed by distributing the probability mass over the multiple intervals and assigning probability mass zero to all delays in between.

Example 43. Let us now use Algorithm 23 to perform an SMC analysis of the same simple probabilistic-timed BRP model that we used in the previous example with the same parameters for the statistical evaluation, but allowing TD_{min} and TD_{max} to have different values. We observe from the results that the model does not have timing anomalies (i.e. situations where, for example,

longer local delays at some points lead to a reduction in an overall global delay): If we use $\mathfrak{S}_{\text{ASAP}}$, this has the same effect as setting TD_{max} to the value of TD_{min}. Conversely, using the $\mathfrak{S}_{\text{ALAP}}$ scheduler is the same as setting TD_{min} to the value of TD_{max}. SMC with $\mathfrak{S}_{\text{INV}}$ leads to the same results as using $\mathfrak{S}_{\text{ALAP}}$. Finally, if we use $\mathfrak{S}_{\text{St}}$, we get results that are between those provided by the $\mathfrak{S}_{\text{ASAP}}$ and $\mathfrak{S}_{\text{ALAP}}$ time schedulers.

5.6.3 Implicit Stochastic Semantics

Time schedulers as presented above have a *global* view of delays in a PTA model. However, when the model at hand is actually a network of PTA, taking its component structure into account can allow the specification of even more interesting time schedulers. One particular such scheduler has been implemented in the UPPAAL SMC tool [DLL+11b] to allow SMC of timed and probabilistic timed automata specified with UPPAAL's graphical frontend. Although such an analysis is not, in general, sound according to the PTA analogue of Definition 53, it is interesting to look into how UPPAAL SMC resolves nondeterminism and what the consequences are.

Component-wise stochastic scheduling In order to use UPPAAL SMC, an NPTA model must conform to certain restrictions that are necessary for the time scheduler used and that improve simulation performance [DLL+11a, Hof13]:

- Only *broadcast* synchronisation can be used: On edges, the actions (except for τ) are suffixed by either "?" or "!" to indicate that they represent an input or an output of the component automaton, respectively. During parallel composition, for an action $\mathtt{a} \neq \tau$, an edge labelled $\mathtt{a}!$ can always be taken, irrespective of the other component automata. However, it then synchronises with $\mathtt{a}?$-labelled edges in all the automata in which such an edge is enabled. An $\mathtt{a}?$-labelled edge cannot be taken alone. In this way, broadcast synchronisation makes sure that all components that are ready to synchronise do so, but the other components do not prevent the synchronisation from happening. It thus provides input-enabledness for all components on all actions $\mathtt{a}$.
- An action $\mathtt{a}$ can only appear as an output $\mathtt{a}!$, and a clock c can only be modified (i.e. reset), in at most one component automaton.
- The automata must be well-formed, i.e. it must not be possible to reach a location such that its invariant is immediately violated. This is only due to UPPAAL's use of a strong invariant semantics.
- Guards and invariants must be basic diagonal-free clock constraints. Location invariants must correspond to upper bounds on the progress of time, or a *rate expression* must be associated to the location.

For a model that satisfies these requirements, UPPAAL SMC then selects delays and edges according to the following simulation procedure [DLL+11a, Hof13]:

1. For each component automaton M, calculate the upper bound d^M_{max} on the progress of time given by its current location's invariant. Then select the delay d^M as follows:
   (a) If $d^M_{max} = \infty$, sample d^M from the exponential distribution whose rate is given by the current value of the location's rate expression.
   (b) If $d^M_{max} = x \in \mathbb{R}^+_0$, compute d^M_{min} as the infimum delay after which at least one edge is enabled, then sample d^M from the continuous uniform distribution over the interval $[d^M_{min}, d^M_{max}]$.
2. Let $M_{win} = \arg\min_M d^M$ bet the component automaton that has picked the smallest delay, i.e. the one that has "won the race" between the components.
3. Let time advance by $d^{M_{win}}$ time units.
4. Let F be the set of edges of M_{win} that are now enabled and that are labelled with an output or with τ.
   (a) If $F = \varnothing$, go to step 1.
   (b) Otherwise, pick an edge $e \in F$ according to $\mathscr{U}(F)$, perform that edge, and go to step 1.

Edges can additionally be assigned weights in order to make some of them more likely than others in step 4b.

The basic idea of selecting delays and edges in UPPAAL SMC is similar to the stochastic scheduler $\mathfrak{S}_{\mathrm{St}}$. The key difference is that the UPPAAL SMC method is built around the idea of independence of components: Every component selects a delay on its own, without taking the state of the other components into account; only after all components have made their decision, a race semantics simply selects the one that chose the smallest delay to go forward and execute one of its (output) edges. In a sense, the resulting scheduler is thus *distributed*: Its decisions are only based on local knowledge.

Surprising consequences Although the scheduling policy of UPPAAL SMC may appear simple and intuitive at first glance, it has surprising consequences that, again, highlight that performing SMC with some arbitrary scheduler cannot, in general, deliver trustworthy results:

Example 44. If we perform an SMC analysis of the PTA shown in Figure 5.12 using UPPAAL SMC to compute the probabilities to reach locations l_1 or l_2, respectively, within some time bound ≥ 10, we obtain the following results:

$$[\![\mathrm{P}_{\max}(\diamond\, \mathit{one})]\!] \approx 0.1 \text{ and } [\![\mathrm{P}_{\max}(\diamond\, \mathit{two})]\!] \approx 0.9$$

Clearly, these values are far from the actual probabilities, which are both 1, but this is of course to be expected in an unsound SMC approach. However, what

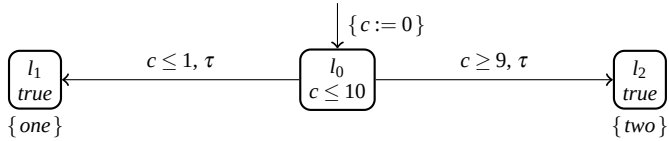


Figure 5.12: A PTA with counterintuitive semantics in UPPAAL SMC [Hof13]

should be surprising, knowing the way that UPPAAL SMC selects delays and edges, is that the edges to reach each of the locations are both enabled for one tenth of the total time, yet the reachability probabilities are different. This is due to the fact that the uniform distribution used in step 1b is over the entire interval of time between the first edge being enabled and the upper bound specified by the invariant. In particular, whether edges become disabled again inside this interval is not taken into account. Thus, when a delay in the interval $(1,9)$ is sampled, which happens with probability $8/10$, step 4a has to be applied, and that probability mass is effectively added to the probability of reaching location l_2.

Arguably, this example shows once again that using particular schedulers to perform SMC for nondeterministic models, no matter whether they resolve discrete nondeterminism as for MDP or additionally nondeterministic delays as in this case, is dangerous unless one fully understands how the scheduler behaves for the entirety of the model at hand. Usually, reaching this level of understanding is as hard as solving the verification problem for the model in the first place.

5.6.4 SMC for General PTA

So far, we have not seen a way to perform sound SMC for general PTA, i.e. those that may have discrete nondeterministic choices as well as nondeterministic delays. In fact, to the author's knowledge, no such approach currently exists, but there are some obvious possibilities whose practical viability, however, is currently doubtful:

– We saw in Section 5.5 that the **region graph** is PTCTL-equivalent to a PTA's TPTS semantics. As the region graph is an MDP, we can apply all techniques that allow SMC for nondeterministic MDP to the region graph of a general PTA. Unfortunately, we also saw in Section 4.8 that there is currently no convincing SMC technique for general nondeterministic MDP. While the region

graph may be "less nondeterministic" than the original PTA if nondeterministic delays correspond to exactly one region, we expect many nondeterministic choices between letting time pass (i.e. advancing to the successor region) or taking an edge to remain.
– In a similar direction, we could use the **digital clocks** approach to transform the PTA into an MDP instead. The main advantage of using digital clocks compared to regions for exhaustive model checking was that we achieve smaller state spaces at the cost of some expressivity. However, for a "true" SMC approach that has (near-)constant memory usage, the size of the state space does not matter. Still, since a digital clocks valuation covers more TPTS states than a region, the digital clocks MDP may contain slightly fewer nondeterministic choices than the region graph.
– In both of the above approaches, we could use **POR or confluence reduction** to perform SMC if the nondeterminism is spurious. However, as others have observed in attempts to define a useful partial order reduction technique for TA (see e.g. the overview of Minea [Min99]), the passage of time is an implicit synchronisation between all the components in a network of (P)TA. This means that many transitions are dependent in the sense of a condition like J1 of Section 4.7.1. It remains to be seen whether a reduction technique that works directly on the concrete state space, like our confluence-based technique for MDP, could overcome this problem. In any case, in order for such an approach to work, all nondeterministic delays would have to be spurious, i.e. it must not matter for the properties at hand which particular point in time is selected. We will see a model that is likely to fulfill this condition in the form of the BRP model with $TD_{min} \neq TD_{min}$, but without properties that have time bounds or references to rewards, in the next section.
– Finally, we may try to use **zones** to represent many regions at once during simulation. The hope is that all edges will be enabled over the entire duration represented by a particular zone and the nondeterminism over time thus does not appear as nondeterminism in the zone graph. However, for this to work, we would first need to prove whether SMC with zones would actually allow approximations of the actual reachability probabilities or merely of upper bounds as was the case for exhaustive model checking (see Section 5.5). Additionally, domain experts expect that any implementation of such an approach would not be competitive in terms of runtime due to the computational overhead involved in dealing with zones [Dav13].

Unsurprisingly, none of these ideas have been implemented in a tool or evaluated on case studies so far. It however remains future work to find out whether they actually do not work well, or whether there are relevant models that are amenable to an SMC analysis with one of these techniques.

5.7 A Bounded Retransmission Example

To conclude this chapter on probabilistic timed automata, let us look at an example PTA model that a) represents a real-life communication protocol, that b) is parameterised to allow the analysis of more or less realistic instances with exploding state spaces, and that c) is amenable to both exhaustive and statistical model checking: the bounded retransmission protocol (BRP).

This protocol, originally developed by Philips, has been studied with various techniques and focussing on many different aspects of correctness and performance since the early 1990s [HSV93, GvdP96, DKRT97, DJJL01]. Notably, a timed automata model was used to study the influence of different timeout values on the correctness of the procotol [DKRT97] using UPPAAL, and a probabilistic, but untimed, PRISM model was used for a quantitative probabilistic analysis [DJJL01]. The first model that combined these aspects [HH09] was built as a network of PTA in MODEST and is what we study in this section as "the (full) BRP model".

The Model

The simple BRP model that we have developed as a running example throughout this thesis, starting from the VLTS model of Example 8 and ending with the probabilistic timed VPTA model of Example 35, actually contains all the important ideas of the full BRP model: Data is transferred from a sender to a receiver over lossy communication channels, and there is an upper bound on the number of retransmissions. The main difference between the full model and the simple models is that the scenario of the full model is the transmission of an entire file consisting of N chunks of data (called *frames* here), and both sender and receiver contain more complex logic to keep track of the current state. In particular, they determine whether they are at the start of the file, in the middle, and whether they have given up such that the transmission failed or they are unsure if it was successful (which can happen if the very last acknowledgment is lost repeatedly). Additionally, instead of having upper and lower bounds on the transmission delay, it is fixed to be nondeterministic over the interval $[0, TD]$. Furthermore, the probability of message loss of the channel from sender to receiver is 0.02, while that of the channel that carries acknowledgments back from the receiver to the sender is just 0.01, modelling the fact that acknowledgments are shorter and thus less likely to be hit by interference. The parameters of the full model are thus N, the number of chunks/frames to transmit; MAX, the maximum number of retransmissions per frame (which was fixed to 1 in our simple model); and TD, the maximum transmission delay. The

```
bool ff, lf, ab; // channel data: first/last frame, alt. bit
bool premature_timeout;
process Receiver() {
   bool r_ff, r_lf, r_ab, bit;
   clock c;
   // receive first frame
   if(ff) {
      get_k {= c = 0, bit = ab, r_ff = ff, r_lf = lf, r_ab = ab =}
   } else {
      get_k {= c = 0, premature_timeout = true =}; stop
   };
   do {
   :: invariant(c <= 0) {
         if(r_ab != bit) {
            // repetition, re-ack
            put_l
         }
         else {
            // report frame to upper layers
            if(r_lf) { r_ok }
            else if(r_ff) { r_fst }
            else { r_inc };
            put_l {= bit = !bit =} // ack
         }
      };
      invariant(c <= TR)
      {
         alt {
         :: // receive next frame
            get_k {= c = 0, r_ff = ff, r_lf = lf, r_ab = ab =}
         :: // timeout
            when(c == TR)
            if(r_lf) {
               // we just got the last frame, though
               r_timeout; break
            }
            else {
               r_nok;
               // abort transfer
               r_timeout; break
            }
         }
      }
   };
   Receiver()
}
```

Figure 5.13: The receiver process for the bounded retransmission protocol

MODEST code of the full BRP model is part of the MODEST TOOLSET download, and we show its receiver process, which gained the most in complexity compared to the simple models, in Figure 5.13.

Properties

We analyse the following five properties on the full bounded retransmission protocol model using exhaustive and statistical model checking:

– T_{A1} checks for the absence of premature timeouts in the receiver, i.e. for the situation that the receiver thinks that the transmission of a file was aborted by the sender when in fact it was not. The result is *true* if no premature timeout is possible, and *false* otherwise. We expect the verification result $[\![T_{A1}]\!] = \mathit{true}$ if the protocol is correct. It should be noted that this property is not a probabilistic timed reachability or expected-reward property, but a non-probabilistic invariant property that is specified in MODEST as

```
property T_A1 = A[](!premature_timeout);
```

where the Boolean variable `premature_timeout` is set inside the receiver process as shown in Figure 5.13.

– P_A is a probabilistic reachability property that queries for the maximum probability that eventually the sender reports a certain unsuccessful transmission but the receiver got the complete file. In MODEST syntax, the property is written as

```
property P_A = Pmax(<> s_nok_seen && r_ok_seen);
```

where the two Boolean variables observed in the property are set in the model when the corresponding conditions are satisfied for the first time; this can be seen in Figure 5.13 for `r_ok_seen`. We expect the verification result to be 0 if the protocol is correct, i.e. the view of sender and receiver of what has been successfully transmitted should not go out of sync.

– P_1 is similar to P_A: This probabilistic reachability property asks for the maximum probability that eventually the sender does not report a successful transmission, but rather that it reports that transmission was not successful ("`nok`") or that it does not know whether it completed successfully ("`dk`" for "don't know"). As mentioned, the latter can happen if the acknowledgments for the last frame get lost. The property in MODEST is

```
property P_1 = Pmax(<> s_nok_seen || s_dk_seen);
```

In this case, we expect some—hopefully low—probability > 0: Due to the bound on the number of retransmissions, an unsuccessful transmission is actually possible.

– D_{max} is a probabilistic timed reachability property,

```
property D_max = Pmax(<> s_ok_seen && time <= 64);
```

in MODEST. It asks for the the maximum probability that the sender reports a successful transmission within 64 time units. The result for this property is expected to vary with all three of the model parameters.

– E_{max}, the last property we check, is an expected-reward property that computes the maximum expected time until the transfer of the first file is finished (successfully or unsuccessfully). In MODEST, it is written as

```
property E_max = Xmax(time | first_file_done);
```

using the predefined global time reward `time`.

Property T_{A1} was first analysed in [DKRT97]. P_A and P_1 are from [DJJL01]. D_{max} and E_{max} combine probabilities and time and were introduced together with the PTA model we study here in [HH09].

Analysis and Results

For exhaustive model checking, we first analyse the nondeterministic overapproximation of the model (cf. Definition 67) using mctau, which relies on UPPAAL for a timed automata analysis using zone-based techniques. We then use the digital clocks approach with both mcpta, which relies on PRISM for the MDP analysis (using PRISM's default "hybrid" engine that partially relies on variants of binary decision diagrams to reduce memory usage), and mcsta, which is a fully explicit-state model checker. Finally, we perform SMC using modes with the ALAP scheduler, for which we know (by manual analysis of the behaviour of the model) that it delivers the actual maximum probabilities/values for all the properties except for D_{max}. As in Example 42, we generate $k = 18450$ simulation runs to get $\varepsilon = 0.01$ and $\delta = 0.95$ in the APMC method. All measurements are then performed on a 1.7 GHz Intel Core i5-3317U system with 4 GB of RAM running 64-bit Windows 8.1.

The analysis results are shown in Table 5.5. For each model instance and analysis tool used, we show the probabilities respectively values we get for the different properties. mcsta and modes currently do not support the nonprobabilistic property T_{A1}, but would support computing the *probability* of a premature timeout. Conversely, mctau cannot handle the reward-based property E_{max}. In column "states", we give the maximum number of states that had to be kept in memory at any point in time during the analysis, while column "time" reports the total runtime needed to complete the analysis of all five properties. We should note that the number of states is an indicator of memory usage, but in practice, even similar numbers of states may require vastly different amounts

parameters	tool	T_{A1}	P_A	P_1	D_{max}	E_{max}	states	time
$N = 16$, $MAX = 2$, $TD = 1$	mctau	*true*	0	$[0,1]$	$[0,1]$	–	833	0 s
	mcpta	*true*	0	$4.23 \cdot 10^{-4}$	0.99958	33.473	170 885	6 s
	mcsta	–	0	$4.23 \cdot 10^{-4}$	0.99958	33.473	206 498	2 s
	modes	–	≈ 0	$\approx 3.8 \cdot 10^{-4}$	≈ 0.9996	≈ 33.46	2	13 s
$N = 16$, $MAX = 2$, $TD = 4$	mctau	*true*	0	$[0,1]$	$[0,1]$	–	833	0 s
	mcpta	*true*	0	$4.23 \cdot 10^{-4}$	0.99958	132.413	758 163	26 s
	mcsta	–	0	$4.23 \cdot 10^{-4}$	0.99958	132.413	789 271	12 s
	modes	–	≈ 0	$\approx 3.3 \cdot 10^{-4}$	≈ 0	≈ 132.42	2	13 s
$N = 64$, $MAX = 5$, $TD = 1$	mctau	*true*	0	$[0,1]$	$[0,1]$	–	7934	1 s
	mcpta	*true*	0	$4.48 \cdot 10^{-8}$	0.99999	133.897	2 212 828	178 s
	mcsta	–	0	$4.48 \cdot 10^{-8}$	0.99999	133.897	2 422 332	37 s
	modes	–	≈ 0	≈ 0	≈ 0	≈ 133.88	2	46 s
$N = 64$, $MAX = 5$, $TD = 4$	mctau	*true*	0	$[0,1]$	$[0,1]$	–	8151	1 s
	mcpta	*true*	0	$4.48 \cdot 10^{-8}$	0.99876	529.692	4 916 483	471 s
	mcsta	–	0	$4.48 \cdot 10^{-8}$	0.99876	529.692	5 002 765	149 s
	modes	–	≈ 0	≈ 0	≈ 0	≈ 529.73	2	47 s

Table 5.5: Analysis results and performance for the full BRP model

of memory depending on how the state space is represented in memory. In particular, we observed that mcsta consistently used between five and ten times as much memory as mcpta. This is likely due to PRISM's use of decision diagrams, which however comes at an obvious runtime cost.

The first observation we make when looking at the results that are shown in Table 5.5 is the clearly visible difference between the exhaustive and statistical model checking approaches. SMC needs only two states in memory at any time (it cannot do with only one because of the atomic assignment semantics of MODEST, which means that assignments cannot be performed on-the-fly on a single state), while exhaustive model checking needs to store up to about five million states on which to perform value iteration. On the other hand, the difference in runtime is not so clear; SMC is faster here for the larger instances, but does not produce useful results for at least P_1 because the actual probability is too small, i.e. the event to observe is too rare. Note that the reported result is still correct w.r.t. the confidence $\varepsilon = 0.01$ and $\delta = 0.95$ that we used. In order

to get numbers $\neq 0$, we would need a much smaller value for ε, consequently many more simulation runs, and thus also a much higher runtime. SMC however works well to estimate expected rewards, but it must be noted that the APMC method applies only to the estimation of probabilities. The reported value for E_{max} is thus simply the mean of the observations during simulation, without any attached confidence. However, we could, for example, have computed confidence intervals for it.

We furthermore see that the performance of both SMC and the analysis with mctau appear immune to changes in *TD*, i.e. to longer (nondeterministic) delays in the model. For SMC, this is because Algorithm 23 jumps from one point in time directly to the next point at which either the state labelling changes or an edge becomes enabled, no matter how much time passes in between. For mctau, this is rooted in UPPAAL using a zone-based approach, which can represent larger intervals of time within a single zone. Consequently, the state spaces that UPPAAL explores are much smaller than the ones of the digital clocks-based approaches, which suffer from state space explosion when *TD* is increased. mcsta generates slightly larger state spaces than mcpta because it currently does not perform certain optimisations such as an active clocks reduction.

Finally, we observe that mctau, despite working only on the nondeterministic overapproximation of the PTA, is able to give us useful results: We can already be sure that T_{A1} and P_A do deliver the expected results without having to perform the more expensive probabilistic analysis with mcpta or mcsta. Also as expected, SMC with modes and the ALAP scheduler gives us results consistent with exhaustive model checking for all properties save D_{max}. This is because it is actually advantageous to complete transmissions as *soon* as possible if one wants to *maximise* the probability of succeeding within a time bound. We could thus informally say that the switch between ASAP and ALAP is done automatically in exhaustive model checking depending on the property, whereas it is a decision of the user when performing SMC. We argue that this shows once again that using some scheduler to perform SMC is not a good idea in general and that there is a dire need for sound techniques to be developed instead.

5.8 Summary and Discussion

We extended MDP with real-time aspects in this chapter, which led to the model of probabilistic timed automata. They can be seen as the orthogonal combination of MDP and timed automata, the latter of which we defined as a submodel

of PTA here. An alterative name would thus be “timed MDP”. PTA are particularly well-suited to study communication protocols, as substantiated by our running example and full BRP case study. After the definition of PTA and their semantics, we have focused on the difference between PTA with invariants and the alternative model of PTA with deadlines. PTA can be modelled in MODEST, and we presented the necessary language constructs. For verification, we specified probabilistic timed reachability properties as an extension of the probabilistic reachability properties we already considered for MDP, and we introduced rewards and the corresponding class of expected-reward properties. We then reviewed the current state of the art in exhaustive PTA model checking, before looking into how statistical model checking could be applied.

As mentioned, there are two “competing” ways to restrict the passage of time in PTA: invariants and deadlines. In line with popular tools such as UPPAAL and PRISM, we focus on PTA with invariants. As a core contribution of this thesis, we have shown that the two ways differ in expressiveness and modelling convenience, but that there are large subclasses of PTA with invariants respectively deadlines that can be converted into each other. We provided such a transformation procedure for both directions, where the case of transforming deadlines into invariants was more involved. We have shown that the transformation is correct.

In our comparison, we took the definition of deadlines verbatim as originally given in [BDHK06]. It would be simple to modify this to achieve the same basic expressivity as invariants (while the compositionality advantages remain unchanged) by replacing the strict comparison by a non-strict one in the relevant time progress condition. The question, then, is whether the resulting type of deadlines is more or less useful than before, and whether the semantics still matches the intuitive understanding of what a deadline resp. an urgent edge represent. When it comes to compositionality, deadlines provide the ability to use various operators to combine the constraints of synchronising edges. In particular, the choice of operator is not limited to conjunction and disjunction [BS00], which we focused on. It may be worthwhile to investigate whether other operator combinations are elegant and useful in the setting of PTA and MODEST.

The problem of exhaustive model checking of PTA appears to be largely solved: There is a range of methods with different strengths and weaknesses, and useable tool implementations exist. While there is still room for the development of new methods that perform better, or that work well with complex logic-based properties, we instead focussed more on the problem of SMC for PTA, for which no reasonable solution currently exists. We have seen that it should be possible to adapt methods developed for MDP to PTA by using reductions such as the digital clocks approach, but it remains doubtful whether

these methods (in particular the POR- and confluence-based ones) can deal with the kinds of nondeterminism typically present in PTA, and whether they scale to large models (in particular concerning the more memory-intensive approaches such as the ones based on learning). In summary, no sound technique for SMC of general PTA currently exists, but further research in the area of SMC for MDP can likely be transferred to the PTA setting to a large extent.

Finally, in our study of the bounded retransmission protocol as a large example of a PTA model, we saw interesting tradeoffs between, and the characteristic behaviours of, exhaustive and statistical model checking. The model is a rather straightforward extension of the ideas that we introduced in the running example throughout this thesis so far, and highlights the strength of PTA in representing the aspects necessary to model communication protocols and the settings they need to operate in. This has been recognised early on in the development of PTA, and a number of different protocols have already been studied. These include the original PTA case study of the IEEE1394 root contention protocol [KNS03] and a safety-critical hard real-time system in the form of a wireless bike brake [GHK+11]. PTA with rewards have been used to study the energy implications of different options of IEEE 802.15.4 [GHP07], which is the basis of e.g. the ZigBee communication protocols, and to compare the energy efficiency of simple and distributed slotted Aloha MAC protocols as used in wireless sensor networks [YBKK11].

6 Stochastic Timed Automata

The model of probabilistic timed automata that we have seen in the previous chapter is suitable for modelling so-called *hard real-time* systems. In that setting, delays have exact upper and lower bounds, and timing requirements are strict. However, in many settings, this is either unrealistic (we may not know the exact bound, but merely what the average delay is) or it leads to models and requirements that are too strong w.r.t. the actual system (we may not even be interested in worst-case timings, but be satisfied with a mean response time that is low enough).

Consider the example of the electronic trading system that we mentioned earlier: If part of the communication necessary to execute an order is performed over the Internet, we will not be able to guarantee an upper bound on the time between order placement and execution due to the Internet's best-effort-only nature. However, we can probably work with an assumption that, given a particular connectivity setting, communication finishes on average within t time units with some quantifiable deviation. In that case, we want to be able to still guarantee, for example, that the expected time between order placement and execution is within $f(t) \pm t'$ in at least 95% of all cases.

As a standard example where it is not the requirements that need to be softened, but where we need stochastic delays inside a model, consider the handling of some kinds of customers at a service desk. This could be a cash register in a supermarket, or a server handling requests on a network. In the server case, let us assume that the service time is dominated by the seek time incurred when fetching some file from a hard disk. If we know some characteristics of the disk, we should be able to obtain a hard upper bound st_{max} on that time (e.g. for the worst case that the file is at one "end" of the platter and the disks heads are as far away as possible). Still, using a nondeterministic delay of $[0, st_{max}]$ may lead to an overly pessimistic analysis when we could also assume that, in normal operation, the disk heads are always at a uniformly distributed random position; we thus would like to model the delay as

exactly following $\mathscr{U}(0, st_{max})$ instead. For the case of the customers, on the other hand, we rarely have such inside knowledge, but we still need to include their *interarrival time*, i.e. the time between the arrival of one customer and the next, in the model. When all that we know is that they arrive at some rate λ, e.g. $\lambda = 5\,\text{customers}/\text{minute}$, we like to take the (continuous) distribution that maximises entropy for under this knowledge, which means that we model interarrival times as following the exponential distribution with rate λ.

For cases like these two examples where delays and time bounds are not "exact" or "strict", but rather (are allowed to) follow some probability measure over time, we are dealing with *soft real-time* systems. A number of modelling formalisms exist to capture such systems, including continuous-time Markov chains (CTMC) and derived models such as Markov automata [EHZ10], which are based on purely exponentially distributed delays, or the more flexible, but deterministic, generalised semi-Markov processes (GSMP, [HS87]). In this thesis, we focus on an extension of PTA that allows soft real-time aspects to be included in addition to hard real-time ones in a single model. Following the convention of using the word stochastic to refer to uses of continuous probability measures, we call this model *stochastic timed automata* (STA, [BDHK06]). On the level of syntax, they only add the ability to assign to variables values sampled from continuous probability measures and values selected non-deterministically from some (uncountable) sets. In the semantics, however, some important changes need to be made to accommodate these new features. In particular, we from now on use the stochastic-nondeterministic semantics for expressions and updates that we defined in Section 2.3.

We start this chapter with the customary introduction of the extensions in syntax and semantics necessary to extend PTA to STA (Section 6.1). This includes the usual overview of submodels, where however, in addition to precisely characterising the relationship to PTA, we also present a brief outlook on CTMC-based models and the challenges involved in mapping them to STA and vice-versa. We then show how to write STA models in MODEST (Section 6.3) and formally define the meaning of probabilistic reachability and expected-reward properties (Section 6.4). In Section 6.5, we show how to analyse these properties with an exhaustive model checking technique, which is the main contribution of this chapter: an algorithm to compute upper/lower bounds on maximum/minimum reachability probabilities and expected cumulative reward values in a given STA. The algorithm uses abstraction to convert the STA into a PTA, which can then be analysed using existing PTA model checking techniques (cf. Section 5.5). We show the correctness of the abstraction for the considered properties. The underlying theory was originally developed for stochastic hybrid systems [FHH+11, Hah13]; we explain how we take advantage

of the specialisation to timed systems to improve scalability, usability and applicability. We have implemented the new approach as part of the MODEST TOOLSET, which allows us to investigate its effectiveness and efficiency using four different example models. Finally, in Section 6.6, we briefly investigate to what extent the statistical model checking methods we have presented for PTA in the previous chapter can be transferred to the continuous-probability setting of STA.

Origins

Stochastic timed automata were originally introduced to define the symbolic semantics of the MODEST language [BDHK06]. The definitions that we give in Section 6.1 are based on a combination of [HHH14] and [HHHK13]; in both cases, the majority of the definition of the concrete semantics was originally written by Ernst Moritz Hahn. The example of Figure 6.1 was designed by the author for [HHH14]; likewise, Example 49 was worked out by the author and first appeared in that same article.

The STA analysis method we present in Section 6.5 has been developed in joint work with Ernst Moritz Hahn and Holger Hermanns [HHH14]. It is an adaptation to the STA setting of more general theory originally developed by Moritz for his Ph.D. thesis [Hah13]. Aside from triggering the research in the first place, Moritz wrote the correctness proof (in Section 6.5.1), while the rest of the section has been written by and is based on work (in particular the implementation in mcsta and the selection and evaluation of the case studies) performed by the author. The overview of related work has benefited from input by Holger.

6.1 Definition

Stochastic timed automata are by definition a model with variables, but in contrast to e.g. VMDP (Section 4.2), we do not restrict these variables to have countable domains. This allows us to not only include clocks, but also "normal" variables whose domain is $\mathbb{R}$, too. These variables can then be used for the two new features compared to PTA: the ability to assign to such real variables 1) values sampled from some arbitrary distribution or measure or 2) values selected nondeterministically from some set. We can thus use assignments like $x_1 := \text{UNI}(0, \sqrt{2})$ to sample a value out of $[0, \sqrt{2}]$ according to the continuous uniform distribution, $x_2 := \text{EXP}(\lambda)$ to select y using the exponential distribution with rate λ, $x_3 := \text{NORM}(5, 2)$ to sample the normal distribution with mean 5 and standard deviation 2, or $y := \mathsf{any}(z \mid z > \sqrt{2})$ to nondeterministically select a real value greater than $\sqrt{2}$, to name a few examples.

We want to make sure that, when we allow these non-clock variables to occur in a model, clocks still remain restricted to expressions that conform to the clock constraint pattern. To achieve this, we extend our definition of clock constraints in a conservative way such that they also allow Boolean expressions over non-clock variables as well as comparisons between clocks and non-clock variables. From now on, clock constraints over a set of clocks $\mathscr{C}$ are thus expressions in *Bxp* of the form

$$\mathscr{CC}(\mathscr{C}) ::= b \mid \mathscr{CC} \wedge \mathscr{CC} \mid \mathscr{CC} \vee \mathscr{CC} \mid c \sim e \mid c_1 - c_2 \sim e$$

where $\sim \in \{>, \geq, <, \leq, =, \neq\}$, $c, c_1, c_2 \in \mathscr{C}$ and $b \in Bxp$, $e \in Axp$ are clock-free expressions. With this extension, the model of stochastic timed automata can be defined as follows:

Definition 81 (STA)**.** A *stochastic timed automaton* (STA) is an 8-tuple

$$\langle Loc, Var, A, E, l_{init}, Inv, AP, L \rangle$$

where
– Loc is a countable set of locations,
– $Var \supseteq \mathscr{C}$ is a finite set of variables with a subset $\mathscr{C}$ of clock variables,
– $A \supseteq \{\tau\}$ is the automaton's countable alphabet,
– $E \in Loc \to \mathscr{P}(\mathscr{CC} \times A \times (Upd \times Loc \to Axp))$ is the edge function, which maps each location to a set of edges, which in turn consist of a guard, a label and a symbolic probability distribution over updates and target locations, with the restriction that assignments involving a clock $c \in \mathscr{C}$ must be of the form $c := 0$,
– $l_{init} \in Loc$ is the initial location,
– $Inv \in Loc \to \mathscr{CC}$ is the invariant function, which maps each location to a clock constraint that allows time to pass as long as it evaluates to *true*,
– AP is a set of atomic propositions, and
– $L \in Loc \to \mathscr{P}(AP)$ is the location labelling function.

The usual notation that we already used for previous models, such as using arrows to denote edges, can be applied to STA analogously. We can also extend STA with rewards as introduced for PTA in Definition 71, and the parallel composition operator of PTA (Definition 63) applies to STA analogously, too. Let us illustrate the capabilities of STA with a synthetic example that highlights all the essential features of the model:

Example 45. The graphical representation of an example STA with reward r is shown in Figure 6.1. Out of location l_2, the edge to l_4 can only be taken after a deterministic delay of 16 time units, while the one back to l_0 can be taken after any delay nondeterministically chosen out of $[8, 16]$. After 16 time units,

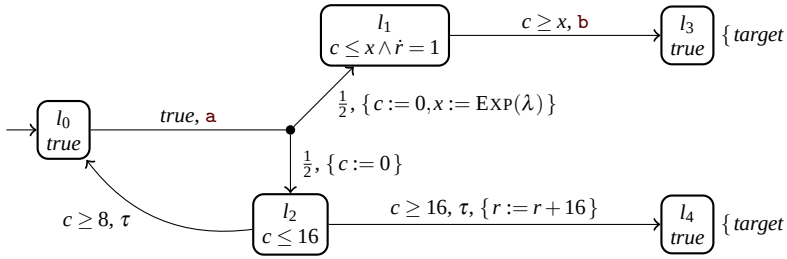


Figure 6.1: An example stochastic timed automaton

the choice of edge in l_2 thus becomes nondeterministic. The delay incurred in l_1, on the other hand, is stochastic: $x := \text{EXP}(\lambda)$ assigns to x a value sampled from the exponential distribution with rate λ, thus the delay is exponentially distributed with rate λ. The reward r keeps track of the time spent in l_1, and is increased by 16 upon entering l_4.

Semantics

The semantics of an STA is again a TPTS. We can in fact reuse most of Definition 61 verbatim except for the replacement of $\mathscr{C}$ by *Var* and the *jump* inference rule. The latter now needs to take into account the *symbolic* distribution in the STA as well as the sampling and nondeterministic expressions in the updates. Consider a location l and a symbolic probability distribution

$$m = \{ \langle U_1, l_1 \rangle \mapsto w_1, \langle U_2, l_2 \rangle \mapsto w_2, \dots \} \in \mathit{Upd} \times \mathit{Loc} \to \mathit{Axp}$$

and a distribution $\mu \in \text{Prob}(\mathit{Loc} \times \mathit{Val})$. For $i \in \mathbb{N}^+$, we let

$$M_i \stackrel{\text{def}}{=} \{ \mathscr{D}(l_i) \} \otimes [\![U_i]\!](v)$$

be the set of probability measures resulting from update U_i (remember that, by Definition 14, we have that $[\![U_i]\!] \in \mathit{Val} \to \Delta(\Sigma_{\mathit{Var}})$) extended to include the target locations. Let

$$w \stackrel{\text{def}}{=} \sum_i [\![w_i]\!](v) \in \mathbb{R}^+$$

be the total sum of the weights. Then the predicate $\mathit{Jmp}(v, l, m, \mu)$ holds if there are $\mu_i \in M_i \subseteq \text{Prob}(\mathit{Loc} \times \mathit{Val})$ such that μ is their weighted sum, i.e.

$$\forall A \in \mathscr{P}(\mathit{Loc}) \otimes \Sigma_{\mathit{Var}} : \mu(A) = \sum_i \frac{[\![w_i]\!](v)}{w} \mu_i(A).$$

The distribution μ over jump targets is specified such that the successor location is chosen according to the relative weights of the weight expressions (with the

usual requirements of non-zero weights and a positive finite sum as before). For a fixed successor location l_i, the distribution over the actual assignments results from a nondeterministic choice between the possible distributions represented by U_i. The predicate *Jmp* can now be used for the definition of the semantics of STA:

Definition 82 (Semantics of STA)**.** The semantics of an STA

$$M = \langle Loc, Var, A, E, l_{init}, Inv, AP, L \rangle$$

is the TPTS

$$[\![M]\!] = \langle Loc \times \mathrm{Val}(Var), \mathscr{P}(Loc) \otimes \Sigma_{Var}, \mathbb{R}_0^+ \uplus A, T_M, \langle l_{init}, \mathbf{0}_{Var} \rangle, AP, L_M \rangle$$

where $L_M(\langle l, v \rangle) = L(l) \cup \{\, e \in AP \cap Bxp \mid [\![e]\!](v) \,\}$ and T_M is the smallest function such that the following two inference rules are satisfied:

$$\frac{l \xrightarrow{g,a}_E m \quad Jmp(v,l,m,\mu) \quad [\![g]\!](v)}{\langle l, v \rangle \xrightarrow{a}_{T_M} \mu} \ (jump)$$

$$\frac{t \in \mathbb{R}^+ \quad \forall t' \leq t \colon [\![Inv(l)]\!](v+t')}{\langle l, v \rangle \xrightarrow{t}_{T_M} \mathscr{D}(\langle l, v+t \rangle)} \ (delay)$$

The semantics of rewards for STA is identical to their semantics for PTA, which is why we do not repeat Definition 73 here. The same applies to parallel composition.

6.2 Submodels

Stochastic timed automata are an extension of PTA, but by using the exponential distribution in sampling expressions, we should also be able to express continuous-time "Markovian" models such as continuous-time Markov chains as special cases of STA, as already postulated in Figure 1.2 in the introduction.

Probabilistic Timed Automata

If we do not allow non-clock variables and consequently cannot use sampling or nondeterminism in assignments (as clocks can only be reset to zero), an STA becomes a PTA:

Proposition 7. An STA

$$M = \langle Loc, Var, A, E, l_{init}, Inv, AP, L \rangle$$

where $Var = \mathscr{C}$ for a set of clock variables $\mathscr{C}$ is isomorphic to the PTA

$$M' = \langle Loc, \mathscr{C}, A, E', l_{init}, Inv, AP, L \rangle$$

where $l \xrightarrow{g,a}_{E'} \mu' \in E' \Leftrightarrow l \xrightarrow{g,a}_{E} \mu$ with $\mu'(\langle C, l\rangle) = \mu(\langle \{ c := 0 \mid c \in C \}, l\rangle)$ for all $C \subseteq \mathscr{C}$. In that case, we also have $[\![M]\!] \cong [\![M']\!]$, and we say that M *is* a PTA, and every PTA *is* an STA according to this correspondence.

Models Based on the Exponential Distribution

Because the exponential distribution can be used in assignments, we can model exponentially distributed delays in STA by following a simple recipe: With a dedicated clock c and non-clock variable x, we reset c to zero and assign to x a value sampled from the exponential distribution with rate λ. In the following location, we have invariant $c \leq x$ and a single outgoing edge with guard $c \geq x$. The time that has to be spent in that location then follows the exponential distribution with rate λ. The basic model that relies on exponentially distributed delays is that of continuous-time Markov chains:

Definition 83 (CTMC)**.** A *continuous-time Markov chain* (CTMC), for the purpose of this thesis, is a 5-tuple

$$\langle S, T, s_{init}, AP, L\rangle$$

where
– S is a countable set of states,
– $T \in S \to \mathscr{P}(\mathbb{R}^+ \times S)$ is the rate-transition function,
– $s_{init} \in S$ is the initial state,
– AP is a set of atomic propositions, and
– $L \in S \to \mathscr{P}(AP)$ is the state labelling function.

We assume w.l.o.g. that CTMC are deadlock-free, i.e. $T(s) \neq \varnothing$ for all $s \in S$. CTMC are the continuous-time counterpart of DTMC; similarly, they are mathematically just instances of stochastic processes (see e.g. [GS01, Section 6.9]), but we use an equivalent automata-based definition here. Informally, the meaning of a CTMC as defined above is as follows: When in some state $s \in S$, an outgoing transition $s \xrightarrow{\lambda} s'$ is taken after an amount of time that is exponentially distributed with rate λ. If there are multiple outgoing transitions, this becomes a race: In any particular run, the transition that "picks" the shortest delay is taken. An equivalent formulation is thus that the time spent in state s is exponentially distributed with rate parameter $\lambda(s) = \sum_{\langle \lambda, s'\rangle \in T(s)} \lambda$, and the target state is then chosen according to the distribution $\{ s' \mapsto \lambda/\lambda(s) \mid \langle \lambda, s'\rangle \in T(s) \}$. This exploits the fact that the minimum of two exponential distributions is an exponential distribution with the sum of the rates. A CTMC can be encoded as an STA:

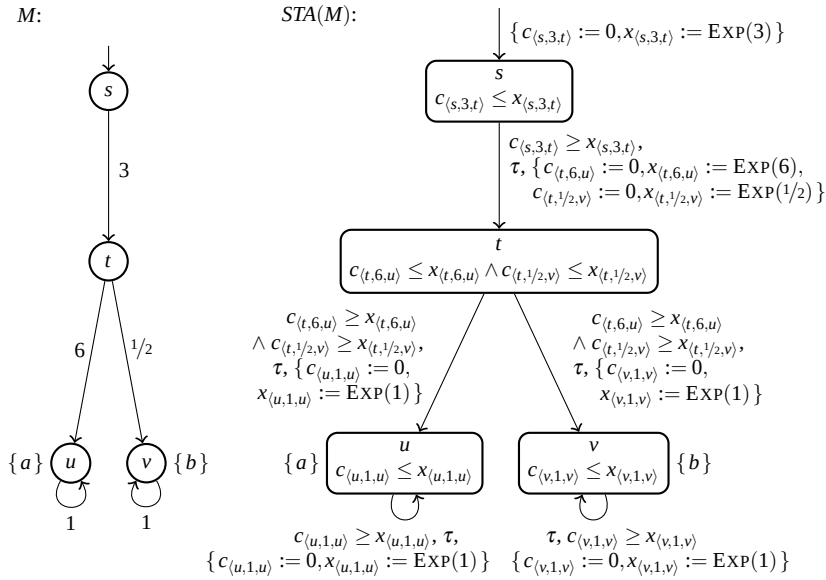


Figure 6.2: A CTMC and the corresponding STA

Definition 84. For a CTMC $M = \langle S, T, s_{init}, AP, L\rangle$, the corresponding STA is

$$STA(M) = \langle S \uplus \{ l_{init} \}, \mathscr{C}_T \uplus Var_T, \{ \tau \}, E, l_{init}, Inv, AP, L\rangle$$

with a set of clocks $\mathscr{C}_T = \bigcup_{s\in S}\{ c_{tr} \mid tr \in T(s) \}$, a set of non-clock variables $Var_T = \bigcup_{s\in S}\{ x_{tr} \mid tr \in T(s) \}$ with domain $\mathbb{R}_0^+$, and the edge function

$$\begin{aligned} E(s) = {} & \{ l_{init} \mapsto \{ \langle true, \tau, m(s_{init})\rangle \} \} \\ & \cup \{ s \mapsto \{ \langle c_{tr} \geq x_{tr}, \tau, m(s')\rangle \mid tr = \langle \lambda, s'\rangle \in T(s) \} \mid s \in S \} \end{aligned}$$

where $m(s') = \{ \langle \bigcup_{tr=\langle\lambda,s''\rangle\in T(s')}\{ c_{tr} := 0, x_{tr} := \textsc{Exp}(\lambda) \}, s'\rangle \mapsto 1 \}$ performs the assignments for the variables needed by the transitions of the target state. The invariant function is $Inv = \{ l_{init} \mapsto false \} \cup \{ s \mapsto \bigwedge_{tr\in T(s)} c_{tr} \leq x_{tr} \mid s \in S \}$.

Figure 6.2 shows an example for a CTMC and its corresponding STA. In the opposite direction, we could formulate restrictions on STA such that the transformation is applicable in both ways.

This correspondence between CTMC and STA relies on the intuitive understanding that the STA explicitly expresses the (implicit) semantics of the CTMC and two corresponding models are thus in some sense “equivalent”. For example, the reachability probability for some set of states are expected to

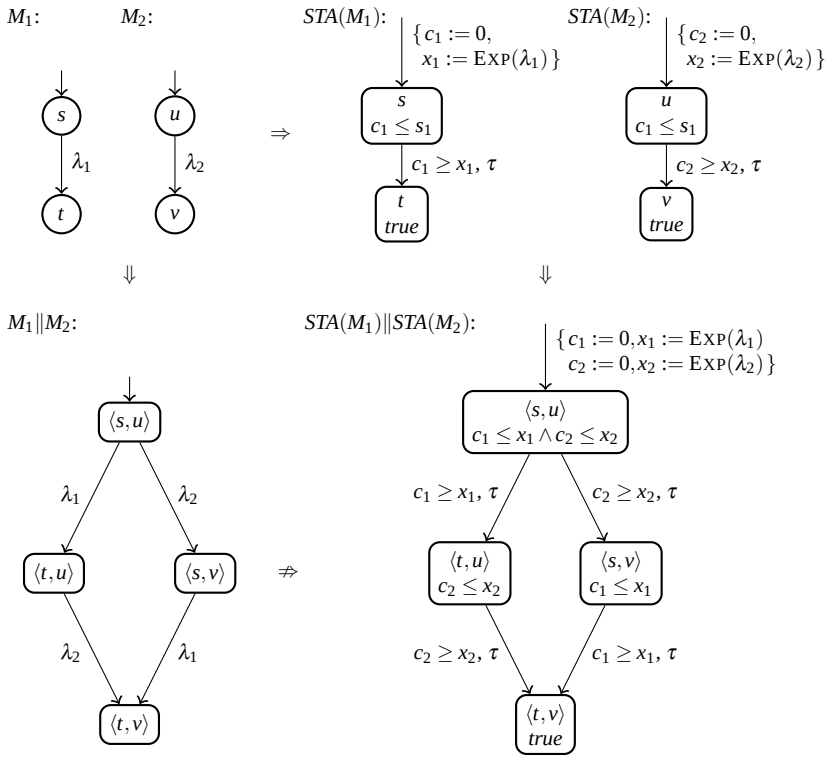


Figure 6.3: Non-compositionality of the CTMC-to-STA correspondence

be the same in the CTMC and in its corresponding STA, so the transformation preserves probabilistic reachability (modulo a formal definition of reachability properties for CTMC that we do not give in this thesis). However, it is not clear how far this equivalence goes, or how to capture it formally; in particular, the completely different semantics of CTMC (which directly correspond to stochastic processes and thus *are* their own semantics in a certain sense) and STA (in terms of dense probabilistic transition systems) makes reasoning about paths, bisimulations and similar notions difficult. Additionally, the transformation is not compositional, as the following example shows:

Example 46. Figure 6.3 shows two CTMC M_1 and M_2 (with the self-loops in the final states omitted and variable names shortened for clarity) as well as their corresponding STA $STA(M_1)$ and $STA(M_2)$. In contrast to DTMC,

the natural parallel composition operation for CTMC uses an interleaving semantics [HZ11]. The resulting parallel composition $M_1 \| M_2$, which is a CTMC, is shown on the bottom left of the figure. On the bottom right is the parallel composition of the two STA. The parallel composition operator used in that case is the one for STA, and the result is an STA. Alas, $STA(M_1) \| STA(M_2)$ is not the STA that corresponds to $M_1 \| M_2$, or, in other words, $STA(M_1) \| STA(M_2)$ and $STA(M_1 \| M_2)$ are not isomorphic. We see that in the former, the two exponential distributions are sampled only once, at the very beginning; then both clocks start to run and are never reset. In the latter, the (remaining) distributions would be resampled and the clocks reset on the τ-labelled edges as well.

Because of the exponential distribution's memoryless property, the parallel composition of STA corresponding to CTMC and the STA corresponding to the parallel composition of CTMC as shown in the previous example should actually behave "the same": It should not matter if we reset the timer for an exponentially distributed delay at any point or not. Defining a useful relation according to which two such STA would be equivalent has been a long-standing problem (that is not the focus of this thesis) to which a first solution has only appeared very recently [HKK14].

Nondeterministic memoryless-stochastic models We mention two particular extensions of CTMC that add nondeterministic choices in a compositional way, namely *interactive Markov chains* (IMC, [Her02]) and *Markov automata* (MA, [EHZ10]). IMC introduce a second kind of *interactive* transitions labelled with actions, whereas the usual rate-labelled transitions are now called *Markovian*. When a state has multiple outgoing interactive transitions, the selection of transition to take becomes a nondeterministic choice just like in LTS. An interesting point is that a *maximal progress* assumption is applied for closed (networks of) IMC: If a state has both interactive and Markovian transitions, the former take precedence and one of them is executed without delay. MA, as an extension of IMC, add probabilistic branching to interactive transitions, in the same way that branching is added when going from LTS to MDP. They are thus a generalisation of both IMC and MDP (cf. Figure 1.2).

For IMC and MA models, Definition 84 can be easily extended to provide corresponding STA, too; all the caveats we described for this correspondence then apply in the same way. Additionally, the issue of compositionality is compounded by the fact that the values sampled from distributions in an STA become part of the state space of its TPTS semantics. As a consequence, schedulers can use this knowledge to make different decisions depending on the outcome of a sampling assignment, which would not be possible in the original

IMC or MA. With increased scheduling power, certain properties may now lead to different results; we refer the reader to [HKK14] for details on this problem.

6.3 Modelling

The extensions we need to make to the MODEST modelling language to support the creation of STA models are very small: We only need to add support for sampling from probability distributions and measures and for nondeterministic selections to the expressions used on the right-hand sides of assignments. Formally, all we need to do is to say that these expressions need to be in *Sxp* as before, but keep in mind how the definition of *Sxp* changed for this chapter.

The MODEST TOOLSET implementation uses the same syntax for nondeterministic selections with any that we used in this thesis so far, and supports the following distributions in assignments:

- DISCRETEUNIFORM(n,m): the (discrete) uniform distribution $\mathscr{U}(n,m)$ for two integer parameters n and m with $n \leq m$, which corresponds to a random variable with mass function $\{k \mapsto 1/(m-n+1) \mid k \in \{n,\ldots,m\}\}$;
- BINOMIAL(p,n): the binomial distribution for a real parameter $p \in [0,1]$ and an integer $n \in \mathbb{N}^+$, which corresponds to a random variable of the number of successes in n independent Bernoulli trials, each of which has success probability p. Its mass function is $\{k \mapsto \binom{n}{k} \cdot p^k \cdot (1-p)^{n-k} \mid k \in \{0,\ldots,n\}\}$.
- POISSON(λ): the Poisson distribution with rate $\lambda \in \mathbb{R}^+$, whose probability mass function is $\{k \mapsto (\lambda^k/k!) \cdot \mathrm{e}^{-\lambda} \mid k \in \mathbb{N}\}$.
- GEOMETRIC(p), which takes the probability $p \in (0,1)$ as a parameter and has mass function $\{k \mapsto p \cdot (1-p)^{k-1} \mid k \in \mathbb{N}^+\}$.
- EXPONENTIAL(λ) or EXP(λ): the exponential distribution with rate $\lambda \in \mathbb{R}^+$, whose cdf is $\{x \mapsto 1-\mathrm{e}^{-\lambda x} \mid x \in \mathbb{R}\}$.
- UNIFORM(a,b) or UNI(a,b): the continuous uniform distribution for reals a, b with $a \leq b$ and cdf $\{x \mapsto 0 \mid x < a\} \cup \{x \mapsto \frac{x-a}{b-a} \mid x \in [a,b)\} \cup \{1 \mid x \geq b\}$.
- NORMAL(μ,σ) or NORM(μ,σ): the normal distribution with mean $\mu \in \mathbb{R}$ and standard deviation $\sigma \in \mathbb{R}^+$, which delivers normally distributed real values around μ that are within $[\mu-\sigma,\mu+\sigma]$ in about 68.2% and within $[\mu-2\sigma,\mu+2\sigma]$ in about 95.4% of all cases.

Example 47. In Example 35, we changed our simple BRP model to use a lossy channel that additionally incurred a transmission delay (Figure 5.5). That delay was modelled as nondeterministically chosen out of the interval $[TD_{min}, TD_{max}]$ for each transmission. In Figure 6.4, we show a different variant of this lossy channel in MODEST. This time, the transmission delay is still at least TD_{min} and at most TD_{max}, but the concrete value is selected according to the uniform distribution over this interval. The model of Figure 6.4 can thus be seen as a

```
process Channel()
{
   clock c;
   real x;

   snd palt {
   :98: {= c = 0, x = Uniform(TD_MIN, TD_MAX) =};
        invariant(c <= x) alt {
        :: when(c >= x) rcv
        :: snd {= collision = true =}; stop
        }
   : 2: {= c = 0, x = Uniform(TD_MIN, TD_MAX) =};
        invariant(c <= x) alt {
        :: when(c >= x) tau
        :: snd {= collision = true =}; stop
        }
   };
   Channel()
}
```

Figure 6.4: The lossy channel with stochastic transmission delay in MODEST

specific implementation, i.e. resolution of the nondeterministic choice, of the nondeterministic channel of Figure 5.5. (In fact, this is the effect that using the time scheduler $\mathfrak{S}_{\mathrm{St}}$ of Section 5.6.2 during simulation would have.) When analysing the nondeterministic channel (without predetermining a scheduler), all possible (combinations of) delays in $[TD_{min}, TD_{max}]$ have to be considered, including the uniform selection; the new stochastic channel of Figure 6.4 explicitly forces this resolution to be used. We have thus added more information to the model and thereby reduced the choices that can be made.

While STA allow a lot of flexibility in how probability measures and sampling can be used, their most common purpose is to model a stochastic delay as shown in the example we have seen so far. For this particular use of modelling a delay whose upper and lower bounds are the same and follow a particular (continuous) probability distribution, the MODEST TOOLSET provides a convenient `delay` shorthand:

$$\texttt{delay}(e_{delay}, e_{cond})\ P \stackrel{\text{def}}{=} Delay()$$

where $e_{delay} \in Sxp$, $e_{cond} \in Bxp$ and $Delay$ is a new, unique process name for every occurrence of the `delay` shorthand that is defined as

$$\texttt{process}\ Delay()\ \{\ \texttt{clock}\ c;\ \texttt{real}\ x = e_{delay};\ \texttt{invariant}(c \leq x \vee \neg e_{cond})\ \texttt{when}(c \geq x \wedge e_{cond})\ P\ \}.$$

```
const int C = 5; // maximum queue length
int(0..C) q;     // current queue length

process Arrivals()
{
   do {
   :: delay(Exp(10/60), q < C) {= q = q + 1 =}
   }
}

process Server()
{
   do {
   :: when(q > 0) invariant(q <= 0) {= q = q - 1 =};
      delay(Norm(10, 2)) tau {==} // process customer/request/...
   }
}

par {
:: Arrivals()
:: Server()
}
```

Figure 6.5: An M/G/1/C queueing system in MODEST

In this way, we can simply write e.g. `delay`$(\text{EXP}(\lambda), \mathit{true})$ instead of manually introducing a new clock, a new real variable, resetting the clock, assigning to the real variable a value sampled from $\text{EXP}(\lambda)$, and then using a combination of guards and deadlines to achieve the desired delay like in the previous example. The condition expression e_{cond} is necessary to allow postponing the execution of P when the delay has expired, but some other (untimed) condition is not yet satisfied. We need to include it in the definition of the shorthand because there is no way to achieve a disjunction in an invariant when the operands must appear in different syntactical constructs (cf. the invariant function shown in Table 5.2). It is an optional parameter of the shorthand that defaults to *true*.

Example 48. Let us now use the `delay` shorthand to build a concise model of an *M/G/1/6 queueing system* in MODEST as shown in Figure 6.5. It represents a stream of customers (or requests) that arrive at a single service desk (or a server), where they are processed one after the other. The time between customer arrivals is exponentially distributed (i.e. it is *memoryless*: *M*) with rate $\frac{1}{6}$. The time it takes to process one customer follows the normal distribution (i.e. it is *generally* distributed: *G*) with mean 10 and standard deviation 2. Since clocks cannot be negative, it is implicitly truncated to values ≥ 0 when we compare the result to a clock. If we interpret a time unit as 10 seconds, this means one

customer arrival per minute on average and a service time around 100 s that is in $[60\,\text{s}, 140\,\text{s}]$ in about 95 % of all cases. The queue has length $\texttt{C} = 5$, not counting the customer being served, and is initially empty. To make sure that customer arrivals are blocked when the queue is full, we use the `delay` shorthand's ability to include the condition $\texttt{q} < \texttt{c}$ in the `Arrivals` process.

This section marks the last point in this thesis where we extend the MODEST language. As mentioned, the complete syntax and semantics in terms of STA is given in Appendix A. An extended version with support for complex continuous dynamics, its symbolic semantics in terms of stochastic hybrid automata, and their concrete semantics as NLMP can be found in full detail in [HHHK13].

6.4 Properties

The semantics of STA is defined in terms of TPTS, just like that of PTA. In Section 5.4, we introduced probabilistic timed reachability and expected-reward properties for PTA, but their semantics was actually defined for the TPTS semantics of PTA. We can consequently use these two classes of properties and their semantics without changes for STA, too. As a reminder, we thus consider the following forms of properties:

$$\text{Probabilistic timed reachability: } \mathrm{P}_{\max}(\diamond\,\phi), \quad \mathrm{P}_{\min}(\diamond\,\phi) \quad \text{and } \mathrm{P}(\diamond\,\phi) \sim x$$
$$\text{Expected-reward properties: } \mathrm{X}_{\max}(r \mid \phi),\ \mathrm{X}_{\min}(r \mid \phi) \text{ and } \mathrm{X}(r \mid \phi) \sim x$$

where the ϕ are state formulas that may include clock constraints and r is a reward.

Example 49. We are interested in the probability of reaching a *target*-state, i.e. one whose location is l_3 or l_4, within at most t time units in the STA of Example 45. The minimum probability is $[\![\mathrm{P}_{\min}(\diamond\,\mathit{target})]\!] = 0$ because the invariant of l_0 allows us to stay there forever. If $t < 8$, we can only reach l_3 and thus compute the maximum probability $[\![\mathrm{P}_{\max}(\diamond\,\mathit{target})]\!]$ based on the exponential distribution's cdf: it is

$$p = \frac{1}{2} \cdot (1 - \mathrm{e}^{-\lambda t}).$$

If $t \geq 16$, we can also reach l_4 and the result is $p + 1/2$. For $t \in [8, 16)$, we get

$$p' = \frac{1}{2} \cdot (1 - \mathrm{e}^{-\lambda t}) + \frac{1}{4} \cdot (1 - \mathrm{e}^{-\lambda (t-8)})$$

by going back to l_0 from l_2 as soon as possible. Observe that $p = p'$ for $t = 8$, but for $t = 16$, $p' \neq 1/2 + p$: here, the nondeterministic choice available in l_2 makes an important difference.

Now, let us look at the (time-unbounded) minimum and maximum expected reward r when we reach l_3 or l_4. By definition, since there is a scheduler that reaches those locations with probability less than 1 (by staying in l_0 forever), the maximum value is $[\![X_{\min}(r \mid \mathit{target})]\!] = \infty$. If $\lambda \geq 1/16$, the minimum value that we can achieve is $1/\lambda$ by always returning to l_0 from l_2; otherwise, it is $1/2 \cdot (16 + 1/\lambda)$. If we set the invariant of l_0 to $c \leq 0$, the maximum is finite and we can make a similar calculation.

As we would expect, there is no need to extend the MODEST syntax for properties in any way in order to support STA, either.

Example 50. When studying the M/G/1/6 queueing system introduced in Example 48, we are interested in the following values:
– the probability p that the queue is full and $\leq t_p$ time units have elapsed,
– the expected time t until the queue is full for the first time, and
– the expected number c of customers served before the queue becomes full.
In MODEST, the properties that give us the minimum and maximum value for p are

```
property QuOverflowPrMin = Pmin(<> (q == C && time <= t_p));
property QuOverflowPrMax = Pmax(<> (q == C && time <= t_p));
```

where `time` is a the global clock variable/global time reward mentioned in Section 5.4. We also use the latter in order to compute t via the properties

```
property MinTimeBeforeOverflow = Xmin(time | q == C);
property MaxTimeBeforeOverflow = Xmax(time | q == C);
```

We need to add an explicit reward variable `served_count` that is increased by 1 in the previously empty second update in process `Server` of Figure 6.5 in order to be able to determine c using the properties

```
property MinCustBeforeOverfl = Xmin(served_count | q == C);
property MaxCustBeforeOverfl = Xmax(served_count | q == C);
```

As a queueing system where all delays follow continuous distributions that in particular do not assign probability mass $\neq 0$ to any single point in their support, the M/G/1/6 queuing system model is a fully deterministic STA (i.e. its TPTS semantics does not contain any nondeterministic choice between action-labelled transitions and the STA is time-deterministic in the spirit of the definition of time-determinism for PTA that we gave in Section 5.6.1). The values for the `Max`/`Min` pairs of properties given above should thus be the same in all three cases, and we will exploit this fact when we come back to the model in Section 6.5.3.

6.5 Model Checking

We have seen in the previous chapter that various exhaustive model checking techniques for PTA have been developed over the past decade. However, an exhaustive analysis method for STA, in particular one that soundly supports (most of) the various forms of nondeterminism that can be present in an STA, has been out of reach until recently, when Ernst Moritz Hahn developed exhaustive model checking techniques for stochastic hybrid automata (SHA) that allow safety verification [FHH+11] and computing bounds on reward-based properties [Hah13]. SHA generalise STA by adding continuous variables whose evolution over time can be described by differential equations and inclusions in invariants. The safety verification approach has been (partly) implemented as part of the MODEST TOOLSET in the prohver tool [HHHK13]. The implementation suffered from the usual limited scalability of hybrid systems analysis techniques and was encumbered by modelling restrictions and portability problems stemming from the use of PHAVER [Fre05] as a backend to handle the continuous behaviour.

We present a specialisation of these SHA analysis techniques here: We use a combination of abstraction and probabilistic model checking to compute bounds on reachability probabilities and expected reward values for STA. For maximum probabilities or rewards, we obtain upper bounds; for minimum probabilities or rewards, we obtain lower bounds. This works as follows: First, the continuous distributions that occur in the STA are abstracted by a combination of discrete probabilistic choices and continuous nondeterminism. The result is a PTA, which is then analysed using the digital clocks approach as described in Section 5.5. The results are upper/lower bounds on the corresponding values in the original STA. We describe the method in more detail and prove its correctness in Section 6.5.1. An implementation is available in the form of the mcsta tool within the MODEST TOOLSET. We describe implementation aspects in Section 6.5.2, and then apply mcsta to four example models to study its effectivity and efficiency in Section 6.5.3.

By specialising for STA, we gain scalability, improve usability by requiring less user input and improving automation, and are able to compute useful lower bounds on minimum probabilities. Figure 6.6 contrasts the SHA and STA approaches and their implementations in prohver and mcsta graphically.

Related work Kwiatkowska et al. [KNSS00] have pioneered the foundational basis of STA model checking with their work on timed automata with generally distributed clocks, verified against properties in probabilistic timed

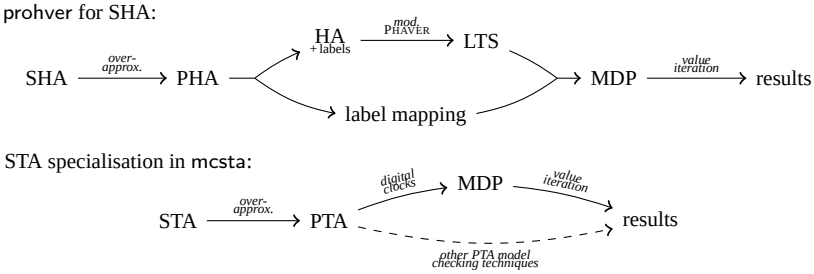


Figure 6.6: The verification approaches for SHA and STA compared

CTL. They use a semantics based on the region graph where regions are further partitioned to cater for the stochastic behaviour. The main differences to what we present in this paper are that our approach can handle distributions with unbounded support (e.g. the exponential and normal distributions), supports expected rewards, and that we avoid the region construction. We also show a working implementation, which instead currently uses a digital clocks semantics, but this can be interchanged with other approaches, such as the one based on stochastic games.

Other related approaches that we find are based on statistical model checking [DLL+11b], numerical discretisation [LHK01], plain discrete event simulation [HS00], or state classes [BBH+13] (on a different model also called STA). However, all of these either implicitly or explicitly exclude the presence of nondeterminism, and thus work in the realm of generalised semi-Markov processes (GSMP, [HS87]) instead. As an example, consider the "STA" model of [BBJM12] (which is closely related to the one of [BBH+13]): There, a single distribution is sampled on every edge, the result being the exact sojourn time in the following location. In comparison, our model of STA also supports continuous and discrete nondeterminism as well as multiple samplings per edge and multiple sampled variables that can memorise their values over several edges/locations.

In particular, the method we present here is geared towards correctly handling the general combination of stochastics and nondeterminism. Dedicated approaches for deterministic models consequently provide better precision or performance for that special case. We come back to this tradeoff in our evaluation in Section 6.5.3, where we look at two deterministic models for comparison, and two nondeterministic case studies that can only be handled correctly with our new approach.

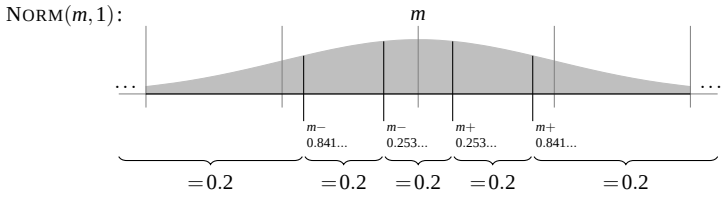


Figure 6.7: Overapproximation of a continuous distribution [HHHK13]

6.5.1 Bounds for Reachability and Rewards

We now present details about the abstraction mechanism used to replace continuous probability distributions by the combination of discrete ones and continuous nondeterminism. We show that this transformation preserves upper bounds for minimum and maximum reachability and expected rewards. After that, we discuss the error introduced by the abstraction as well as the use of the digital clocks technique to analyse the resulting PTA, and how this error can sometimes be reduced at the cost of larger state spaces.

Abstracting Continuous Distributions

In the first step of our analysis method (cf. Figure 6.6, bottom), discrete probability distributions are used to overapproximate continuous ones by introducing additional nondeterminism: The distribution's support is divided into a number of intervals and the probability of each interval is computed. The continuous sampling is then replaced by a probabilistic choice over the intervals with the computed probabilities, followed by a nondeterministic choice of which concrete value to pick from the chosen interval.

Example 51. For the sampling expression NORM$(m,1)$, Figure 6.7 shows a decomposition into five intervals with probability mass 0.2 each. In MODEST, we would replace the action with the corresponding sampling assignment,

```
a {= x = Norm(m, 1) =}; P,
```

by a `palt` with nondeterministic assignments for the intervals:

```
a palt {
:0.2:{= x=any(y |                          y <= m-0.841...) =}; P
:0.2:{= x=any(y | m-0.841... <= y && y <= m-0.253...) =}; P
:0.2:{= x=any(y | m-0.253... <= y && y <= m+0.253...) =}; P
:0.2:{= x=any(y | m+0.253... <= y && y <= m+0.841...) =}; P
:0.2:{= x=any(y | m+0.841... <= y                   ) =}; P
}
```

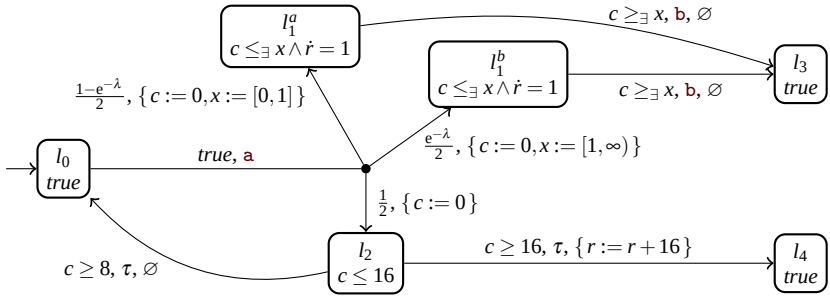


Figure 6.8: A PTA abstraction of the example STA

Observe that the singleton overlaps between the intervals do not matter as they all have a probability mass of zero.

When using prohver, the probabilities for the intervals created in the abstraction step had to be concrete real values so they could be split off to obtain a nonprobabilistic hybrid automaton for use by PHAVER. In our new approach, we can map to PTA with probabilities that depend on state variables (but not on clocks or variables that were previously sampled). This makes it possible to analyse, for example, population models as they typically arise in systems biology applications where the rates of exponential distributions depend on the current number of individuals of the involved species.

Furthermore, since PTA allow only integer clock constraints, the choice of intervals is limited to those with integer bounds. We use this as an opportunity to improve usability by removing from the user the burden of specifying the set of intervals to use. Instead, we always overapproximate continuous distributions with intervals of unit width 1 aligned on integer bounds. For distributions with unbounded support, such as the exponential or normal distribution, we generate as many unit width intervals as needed to cover a probability mass of $1-\rho$ and then add half-open intervals for the residual of the support. Instead of a set of intervals as with prohver, the only parameter of our approach is this *residual probability* ρ. We use a default of $\rho = 0.05$ unless stated otherwise.

Example 52. For the STA introduced in Example 45, we show the PTA overapproximation for the case that a single unit width interval is sufficient to cover $1-\rho$ probability in Figure 6.8. With $\rho = 0.05$, this is ensured provided $\lambda \geq 3$. We use $\geq_\exists$ and $\leq_\exists$ to denote interval comparisons. They are satisfied whenever there exists some value in the interval such that the concrete comparison is satisfied. This amounts to a comparison with the upper bound for $\leq_\exists$ and with the lower bound for $\geq_\exists$ when the interval operand is on the right-hand side.

Correctness

We now show that, in the PTA that is constructed as described above, the maximum/minimum reachability probabilities and expected reward values are indeed upper/lower bounds for the corresponding values in the original STA. We first have to define the effect of abstraction more formally:

Definition 85. Consider an STA $M = \langle Loc, Var, A, E, l_{init}, Inv, AP, L\rangle$ and a (potentially infinite) family of sets $\mathscr{A} = \langle B_i\rangle_{i\in I}$ such that $\bigcup_i B_i = Loc \times Val$. Each abstract state $B_i \subseteq Loc \times Val$ subsumes certain concrete states of $[\![M]\!]$, and all states are covered. We require that an abstract state only subsume concrete states of the same location. Assume $\langle l_{init}, \mathbf{0}_{Var}\rangle \in B_{\text{init}}$, and B_i, B_j are disjoint for $i \neq j$. The *abstraction TPTS* is defined as

$$\text{abs}(M,\mathscr{A}) \stackrel{\text{def}}{=} \langle \mathscr{A}, \Sigma_{\mathscr{A}}, \mathbb{R}_0^+ \uplus A, T_M^{\text{abs}}, B_{\text{init}}, AP, L_{\mathscr{A}}\rangle$$

where T_M^{abs} is defined similar to Definition 82 with the *jump* rule being

$$\frac{l \xrightarrow{g,a}_E m \quad \langle l,v\rangle \in B_i \quad Jmp(v,l,m,\mu) \quad [\![g]\!](v)}{B_i \xrightarrow{a}_{T_M^{\text{abs}}} [\forall j \in I\colon B_j \mapsto \mu(B_j)]}$$

where by $[\forall i\colon x_i \mapsto p_i]$ (or $[x_1 \mapsto p_1, \ldots, x_n \mapsto p_n]$) we denote the distribution $\sum_i p_i \mathscr{D}(x_i)$. We require $\mathscr{A}$ to be defined such that all induced $[\forall j\colon A_j \mapsto \mu(A_j)]$ have finite support. Timed transitions are defined accordingly. The labelling function $L_{\mathscr{A}}$ is $\{B_i \mapsto \bigcup_{s\in B_i} L_M(s) \mid i \in I\}$ with L_M as in Definition 82. We assign rewards to abstract states according to the rate assigned to its location and the rewards of the edges originating from there.

In the context of this paper, $\mathscr{A}$ is obtained by splitting the possible values sampling variables can take into unit width or half-open intervals. This construction ensures the finite-support requirement. For instance, for a single sampling variable x, all concrete states where x is sampled to take values in the range between 1 and 2 are subsumed by a single abstract state. For multiple sampling variables, abstract states are built from the cross product of intervals.

Lemma 5. For an STA M with abstraction sets $\mathscr{A}$ as above and some set of states B, the maximal (minimal) probability or reward to reach B in $\text{abs}(M,\mathscr{A})$ is not lower (not higher) than the maximal (minimal) probability/reward value in $[\![M]\!]$.

Proof ([HHH14]). We only consider disjoint abstract states. Non-disjoint ones (from overlapping intervals) would not affect correctness, but induce imprecision due to additional transitions in the abstraction. Let

$$M = \langle Loc, Var, A, E, l_{init}, Inv, AP, L\rangle$$

and $\mathscr{A} = \langle B_i \rangle_{i \in I}$. We define the *intermediate abstraction* TPTS

$$M' \stackrel{\text{def}}{=} \langle Loc \times Val, \mathscr{P}(Loc) \otimes \Sigma_{Var}, \mathbb{R}_0^+ \uplus A, T_{M'}, \langle l_{init}, \mathbf{0}_{Var} \rangle, AP, L_{M'} \rangle$$

by replacing *jump* of Definition 82 by

$$\frac{l \xrightarrow{g,a}_E m \quad Jmp(v,l,m,\mu) \quad [\![g]\!](v) \quad \langle s'_j \rangle_{j \in I} \text{ s.t. } \forall j \in I \colon s'_j \in B_j}{\langle l, v \rangle \xrightarrow{a}_{T_{M'}} [\forall j \in I \colon s'_j \mapsto \mu(B_j)]}$$

Let f map paths from the intermediate abstraction to the semantics $[\![M]\!]$, so for $\beta = s_0 a_0 [\forall j \colon s'_j \mapsto \mu_0(B_j)] s_1 a_1 \ldots$ we have $f(\beta) \stackrel{\text{def}}{=} s_0 a_0 \mu_0 s_1 a_1 \ldots$.

For $\sigma \in \mathfrak{S}_{[\![M]\!]}$ we construct $\sigma' \in \mathfrak{S}_{M'}$. Consider some path β for which $\text{last}(\beta) = \langle l, v \rangle$. W.l.o.g. consider a subset $A = \{a\} \times A_{\text{dist}} \subseteq A \times \text{Prob}(S)$ of the possible successors when choosing edge $e = l \xrightarrow{g,a} m \in E$ with $\langle l, v \rangle \xrightarrow{a}_{T_{M'}} \mu$. Let $\langle S_i \rangle_{i \in I} \subseteq \mathscr{A}_i$ be the finite set of abstract states for which $\mu(S_i) > 0$. We define $\mu_i \in \text{Prob}(S_i)$ as

$$\mu_i(A_i) \stackrel{\text{def}}{=} \frac{\mu(A_i)}{\mu(S_i)}$$

for measurable $A_i \subseteq S_i$ and write $\mu_{\text{prod}} \in \text{Prob}(\bigtimes_i S_i)$ for their product measure. Define

$$U \stackrel{\text{def}}{=} \{[\forall i \colon s'_i \mapsto \mu(S_i)] \mid \forall i \colon s'_i \in S_i\},$$

let the function g be given by

$$g([s'_1 \mapsto p_1, \ldots, s'_n \mapsto p_n]) \stackrel{\text{def}}{=} \langle s'_1, \ldots, s'_n \rangle,$$

and let $\mu(B) \stackrel{\text{def}}{=} \mu_{\text{prod}}(g(B))$. Then we set

$$\sigma'(\beta)(A) \stackrel{\text{def}}{=} \mu(A_{\text{dist}} \cap U) \cdot \sigma(f(\beta))(\{\text{edge } e \text{ chosen}\}).$$

In this way, σ' for M' simulates the continuous probability distributions in $[\![M]\!]$ s.t. measures on paths when using σ and σ' agree [Hah13, Theorem 4.22]. This implies that reachability probabilities agree, as do reward values when using equivalent reward structures.

Because distributions in M' and $\text{abs}(M, \mathscr{A})$ have finite support, one can define a finite automata simulation relation [SL95] such that $\langle l, v \rangle \preceq B_i$ if $\langle l, v \rangle \in B_i$, from which one concludes that $\text{abs}(M, \mathscr{A})$ also bounds reachability probabilities of M'. Using extensions of simulation relations similar to e.g. [Hah13, Definition 7.26] one can also bound reward values in this way. □

Digital Clocks and Scaling Time

We model-check the PTA resulting from the abstraction of the continuous distributions using the digital clocks approach (see Section 5.5). This also motivates our choice of unit interval width: In the digital clocks MDP resulting from

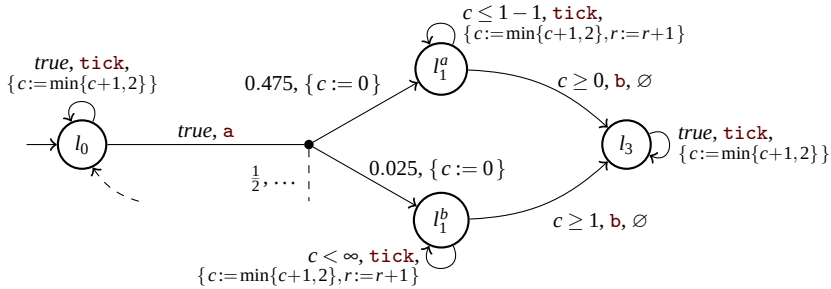


Figure 6.9: Digital clocks MDP of the PTA abstraction (explicit intervals)

the replacement of clocks by integer variables, all delay transitions are of unit duration, and thus all integer time points are anyway enumerated in its state space. If instead one resorts to an approach which can symbolically represent longer delays, e.g. using zones, then the use of larger or more flexible intervals seems worthwhile to investigate.

Example 53. The digital clocks MDP for the PTA from the previous example is shown in Figure 6.9. The delay transitions are labelled `tick`. We have excluded the non-stochastic part (locations l_2 and l_4) and merged the interval-valued variable x into the locations to show the concrete comparisons on the edges of l_1^a and l_1^b. We have also included the concrete probabilities for $\lambda \approx 3$. The maximum probability of reaching location l_3 or l_4 in this MDP in at most $t \in \mathbb{N}$ time units is 0.475 for $t = 0$ and 0.5 for $1 \leq t \leq 7$. We know from Example 49 that the actual probability in the STA is $\frac{1}{2} \cdot (1 - \mathrm{e}^{-\lambda t}) < 0.5$. In our case of $\lambda \approx 3$, this is 0 for $t = 0$, approx. 0.475 for $t = 1$ and very close to 0.5 for $t = 7$. The error is thus between 0.475 and almost 0 depending on t.

For reward r, the maximum value is ∞ even if we remove the `tick`-edge from l_0: We can stay in l_1^b forever due to the right-open interval created for the unbounded exponential distribution. The minimum value computed in this MDP is $0.475 \cdot 0 + 0.025 \cdot 1 = 0.025$, whereas the actual value for $\lambda \approx 3$ is $\approx \frac{1}{3}$.

The example shows that the error introduced by the abstraction of the continuous distributions is highly dependent on the variance of the distributions in relation to the minimum interval width of 1 required to use PTA. In models where the dependence between time and property values is similarly direct as in this example, we can get more accurate results at the cost of larger MDP state spaces by *scaling time*: Both the results of the sampling and the non-interval values that clocks are compared to (including those in the property to verify)

are multiplied by some factor $d \in \mathbb{N}^+$. (For the exponential distribution, for example, the former can be achieved by dividing the rate by d.)

Example 54. By scaling time by a factor of $d = 2$ in the STA that the previous examples were based on, two unit width intervals are used for $r = 0.05$ and $\lambda \approx 3$, with probabilities 0.388 and 0.087. The upper bound for the probability drops to 0.388 for $t = 0$ and 0.475 for $t = 1$; the lower bound for the min. expected reward rises to 0.137.

Although scaling time *can* lead to tighter bounds, there is another, independent cause of overapproximation error, which is due to the digital clocks requirement of closed clock constraints: All adjacent intervals have a singleton overlap, and we can only refer to exactly these overlapping values in clock constraints and properties. They have probability 0 in the STA, but not in the PTA, which leads to e.g. the upper bounds for time-bounded reachability probabilities being "one step ahead": In Example 54, the upper bound computed for $t = 0$ is the actual probability for $t = 1$, the bound for $t = 1$ is the probability for $t = 2$, and so on. The use of a PTA model checking technique that allows non-closed clock constraints clearly needs to be investigated as future work.

6.5.2 Implementation

We have implemented our STA model checking approach in the new mcsta tool within the MODEST TOOLSET. At the current stage, the tool supports the continuous uniform, exponential and normal distributions. To make the transformation from continuous to discrete distributions possible in a setting where sampling expressions can in principle refer to arbitrary state variables and combine sampling in arbitrarily complex and nested expressions, we impose the following restrictions: When an assignment contains a continuous probability distribution, it must be of one of the following forms, where x is a variable of type `real`:

– $x := \text{UNI}(lower, upper)$ for the uniform distribution, where *lower* resp. *upper* are expressions in *Axp* of real type for which a concrete lower bound $lb \in \mathbb{R}$ resp. a concrete upper bound $ub \in \mathbb{R}$, i.e. values such that

$$\forall v\colon lb \leq [\![lower]\!](v) \wedge [\![upper]\!](v) \leq ub,$$

can be determined with $lb \leq ub$. The abstraction intervals are then

$$[\lfloor lb \rfloor, \lfloor lb \rfloor + 1], \ldots, [\lceil ub \rceil - 1, \lceil ub \rceil]$$

and the expressions for the intervals' probabilities are built according to the uniform distribution's cdf, that is $cdf_{\text{UNI}}(x) = (x - lower)/(upper - lower)$.

– $x := \mathit{offset} + \text{EXP}(\mathit{rate})$ for the exponential distribution, where *offset* is an expression in *Axp* of integer type and *rate* is an expression in *Axp* of real type. Here, we only require that a concrete lower bound $\lambda \in \mathbb{R}^+$ can be determined for the expression *rate*. The abstraction intervals are then

$$[\mathit{offset}, \mathit{offset}+1], \ldots, [\mathit{offset}+n-1, \mathit{offset}+n] \text{ and } [\mathit{offset}+n, \infty)$$

where $n \in \mathbb{R}$ is calculated as $n = \lceil \frac{-\ln \rho}{\lambda} \rceil$ (using the quantile function of the exponential distribution). The expressions for the intervals' probabilities are built according to the exponential distribution's cdf: $\mathit{cdf}_{\text{EXP}}(x) = 1 - \mathrm{e}^{-\mathit{rate} \cdot x}$.

– $x := \text{NORM}(m, \sigma)$ for the normal distribution, where the mean m is an expression in *Axp* of integer type and the standard deviation σ is a concrete value in $\mathbb{R}^+$. The abstraction intervals are

$$(-\infty, m-n], \ldots, [m-1, m], [m, m+1], \ldots, [m+n, \infty).$$

Since neither the quantile function nor the cdf of the normal distribution have a closed-form solution, we require σ to be a concrete value to precompute n and the actual interval probabilities based on σ and ρ close to `double` precision.

In this way, we can construct a set of intervals with integer bounds for all three distributions. These three examples also show a general recipe to support other continuous distributions using their quantile and cumulative distribution functions; if a distribution has unbounded support, we need to be able to evaluate its quantile function during transformation to get the necessary number of intervals based on r. In any case, we need to be able to express its cumulative distribution function as an expression allowed by the MODEST TOOLSET to support parameterisation by state variables, or merely compute it otherwise. In case a distribution is parameterised by an expression that contains state variables, we may generate more intervals than necessary for some valuations, which then have zero probability. For example, we generate two intervals for $x := \text{UNI}(0, 2i)$ when i has domain $\{0, 1\}$ since the upper bound of expression $2i$ is 2. However, since the probabilities are preserved as expressions, the probability of $[1, 2]$ will evaluate to 0 for all states where $i \neq 2$.

6.5.3 Evaluation

To find out whether the exhaustive STA model checking technique is effective and efficient in practice, we have applied it to four different examples. We are interested in how close the computed bounds are to the actual values (effectiveness), and how large the state spaces of the underlying MDP become[1]

[1] Memory was the limiting factor in all examples; runtime was always below 3 minutes.

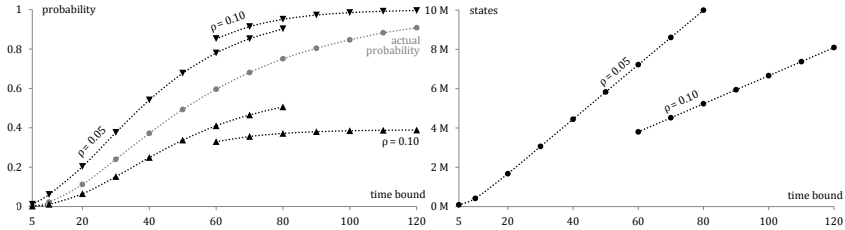


Figure 6.10: Reachability results and state space sizes for the M/G/1 example

(efficiency). All measurements presented in the remainder of this section were performed on the same 1.7 GHz Intel Core i5-3317U system with 4 GB of RAM running 64-bit Windows 8.1.

The first two models we present to evaluate effectiveness are deterministic. As mentioned, our method is not targeted for this special case, so we expect correct and useful, but not very tight, computed bounds. Specialised methods will perform better or be more precise in these cases. The last two models, however, contain continuous and discrete nondeterminism, so our technique is currently the only one available for verification.

M/G/1 Queueing System with Normal Distribution

Our first example takes up the *M/G/1/6 queueing system* introduced in Example 48 (with small modifications that do not affect the intended semantics but make for smaller state spaces). We check properties that query for the values p, t and c described in Example 50. Since nondeterminism is absent in this model by construction, we can perform statistical model checking with modes to obtain good approximations of t and c as well as p when it is not too small (i.e. not a *rare event*).

The results of computing upper and lower bounds on p using our implementation are shown in Figure 6.10. On the left, we show the computed bounds for different values of t_p as black triangles. We see that there is a noticeable approximation error, but the general evolution of the probability over time is preserved. After $t_p \approx 80$, the lower bound shows no significant improvements. For $t_p \geq 90$, we ran out of our 4 GB of RAM, so we increased the residual probability parameter ρ to 0.1. The number of concrete states in the MDP of the digital clocks semantics is shown on the right of Figure 6.10. We see that it increases linearly with t_p and can be reduced significantly by increasing ρ, i.e. by lowering the number of abstraction intervals used for the exponential and normal distributions.

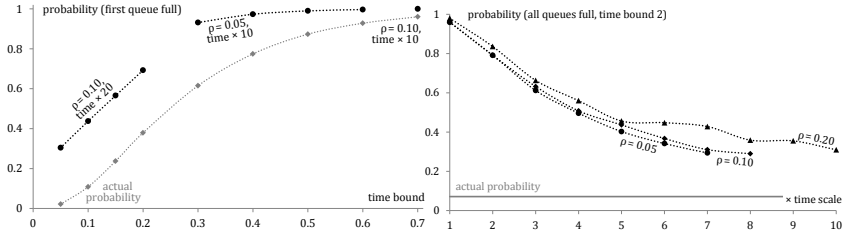


Figure 6.11: Reachability results and time scale effects for the tandem queues

Asking for *minimum* expected rewards, we compute the bounds $t \geq 43.4$ and $c \geq 3.52$ for the other two values. As we do not need a global clock to check a time bound like t_p here, the underlying MDP has just 136 767 states. State space exploration and computation of both bounds takes only 2.3 s in total. If we ask for *maximum* expected rewards, we get bounds of ∞ due to the right-open intervals created by the abstraction of the unbounded distributions (cf. Example 53). Simulation tells us that $t \approx 61$ and $c \approx 6.2$ for this deterministic model.

Tandem Queueing Network

We next look at a model from the PRISM benchmark suite [KNP12]: the *tandem queuing network* of an M/Cox$_2$/1/4 queueing system followed by an M/M/1/4 queueing system [HMKS99]. It is a CTMC and we can thus model it as an STA without nondeterminism. The MODEST code for this example is part of the MODEST TOOLSET download.

We experiment with scaling time as described in Section 6.5.1. We compute the maximum probability p_{ff} of the first queue being full in time t, trying to use as low a value of $\rho \geq 0.05$ and as high a time scaling factor without running out of memory. The result is shown on the left of Figure 6.11. The computed upper bounds are further from the actual probability than in the previous example. This is likely due to the effect of errors adding up with four subsequent stochastic delays that are individually abstracted. While we get correct results, using specialised CTMC model checking techniques would obviously be a better choice here. (Indeed, e.g. PRISM can perform a precise analysis of large instances of this model with moderate runtime and memory usage.)

The second property we look at is the maximum probability p_{af} of both queues becoming full within time t. This happens at a vastly different time scale: While the first queue is full with probability close to zero for $t = 1$, p_{af} only starts to approach 0.5 when t is on the order of 50. We thus focus on the effect of scaling time on the approximation error for fixed time bound $t = 2$.

model	type	Pr_{max}	$[E^{\wedge}_{min}, E^{\wedge}_{max}]$ µs	$[E^{\vee}_{min}, E^{\vee}_{max}]$ µs	$[E^{1}_{min}, E^{1}_{max}]$ µs	states	time
wlan	PTA	0.1836	$[1325, 6280]$	$[450, 4206]$	$[450, 5586]$	104 804	8 s
$wlan^{uni}$	STA	0.1366	$[2325, 4607]$	$[950, 3018]$	$[950, 3880]$	264 240	15 s

Table 6.1: Results and comparison for the WLAN example

The results are shown on the right of Figure 6.11. In this case, the impact of increasing ρ is not as significant, but we see that the error can be significantly reduced by scaling up time.

Finally, we compute bounds on the expected times t_{ff} until the first queue becomes full and t_{af} until both are full. As we increase the time scaling, we go from lower bounds $t_{ff} \geq 0.000012$ and $t_{af} \geq 0.56$ for time scale $d = 1$ with 9 557 MDP states, computed in 0.1 s, to $t_{ff} \geq 0.108710$ and $t_{af} \geq 5.87$ for $d = 10$ with 3 662 958 states, computed in 108 s. Again, upper bounds (i.e. maximum expected rewards) are all ∞. From simulation, we get $t_{ff} \approx 0.29$ and $t_{af} \approx 17.9$.

Wireless LAN with Uniform Transmission Time

Departing from queueing systems, we now look at the model of a communication protocol: the carrier-sense multiple-access with collision avoidance (CSMA/CA) part of IEEE 802.11 *WLAN*. We take the MODEST PTA model of [HH09] that was originally described as a PRISM case study[2] and replace the nondeterministic choice of transmission delay out of $[200, 1250]$ µs (with a unit of time representing 50 µs) by a uniformly distributed choice over the same interval. This is the same kind of transformation as we performed on the lossy channel of our simple BRP model in Example 47. The result here is also an STA that is no longer a PTA, but that still contains (other) nondeterministic choices. Again, both MODEST models are part of the MODEST TOOLSET download.

Model checking results for the original PTA model (*wlan*) and the new STA model ($wlan^{uni}$) are shown in Table 6.1. We see that the state space of the underlying MDP is larger when the uniform distribution is used. This is because the states not only contain explicit values for all clocks as in the original PTA, but additionally 21 different concrete intervals that overapproximate the result of sampling from $\text{UNI}(4, 25)$. The blowup thus stays far below the worst-case factor of 21 here.

We analyse six time-unbounded properties. The first is Pr_{max}, the maximum probability that either of the two modelled senders' backoff counters

[2] http://www.prismmodelchecker.org/casestudies/wlan.php

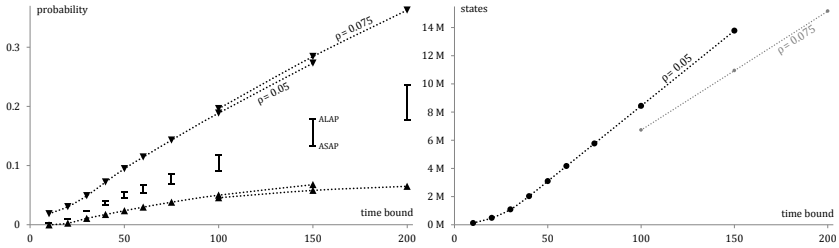


Figure 6.12: Model checking results and state space sizes for the file server example

reaches the upper bound of 2. The other five are $E^{\wedge}_{min}/E^{\wedge}_{max}$, $E^{\vee}_{min}/E^{\vee}_{max}$ and E^{1}_{min}/E^{1}_{max}, the minimum/maximum expected times until both senders, either of them, or the one with id 1 correctly deliver their packets. Because of the model being nondeterministic, we cannot use simulation or any other technology to compute the actual values. However, the computed bounds are plausible if we assume that in the PTA, the longest/shortest possible transmission delay maximises/minimises the values. The STA is thus indeed expected to show less extremal behaviour.

File Server

As a final example, we analyse another model that combines all essential features of STA and cannot be model checked with any other approach we know of (except prohver). It represents a single-threaded *file server* with slow archival storage:

– Requests arrive to a single queue of length $C = 5$. Their interarrival time is exponentially distributed with rate $\frac{1}{8}$.
– File sizes are uniformly distributed over some range such that sending the file back to a client takes time uniformly distributed over $[1,3]$.
– 2 % of all files have been moved to a slow archival storage. Retrieving a file from normal storage is instantaneous, but retrieving it from archival storage may take between 30 and 40 time units. We do not know anything else about the archive (e.g. whether it is tapes, a slow disk, goes to standby quickly etc.) so we model this delay as nondeterministic.

We thus have continuous stochastic delays, a probabilistic choice and nondeterministic delays. Additionally, we model the initial queue length as uniformly distributed in $\{0,\ldots,\lfloor\frac{C}{2}\rfloor\}$. The MODEST code for this model (with small changes for clarity of presentation) is shown in Figure 6.13.

```
const int C = 5;              // queue size
const real T_BOUND;           // time bound for reachability
const real LAMBDA = 1 / 8;    // request arrival rate
const real ARCH_PR = 1 / 50; // fraction of files in archive
const real FILE_MIN = 1;      // min. time to send regular file
const real FILE_MAX = 3;      // max. time to send regular file
const real ARCH_MIN = 30;     // min. time to send archived file
const real ARCH_MAX = 40;     // max. time to send archived file

int(0..C) q;

property QuOverflowProbMax = Pmax(<> (q == C && time <= T_BOUND));
property QuOverflowProbMin = Pmin(<> (q == C && time <= T_BOUND));
property QuOverflowTimeMin = Xmin(time | q == C);
property QuOverflowTimeMax = Xmax(time | q == C);

process Arrivals()
{
   clock c; real x;

   do {
   :: when(c >= x && q < C) invariant(c <= x || q == C)
      tau {= q = q + 1, c = 0, x = Exp(LAMBDA) =}
   }
}

process Server()
{
   clock c; real x;

   delay(0, q > 0) palt {
   :  ARCH_PR: {= q = q - 1, c = 0 =};
      when(c >= ARCH_MIN) invariant(c <= ARCH_MAX)
      tau {= c = 0, x = Uni(FILE_MIN, FILE_MAX) =}
   :1-ARCH_PR: {= q = q - 1, c = 0, x = Uni(FILE_MIN, FILE_MAX) =}
   };
   when(c >= x) invariant(c <= x) tau {= c = 0, x = 0 =};
   Server()
}

par {
:: Arrivals()
:: Server()
:: delay(0) {= q = DiscreteUniform(0, floor(C/2)) =}
}
```

Figure 6.13: MODEST model of a file server with slow archival storage

We are interested in the probability p that the request queue becomes full within time t_p, and the minimum (i.e. worst-case) expected time t until this happens. For t, we obtain a lower bound of 462 time units from an MDP with 107 742 states in 6 s. For p, the results are shown in Figure 6.12. On the right, we see that the number of MDP states again grows linearly with the time bound. On the left, we have plotted the computed upper/lower bounds using small triangles.

Due to the nondeterministic delay, we cannot use simulation. However, we can instruct modes to resolve that delay by scheduling events either as soon or as late as possible (ASAP/ALAP). Simulating these deterministic variants of the model gives us $t \approx 1012$ for ASAP and $t \approx 721$ for ALAP. For p, the simulation results are included on the right of Figure 6.12. Observe that the highest probability we see is around 0.35. The results that we get via our new approach are clearly useful for verification: They are safe bounds whereas we do not know anything about the relationship between simulation results and the actual values.

6.6 Statistical Model Checking

The main problem in performing statistical model checking of STA models is the same as for PTA: continuous nondeterministic choices. The other addition, sampling from continuous probability distributions, does not complicate simulation because this sampling can just be done while exploring a path, in the same way that probabilistic choices were simulated. However, it may complicate correctness arguments for advanced SMC techniques, like possible extensions of the POR and confluence-based approaches. Since the fundamental issues are so similar to PTA, in this section we just briefly review the notion of time-deterministic STA, update the path generation procedure of Algorithm 23 to handle sampling and the `any` construct, and give a brief overview of how SMC for general STA is more challenging than for PTA.

Time-deterministic STA

We have seen that the subclass of time-deterministic PTA was easy to simulate. Let us now define a similar subclass of STA. In addition to the requirements that were already in Definition 79, we also have to ensure that there is no continuous nondeterminism due to the `any` construct: This could be used to make a subsequent delay nondeterministic (in the same way that sampling can be used to make a subsequent delay stochastic), but even when the result is not used for delays, it cannot be simulated. We therefore define:

Definition 86 (TDSTA)**.** An STA M with TPTS semantics

$$[\![M]\!] = \langle Loc \times Val, \Sigma_{Loc \times Val}, \mathbb{R}^+ \uplus A, T_M, \langle l_{init}, \mathbf{0} \rangle, AP, L_M \rangle$$

is *time-deterministic*, or equivalently a *time-deterministic stochastic timed automaton* (TDSTA), if AP is countable and for all reachable states $\langle l, v \rangle$ of $[\![M]\!]$, we have that if there is an action-labelled transition $\langle a, \mu \rangle \in T_M(\langle l, v \rangle)$, then there is no delay-labelled transition in $T_M(\langle l, v \rangle)$, and furthermore $T_M(\langle l, v \rangle)$ is finite.

We use the restriction to finite sets of action-labelled transitions in order to disallow the use of `any` to create continuous nondeterministic assignments. Since we assume STA to be finitely branching, such (countably or uncountably) infinite sets can only be created by the *jump* inference rule through use of the `any` construct.

The semantics of a TDSTA, however, is not an MDP because sampling from continuous probability distributions is still allowed. In case the semantics is deterministic (i.e. there is no discrete nondeterminism in the STA either), however, a TDSTA is a GSMP, and as such can be simulated easily. In particular, the choice of time scheduler in the `simulate` algorithm we describe below does not affect the SMC results if the model at hand is a TDSTA. (The choice of resolver (and any-resolver), however, remains important if discrete nondeterminism is present.)

Path Generation for STA

In order to generate paths through STA with user-specified resolutions of all nondeterminism, we need an *any-resolver* $\mathfrak{N}$ to handle nondeterminism in updates due to the `any` construct in addition to the resolver $\mathfrak{R}$ for discrete nondeterminism and the time resolver $\mathfrak{S}$ for nondeterministic delays. It then takes only small modifications to make Algorithm 23 applicable to STA. The result is shown as Algorithm 24; the only relevant changes are in lines 24 and 30 where we first let $\mathfrak{N}$ select one of the possible measures over valuations allowed by the chosen updates, and then perform the actual sampling to get the target valuation v.

SMC for General STA

As for PTA before, when we have to analyse a general STA that is not necessarily time-deterministic using SMC, we should actually not use implicit schedulers and resolvers if we want to obtain sound results. However, we saw that there is currently no technique to do so for PTA, and consequently no technique to solve the same problem for the more general model of STA, either. Similar to the ideas of Section 5.6.4, we could try to use the concepts of our

function `simulate`($M = \langle Loc, \mathscr{C}, A, E, l_{init}, Inv, AP, L\rangle$, $\mathfrak{R}$, $\mathfrak{S}$, $\mathfrak{N}$, ϕ, d)
  $\langle l, v\rangle := \langle l_{init}, \mathbf{0}\rangle$, $seen := \varnothing$, $i := 1$
  while $i \leq d$ **do**
    if $\phi(L_M(\langle l, v\rangle))$ **then return** *true*
    else if $\langle l, v\rangle \in seen$ **then return** *false*
    $I_i' := \{\, t \in \mathbb{R}_0^+ \mid [\![Inv(l)]\!](v+t) \wedge \nexists t' \in \mathbb{R}_0^+ .\ t' < t \wedge \neg [\![Inv(l)]\!](v+t') \,\}$
    $I_g := \{\, t \in \mathbb{R}_0^+ \mid \exists l \xrightarrow{g,a} \mu .\ [\![g]\!](v+t) \,\}$
    if $I_i' \cup \{0\} = \{0\} \wedge I_g \cap \{0\} = \varnothing$ **then return** *false* // *timelock*
    $\mu_{\mathfrak{s}} := \mathfrak{S}(\langle\langle l, v\rangle, I_i' \cup \{0\}, I_g\rangle)$ // $\cup\{0\}$ *for weak invariants*
    $t_{\mathfrak{s}} :=$ choose a delay $t_{\mathfrak{s}}$ randomly according to $\mu_{\mathfrak{s}}$
    repeat
      $t := \min(\{t_{\mathfrak{s}}\} \cup \{\, t' \in \mathbb{R}^+ \mid L_M(\langle l, v\rangle) \neq L_M(\langle l, v+t'\rangle) \,\})$
      if $t = \infty$ **then return** *false*
      $v := v + t$, $t_{\mathfrak{s}} := t_{\mathfrak{s}} - t$, $i := i + 1$
      if $\mu_{\mathfrak{s}}$ is Dirac **then** $seen := seen \cup \{\, \langle l, v\rangle \,\}$ **else** $seen := \varnothing$
      if $\phi(L_M(\langle l, v\rangle))$ **then return** *true*
      else if $\langle l, v\rangle \in seen$ **then return** *false*
      else if $i > d$ **then return** *unknown*
    until $t = 0$
    if $\exists l \xrightarrow{g,a} \mu .\ [\![g]\!](v)$ **then**
      $\mu_{\mathfrak{R}} := \mathfrak{R}(\langle l, v\rangle)$
      $v :=$ choose an edge $\langle g, a, v\rangle$ randomly according to $\mu_{\mathfrak{R}}$
      $\langle U, l'\rangle :=$ choose update, location randomly according to v
      $\nu' := \mathfrak{N}([\![U]\!](v))$ // *let* $\mathfrak{N}$ *select a measure over valuations*
      if $\mu_{\mathfrak{R}}$, ν and ν' are Dirac **then**
        $seen := seen \cup \{\, \langle l, v\rangle \,\}$
      else
        $seen := \varnothing$
      end
      $v :=$ choose target valuation randomly according to ν'
      $l := l'$, $i := i + 1$
    end
  end
  return *unknown*

Algorithm 24: Path generation for an STA, with any-resolver $\mathfrak{N}$

exhaustive model checking approach for STA in order to turn STA into PTA on-the-fly and then use sound SMC techniques for PTA, if they existed. However, due to the problems that we already face with PTA, no such approach has yet been investigated in detail or even implemented.

6.7 Summary and Discussion

By adding the possibility to sample from continuous probability distributions and to use continuously nondeterministic assignments, we extended the model of PTA to obtain stochastic timed automata (STA). Our notion of STA is a more general one, in particular in the nondeterministic choices it allows, than some similarly-named formalisms [BBH+13, BBJM12]. After the definition of the model and its semantics, which included the move from a deterministic to a more involved stochastic-nondeterministic semantics for expressions and assignments, we showed how to write STA models in MODEST and that the property classes we defined for PTA can be applied to STA just as well. We then presented as a core contribution of this thesis the first fully-automated exhaustive model checking approach for general STA. Finally, we briefly reviewed possibilities to make SMC work for STA.

STA are a very expressive model, and in particular do not only cover originally automata-based models such as PTA and MDP as special cases, but also the growing family of models based on continuous-time Markov chains. Investigating these submodels from the STA perspective raises a number of interesting questions related to the effects of memoryless sampling in compositional models as well as to the consequences and to the desirability of including more information (in this case, the knowledge of the sampling results) in the state spaces of the underlying semantics, which makes this information available to the schedulers we use to reason about nondeterminism. Although first steps to answer these questions have been taken [HKK14], a lot of additional research is possible and necessary in this area.

The exhaustive model checking approach that we presented works for STA with general, unbounded distributions as well as nondeterministic choices. It provides upper bounds for maximum and lower bounds for minimum reachability probabilities and expected rewards. It is similar to the idea of Kwiatkowska et al. [KNSS00], but allows unbounded distributions and also handles reward-based properties. We investigated causes of approximation error and showed that it can often be reduced by scaling time. In experiments performed with our implementation from the MODEST TOOLSET, we saw that the approach works well in practice, but that state space explosion is a significant problem for time-bounded properties. As future work, we would like to investigate the

use of an analysis technique for PTA that supports open clock constraints to obtain tighter bounds, such as the game-based approach (cf. [KNP09] and Section 5.5). An open question is whether we can we prove, like [KNSS00], that time scaling cannot increase the approximation error, even on models with timing anomalies.

Finally, we gave a brief overview of the issues of performing SMC for STA, which turned out to be mostly the same as for PTA. This is because handling continuous probability distributions during simulation is no conceptual problem; it is only the any construct for nondeterministic assignments that adds some complication.

7 Self-Stabilising Photovoltaic Power Generation

The electricity markets in Europe, Asia, and the Americas are evolving towards decentralised structures, essentially rooted in political decisions to counter the worldwide climate change. While large conventional power plants dominated electric power generation up to now, the future will see a drastic increase in the number of distributed microgenerators based on renewable energy sources such as solar and wind power. Electric power grids thus move from a setting in which production was assumed fully controllable so as to always match the uncontrollable, but well-predictable consumption to a setting where the production side becomes uncontrollable, too. External influences such as changing weather conditions can imply drastically higher fluctuations in available electric power. The problem has two challenging facets, namely power grid stability and economy of power production/supply. The stability of the distribution grids is a priority concern because reliable distribution is a prerequisite for economic use of power, whether or not renewable. This asks for improved and better-coordinated diagnostic and prediction techniques, as well as orchestrated demand-side mechanisms to counter critical grid and/or generation situations. These challenges are amplified by the difficulty of centrally controlling the vast number not only of electricity users, but nowadays also of geographically distributed microgenerators.

One of the fastest changing power grids is the German one, owed to substantial increases in wind and solar energy production. This is a consequence of the legal framework enforced by Federal legislation over the last decades: It is characterised by an emphasis on microgenerated power, which enjoys priority in the sense that it must be absorbed by the power grid, unless the grid operates in emergency mode; in that case, however, the wasted power must still be monetarised in the accounting as if it had been fed into the grid. For these reasons, microgenerators of photovoltaic (PV) power have been rolled out massively on the rooftops of residential buildings all over the country. In fact, the growth anticipations for PV microgeneration have in the past years been surpassed by

a large margin: 7.5 gigawatts (GW) have been installed in 2011, while the German government had announced a target growth rate of 1.5 GW per year in 2009.

This growth creates problems. About 75% of the PV microgenerators rolled out are non-measured and cannot be remotely controlled. Since 2007, a regulation (EN 50438:2007) was in place that enforced a frequency-based distributed control strategy. A too high frequency is an indicator for overproduction of power. The regulation stipulated that a PV microgenerator must shut off once it locally observes the frequency to overshoot 50.2 Hz. While this was initially meant as a way to stabilise the grid by cutting overproduction, it later surfaced that due to the high amount of PV generation, an almost synchronous distributed decision to take out this portion may induce a sudden frequency drop, followed by the PV generators joining back in, and so on. It hence may lead to very critical, Europe-wide frequency oscillations. As a consequence, new control strategies have been developed, especially by the VDE, the German Association for Electrical, Electronic & Information Technologies [BBZL11].

To develop robust and correct mechanisms that do not create unexpected instability, mathematically well-founded models of electric power grids and their components are needed. However, the modelling space is huge, and a precise model reflecting all components in a detailed, physically exact manner will be very complex (if at all possible), and virtually impossible to analyse. Instead, suitable abstractions need to be developed, tailored to the fragments of the system under consideration and to the aspects of interest. This will be the topic of the first part of this case study chapter, where we give an overview of the various system aspects of electric power grids, in particular of last-mile microgrids with a significant fraction of microgeneration, and the future challenges faced in such grids (Section 7.1). We also take a look at the modelling challenges encountered in the study of such systems, surveying different modelling and abstraction approaches suitable for different system aspects and measures.

A central issue to ensure the stability of future power grids is proper control for the increasing number of microgenerators. Due to their distributed deployment, decentralised control strategies offer several advantages over centralised management approaches. The ideal is that of a network of independent generators whose control algorithms lead to a self-stabilising system. In the second part of this chapter, we thus focus on the study of control algorithms for photovoltaic microgenerators as an example for the modelling and analysis of future power grids. We present a potpourri of alternative strategies that take up and combine ideas from distributed communication protocol design in Section 7.2. Among them, we find the concept of additive-increase/multiplicative-decrease known from TCP, the idea of exponential backoff as used in CSMA variations,

and adaptive probabilistic switching similar to what is done in 802.11e. We model these strategies formally in MODEST (Section 7.3) and simulate the models with the help of the modes simulator (Section 7.4). We study the oscillatory effect induced by the previous legislatory framework and analyse the currently required approach developed by the VDE. We explore the properties of all alternatives with respect to stability, availability, goodput and fairness. It turns out that Internet-inspired mechanisms to break synchrony, especially by using randomisation, can be considered as one decisive piece in the puzzle of making the power grid future-proof.

Related work The modelling of power grids based on behavioural models with strictly formal semantics is gaining momentum. The most closely related work is likely the paper by [CFK+12], who analyse a multi-player game based on a recently proposed distributed demand-side microgrid management approach [HS11]. Efficiency aspects of demand side management approaches have also been studied [TA09, LKR11]. Other tangible work includes the application of probabilistic hybrid automata with distributed control to the power grid domain [MPL11], and work on network calculus in battery buffered end user homes [BT12].

Origins

This chapter combines material from two publications to which the author was the main contributor: [HH12] is the basis for sections 7.1 to 7.3, while the simulation study of [HHB12] now mainly forms Section 7.4. For the latter, the author *designed* the simulation experiments, which were subsequently *executed* by Pascal Berrang, who also created the graphs and traces shown in this chapter from the collected data.

7.1 Last Mile Power Microgrids

The power distribution grid is hierarchically structured, with a grid of long-distance high-voltage lines (380 kV) as the top layer, down to the leaves which traditionally connect end consumers to the upper layers, using 400 V three-phase current (or 230 V per phase). Typically, these last miles have a tree-like structure through which electric power is distributed towards the leaves from a root. This root is a transformer, which constitutes the connection to the upper layer. Since these grids are relatively small (comprising at most a few hundred residential homes or business customers) and have a clear point of separation from the remaining grid, yet may themselves contain multiple independent microgenerators, we call these last miles *power microgrids*. Figure 7.1 gives a

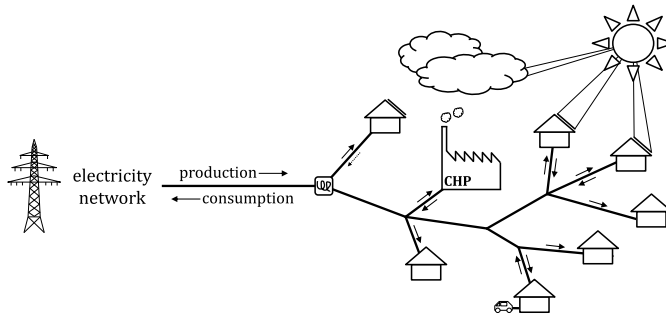


Figure 7.1: Power microgrids

schematic overview of an exemplary power microgrid consisting of seven residential homes and a small industrial customer. In this chapter, we put focus on these leaves, which is where the masses of PV microgenerators are installed. The power management of these last miles is a particular challenge because of their sheer number, the lack of measurement and reporting infrastructure, and the volatility of the photovoltaic production. These problems call for a highly automatic, decentralised and flexible grid management—a very challenging and pressing problem.

7.1.1 Elements of Power Microgrids

A model of a power microgrid needs to take five central aspects into account: (1) the influence of the wide-area power grid it is connected to, (2) the local consumption of electric power, (3, 4) the grid-local electric power generation—which can be further divided into (3) potential and (4) actual generation, i.e. the amount of electric power that can be produced in ideal external circumstances (such as weather and time of day), and the amount that is actually produced after control algorithms inside the generators have been applied—and finally (5) the geographic topology and capacities of the cabling inside the microgrid.

Wide-area connection A power microgrid usually has a single connection point to the wide-area electric power grid. This is a transformer station that converts the network's high voltage to the grid's 400 V three-phase current. Traditionally, electric power flows from large conventional (mostly thermal) power plants through the wide area grid into the microgrids. This infeed is controlled by grid coordinators based on predictions of the local consumption of all the microgrids [HW12], corrected by runtime observations. Runtime

deviations must be corrected in order not to destabilise any grid. Due to the physical limitations related to the power plants in use, only a small fraction of the total generation potential can be employed for runtime adaptation. This is mainly realized with the help of pump-storage plants, where subtracting power is achieved by pumping up water, while adding power is achieved by the reverse, turning water downfall into electric power.

As part of the interconnection to the wide-area grid there is also a safety "fuse", a device that may disconnect the microgrid, intended as a preventative measure both for local events, e.g. to prevent fatal accidents when a cable is damaged during excavation works, as well as for interference from the wide area grid, e.g. to prevent excessive infeed that would exceed the electric power flow capacity in the microgrid. With increasing microgeneration inside microgrids, this safety device may actually turn into a problem, for example by disconnecting the microgrid in case local overproduction exceeds the fuse specifications.

Local consumption At the leaves of the cabling inside the microgrid are residential homes and business customers. In the past they acted only as electric power consumers. The consumption of an individual leaf ultimately depends on a number of factors and decisions by its "inhabitants", yet it roughly follows certain patterns over the course of a day. Variations may be due to external factors such as temperature, influencing the electric power needed for heating or cooling. As such, consumption is uncontrollable, but predictable within certain error bounds. There is a recent trend to make consumption more controllable via so-called demand-side mechanisms [HW12], which intend to control the energy consumption of schedulable devices such as off-peak storage heaters and air conditioners. The decisions are to be based on electric power costs or grid stability conditions.

Generation potential More and more traditional consumers at the leaves of the microgrid are turning into producer-consumers (a.k.a. "prosumers"). At certain times, they may produce more electric power than they consume. The potential output of the microgenerators installed at these leaves depends first and foremost on the type of generator: Combined heat and power plants (CHP) can essentially operate on demand, independent of external circumstances, while microgenerators based on renewable energy sources such as wind and solar power are inherently dependent on natural phenomena. These vary over time in an uncontrollable manner. Wind turbines show relatively moderate fluctuations since wind intensity usually changes only gradually; the amount of available

solar power, however, can change rapidly and significantly when cloud coverage changes quickly.

Actual generation To avoid grid instability, the consumption and production of electric power need to be matched continuously in real time. The actual electric power emitted into the grid by a locally installed microgenerator may affect this stability. With the further increase of these sources, effective control mechanisms are needed in order to avoid over- or underprovisioning of power. Technically, it is no problem to reduce the output of all relevant types of generators—the problem is to decide when to do so, by which amount, when to switch the generators back on, and by how much. Control algorithms are thus an important aspect of future microgenerators. They are expected to have significant influence on the behaviour of future power microgrids.

Local grid topology The topology and spatial layout of the microgrid in terms of cable lengths and diameters clearly impacts its behaviour. The grids have been rolled out in the past with the sole perspective of distributing power downstream, i.e. towards the leaves of the last miles. Now there might be upstream power flow in some parts of the grid. It is easy to come up with scenarios where this may result in stability violations (such as excessive voltage) inside the grid that remain unnoticed at the leaves and at the root. The proper reflection of these influences in a way that generalises to arbitrary last miles is very difficult, because it crucially depends on a specific layout.

7.1.2 Modelling and Abstraction Choices

Since a full model of all individual components of a power microgrid and their precise behaviour is extremely difficult to build and most probably entirely impossible to analyse, the various components have to be represented at appropriate levels of abstraction in a model. These abstractions have to be chosen carefully to make modelling and analysis feasible, yet provide sufficient information to extract reliable answers to the questions of interest from the model.

A first candidate for abstraction is the contribution of the wide-area grid. A detailed modelling of the wide-area grid is clearly out of the scope of a model focussed on just a single power microgrid, while the reverse, the impact of a single microgrid on the behaviour of the entire (e.g. European) electric power grid can be considered negligible. It is thus reasonable to represent the influence of the wide-area grid in the form of a profile, i.e. a deterministic or stochastic function mapping time to the amount of electric power provided. This is an instance of what is called a *load profile*, and is itself assumed independent of

what happens inside the particular microgrid. In addition, the safety fuse at the root we mentioned does not need to be explicitly modelled; instead it is present in the analysis as part of the characterisation of what "unsafe" or "unstable" states need to be avoided.

When it comes to modelling consumer behaviour, the abstraction level depends on the intended modelling purpose. If the focus is on the effects of consumer behaviour, as in a study of demand-side management mechanisms, a detailed consumer model and the explicit representation of individual consumers are obvious necessities. If this is not the focus, two choices are to be made: Should consumers be represented individually or in aggregated form (i.e. as a load profile), and how detailed does the individual or aggregated consumer model need to be? Modelling consumers individually allows the differentiation of consumer types (e.g. into households and businesses) to be represented directly. These distinctions would only lead to variations of the chosen load profile otherwise. Another fundamental question is whether to use a deterministic, stochastic or nondeterministic model of consumption. While deterministic models are often easier to analyse, they embody the risk of exhibiting or causing spurious oscillations or correlations mainly because they may ignore differences between the participants. A stochastic model typically is a good way to avoid these phenomena by assigning probabilities to different behaviours that are all considered part of the model. When it is not possible to assign probabilities to behaviours, nondeterministic models may capture all possible alternatives, but may often turn out to be hard or impossible to analyse.

The modelling spectrum on the power generation side is similar to that on the consumer side. Given a fixed set of generators of different types, a (deterministic or stochastic) load profile is a good representation of the potential generation. It can represent how the external influences on generation potential vary over time, and since a power microgrid covers only a very restricted geographic area (of maybe 1 km^2), it can be considered constant throughout the geographic dimension, since local differences in wind or cloud cover are negligible at this resolution. With respect to actual generation, a load profile may be a good first step, but may hide interesting behaviour that can result from inappropriate control algorithms. For example, the currently deployed control algorithm for PV generators in Germany can lead to oscillating behaviour in times of high potential generation once an unsafe grid state is reached (see Section 7.2.2). In order to study, for example, whether certain demand-side mechanisms can avoid or buffer these oscillations, one would need at least a simple behavioural model of the actual generation.

Finally, the role played by the grid topology is closely tied to the way the physical aspects of electric power are represented in the model. Intertwined differential equations or calculations with complex numbers are the norm, needed to provide nontrivial answers about frequency and voltage. They are achievable for specific layouts. A common abstraction that helps to provide valid answers on a more abstract level assumes the local grid to behave like a perfect "copper plate", thus eliminating any spatial considerations.

7.1.3 Properties and Challenges

As the installed microgeneration capacity increases, the effect of power microgrids on the whole network gets more significant. At the same time, as most microgenerators are based on renewable energy sources, the volatility in the microgrids' behaviour becomes an important concern. There are two core objectives of microgrid and microgenerator management: economy and stability, which are deeply intertwined, yet often conflicting interests.

Objectives and tradeoffs In European legislation, an electric power grid has two distinct modes of operation: *emergency operation*, where direct intervention of the grid coordinator is needed to drive the grid to a safe state, possibly impacting service levels on the consumer side, and *normal operation*, where market incentives drive the decisions of the participants. The stability of the grid is a priority concern because reliable distribution is a prerequisite for economic use of energy. However, the most economically beneficial decisions for individual participants may sometimes run counter to the goal of a stable grid. Grid instability is caused by over- or underproduction, respectively under- or overconsumption, i.e. the electric power production does not match the current consumption. It can be stabilised by suitably adjusting production, consumption, or both.

On the production side, the main issue is to avoid overproduction: While some generation technologies such as CHP are perfectly controllable, the upper limit on potential generation of renewable electric power is dependent on natural phenomena; control strategies for these microgenerators can thus only reduce production compared to their genuine potential. On the other hand, the economic interest of microgenerator owners is to feed as much energy into the grid as possible. In this sense, grid stability and production economy are conflicting interests. Control strategies on the production side, whose overriding goal is to ensure grid stability, thus have to be evaluated for efficiency and fairness in the economic sense as well.

In contrast to this, economic interests can be used as a way to guide the consumption side to a behaviour that is beneficial to stability: Over- and underproduction ideally have a direct effect on the price of electricity, which can drive demand in the desired direction. Nevertheless, the study of effective demand-side mechanisms that lead to compensation of production volatility, with or without economic aspects, is an area as widely open for research as the production side.

Properties We propose the following set of measures to evaluate production control algorithms and demand-side mechanisms, which we collectively call *strategies*, for electric power microgrids:

– **Stability** is the ability of a strategy to keep the grid in a safe state with a minimum of oscillation between safe and unsafe states.
– **Availability** is the overall fraction of time that the grid spends in a safe state.
– **Output** measures the (total or individual, cumulative or averaged) electricity output of the relevant microgenerators, which is usually proportional to the financial rewards of the respective operators.
– **Goodput** relates output to availability: the amount of electric power a generator can add to the grid while the grid is in a safe state.
– **Quality of Service** measures negative impacts on the consumer side. While closely tied to availability, quality of service can also vary while the grid is in a safe state, for example if service reductions are used to achieve safety.
– **Fairness** is the degree to which a strategy manages to distribute adverse consequences equally among the participants. When the grid state does not allow all generators to operate at full power, for example, will each of them be allowed to provide an equal share of the allowed power generation?

Formal Modelling Challenges Power microgrids are complex systems that require expressive modelling formalisms to capture the entirety of their behaviour. Even if only abstracted subsets of a microgrid shall be represented, features such as real-time behaviour and stochastics are necessary, e.g. to model delayed reactions by the grid controller as well as stochastic load profiles or randomised algorithms. In order to faithfully represent the precise physical behaviour of the electric components together with a discrete control strategy, a versatile modelling formalism is a necessity. A more exhaustive discussion on what kinds of modelling features are needed for this problem domain can be found in [HW09]. However, there is an inherent tradeoff between expressivity and the analysis effort needed to compute results. Every modelling study thus needs to precisely identify the aspects to be included in the model as well as the kinds of properties to be analysed so as to make it possible to select the best

matching formalism that is still sufficiently expressive. In the remainder of this chapter, we will use the MODEST language, but most models will correspond to STA without nondeterminism. We can thus use the modes tool for a sound simulation analysis.

7.2 Decentralised Stabilisation Techniques

A major portion of the photovoltaic (PV) microgeneration capacity is mounted on the rooftops of private households, and is as such connected to the last mile. The often excessive volatility of solar production asks for a highly flexible grid management on this level. For the remainder of this chapter, we therefore focus on control strategies for PV microgenerators. As outlined in the previous section, the goal of such a strategy is to reduce actual power output compared to the potential generation whenever this is necessary to maintain grid stability. Otherwise it should allow the output of as much electric power as can be generated. Let us first take a deeper look at what constitutes a "safe state" for power (micro-)grids. There are three fundamental dimensions to stability:

– In Europe, the target **frequency** is 50 Hz. If the frequency leaves the band of 49.8 to 50.2 Hz, this is a serious Europe-wide phenomenon.
– In the end customer grid, the downstream customers may witness considerable **voltage** fluctuations because of upstream fluctuations in production and consumption. Deviations of more than 10 % are not tolerable.
– There are individual limits on the **capacity** of grid strands with respect to energy, i.e. the product of voltage and amperage.

The capacity limits are due to the local grid layout and the "fuse" at the connection point to the upper layers. Voltage has a direct linear dependency to production/consumption and is thus a good measure of the grid state. However, voltage changes are local phenomena, entangled with phase drifts in the last mile and intimately tied to the grid topology and the distances and cabling between producers and consumers. Therefore, the frequency is often used instead of voltage as a measure of the grid state, although frequency drifts usually affect the entire European grid and not only a specific last mile and are subject to dampening effects. An approximately linear dependency between production/consumption and frequency is known, albeit being an indirect effect of physical realities. However, it is still considered an appropriate abstraction by domain experts [Leh12, Wie12]. Roughly, a change in production/consumption of 15 GW approximately corresponds to a 1 Hz change in frequency in the European grid. The installed PV generation capacity in all of Germany in mid-2012 thus corresponded to a frequency spread of about 1.7 Hz.

7.2.1 Centralised vs. Decentralised Control

Photovoltaic microgenerators are difficult to manage. First, this is due to their sheer number, which leads to problems of scalability for any centralised approach. A second problem is their distributed nature: There is currently no measurement, logging and reporting infrastructure in place that enables the collection of accurate and up-to-date information about the state of the grid participants, and there is no communication infrastructure that allows safe remote control. These are two good reasons to consider highly local, decentralised and automatic grid management approaches. Additionally, decentralised approaches that do not need any transmission of information to central coordinators are inherently preferable from a privacy perspective.

The design of a highly local, highly automatic, highly decentralized, and highly flexible grid management is a challenging and pressing problem. It resembles the field of *self-stabilising system* (SSS) design [Dol00]. SSS are built from a number of homogeneous systems that follow the same algorithmic pattern, with the intention that their joint execution results in a stable global behaviour, and can recover from transient disturbances. Compared to the setting usually considered in SSS, there are however some important differences: In a power grid, destabilisation threats must be countered within hard real-time bounds. This is usually not guaranteed for SSS. On the other hand, in SSS usually no participant is considered to have knowledge about the global system state, while in a power grid, the participants do in principle have access to a joint source of localized information by measuring amperage, voltage and frequency.

7.2.2 Current Approaches

As of 2012, about 75 % of the PV microgenerators rolled out in Germany so far were non-measured and could not be remotely controlled. Since 2007, a regulation was in place that enforced a frequency-based distributed control strategy (EN 50438:2007). It stipulated that a microgenerator must shut off once the frequency is observed to overshoot 50.2 Hz. We call this the **on-off** controller. While this was initially meant as a way to stabilise the grid by cutting overproduction, it later surfaced that due to the high amount of PV generation, an almost synchronous distributed decision to take out this portion may induce a sudden frequency drop, followed by the PV generators joining back in, and so on. It hence may lead to critical Europe-wide frequency oscillations.

Due to the obvious problems that widespread use of these rules may lead to, new requirements have been developed as part of VDE-AR-N 4105 [BBZL11].

Consequently, PV generators are since 2012 required to implement the following **linear** control scheme:

– As long as the observed frequency is below 50.2 Hz, the generator may increase its output by up to 10 % of the maximum output that it is capable of per minute.
– When the observed frequency crosses the 50.2 Hz mark, the current output of the generator is saved as p_m. When the frequency f is between 50.2 and 51.5 Hz, the generator must reduce its output linearly by 40 % per Hertz relative to p_m, i.e. its output is given by the function

$$output(f) = p_m - 0.4 \cdot p_m \cdot (f - 50.2).$$

– In case the observed frequency exceeds 51.5 Hz, the generator has to be switched off immediately and may only resume production once the frequency has been observed to be below 50.05 Hz for at least one minute.

As we will see (Section 7.4), this relatively complex algorithm is designed to dampen the effect of PV generation spikes and to avoid introducing oscillatory behaviour, but not to actively steer the system towards a safe state where the frequency is below 50.2 Hz.

7.2.3 Probabilistic Alternatives

If we look at the PV control problem in a more abstract way, it turns out to be remarkably similar to problems solved by communication protocols in computer networks such as the Internet: Limited bandwidth (in our case, capacity of the power grid to accept produced electric power) needs to be shared between a number of hosts (in our case, generators) in a fair way. We thus consider several new control algorithms inspired by concepts from communication protocols, most of which use randomisation to break synchrony and avoid deterministic oscillations. We assume that the controllers run a loop of two steps, with every pair of steps separated by some delay: First, the current grid frequency is measured; then, based on this measurement, the generator output is changed. We describe this model in more detail in Section 7.3. The new control algorithms we consider are the following:

Additive increase, multiplicative decrease The first new control algorithm that we study is inspired by the way the Internet's *Transmission Control Protocol* (TCP) achieves fair usage of limited bandwidth between a number of connections: Bandwidth usage is increased in constant steps (additively), and when a message is lost (taken as an indication of buffer overflows due to congestion), it is reduced by a constant factor (multiplicatively). This *additive-increase,*

multiplicative-decrease (AIMD) policy ensures that several users of the same connection eventually converge to using an equal share of the bandwidth. We directly transfer this approach to PV generators: Power output is increased in constant steps below 50.2 Hz, and if the frequency is measured above 50.2 Hz, the output is scaled down by a constant *factor*.

Probabilistic on-off Our hypothesis is that probabilistic strategies may improve stability without requiring fine-grained modifications of the generators' power output as in AIMD. After each frequency measurement, our first (very simple) randomised controller proceeds like the on-off controller with probability 0.95 or switches off with probability 0.05 to introduce some disturbance to avoid oscillations.

Dynamic die We next use a p-sided die, $p \in \mathbb{N}$, instead of fixed probabilities to decide between proceeding like the on-off controller or unconditionally switching off, the latter corresponding to exactly one side of the die. p increases exponentially when the frequency is low, making it more probable for the generator to switch on, and conversely decreases exponentially when the frequency is above the 50.2 Hz threshold, thus increasing the probability of the generator staying off for some time even when the frequency drops below the threshold again.

Frequency-dependent probabilistic switching Next up, instead of influencing the probability of switching on or off via a stepwise increase or decrease of die size, the frequency-dependent controller makes the probability a function of the currently observed frequency. The probability function has been chosen such that the probability of switching on depends linearly on frequency, being 1 at 50.0 Hz, 0.5 at 50.2 Hz and 0 at 50.4 Hz.

Exponential backoff The previous three policies all decided in a probabilistic manner whether to change the power output or not. In a departure from this, our next controller will unconditionally switch to full power when the current frequency allows and switch off when 50.2 Hz are exceeded, but then wait a probabilistically chosen amount of time before measuring and potentially switching on again.

The precise scheme that we use is *exponential backoff* with *collision detection*. In the computer networks domain, this is commonly employed in CSMA/CD-based (*carrier sense multiple access* with collision detection) medium access protocols such as Ethernet: When one device connected to the

shared medium (e.g. the cable) has data to send, it first *senses* the carrier to determine whether another device is currently sending. If not, it sends its data immediately. However, if the channel is occupied or if the sending is interrupted by another device starting to send as well (a collision), it waits a number of time slots before the next try. This number is sampled from a uniform distribution over a range such as $\{1, \ldots, 2^{bc}\}$, where *bc*, the *backoff counter*, keeps track of the number of collisions and of the number of times that the channel was sensed as occupied when this message should have been sent. The range of possible delays increases exponentially, thus the policy's name; its goal is to use randomisation to prevent two devices from perpetually choosing the same delay and thus always colliding, and to use an exponential increase in the maximum waiting time in order to adapt to the number of devices currently having data to send (again, in order to avoid continuous collisions).

The goals of exponential backoff in network protocols closely match our goals in designing a power generation control scheme: We want all generators to be able to feed power into the grid when it is not "occupied", i.e. when the frequency is below the threshold of 50.2 Hz, and we want to avoid "collisions", i.e. several generators switching on at about the same time and thus creating frequency spikes above that threshold.

Frequency-dependent switching with exponential backoff Our last controller combines the randomisation of switching decisions from the frequency-dependent controller with probabilistic waiting times according to an exponential backoff scheme.

7.3 Modelling Decentralised Controllers

To evaluate the behaviour of the different PV generator control strategies introduced in the previous section, we build MODEST models for power microgrids that use these controllers. As our focus is on the generator control aspect, we chose to use the following abstractions:

– **Physics**: We abstract from the detailed physical characteristics (cf. the *local grid topology* element of Section 7.1.1) by looking only at the frequency observed. Since our focus is on effects of overproduction, we only consider the frequency range above 50 Hz, thus representing 50 Hz as frequency value 0 in our model. This value is assumed when all PV generators are switched off and there is no (= zero) influence from the upper layer. We treat the grid as a "copper plate" where adding power has a direct linear effect on the frequency, so we can describe the grid frequency as the sum of the generator outputs plus the in-feed from the upper layer minus the consumption. Notably, we could

equally well use the observed voltage as a reference quantity for the modelling instead of the frequency in our models since the models are sufficiently abstract. Since frequency changes are a Europe-wide phenomenon and not restricted to a specific last mile, we also exaggerate the influence of each single PV generator.
- **Consumption and upper layer in-feed**: The influence from the upper layer power grid on our last mile as well as the local consumption within the last mile is modelled as an abstract (randomised or deterministic) load profile.
- **Generation potential**: We model the "worst case" of a maximally sunny day. Each PV generator is assumed to be able to contribute the full amount of power it is capable of (given by a constant *MAX*) into the grid at any time.

7.3.1 A Model Template for Power Microgrids

The detailed models of the control strategies all fit into the same model template shown in Figure 7.2. The control strategies become part of a `Generator` process, while a `LoadProfile` process represents the wide-area influence and local consumption; the entire system is finally specified as the parallel composition of G instances of `Generator` plus a single `LoadProfile` instance. This template shows a few more noteworthy modelling choices and abstractions:

A global array `output` of real-valued variables keeps track of each individual generator's current power generation. Each generator repeatedly measures the grid's current frequency, uses this value to decide whether and in which way to modify its own power output, and finally updates its output according to this decision. Each of these measure-update cycles takes M time units, with $D \leq M$ time units passing between the measurement and the change of power output. This delay allows us to model decision and reaction times as well as the time it actually takes for the changes made by one generator to be observed by the others. Higher values of D will thus lead to decisions being made on "older" data, while $D = 0$ implies that every change is immediately visible throughout the last mile. We have thus chosen a discrete measure-update-wait approach; an alternative is to make the generators reactive, i.e. to observe the evolution of the frequency and react when relevant thresholds are crossed.

By use of the `GeneratorInit` process, each generator begins operation after a random, uniformly distributed delay in the range between 0 and M time units; measurements will thus be performed asynchronously. Less realistic, but easier to analyse alternatives would be to have the generators perform their decisions in a fully synchronous manner, or at the same point of time, but in a

```
action init;

const int TIME_BOUND; // analysis time bound
const int G; // number of generators
const int M; // measure every M time units
const int D; // changes take D time units to take effect (D <= M)
const real B = 0.3; // frequency when all generators are on
const real MAX = B / G; // max output of a generator
const real L = 0.1; // max wide-area influence + local consumption

real input; // background generation, in [0, L]
real[G] output; // generator output, each in [0, MAX]

function real freq() = input + /* sum over output array */;

reward r_availability, r_output, r_goodput;
property Availability = Xmax(r_availability | time == TIME_BOUND);
property Output = Xmax(r_output | time == TIME_BOUND);
property Goodput = Xmax(r_goodput | time == TIME_BOUND);

process GeneratorInit(int(0..G) id) {
   // Generators are initially in a random state
   urgent init {= output[id] = Uniform(0, MAX) =};
   // Each generator "starts" after a random delay in [0, M]
   delay(Uniform(0, M)) Generator(id)
}

process Generator(int(0..G) id) {
   action measure, update;
   real fm; // frequency measurement
   clock c = 0;

   process Measure() {
      measure {= fm = freq(), c = 0 =}
   }

   /* control algorithm modelled will be inserted here */
}

process LoadProfile() {
   /* load profile model will be inserted here */
}

par {
:: GeneratorInit(0)
   /* ... */
:: GeneratorInit(G - 1)
:: LoadProfile()
:: invariant(der(sumoutput) == (freq() - input) / TIME_BOUND
   && der(goodput) == freq() > 0.2 ? 0 : (freq() - input) / TIME_BOUND
   && der(availability) == freq() > 0.2 ? 0 : 1 / TIME_BOUND) stop
}
```

Figure 7.2: A model template for power microgrids

```
process Generator(int(0..G) id)
{
   /* ...template code... */

   Measure();
   when(c >= D) urgent(c >= D)
      update {= output[id] = fm >= 0.2 ? 0 : MAX =};
   when(c >= M) urgent(c >= M)
      Generator(id)
}
```

Figure 7.3: 50.2 Hz on-off controller

certain order. However, we have observed that in particular the second alternative generates extreme results (e.g. for fairness) that are clearly artefacts of that abstraction.

7.3.2 Control Strategy Models

We now explain how to model the control strategies described in sections 7.2.2 and 7.2.3 in MODEST to fit into the template introduced above. We omit some of the strategies where the MODEST code would have been mostly repetitive.

Current approaches The MODEST code for the simple on-off strategy that turns the generator off when a frequency of at least 50.2 Hz is observed and turns it to full power in all other cases is shown in Figure 7.3. A direct implementation of the new control scheme according to VDE-AR-N 4105 is shown in Figure 7.4. The switch between normal and emergency mode is obvious in the model. Note that `when urgent`(e) P is a shorthand for `when`(e) `urgent`(e) P.

Probabilistic Alternatives Figure 7.5 shows the model of the AIMD controller. In this case, we chose 10% of the maximum generator output as the constant value when increasing, and $2/3$ as the decrease factor. The latter has shown to provide a good tradeoff between availability and goodput when we compared our analysis results (see next section) for different reduction factors. The MODEST code for the frequency-dependent probabilistic switching controller is shown in Figure 7.6. As described previously, we have chosen a linear function over the range of $[50.0\,\text{Hz}, 50.4\,\text{Hz}]$ for the mapping from measured frequency to switch-off probability. At the critical threshold of 50.2 Hz, the probability of switching off will thus be $1/2$. Finally, the controller based on the exponential backoff approach can be seen in Figure 7.7; the combination with

```
process Generator(int(0..G) id)
{
   real p_m = output[id];
   /* ...template code...  */
   process NormalOperation() {
      alt {
      :: when(fm < 0.2)
         // Increase by 10% of MAX per minute
         update {= output[id] += (0.1 * MAX) / MINUTE,
                   p_m        += (0.1 * MAX) / MINUTE =};
         when urgent(c >= M) Measure()
      :: when(0.2 <= fm && fm < 1.5)
         // 40% gradient
         update {= output[id] = -0.4 * p_m * (fm - 0.2) + p_m =};
         when urgent(c >= M) Measure()
      :: when(1.5 <= fm)
         // Switch off
         EmergencySwitchOff()
      };
      when(c >= D) urgent(c >= D) NormalOperation()
   }
   process EmergencySwitchOff() {
      bool waiting;
      clock minute;
      // Switch off
      update {= output[id] = 0, p_m = 0 =};
      // Wait for frequency to be below 50.05 Hz for one minute
      do {
      :: when urgent(waiting && minute >= MINUTE) break
      :: when urgent(c >= M && !(waiting && minute >= MINUTE))
         Measure();
         urgent alt {
         :: when(fm <= 0.05 && !waiting)
            {= waiting = true, minute = 0, c = 0 =}
         :: when(fm <= 0.05 &&  waiting)
            {= c = 0 =}
         :: when(fm > 0.05)
            {= waiting = false, c = 0 =}
         }
      }
   }
   Measure();
   when urgent(c >= D) NormalOperation()
}
```

Figure 7.4: Model of the controller according to VDE-AR-N 4105

```
process Generator(int(0..G) id) {
   /* ...template code...  */
   Measure();
   when urgent(c >= D) alt {
   :: when(fm <  0.2)
      {= output[id] = min(MAX, output[id] + 0.1 * MAX) =}
   :: when(fm >= 0.2) {= output[id] *= 2/3 =}
   };
   when urgent(c >= M) Generator(id)
}
```

Figure 7.5: Model of additive increase, multiplicative decrease of frequency

```
process Generator(int(0..G) id) {
   /* ...template code...  */
   Measure();
   when urgent(c >= D) update palt {
      :max(0, 0.4 - fm): {= output[id] = MAX =}
      :       fm       : {= output[id] = 0   =}
   };
   when urgent(c >= M) Generator(id)
}
```

Figure 7.6: Model of the frequency-dependent prob. switching controller

```
process Generator(int(0..G) id) {
   int bc; // backoff counter
   int backoff; // number of slots to wait till next try
   /* ...template code...  */
   process Gen() {
      Measure();
      when urgent(c >= D) alt {
      :: when(backoff > 0) update {= backoff-- =}
      :: when(backoff == 0) alt {
         :: when(fm <  0.2) {= output[id] = MAX, bc = 0 =}
         :: when(fm >= 0.2) {= output[id] = 0, bc++,
               backoff = DiscreteUniform(0, (int)pow(2, bc)) =}
         }
      };
      when urgent(c >= M) Gen()
   }
   Gen()
}
```

Figure 7.7: Model of the controller with exponential backoff

frequency-dependent switching is just a simple replacement of the `when` conditions in exponential backoff with a probabilistic alternative (`palt`) that uses the chosen probability function.

7.4 Evaluation

We use the modes discrete-event simulator for MODEST to simulate our last mile model with the different control strategies using $G = 32$ generators and evaluate them (cf. Section 7.1.3) for **stability**, i.e. how prone the algorithms are to the delay incurred prior to their reactions taking effect and how well they manage to keep the system in the safe state where the frequency is below the threshold of 50.2 Hz with a minimum of oscillation; for **availability**, i.e. we look at the overall fraction of time that the system spends in the safe state, and relate this to the **goodput**, which is the average amount of power a generator can add to the system while keeping it safe; and for **fairness**, i.e. whether each generator will be able to provide an equal share of the allowed power generation when the system state does not allow all generators to operate at full power, or whether any of the control schemes continually put certain generators at a disadvantage.

7.4.1 Stability

In order to find out how prone the different algorithms are to frequency oscillations, we perform two simulation runs per algorithm: one for the case where there is no delay between a generator's frequency measurement and the time at which the effect of its modification of power output can be observed over the entire branch of the grid ($D = 0$), and one for the case where this delay amounts to nine tenths of an entire measurement cycle ($D = 9$, $M = 10$). We then plot and compare the overall system frequency over time. To allow a proper comparison, we use a deterministic background load process. The initial power generation for each generator is randomly determined.

Findings The simulation plots are shown in figures 7.8 to 7.15, with frequency values on the y-axis and simulation time on the x-axis. The safe area below 50.2 Hz is highlighted. The system frequency is the upper (blue) curve, while the lower (red) curve is a plot of the background load for reference. The left-hand plots are for the immediate case ($D = 0$) while the right-hand plots show the behaviour in the delayed case ($D = 9$).

We first see (Figure 7.8) that the on-off controller indeed produces extreme frequency oscillations, at least in the delayed setting. Its behaviour in the immediate setting, the details of which are not visible due to the scale of the graph, is also very predictable (modulo changes in background load): In a system with G generators, it enters a cycle of length $G+1$ that starts when a sufficient number of generators is off such that the system frequency is just below 50.2 Hz. The next generator to act that is currently off will then switch on, pushing the frequency above 50.2 Hz and thereby forcing the next generator that is on to switch off. The set of generators that are off thus moves through the total set of generators in a round-robin fashion; assuming that K out of the G generators can be switched on simultaneously without exceeding 50.2 Hz, the frequency will be below that threshold for a fraction $(K+1)/(G+1)$ of the total time, giving a simple formula for system availability (albeit for a very simple setting). We suspect that this behaviour is more of a modelling artefact than realistic, though, and that a (slightly or severely) delayed setting is a better approximation of reality.

The linear controller according to VDE-AR-N 4105 is a clear improvement (Figure 7.9): It exhibits no oscillations or frequency jumps at all and instead mostly follows the background load. It completely fails at keeping the frequency in the safe area below 50.2 Hz, however. Such a direct frequency control was most probably not the intention of this controller's designers, though—in fact, if one treats the background load as the cause for the unsafety in this case, one can argue that, if this background load is to a large extent controllable (as is the case when most power is conventionally generated), modifications of the background load will suffice to keep the frequency in the desired range because this linear controller will *not interfere* with such stabilisation attempts.

The AIMD controller is essentially the on-off controller improved to proceed in smaller steps, and this shows in the frequency plots (Figure 7.10), which show a highly dampened version of the on-off controller's behaviour. Indeed, the interesting point about the AIMD controller is not so much its improved stability but the question whether AIMD brings the same fairness into distributed power generation that it brings to TCP in the computer networks setting.

We now come to the family of probabilistic on-off controllers employing various degrees of randomisation based on different approaches. Our first observation is that, from the stability perspective, neither the probabilistic on-off controller, nor its variant with dynamic die size, nor the exponential backoff-based controller offer any advantage over the simple deterministic approach (figures 7.11, 7.12 and 7.14). It is particularly surprising that both approaches to randomisation (random switching and randomised delays) fail to improve

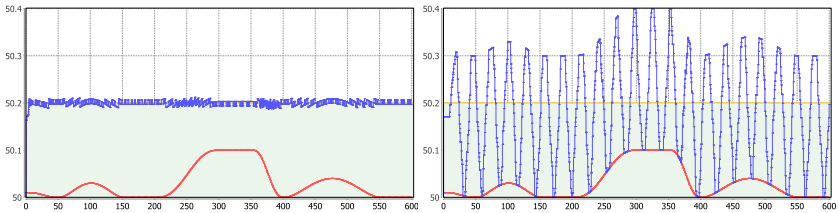


Figure 7.8: Behaviour of the 50.2 Hz on-off controller

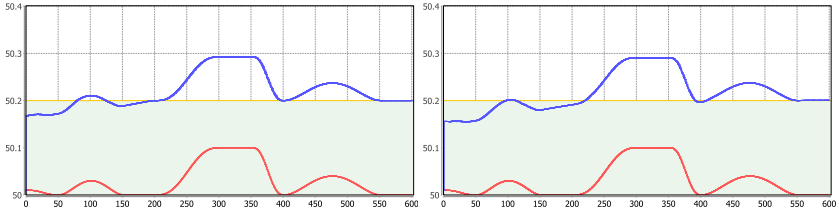


Figure 7.9: Behaviour of the linear controller according to VDE-AR-N 4105

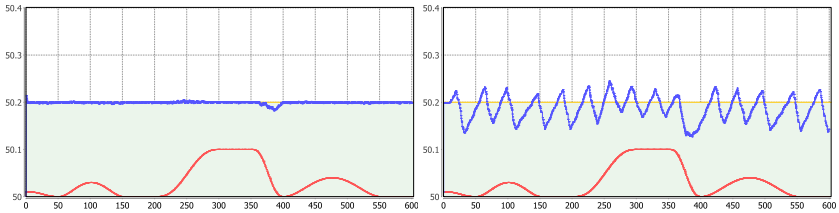


Figure 7.10: Behaviour of the AIMD controller

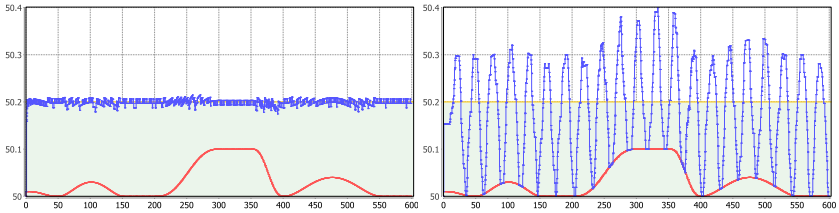


Figure 7.11: Behaviour of the probabilistic on-off controller

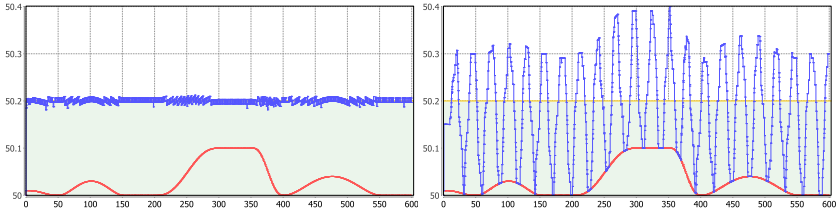


Figure 7.12: Behaviour of the probabilistic on-off controller with dynamic die

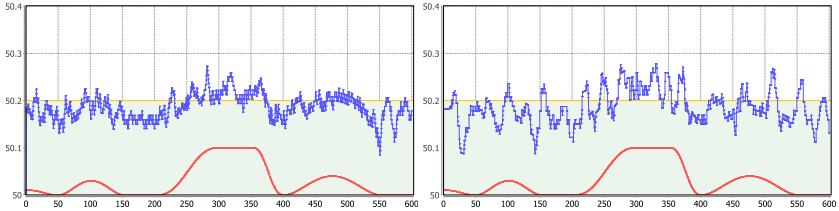


Figure 7.13: Behaviour of the frequency-dependent prob. on-off controller

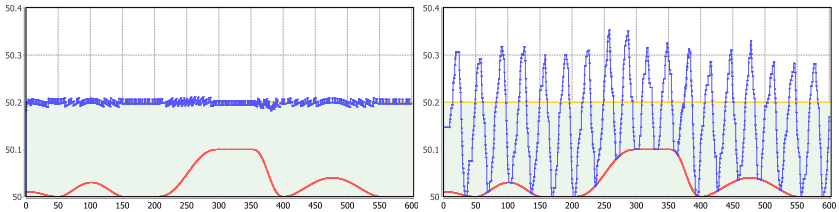


Figure 7.14: Behaviour of the on-off controller with exponential backoff

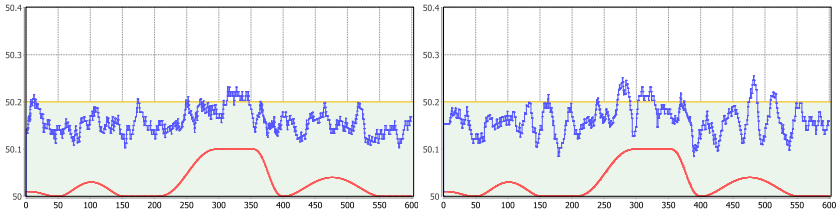


Figure 7.15: Behaviour of the frequency-dependent controller with exp. backoff

stability (except for a very, very slight reduction in the amplitude of the oscillations when using exponential backoff in the delayed setting). For the randomised delay controllers, this appears to be due to the decisions in the delayed case being effectively 90 % synchronous, which means that almost all generators will decide to switch on as soon as the system has become safe for low backoff counter values, which leads to the backoff counter being reset frequently and never even reaching the higher values needed to break synchronicity.

The frequency plots obtained from the two frequency-based on-off controllers, without (Figure 7.13) and with exponential backoff (Figure 7.15), tell a very different story: In both cases, the behaviour of the system is very different from the other on-off controllers, with the frequency changes in the delayed setting being much closer in magnitude to the AIMD controller. This indicates that it is indeed possible to achieve the effect of fine-grained deterministic control using a suitable number of probabilistic controllers employing an adequate randomisation scheme instead. Evaluating the addition of exponential backoff in this case is somewhat difficult: We see that the system appears to remain in the safe area for longer amounts of time when exponential backoff is used, but this effect might potentially be achieved by skewing the probabilities to favor switching off in the non-backoff controller as well. Still, it appears that the controller using exponential backoff copes better with the increase of background load in the middle of the simulation run, so we tend to consider this a beneficial addition in terms of improving stability.

7.4.2 Availability and Goodput

We obtain results for the system availability by including a (reward-based) property in the model framework that evaluates to the fraction of time that the frequency was below 50.2 Hz in a simulation run. We perform 1000 simulation runs in order to average out the availability numbers over the probabilistic decisions, which in this case also come from using a random background load. Instead of merely showing the availability numbers on their own, we chose to also compute the *goodput* of the generators, i.e. the amount of "useful" power generated, with power being of no use when the system is unavailable. We then show, in Figure 7.16, availability compared to total goodput for all generators per time unit for the different controllers.

Findings In terms of availability, four algorithms behave very similarly: The on-off controller, the probabilistic on-off controller, its variant with dynamic die size, and the basic frequency-dependent probabilistic on-off controller. The

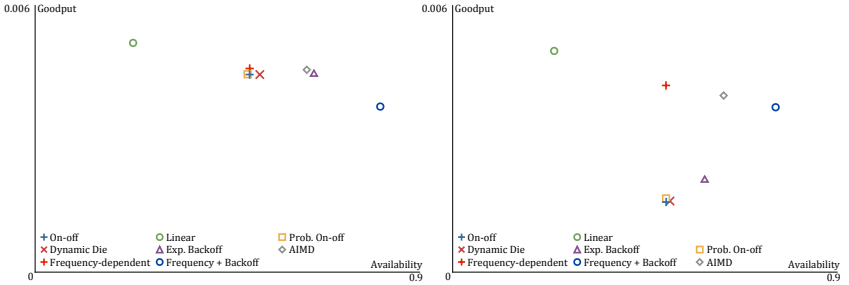


Figure 7.16: Availability vs. total goodput for $D = 0$ (left) and $D = 9$ (right)

first three already exhibited almost identical behaviour in the plots used to evaluate stability, so their being so close in terms of availability comes as no surprise. The two interesting points about this cluster of algorithms are that 1) the frequency-dependent controller with its vastly different behaviour is part of it—however, in the delayed case, it makes this difference shown by providing notably better goodput, and 2) that the on-off controller with exponential backoff is not part of it, showing similar goodput, but increased availability—this being where the reduction in oscillation amplitude compared to the other severely oscillating controllers, as observed in Figure 7.14, probably comes in.

The linear controller according to VDE-AR-N 4105 is a clear outlier in the negative sense for availability; as observed before, however, its goal is probably not to actively stabilise the system but to avoid introducing additional instability. The two positive outliers are the AIMD controller and the frequency-dependent controller with exponential backoff. Both manage to improve availability and goodput, with the AIMD controller favouring goodput and the frequency-dependent exponential backoff controller providing higher availability at slightly lower goodput, i.e. more severe but shorter spikes into the unsafe area above 50.2 Hz.

7.4.3 Fairness

Another aspect worth studying is fairness: We want to find out if some of the controllers allow certain generators to produce significantly more power than others. Since all generators are identical, any such difference will be due to the random initial power generation. As such, it does not make sense to compute averages over several runs, so we perform one very long simulation run instead, using a scaled version of the usual deterministic load profile. The range of values we obtain, with one value being the accumulated output of one of the

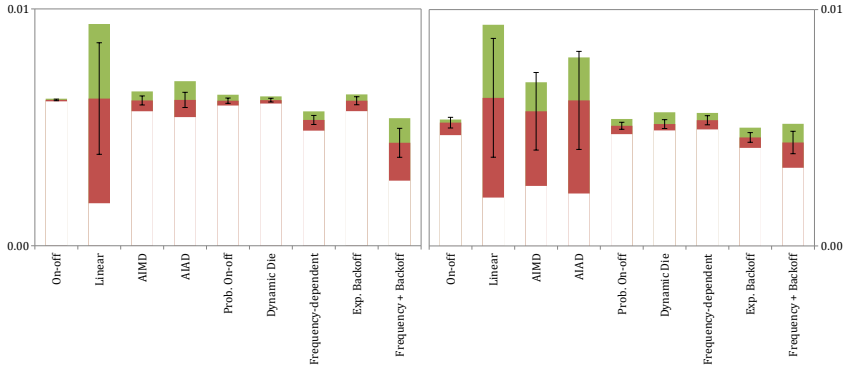


Figure 7.17: Generator output ranges for $D = 0$ (left) and $D = 9$ (right)

generators over the length of the run, is shown in Figure 7.17. The lower (red) area is below the mean output, the upper (green) area is above the mean, and the black line indicates one standard deviation from the mean in both directions.

Findings The most unfair controller clearly is the linear one. This is not unexpected: It does not include any radical changes of output, instead trying to progress in small steps. A generator that initially has a high output will thus more or less remain at high output, while a generator that starts low or off does not get a chance to obtain a significant increase in participation. The main point of interest concerning fairness is whether the AIMD approach will result in a fair sharing of the available production possibility in the same way that it results in a fair sharing of bandwidth when used in TCP. Surprisingly, while it is indeed better than its fairness-ignoring cousin with additive decrease (AIAD), both are still comparatively unfair relative to most other controllers in the delayed setting. This can again be explained by the incremental nature of AIMD, which makes it similar to the linear controller, compared to the very randomised on-off behaviour of the remaining controllers, which show a high degree of fairness both for $D = 0$ and for $D = 9$, with the frequency-dependent controller with exponential backoff being a little worse than the others.

7.4.4 Scaling the Model

One final point of interest is whether the controllers are sensitive to the particular simulation setting of $G = 32$ generators with a total contribution to frequency of at most 0.3 Hz. The number of 32 generators is realistic, if not on

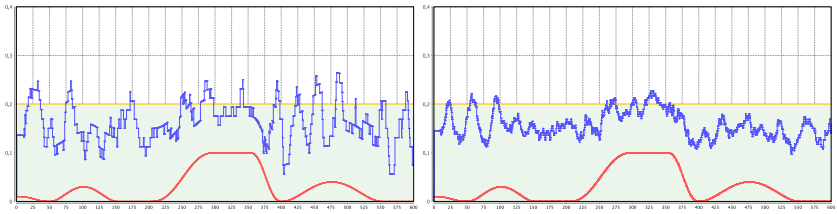


Figure 7.18: The frequency-dependent controller with backoff, scaled setting

the high side, for a typical last mile branch today. However, as mentioned, frequency is a Europe-wide phenomenon, while voltage is highly local, so it makes sense to study both lower (more realistic for voltage) and higher numbers of generators (more realistic for frequency). We thus performed the simulation analysis for stability and availability/goodput for $G \in \{2,4,8,16,64\}$ as well, in two different settings:

– **Scaled setting**: The contribution to frequency of each generator is scaled such that the total contribution of all generators is the same as for $G = 32$. This will allow us to find out how the number of generators affects the controllers' behaviour.
– **Unscaled setting**: The contribution of each individual generator is constant (namely $0.3\,\text{Hz}/32 = 0.009375\,\text{Hz}$, as in all models studied in the previous sections), leading to a lower (for $G < 32$) or higher (for $G > 32$) total contribution of all generators. In particular, with a maximum background generation of 0.1 Hz, the system cannot become unsafe for up to 10 generators, but it can reach up to 50.7 Hz for $G = 64$.

Stability

We found that the number of generators itself (= scaled setting) does not change the system behaviour or the performance of the different controllers in terms of stability significantly. As an example, consider the behaviour of the frequency-dependent controller with exponential backoff: Its behaviour in the scaled setting with $D = 9$ is shown in Figure 7.18, for $G = 16$ on the left and $G = 64$ on the right. Compared to the right-hand plot in Figure 7.15, the only significant difference is the step size of the changes when a generator is switched on or off.

However, once we increase the total frequency contribution (= unscaled setting with $G = 64$, $D = 9$), the two frequency-dependent controllers start behaving differently. The plot on the left-hand side of Figure 7.19 shows the behaviour of the controller *without* exponential backoff: It quickly starts to induce

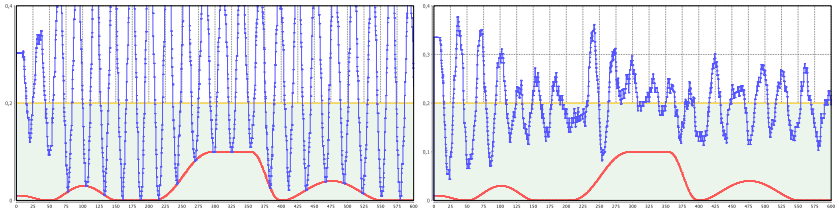


Figure 7.19: The two frequency-dependent controllers in the unscaled setting

severe oscillations. This happens because the probability function used by the controller assigns probability 1 to switching off for frequencies beyond 50.4 Hz, which are more likely to be reached at some point when the total contribution of all generators is as high as 0.6 Hz. Once this happens, almost all generators switch off due to the measurement delay of $D = 9$, only to switch on again with a very high probability in the next "round". The frequency-dependent controller *with* exponential backoff is the only probabilistic one that still works in this setting, albeit not leading to as stable a system as for $G = 32$. Still, as shown on the right-hand side of Figure 7.19, it manages to reduce oscillations over time. Experiments using different and non-linear functions to compute the switch-off probability depending on the frequency indicate that these scaling problems can be reduced by using a "better" probability function [Ber13].

Availability and Goodput

When we look at availability and goodput, we mainly see these findings confirmed: Figure 7.20 shows the evolution of availability and goodput for changing values of G for the three most interesting controllers (again for $D = 9$). This time, we show the average goodput *per generator* and time unit. In the scaled setting (left-hand side), this implies that a controller that is not affected by changes of G will have constant availability, and goodput inversely proportional to G. We clearly see that the on-off and AIMD controllers show this kind of behaviour. The frequency-dependent controller with exponential backoff, however, performs better as G increases: Availability improves, and for twice the number of generators, the goodput is actually larger than half the previous value. This controller's randomised decisions thus manage to create an averaging effect for a higher number of participants and thereby successfully exploit the potential benefits of randomisation.

In the unscaled setting (right-hand side of Figure 7.20), the results are more difficult to interpret. We first observe that it needs at least 11 generators to reach

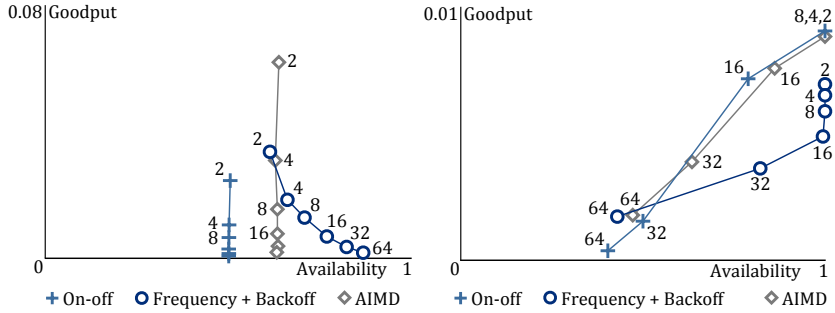


Figure 7.20: Availability vs. goodput per generator, scaled vs. unscaled setting

a system frequency above 50.2 Hz under full background load (22 generators with no background load). For an ideal controller, the goodput per generator should thus be the same for all $G < 11$; after that, an increasing number of generators has to "share" the safe area below 50.2 Hz, thus goodput should again be inversely proportional to G. This appears to be the case for the on-off and AIMD controllers, though both fail to keep availability constant and thus independent of the number of generators. The frequency-dependent controller with exponential backoff again behaves differently: While it starts with lower goodput and does not keep this independent of G, it performs better in terms of availability compared to the on-off and AIMD controllers. As observed in Section 7.4.4, it finally starts to break down when $G = 64$. Still, this shows that combining probabilistic switching with randomized delays can indeed lead to a more robust controller; we expect the use of a better probability function and fine-tuning of the backoff procedure to have the potential for significant improvement of the frequency-dependent controllers with and without exponential backoff.

7.5 Summary and Discussion

In this chapter, we have discussed elementary mechanisms for distributed control of power grids facing considerable infeed of renewable energy. We have focussed on the properties and modelling aspects needed to describe and analyse these systems and the techniques necessary to manage them in a highly flexible, highly automated, and highly decentralized manner.

Another system which is highly decentralized, highly flexible and managed in a highly automated way is the Internet. As we have discussed, certain solutions that have been coined as part of Internet protocols can be adapted to serve

beneficially in future distributed control of power grids. This benefit might not be restricted to concrete Internet solutions, but might more generally also materialise for some of the more universal, genuine Internet design principles, such as:

– Network neutrality and fairness: There is no discrimination in the way the network shares its capacity among its users. Ideally, the net is fair in the sense that if n users are sharing a connection, then on average each user can use about $1/n$-th of the capacity.
– Intelligent edges, dumb core: Putting intelligence into the net itself is much more cost ineffective than placing it at the edges of the networks, i.e. into the end user appliances.
– Distributed design and decentralised control: Distributed, decentralised control is not only a means to assure scalability. It also is a prime principle to protect end user privacy that would be at stake if centralised authorities would collect information for decision making.

There are a number of similarities between the Internet and the power grid, including its excessive size, its hierarchical structure, its organic growth, and its ultimately high dependability. It seems that this implies a number of very good reasons why the future management of power grids should take strong inspirations from the way the Internet is managed. Our research indicates some first concrete examples of this kind:

We have presented a simulation-based evaluation of a potpourri of decentralised stabilisation strategies inspired by concrete Internet solutions. The discussion has focussed on the frequency as a single indicator of the grid state, and has assumed a linear impact of PV production on it. The assumption of linearity is an indirect effect of the physical realities. We could also have used the observed voltage as a reference quantity for the modelling instead of the frequency. For low voltage rotating current, the allowed voltage interval is 440 V to 360 V. In this interpretation, the linearity assumption would directly hold, and the analysis results can therefore be transferred to this interpretation right away. It is worth noting that frequency drifts usually affect the entire European grid, and not only a specific last mile. This also means that the influence of a single last mile on the frequency is in fact negligible, and appears amplified in our studies. In contrast, voltage changes are local phenomena, entangled with phase drifts in the last mile. The influence of a single microgenerator on the observed voltage in a last mile is therefore much more substantial, but may actually be skewed by spatial properties. In fact, we see the development of models that use voltage as the central measure, and that thus need to include more details about the physical and spatial behaviour, as future work. Such models need to include all the details that affect the strategy under study, but

should at the same time still be amenable to an automated analysis. Deterministic variants of hybrid automata may fit this bill, but their faithful simulation (in particular when it is necessary to ensure that no events are ever missed) already poses technical challenges.

At the same time, the discussion has focussed on grid stability, not grid economy. The stability of the distribution grids is a priority concern, because reliable distribution is a prerequisite for economic use of energy, whether renewable or not. Of course, the same basic control algorithms can use intraday and spot market prices in addition to frequency or voltage as indicators for the grid state in times when the grid is operating well (which will hopefully be dominating time-wise anyway). These indicators are nowadays easily accessible in residential areas over the Internet (provided there is power to run the residential Internet connection, a grid stability problem) and can be used for decentralised demand-response management. A massive roll-out of such appliances may however make the power grid fall into a similar trap as the current German on-off controllers do, cf. Figure 7.8. This is because automatic decentralised decisions orchestrated by a central signal may lead to oscillations. Suitable use of randomisation may again help to avoid such undesired effects.

Concerning the modelling and analysis techniques used in this chapter, we have seen that the models are stochastic timed automata without nondeterminism. This makes them amenable to a straightforward simulation, i.e. SMC, analysis without encountering the problems we first described for MDP in Section 4.6 and that reappeared for PTA and general STA. When starting the work, we also studied variants of the model template in which the initial delay or the ordering between the individual generators was nondeterministic, in contrast to the uniform stochastic initial delay we used here. However, it emerged in experiments with hand-picked schedulers that the extremal behaviour would most likely be obtained when all generators execute at the same time, and that this behaviour would be very unrealistic. For this reason, we did not consider nondeterministic models, or the application of exhaustive model checking, any further. Still, with improvements in efficiency of the exhaustive model checking approach for STA described in the previous chapter, it may become feasible to scale time such that the uniform initial delay is represented in sufficient detail in the abstracted model for the unrealistic extremal behaviours to become rare. Since all generators are currently identical in behaviour, symmetry reduction techniques may help, too. Then, a model checking analysis may provide new insights.

Probably the most tangible contribution of this chapter is a first indication that concepts developed for distributed communication protocols may be attractive candidates for components of future power grid stabilisation strategies.

We emphasised that there is an urgent need for better stabilisation strategies in light of the strong growth of PV microgeneration especially in Germany. The Internet-inspired mechanisms to break synchrony, especially by using randomisation, appear—at least to the author—as one decisive piece in the puzzle of making the power grid future-proof.

8 Discussion

On our tour of stochastic timed systems in this thesis, we have given consistent definitions of formal models ranging from the basis of labelled transition systems and discrete-time Markov chains all the way to the expressive nondeterministic-stochastic formalism of stochastic timed automata (STA). We have shown how these models build on each other, and how they can be extended with additional orthogonal features such as rewards, discrete variables, and compositional modelling of networks of automata. Our modelling language of choice is MODEST, and we have step-by-step introduced the necessary syntax and semantics to cover the increasingly expressive formalisms. We have restricted our attention to the analysis of reachability properties and, after we introduced real-time aspects, expected accumulated reward values. Our analysis methods of choice are (exhaustive) model checking and statistical model checking, and we have investigated their applicability to each model. Finally, we concluded with a case study from the area of control algorithms for power microgrids with significant renewable energy infeed.

Achievements

Aside from presenting a consistent view of various related formal models that cover the quantitative aspects needed to study stochastic timed systems, the core achievements presented in this thesis lie in statistical model checking for Markov decision processes (MDP, Chapter 4), the comparison of probabilistic timed automata (PTA) with invariants and PTA with deadlines (Chapter 5), a first fully-automated exhaustive model checking method and implementation for STA (Chapter 6), and an evaluation of the idea to use randomisation and other ideas from Internet protocols to control PV microgenerators (Chapter 7).

SMC for MDP The problem of model checking has in principle been long solved for MDP. However, due to the state space explosion problem, i.e. the excessive memory usage of the model checking techniques, the application

of SMC to MDP models has recently become an area of significant research interest. We have presented two approaches to tackle this problem, which are based on an on-the-fly partial order respectively confluence check. They provide sound results and preserve the general flavour of SMC as a technique that conserves memory at the cost of runtime. Both approaches have been implemented in the modes simulator, which is available as part of the MODEST TOOLSET.

PTA with deadlines As an extension of MDP, performing SMC for PTA faces the same problems as for MDP (and possibly more). On the other hand, but again like for MDP, the classical model checking problem has been mostly solved for PTA. This is why we only summarised the existing model checking techniques and the potential ways to apply SMC to PTA when the problem has been solved for MDP. The core contribution of Chapter 5 instead lies in the comparison of the two existing variants of PTA: Those that provide location invariants to restrict the passage of time (as used in tools like UPPAAL and PRISM), and those that rely on the more compositional deadlines to achieve the same goal (as used in MODEST). We showed that the expressiveness of the two model variants is incomparable, but that there is a large and useful subset of each that can be transformed into the other. Of these transformations, the one from deadlines to invariants is nontrivial, and we have proven the correctness of the algorithm we introduced. Finally, to top off the presentation of PTA, we have given a detailed description of the modelling and analysis of a very typical PTA model of a communication protocol (namely the BRP).

STA model checking STA again inherit the problems of PTA when it comes to SMC. However, previous to this thesis, they had not been amenable to model checking either. All studies that used STA models therefore had to carefully restrict to the fragment of deterministic STA, or with equal care use a hopefully suitable resolver or time scheduler to remove the nondeterminism. Based on existing techniques for stochastic hybrid systems, we have developed, implemented and evaluated the first fully-automated model checking algorithm for general, nondeterministic STA. The resulting mcsta tool is now available as part of the MODEST TOOLSET.

Internet ideas for microgrids Aside from describing the real-life problems and ensuing modelling challenges in the area of distributed microgeneration based on renewable energy sources, in particular on volatile solar power, the main contribution of Chapter 7 is in taking up the insight that there are striking similarities between this area and the Internet. Our formal modelling and

analysis of Internet-inspired control algorithms for photovoltaic microgenerators shows that transferring ideas and solutions from the Internet to the power domain may be a key ingredient for the reliable and economic future power generation largely based on renewable sources.

Limitations and Future Work

We have seen, both in the description and in the evaluation of the techniques and analyses presented in the previous chapters, that important limitations and a number of open problems remain.

SMC for MDP and beyond Although we have presented two original approaches and summarised three other existing techniques to perform SMC for MDP, the general problem must still be considered as unsolved. The two methods that we have introduced work well, but are restricted to certain subclasses of MDP. While some of the technical constraints, such as the limitation of the POR-based approach to interleavings only, can likely be overcome, both approaches are fundamentally only applicable to spuriously nondeterministic MDP. The other three approaches, which tackle the problem for the general case, all currently have individual limitations that reduce their usefulness: They are either not sound in the general case, or their memory usage pattern is more comparable to exhaustive than statistical model checking. With SMC for MDP being an open problem, the same very much applies to SMC for PTA and STA as well.

STA model checking While we have shown that our method to perform model checking of STA works well for some models, we have also seen that it severely suffers from state space explosion, especially when it comes to time-bounded properties. It is thus worthwhile to investigate ways of state space reduction that are applicable to or specifically designed for this new method. In particular, the *essential states reduction* originally developed for the RAPTURE tool [DJJL02, Section 5] appears promising to yield smaller digital clocks MDP. Another option would be to use a zone-based analysis of the abstraction PTA, such as the forwards reachability technique of [KNSS02, DKN04] (cf. Section 5.5). Its main drawback was that it could only deliver upper bounds on maximum reachability probabilities, which is less relevant in the STA case where the computed values are upper bounds from the start. Still, an additional overapproximation error may be introduced. As the game-based approach to

PTA analysis [KNP09] (also described in Section 5.5) has been shown to perform better than digital clocks, it may also be able to speed up the STA analysis. An additional benefit of both of these alternatives to the digital clocks approach is that they can handle open clock constraints and should therefore lead to a lower overapproximation error. On the other hand, they cannot handle rewards, so digital clocks will still be needed for reward-based properties—but these were not the main bottleneck in our experiments anyway. Finally, finding conditions under which scaling time does not increase the approximation error and proving this fact (like it was shown in [KNSS00] for the case of bounded distributions) is already on our agenda for future work. If it can be proven, this would pave the way for an automatic abstraction-refinement technique that could deliver the tightest bounds possible given the available memory and runtime budget.

Microgrid modelling and analysis The main catch of our analysis of the Internet-inspired microgenerator control algorithms was the use of a copper plate model with frequency as the key measurement of the power grid's state. The reality of power microgrids is much more complex, and a significant portion of the overall challenge of installing masses of distributed microgenerators in fact lies in keeping nominal conditions at all points inside these last mile grid strands. Our models thus need to be extended with the necessary physical quantities and laws as well as spatial properties of representative microgrids. This will most likely require a *hybrid* model that can represent both the controllers' discrete decisions and the continuous evaluation of the physical quantities (usually described as differential equations), as well as a suitable and trustworthy hybrid system simulation tool. On the other hand, we have not even taken advantage of many of the techniques already described in this thesis when we analysed the models described in Chapter 7. We need to go back to the existing models and see if we can abstract them further in ways that would make model checking possible while still keeping sufficient detail to show interesting behaviours.

Current Related Work

In the areas of SMC for MDP and control of renewable energy generation, there is important ongoing work by others: We have presented three other approaches to perform something like SMC for general MDP, although they all come with some restrictions concerning soundness, correctness, or memory usage. The field of managing renewable energy production to preserve grid stability (and potentially improve economy) is an extremely wide and active area of research,

and we have only given a small glimpse into past related work at the beginning of Chapter 7. We expect significant developments in this area in the near future.

Due to the fact that on the one hand, PTA model checking is mostly a solved problem, while on the other hand, it is still an open question how to soundly and efficiently apply SMC even just to the submodel of MDP, it appears that there is not a large amount of current work in the area of PTA. In fact, the contributions we presented in this thesis in the area of PTA are among the oldest work we have included, dating back to 2010 and earlier. The analysis of STA, on the other hand, has so far received scant attention where nondeterministic models are concerned even though the pioneering work by Kwiatkowska et al. was already published in 2000, i.e. 14 years before this thesis was written. In fact, we see the analysis of STA as the most promising avenue for future research.

A Modest Syntax and Semantics

Throughout the previous chapters, we have introduced the MODEST modelling language step-by-step for the different models from LTS to STA. As the models became more expressive, however, we did not update the inference rules for constructs already presented earlier and instead relied on an intuitive understanding of how to extend these rules to the new models. For reference, we present a complete formal syntax and semantics of the MODEST language for STA in this appendix.

Origins The material presented in this appendix is an adapted and reduced version of Sections 2 and 3 of [HHHK13], which were written by the author as an extension and revision of the original MODEST syntax and semantics introduced in [BDHK06].

A.1 Syntax

We start our discussion by giving the complete grammar for MODEST processes and process behaviours.

Models, Processes and Declarations

A MODEST model consists of a sequence of *declarations* and a *process behaviour*. Declarations are constructed according to the following grammar:

$$
\begin{array}{lll}
dcl ::= & [\texttt{patient} \mid \texttt{impatient}]\ \texttt{action}\ act;\ \mid & (\textit{actions}) \\
& \texttt{exception}\ excp;\ \mid & (\textit{exceptions}) \\
& type\ var\ [= e];\ \mid & (\textit{variables}) \\
& \texttt{process}\ ProcName(t_1\ x_1, \ldots, t_k\ x_k)\ \{\ dcl\ P\ \} & (\textit{processes})
\end{array}
$$

syntax	type	domain	continuous behaviour
`bool`	Boolean variables	$\{true, false\}$	$\dot{x} = 0$
`int`	unbounded integers	$\mathbb{Z}$	$\dot{x} = 0$
`int`$(e_1..e_2)$	bounded integers	$\{e_1, \ldots, e_2\}$	$\dot{x} = 0$
`real`	static real variables	$\mathbb{R}$	$\dot{x} = 0$
`clock`	clocks	$\mathbb{R}_0^+$	$\dot{x} = 1$
`reward`	rewards	$\mathbb{R}_0^+$	$\dot{x} = 0$, or given by invariants

Table A.1: Types of variables in MODEST

where, for $i \in \{1, \ldots, k\}$, *act*, *excp*, *var*, *ProcName* and the x_i are identifiers (names), *type* and the t_i are types (see Table A.1 for the list of types[1]) and P is a process behaviour. A MODEST model can thus be treated as a process without parameters; when we refer to a process in the remainder of this chapter, we mean a declared process or the model's unnamed top-level process. The declarations of a process and its parameters define the following (finite) sets associated to the process:

– $Act_P = PAct_P \uplus IAct_P \uplus Excp_P \uplus \{\tau, \bot, \natural\}$, the set of actions partitioned into patient and impatient actions, exceptions and the silent action τ, the error action $\bot$ and the break action $\natural$;
– Var_P, the set of variables, which contains both declared variables as well as the process' parameters.

To simplify our definitions w.l.o.g., we assume that any particular patient or impatient action, exception or variable is declared in at most one place in a given model. Properties are also included in the declarations section of a model (but not of a process) using a superset of the syntax described in the previous chapters of this thesis.

Variables can initially be assigned the value of an expression $e \in Sxp$ explicitly; otherwise, they are implicitly initialised to a default value, typically zero. As described in Section 2.3, we treat expressions in an abstract manner: We omit a full grammar in this thesis and only point out the possibility of including the `any` and `der` operators as well as probability distributions as described in the previous chapters.

[1]The implementation in the MODEST TOOLSET also supports fixed-size arrays and user-defined data structures, which are technical extensions but not conceptually relevant for this thesis.

Process Behaviours

The process behaviours are constructed according to the following grammar:

$$
\begin{aligned}
P ::= {} & \mathit{act} \mid \texttt{stop} \mid \texttt{abort} \mid \texttt{break} \mid P_1 ; P_2 \mid \\
& \texttt{when}(e_b)\ P \mid \texttt{urgent}(e_b)\ P \mid \texttt{invariant}(e_i)\ P \mid \texttt{invariant}(e_i)\{P\} \mid \\
& \texttt{alt}\{::P_1 \ldots ::P_k\} \mid \texttt{do}\{::P_1 \ldots ::P_k\} \mid \texttt{par}\{::P_1 \ldots ::P_k\} \mid \\
& \mathit{act}\ \texttt{palt}\{:w_1 : U_1 ; P_1\ \ldots\ :w_k : U_k ; P_k\} \mid \\
& \texttt{throw}(\mathit{excp}) \mid \texttt{try}\{P\}\ \texttt{catch}\ \mathit{excp}_1\{P_1\} \ldots \texttt{catch}\ \mathit{excp}_k\{P_k\} \mid \\
& \texttt{relabel}\{I\}\texttt{by}\{G\}\{P\} \mid \texttt{extend}\{H\}\{P\} \mid \\
& \mathit{ProcName}(e_1, \ldots, e_k)
\end{aligned}
$$

where for $j \in \{1, \ldots, k\}$, $\mathit{act} \in \mathit{PAct}_Q \cup \mathit{IAct}_Q \cup \{\tau\}$, $U_j \in \mathit{Upd}$, $\mathit{excp} \in \mathit{Excp}_Q$, $\mathit{excp}_j \in \mathit{Excp}_Q$, $e_b \in \mathit{Bxp}$, $e_i \in \mathit{Bxp}'$ (see below), $w_j \in \mathit{Axp}$, $H \subseteq \mathit{PAct}_Q \cup \mathit{IAct}_Q$ is a set of observable actions, and I and G are vectors of equal length which have elements in $\mathit{Act}_Q \setminus \{\natural, \bot\}$ such that all elements in I are pairwise different and not equal to τ with the current process or any process that contains the current process as Q. In order to simplify the semantics of process calls ($\mathit{ProcName}(e_1, \ldots, e_k)$), we assume that every process call corresponds to a unique process declaration in the model, which can be achieved by renaming in any case.

As mentioned in Section 5.4.2, the variables $\mathtt{r}_i$, $i \in \{1, \ldots, n\}$, of type `reward` must not be used in any expressions outside of properties except in assignments of the form $\mathtt{r} := \mathtt{r} + e$ and in invariant expressions of the form

$$e_1 \wedge \texttt{der}(\mathtt{r}_k) = e_2^k \wedge \texttt{der}(\mathtt{r}_{k+1}) = e_2^{k+1} \wedge \ldots \wedge \texttt{der}(\mathtt{r}_m) = e_2^m$$

with $e_1 \in \mathit{Bxp}$ and $e_2^j \in \mathit{Axp}$ for $j \in \{k, \ldots, m\}$. We use Bxp' to denote the set of expressions of this form.

Shorthands

Two useful shorthands for more complex process behaviours are the `hide` and `delay` constructs. Using `hide`, which is useful to modify the action alphabet and thus the synchronisation interface of a process behaviour for parallel composition, is equivalent to relabelling a set of actions or exceptions to the silent action τ:

$$\texttt{hide}\{H\}\{P\} \stackrel{\text{def}}{=} \texttt{relabel}\{H\}\texttt{by}\{\tau, \ldots, \tau\}\{P\}.$$

The idea behind the `delay` construct is to provide an easy way to specify that a certain process behaviour should be executed after some precise amount of time. It can be expanded to a process call to newly introduced processes:

$$\texttt{delay}(e_{\mathit{delay}}, e_{\mathit{cond}})\ P \stackrel{\text{def}}{=} \mathit{Delay}()$$

where $e_{delay} \in Sxp$, $e_{cond} \in Bxp$ and *Delay* is a new, unique process name for every occurrence of the `delay` shorthand that is defined as

$$\texttt{process}\ \mathit{Delay}()\ \{\ \texttt{clock}\ c;\ \texttt{real}\ x = e_{delay};\ \texttt{when}(c \leq x \wedge \neg e_{cond})\ \texttt{urgent}(c \geq x \wedge e_{cond})\ P\ \}.$$

Intuitively, $\texttt{delay}(e_{delay}, e_{cond})\ P$ delays the initial behaviour of P until e_{delay} time units have passed and condition e_{cond} becomes true; at that point, the initial behaviour of P becomes urgent, so that it is performed without further delay. Condition e_{cond} is part of the shorthand because it cannot be added by hand since writing e.g. $\texttt{urgent}(c \geq x)\ \texttt{urgent}(e_{cond})$ would be equivalent to $\texttt{urgent}(c \geq x \vee e_{cond})$.

A.2 Symbolic Semantics

In this section, we define the symbolic semantics of a given MODEST process, which is the first of two steps in defining a semantics for MODEST. It consists in transforming the process calculus constructs of a process into a stochastic timed automaton with invariants and deadlines. In this STA, the parts of the model not related to process calculus-based definitions, such as model variables or assignments, will still be maintained in a symbolic form. In absence of non-tail-recursive process calls, the automaton will stay finite, so that it is possible to build it explicitly. The second step of the MODEST semantics is the transformation of this symbolic automaton into a concrete, usually infinite, model. It has already been presented in Section 6.1 for the case of STA with invariants. The addition of deadlines can be handled as described for PTA in Section 5.2.

Compared to the original symbolic semantics of MODEST that was given in [BDHK06], we add two first-level `invariant` constructs that replace the previous shorthand of the same name. The shorthand mapped $\texttt{invariant}(i)\ P$ to

$$\texttt{alt}\ \{\ ::\texttt{when}(i)\ P\ \ ::\texttt{urgent}(\neg i)\ \texttt{when}(\mathit{false})\ \texttt{throw}(\mathit{excp}_{invariant})\ \}$$

where $\mathit{excp}_{invariant}$ was a new exception only used for this purpose. The shorthand took advantage of the fact that guards do not influence deadlines, so the deadline $\neg i$ would still take effect even though the edge it is associated to is disabled. Since this construction simulated an invariant using a deadline, it could not be used to represent all possible invariants (cf. Section 5.2).

The semantics presented here also corrects several minor issues with the original STA semantics; for example, assignments are composed using $\circ$ instead of $\cup$ in several places and the semantics for `par` and `palt` no longer

involves any exceptions, which originally led to the order of the parallel behaviours being relevant for the semantics, which went against the intuition of an associative and commutative parallel composition operator.

Definition 87 (MODEST symbolic semantics)**.** The *symbolic semantics* of a MODEST process P with process behaviour Q is the STA

$$\langle Loc, Var, Act \cup Excp, \rightarrow, l_{init}, Inv', AP, \varnothing \rangle$$

where

– Act, $Excp$ and Var are the respective unions of the sets $Act_P \cup \{\tau, \natural, \perp\}$, $Excp_P$ and Var_P of P and the sets Act, $Excp$ and Var given by the symbolic semantics of the processes that are called from within P's process behaviour,
– the initial values of the variables are determined by setting them to the default value of their type and then applying the update $v_0 = \mathbf{A}(Q) \circ v_{\text{decl}}$ where v_{decl} represents the initial values assigned to all variables in Var according to their declarations, $\mathbf{A}$ is the *assignment collecting function* as defined in Table A.2 and the $\circ$ operator denotes the sequential composition of updates, i.e. the execution of the second operand followed by that of the first operand,
– $l_{init} = Q$,
– the invariant function Inv' is defined as

$$Inv'(P) \stackrel{\text{def}}{=} \begin{cases} e_1 & \text{if } Inv(P) \text{ is equivalent to } e_1 \wedge \texttt{der}\ldots \in Bxp' \\ Inv(P) & \text{otherwise} \end{cases}$$

where Inv is the original invariant function as given in Table 5.2 of Chapter 5,
– AP contains the most top-level expressions in $\text{Bxp}(Var)$ that occur in the properties specified in the model,
– the edge relation $\rightarrow$ is given by the inference rules presented below, and
– the set Loc of locations is the set of reachable process behaviours according to $\rightarrow$

together with a reward $r_{\mathtt{r}} = \langle r^{\mathtt{r}}_{Loc}, r^{\mathtt{r}}_{\rightarrow} \rangle$ for every variable $\mathtt{r}$ of type `reward` where

$$r^{\mathtt{r}}_{Loc}(P) \stackrel{\text{def}}{=} \begin{cases} e_{\mathtt{r}} & \text{if } Inv(P) \text{ is equivalent to } \ldots \wedge \texttt{der}(\mathtt{r}) = e_{\mathtt{r}} \wedge \ldots \in Bxp' \\ 0 & \text{otherwise} \end{cases}$$

and $r^{\mathtt{r}}_{\rightarrow}$ is given by the assignments of the form $\mathtt{r} := \mathtt{r} + e$ in $\rightarrow$.

Note that the STA corresponding to a MODEST process contains both invariants for locations, via the invariant collection function, as well as deadlines on edges. Every edge $\xrightarrow{g,d,a}$ thus has three labels: The guard g, the deadline d and the action (or exception) a. An edge leads to a symbolic probability distribution over pairs of an update and a target process behaviour. When writing such

$\mathbf{A}(P) = \varnothing$	if P has one of the following forms: *act*, `stop`, `abort`, `throw`$(excp)$, `break` or *act* `palt` $\{ : w_1 : U_1 ; P_1 \ \ldots \ : w_k : U_k ; P_k \}$
$\mathbf{A}(P) = \mathbf{A}(Q)$	if P has one of the following forms: $Q ; Q'$, `when`(e) Q, `urgent`(e) Q, `invariant`(e) Q, `invariant`(e) $\{ Q \}$, `try` $\{ Q \}$ `catch` e_1 $\{ P_1 \}$ $\ldots$ `catch` e_k $\{ P_k \}$, `relabel` $\{ I \}$ `by` $\{ G \}$ Q or `extend` $\{ H \}$ Q
$\mathbf{A}(P) = \bigcup_{i=1}^{k} \mathbf{A}(P_i)$	if P has one of the following forms: `alt` $\{ :: P_1 \ldots :: P_k \}$, `do` $\{ :: P_1 \ldots :: P_k \}$ or `par` $\{ :: P_1 \ldots :: P_k \}$
$\mathbf{A}(ProcName(e_1, \ldots, e_k)) = \mathbf{A}(Q) \circ \{x_1 = e_1, \ldots, x_k = e_k\}$	if *ProcName* is declared as `process` $ProcName(t_1\, x_1, \ldots, t_n\, x_k)$ $\{ Q \}$

Table A.2: The assignment collecting function [BDHK06]

functions as sets in inference rules in the remainder of this appendix, we may omit all elements that map to $0 \in Axp$ for brevity.

Inference Rules

The edge relation $\rightarrow$ is given by the following inference rules:

Actions and the like The inference rules for performing an action, including the special action $\flat$ to break out of a loop with the `break` construct, are straightforward:

$$\frac{}{act \xrightarrow{tt,ff,act} \{ \langle \varnothing, \checkmark \rangle \mapsto 1 \}} \ (act) \qquad \frac{}{\texttt{break} \xrightarrow{tt,ff,\flat} \{ \langle \varnothing, \checkmark \rangle \mapsto 1 \}} \ (break)$$

$$\frac{}{\texttt{abort} \xrightarrow{tt,ff,\perp} \{ \langle \varnothing, \texttt{abort} \rangle \mapsto 1 \}} \ (abort)$$

The *successfully terminated process* $\checkmark$ is only used as part of the semantics and cannot be specified syntactically. The `abort` process, which *can* be specified syntactically but more usually occurs as the consequence of an unhandled exception (see below), simply performs the unhandled error action $\perp$ over and over again. There is no inference rule for the `stop` process since its semantics is precisely to do nothing.

Conditions Any process behaviour can be decorated with a guard using the `when` construct, with a deadline using the `urgent` construct, and with an invariant using the `invariant` construct. Guards, deadlines and the $\texttt{invariant}(e)\ P$ form of the `invariant` construct only affect the first, immediate edges resulting from the decorated behaviour and then disappear—

$$\frac{P \xrightarrow{g,d,a} \mathscr{W}}{\texttt{when}(e)\ P \xrightarrow{g\wedge e,d,a} \mathscr{W}}\ (\textit{when}) \qquad \frac{P \xrightarrow{g,d,a} \mathscr{W}}{\texttt{urgent}(e)\ P \xrightarrow{g,d\vee e,a} \mathscr{W}}\ (\textit{urgent})$$

$$\frac{P \xrightarrow{g,d,a} \mathscr{W}}{\texttt{invariant}(e)\ P \xrightarrow{g,d,a} \mathscr{W}}\ (\textit{inv})$$

—while $\texttt{invariant}(e)\,\{\,P\,\}$ is a static operator, i.e. it does not disappear after following one edge; with $Q(P) = \texttt{invariant}(e)\,\{\,P\,\}$, its inference rule reads:

$$\frac{P \xrightarrow{g,d,a} \mathscr{W}}{Q(P) \xrightarrow{g,d,a} \mathscr{W} \circ \mathbf{M}_{\text{inv}}^{-1}}\ (\textit{sinv}) \quad \text{where } \mathbf{M}_{\text{inv}}(\langle U, P'\rangle) \stackrel{\text{def}}{=} \begin{cases} \langle U, Q(P')\rangle & \text{if } P' \neq \checkmark \\ \langle U, P'\rangle & \text{if } P' = \checkmark. \end{cases}$$

The inference rules for `invariant` ignore the actual invariant expression e because it does not become part of the edge relation, but is instead preserved as part of the function *Inv* that maps each location to an invariant.

Sequential composition A process behaviour P' can be performed only after another process behaviour P has successfully terminated when they are composed using the ; operator for sequential composition:

$$\frac{P \xrightarrow{g,d,a} \mathscr{W}}{P;\, Q \xrightarrow{g,d,a} \mathscr{W} \circ \mathbf{M}_{;}^{-1}}\ (\textit{seq}) \quad \text{where } \mathbf{M}_{;}(\langle U, P'\rangle) \stackrel{\text{def}}{=} \begin{cases} \langle U, P';\, Q\rangle & \text{if } P' \neq \checkmark \\ \langle \mathbf{A}(Q) \circ U, Q\rangle & \text{if } P' = \checkmark. \end{cases}$$

Nondeterministic choice A nondeterministic choice between several process behaviours is provided by the `alt` keyword:

$$\frac{P_i \xrightarrow{g,d,a} \mathscr{W}_i \quad (i \in \{1,\ldots,k\})}{\texttt{alt}\ \{ ::P_1 \ldots ::P_k \} \xrightarrow{g,d,a} \mathscr{W}_i}\ (\textit{alt})$$

Loops The semantics of the `do` construct is defined via the auxiliary `auxdo` construct, which is not part of the MODEST syntax. It is used to keep track of the original behaviour of the loop which must be restored after each iteration:

$$\texttt{do}\ \{ ::P_1 \ldots ::P_k \} \stackrel{\text{def}}{=} \texttt{auxdo}\ \{\ \texttt{alt}\ \{ ::P_1 \ldots ::P_k \}\ \}\ \{\ \texttt{alt}\ \{ ::P_1 \ldots ::P_k \}\ \}$$

The semantics of `auxdo` is defined in two inference rules: The first one handles the case that the break action is performed to jump out of the loop, while the second rule defines the semantics of performing a step within the loop, including proceeding to the next iteration once the last step has been performed:

$$\frac{P \xrightarrow{g,d,\flat} \mathscr{W}}{\texttt{auxdo}\ \{\ P\ \}\ \{\ Q\ \} \xrightarrow{g,d,\tau} \{\ \langle \varnothing, \checkmark \rangle \mapsto 1\ \}}\ (\textit{breakout})$$

$$\frac{P \xrightarrow{g,d,a} \mathscr{W} \qquad (a \neq \flat)}{\texttt{auxdo}\ \{\ P\ \}\ \{\ Q\ \} \xrightarrow{g,d,a} \mathscr{W} \circ \mathbf{M}_{\text{do}}^{-1}}\ (\textit{auxdo})$$

where

$$\mathbf{M}_{\text{do}}(\langle U, P' \rangle) \stackrel{\text{def}}{=} \begin{cases} \langle U, \texttt{auxdo}\ \{\ P'\ \}\ \{\ Q\ \} \rangle & \text{if } P' \neq \checkmark \\ \langle \mathbf{A}(Q) \circ U, \texttt{auxdo}\ \{\ Q\ \}\ \{\ Q\ \} \rangle & \text{if } P' = \checkmark. \end{cases}$$

Process calls Let process *ProcName* be declared as

$$\texttt{process}\ \mathit{ProcName}(t_1\, x_1, \ldots, t_n\, x_k)\ \{\ P\ \}.$$

Process call $\mathit{ProcName}(e_1, \ldots, e_k)$ then behaves like its process behaviour P where the variables $x_1, \ldots, x_k$ have been assigned the values of the expressions $e_1, \ldots, e_k$. The assignment of variables is performed just before the process call is executed, which is ensured by the assignment collecting function **A** (see Table A.2). The inference rule for process calls is then

$$\frac{P \xrightarrow{g,d,a} \mathscr{W}}{\mathit{ProcName}(e_1, \ldots, e_k) \xrightarrow{g,d,a} \mathscr{W}}\ (\textit{call}).$$

We note that this rule does not include any renaming or stacking of variables, but this is only possible because of the assumption of unique process declarations for process calls in the previous section, and it only works correctly for tail-recursive models. To give a semantics for arbitrarily recursive models, we would need to extend the inference rule above and the assignment collecting function to include the necessary renamings there. We would also have to account for (countably) infinite sets of variables in the definition of STA.

Probabilistic choice The probabilistic choice, `palt`, is *action-prefixed*: An action *act* is performed and the target of the edge labelled *act* is then the (symbolic) probability distribution over the weighted alternatives that make up the

body of the `palt`. The inference rule thus reads as follows:

$$\frac{}{act\ \texttt{palt}\ \{ : w_1 : U_1 ; P_1\ \ldots\ : w_k : U_k ; P_k \} \xrightarrow{tt,ff,act} \mathscr{W}} \ (palt)$$

where

$$\mathscr{W}(\langle \mathbf{A}(P_i) \circ U_i, P_i \rangle) \stackrel{\text{def}}{=} \sum_{j=1}^{k} \mathbf{Ind}(i,j) \cdot w_j$$

and

$$\mathbf{Ind}(i,j) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } \mathbf{A}(P_i) \circ U_i = \mathbf{A}(P_j) \circ U_j \wedge P_i = P_j \\ 0 & \text{otherwise.} \end{cases}$$

Note that $\mathscr{W}$ still is a symbolic function in $Upd \times Loc \to Axp$. Two problems that are not taken into account by this inference rule are that a weight w_i may be negative, and that the sum of all weights may be zero. As noted in earlier chapters, these are considered modelling errors, i.e. the semantics of a (syntactically valid) MODEST model that contains one or more such errors is not defined. Tool support will check for this and reject such models.

Exceptions Once declared, an exception can be *thrown*...

$$\frac{}{\texttt{throw}(excp) \xrightarrow{tt,ff,excp} \{ \langle \varnothing, \texttt{abort} \rangle \mapsto 1 \}} \ (throw)$$

...and either be caught or ignored by enclosing `try-catch` constructs of the form

$$Q(P) = \texttt{try}\ \{\ P\ \}\ \texttt{catch}\ excp_1\ \{\ P_1\ \}\ \ldots\ \texttt{catch}\ excp_k\ \{\ P_k\ \};$$

if caught, the specified exception handler will be executed:

$$\frac{P \xrightarrow{g,d,a} \mathscr{W} \quad (a \notin \{ excp_1, \ldots, excp_k \})}{Q(P) \xrightarrow{g,d,a} \mathscr{W} \circ \mathbf{M}_{\text{try}}^{-1}} \ (try)$$

$$\frac{P \xrightarrow{g,d,excp_i} \mathscr{W} \quad (i \in \{1, \ldots, k\})}{Q(P) \xrightarrow{g,d,\tau} \{ \langle \mathbf{A}(P_i), P_i \rangle \mapsto 1 \}} \ (catch)$$

where

$$\mathbf{M}_{\text{try}}(\langle U, P' \rangle) \stackrel{\text{def}}{=} \begin{cases} \langle U, Q(P') \rangle & \text{if } P' \neq \checkmark \\ \langle U, \checkmark \rangle & \text{if } P' = \checkmark . \end{cases}$$

$\alpha(P) = \{act\} \setminus \{\tau\}$	if P has the form act
$\alpha(P) = \varnothing$	if P has one of the following forms: `stop`, `break`, `abort` or `throw`$(excp)$
$\alpha(P) = \alpha(Q)$	if P has one of the following forms: `when`$(e)\ Q$, `urgent`$(e)\ Q$, `invariant`$(e)\ \{Q\}$ or $ProcName(e_1,\ldots,e_k)$ and process $ProcName$ is declared as `process` $ProcName(t_1\,x_1,\ldots,t_n\,x_k)\ \{Q\}$
$\alpha(P) = \alpha(P_1) \cup \alpha(P_2)$	if P is of the form $P_1; P_2$
$\alpha(P) = \bigcup_{i=1}^{k} \alpha(P_i)$	if P has one of the following forms: `alt` $\{::P_1 \ldots ::P_k\}$, `do` $\{::P_1 \ldots ::P_k\}$ or `par` $\{::P_1 \ldots ::P_k\}$
$\alpha(P) = \alpha(Q) \cup \bigcup_{i=1}^{k} \alpha(P_i)$	if P has the form `try` $\{Q\}$ `catch` $excp_1\ \{P_1\}\ \ldots$ `catch` $excp_k\ \{P_k\}$
$\alpha(P) = \{a_1 \mapsto a'_1,\ldots,a_k \mapsto a'_k\}(\alpha(Q)) \setminus \{\tau\}$	if P has the form `relabel` $\{a_1,\ldots,a_k\}$ `by` $\{a'_1,\ldots,a'_k\}\ Q$
$\alpha(P) = \alpha(Q) \cup \{act_1,\ldots,act_k\}$	if P has the form `extend` $\{act_1,\ldots,act_k\}\ Q$
$\alpha(P) = \alpha(act) \cup \bigcup_{i=1}^{k} \alpha(P_i)$	if P has the form act `palt` $\{:w_1\colon asgn_1; P_1\ \ldots\ :w_k\colon asgn_k; P_k\}$

Table A.3: The alphabet of a process behaviour [BDHK06]

Parallel composition The process behaviours in a `par` construct run concurrently, synchronising on the actions in their shared alphabet. The alphabet of a process is computed by function α as defined in Table A.3. A parallel composition terminates successfully whenever all its components do so, i.e. $\checkmark \parallel_B \checkmark \stackrel{\text{def}}{=} \checkmark$ for any B.

To define the semantics of parallel composition, we resort to the auxiliary operator $\parallel_B$, with $B \subseteq Act$. The `par` construct is then defined as

$$\texttt{par}\ \{::P_1 \ldots ::P_k\} \stackrel{\text{def}}{=} (\ldots((P_1 \parallel_{B_1} P_2) \parallel_{B_2} P_3)\ldots) \parallel_{B_{k-1}} P_k$$

with

$$B_j = (\bigcup_{i=1}^{j} \alpha(P_i)) \cap \alpha(P_{j+1}).$$

The behaviour of $\|_B$ is that action $a \notin B$ can be performed autonomously, i.e., without the cooperation of the other parallel component:

$$\frac{P_1 \xrightarrow{g,d,a} \mathscr{W} \quad (a \notin B)}{P_1 \|_B P_2 \xrightarrow{g,d,a} \mathscr{W} \circ \mathbf{M}^{-1}_{\mathrm{par}\,P_2}}\ (lpar) \qquad \frac{P_2 \xrightarrow{g,d,a} \mathscr{W} \quad (a \notin B)}{P_1 \|_B P_2 \xrightarrow{g,d,a} \mathscr{W} \circ \mathbf{M}^{-1}_{\mathrm{par}\,P_1}}\ (rpar)$$

where

$$\mathbf{M}_{\mathrm{par}\,P}(\langle U, P' \rangle) \stackrel{\text{def}}{=} \langle U, P' \|_B P \rangle \text{ and } \checkmark \|_B \checkmark = \checkmark,$$

while, if we let $\otimes_a = \wedge$ if $a \in PAct_Q$ and $\otimes_a = \vee$ if $a \in IAct_Q$ for some process Q, the inference rule for synchronisation reads:

$$\frac{P_1 \xrightarrow{g_1,d_1,a} \mathscr{W}_1 \quad P_2 \xrightarrow{g_2,d_2,a} \mathscr{W}_2 \quad (a \in B)}{P_1 \|_B P_2 \xrightarrow{g_1 \wedge g_2, d_1 \otimes_a d_2, a} (\mathscr{W}_1 \times \mathscr{W}_2) \circ \mathbf{M}^{-1}_{\mathrm{par}}}\ (sync)$$

where $(\mathscr{W}_1 \times \mathscr{W}_2)(\langle \alpha_1, \alpha_2 \rangle) \stackrel{\text{def}}{=} \mathscr{W}_1(\alpha_1) \cdot \mathscr{W}_2(\alpha_2)$ for all α_1 and α_2—corresponding to the product of two probability spaces—and

$$\mathbf{M}_{\mathrm{par}}(\langle U_1, P_1' \rangle, \langle U_2, P_2' \rangle) \stackrel{\text{def}}{=} \langle U_1 \cup U_2, P_1' \|_B P_2' \rangle \text{ if } U_1 \text{ and } U_2 \text{ are consistent}$$

where, as before, $\checkmark \|_B \checkmark = \checkmark$. Inconsistent updates are considered a modelling error.

Alphabet manipulation The alphabet of a process can be modified with the `extend` and `relabel` constructs. The `extend` construct merely extends the alphabet of a process (see Table A.3) and may affect behaviour only if it appears within the context of a `par` construct: For $Q(P) = \texttt{extend}\ \{act_1, \ldots, act_k\}\ P$,

$$\frac{P \xrightarrow{g,d,a} \mathscr{W}}{Q(P) \xrightarrow{g,d,a} \mathscr{W} \circ \mathbf{M}^{-1}_{\mathrm{ext}}}\ (extend) \text{ where } \mathbf{M}_{\mathrm{ext}}(\langle U, P' \rangle) \stackrel{\text{def}}{=} \begin{cases} \langle U, Q(P') \rangle & \text{if } P' \neq \checkmark \\ \langle U, \checkmark \rangle & \text{if } P' = \checkmark. \end{cases}$$

The semantics for the `relabel` construct is as in traditional process algebra: Observable actions and exceptions are renamed according to a relabelling function, but the behaviour remains otherwise unchanged. For

$$Q(P) = \texttt{relabel}\ \{a_1, \ldots, a_k\}\ \texttt{by}\ \{a_1', \ldots, a_k'\}\ P,$$

the inference rule is thus

$$\frac{P \xrightarrow{g,d,a} \mathscr{W}}{Q(P) \xrightarrow{g,d,f_Q(a)} \mathscr{W} \circ \mathbf{M}^{-1}_{\mathrm{rel}}}\ (relabel)$$

where

$$\mathbf{M}_{\mathrm{rel}}(\langle U, P' \rangle) \stackrel{\text{def}}{=} \begin{cases} \langle U, Q(P') \rangle & \text{if } P' \neq \checkmark \\ \langle U, \checkmark \rangle & \text{if } P' = \checkmark \end{cases}$$

and

$$f_Q(a) = \begin{cases} a'_i & \text{if } a = a_i \\ a & \text{otherwise} \end{cases}$$

for $i \in \{1, \ldots, n\}$ with the restriction that actions may only be mapped to actions (including τ but not $\natural$ or $\perp$) and exceptions may only be mapped to exceptions or τ.

Bibliography

[AC98] Alan Agresti and Brent A. Coull. Approximate is better than "exact" for interval estimation of binomial proportions. *The American Statistician*, 52(2):119–126, May 1998.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[AY06] Todd R. Andel and Alec Yasinsac. On the credibility of MANET simulations. *IEEE Computer*, 39(7):48–54, 2006.

[Bai98] Christel Baier. On algorithmic verification methods for probabilistic systems. Habilitation thesis, Fakultät für Mathematik & Informatik, Universität Mannheim, 1998.

[BB87] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks*, 14:25–59, 1987.

[BBB+10] Ananda Basu, Saddek Bensalem, Marius Bozga, Benoît Caillaud, Benoît Delahaye, and Axel Legay. Statistical abstraction and model-checking of large heterogeneous systems. In *FMOODS/FORTE*, volume 6117 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2010.

[BBH+13] Paolo Ballarini, Nathalie Bertrand, András Horváth, Marco Paolieri, and Enrico Vicario. Transient analysis of networks of stochastic timed automata using stochastic state classes. In *QEST*, volume 8054 of *Lecture Notes in Computer Science*, pages 355–371. Springer, 2013.

[BBJM12] Patricia Bouyer, Thomas Brihaye, Marcin Jurdzinski, and Quentin Menet. Almost-sure model-checking of reactive timed automata. In *QEST*, pages 138–147. IEEE Computer Society, 2012.

[BBZL11] Jens Bömer, Karsten Burges, Pavel Zolotarev, and Joachim Lehner. Auswirkungen eines hohen Anteils dezentraler Erzeugungsanlagen auf die Netzstabilität bei Überfrequenz & Entwicklung von Lösungsvorschlägen zu deren Überwindung, 2011. Study commissioned by EnBW Transportnetze AG, Bundesverband Solarwirtschaft e.V. and Forum Netztechnik/Netzbetrieb im VDE e.V.

[BCC+14] Tomás Brázdil, Krishnendu Chatterjee, Martin Chmelik, Vojtech Forejt, Jan Kretínský, Marta Z. Kwiatkowska, David Parker, and Mateusz Ujma. Verification of markov decision processes using learning algorithms. In *ATVA*, volume 8837 of *Lecture Notes in Computer Science*, pages 98–114. Springer, 2014.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[BDG06] Christel Baier, Pedro R. D'Argenio, and Marcus Größer. Partial order reduction for probabilistic branching time. *Electr. Notes Theor. Comput. Sci.*, 153(2):97–116, 2006.

[BDHH12] Jonathan Bogdoll, Alexandre David, Arnd Hartmanns, and Holger Hermanns. mctau: Bridging the gap between modest and uppaal. In *SPIN*, volume 7385 of *Lecture Notes in Computer Science*, pages 227–233. Springer, 2012.

[BDHK06] Henrik C. Bohnenkamp, Pedro R. D'Argenio, Holger Hermanns, and Joost-Pieter Katoen. MoDeST: A compositional modeling formalism for hard and softly timed systems. *IEEE Trans. Software Eng.*, 32(10):812–830, 2006.

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In *SFM-RT 2004*, number 3185 in Lecture Notes in Computer Science, pages 200–236. Springer, September 2004.

[BDM+98] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos: A model-checking tool for real-time systems. In *CAV*, volume 1427 of *Lecture Notes in Computer Science*, pages 546–550. Springer, 1998.

[Ber13] Pascal Berrang. Stabilising power micro grids. B.Sc. thesis, Universität des Saarlandes, 2013.

[BFHH11] Jonathan Bogdoll, Luis María Ferrer Fioriti, Arnd Hartmanns, and Holger Hermanns. Partial order methods for statistical model checking and simulation. In *FMOODS/FORTE*, volume 6722 of *Lecture Notes in Computer Science*, pages 59–74. Springer, 2011.

[BGC04] Christel Baier, Marcus Größer, and Frank Ciesinski. Partial order reduction for probabilistic systems. In *QEST*, pages 230–239. IEEE Computer Society, 2004.

[BHH12] Jonathan Bogdoll, Arnd Hartmanns, and Holger Hermanns. Simulation and statistical model checking for modestly nondeterministic models. In *MMB/DFT*, volume 7201 of *Lecture Notes in Computer Science*, pages 249–252. Springer, 2012.

[BHHK10] Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, and Joost-Pieter Katoen. Performance evaluation and model checking join forces. *Commun. ACM*, 53(9):76–85, 2010.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[Blo01] S. C. C. Blom. Partial τ-confluence for efficient state space generation. Technical Report SEN-R0123, CWI, 2001.

[BS00] Sébastien Bornot and Joseph Sifakis. An algebraic framework for urgency. *Information and Computation*, 163(1):172–202, 2000.

[BT12] Jean-Yves Le Boudec and Dan-Cristian Tomozei. A demand-response calculus with perfect batteries. In *MMB/DFT*, volume 7201 of *Lecture Notes in Computer Science*, pages 273–287. Springer, 2012.

[BvdP02] Stefan Blom and Jaco van de Pol. State space reduction by proving confluence. In *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 596–609. Springer, 2002.

[CFHM07] Hyeong Soo Chang, Michael C. Fu, Jiaqiao Hu, and Steven I. Marcus. A survey of some simulation-based algorithms for Markov decision processes. *Communications in Information & Systems*, 7(1):59–92, 2007.

[CFK+12] Taolue Chen, Vojtech Forejt, Marta Z. Kwiatkowska, David Parker, and Aistis Simaitis. Automatic verification of competitive stochastic systems. In *TACAS*, pages 315–330, 2012.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[Cha88] D. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1):65–75, 1988.

[CSS02] David Cavin, Yoav Sasson, and André Schiper. On the accuracy of MANET simulators. In *POMC*, pages 38–43. ACM, 2002.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, 1996.

[dAKN+00] Luca de Alfaro, Marta Z. Kwiatkowska, Gethin Norman, David Parker, and Roberto Segala. Symbolic model checking of probabilistic processes using MTBDDs and the Kronecker representation. In *TACAS*, volume 1785 of *Lecture Notes in Computer Science*, pages 395–410. Springer, 2000.

[Dav13] Alexandre David, May 2013. Private discussion at the 2013 NASA Formal Methods Symposium.

[DJJL01] Pedro R. D'Argenio, Bertrand Jeannet, Henrik Ejersbo Jensen, and Kim Guldstrand Larsen. Reachability analysis of probabilistic systems by successive refinements. In *PAPM-PROBMIV*, volume 2165 of *Lecture Notes in Computer Science*, pages 39–56. Springer, 2001.

[DJJL02] Pedro R. D'Argenio, Bertrand Jeannet, Henrik Ejersbo Jensen, and Kim Guldstrand Larsen. Reduction and refinement strategies for probabilistic analysis. In *PAPM-PROBMIV*, volume 2399 of *Lecture Notes in Computer Science*, pages 57–76. Springer, 2002.

[DKN04] Conrado Daws, Marta Z. Kwiatkowska, and Gethin Norman. Automatic verification of the ieee 1394 root contention protocol with kronos and prism. *STTT*, 5(2–3):221–236, 2004.

[DKRT97] Pedro R. D'Argenio, Joost-Pieter Katoen, Theo C. Ruys, and Jan Tretmans. The bounded retransmission protocol must be on time! In *TACAS*, volume 1217 of *Lecture Notes in Computer Science*, pages 416–431. Springer, 1997.

[DLL+11a] Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikucionis, Danny Bøgsted Poulsen, Jonas van Vliet, and Zheng

Wang. Statistical model checking for networks of priced timed automata. In *FORMATS*, volume 6919 of *Lecture Notes in Computer Science*, pages 80–96. Springer, 2011.

[DLL+11b] Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikucionis, and Zheng Wang. Time for statistical model checking of real-time systems. In *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 349–355. Springer, 2011.

[DN04] Pedro R. D'Argenio and Peter Niebert. Partial order reduction on concurrent probabilistic programs. In *QEST*, pages 240–249. IEEE Computer Society, 2004.

[Dol00] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.

[EC82] E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.*, 2(3):241–266, 1982.

[EH86] E. Allen Emerson and Joseph Y. Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, 1986.

[EHZ10] Christian Eisentraut, Holger Hermanns, and Lijun Zhang. On probabilistic automata in continuous time. In *LICS*, pages 342–351. IEEE Computer Society, 2010.

[EP10] Sami Evangelista and Christophe Pajault. Solving the ignoring problem for partial order reduction. *STTT*, 12(2):155–170, 2010.

[FHH+11] Martin Fränzle, Ernst Moritz Hahn, Holger Hermanns, Nicolás Wolovick, and Lijun Zhang. Measurability and safety verification for stochastic hybrid systems. In *HSCC*, pages 43–52. ACM, 2011.

[FKNP11] Vojtech Forejt, Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Automated verification techniques for probabilistic systems. In *SFM*, volume 6659 of *Lecture Notes in Computer Science*, pages 53–113. Springer, 2011.

[Fre05] Goran Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. In *HSCC*, volume 3414 of *Lecture Notes in Computer Science*, pages 258–273. Springer, 2005.

[GDF09] Sergio Giro, Pedro R. D'Argenio, and Luis María Ferrer Fioriti. Partial order reduction for probabilistic systems: A revision for distributed schedulers. In *CONCUR*, volume 5710 of *Lecture Notes in Computer Science*, pages 338–353. Springer, 2009.

[GHK+11] Hernan Baro Graf, Holger Hermanns, Juhi Kulshrestha, Jens Peter, Anjo Vahldiek, and Aravind Vasudevan. A verified wireless safety critical hard real-time design. In *WOWMOM*. IEEE, 2011.

[GHP07] Christian Groß, Holger Hermanns, and Reza Pulungan. Does clock precision influence ZigBee's energy consumptions? In *OPODIS*, volume 4878 of *Lecture Notes in Computer Science*, pages 174–188. Springer, 2007.

[Gir82] Michèle Giry. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis*, pages 68–85. Springer, 1982.

[GLMS11] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2010: A toolbox for the construction and analysis of distributed processes. In *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 372–387. Springer, 2011.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[God97] Patrice Godefroid. Model checking for programming languages using Verisoft. In *POPL*, pages 174–186. ACM Press, 1997.

[Góm09] Rodolfo Gómez. A compositional translation of timed automata with deadlines to uppaal timed automata. In *FORMATS*, volume 5813 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2009.

[GS01] Geoffrey R. Grimmet and David R. Stirzaker. *Probability and Random Processes*. Oxford University Press, third edition, 2001.

[GvdP96] Jan Friso Groote and Jaco van de Pol. A bounded retransmission protocol for large data packets. In *Algebraic Methodology and Software Technology, 5th International Conference, AMAST '96*,

Munich, Germany, July 1-5, 1996, Proceedings, volume 1101 of *Lecture Notes in Computer Science*, pages 536–550. Springer, 1996.

[GvdP00] Jan Friso Groote and Jaco van de Pol. State space reduction using partial tau-confluence. In *MFCS*, volume 1893 of *Lecture Notes in Computer Science*, pages 383–393. Springer, 2000.

[Hah13] Ernst Moritz Hahn. *Model checking stochastic hybrid systems*. PhD thesis, Universität des Saarlandes, 2013.

[Har10] Arnd Hartmanns. Model-checking and simulation for stochastic timed systems. In *FMCO*, volume 6957 of *Lecture Notes in Computer Science*, pages 372–391. Springer, 2010.

[Hen96] Thomas A. Henzinger. The theory of hybrid automata. In *LICS*, pages 278–292. IEEE Computer Society, 1996.

[Her02] Holger Hermanns. *Interactive Markov Chains: The Quest for Quantified Quality*, volume 2428 of *Lecture Notes in Computer Science*. Springer, 2002.

[HH09] Arnd Hartmanns and Holger Hermanns. A Modest approach to checking probabilistic timed automata. In *QEST*, pages 187–196. IEEE Computer Society, 2009.

[HH12] Arnd Hartmanns and Holger Hermanns. Modelling and decentralised runtime control of self-stabilising power micro grids. In *ISoLA (1)*, volume 7609 of *Lecture Notes in Computer Science*, pages 420–439. Springer, 2012.

[HH14] Arnd Hartmanns and Holger Hermanns. The Modest Toolset: An integrated environment for quantitative modelling and verification. In *TACAS*, volume 8413 of *Lecture Notes in Computer Science*, pages 593–598. Springer, 2014.

[HHB12] Arnd Hartmanns, Holger Hermanns, and Pascal Berrang. A comparative analysis of decentralized power grid stabilization strategies. In *Winter Simulation Conference*. WSC, 2012.

[HHH14] Ernst Moritz Hahn, Arnd Hartmanns, and Holger Hermanns. Reachability and reward checking for stochastic timed automata. *ECEASST*, September 2014. To appear.

[HHHK13] Ernst Moritz Hahn, Arnd Hartmanns, Holger Hermanns, and Joost-Pieter Katoen. A compositional modelling and analysis framework for stochastic hybrid systems. *Formal Methods in System Design*, 43(2):191–232, 2013.

[HHWZ09] Ernst Moritz Hahn, Holger Hermanns, Björn Wachter, and Lijun Zhang. INFAMY: An infinite-state Markov model checker. In *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 641–647. Springer, 2009.

[HHWZ10a] Ernst Moritz Hahn, Holger Hermanns, Björn Wachter, and Lijun Zhang. PARAM: A model checker for parametric Markov models. In *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 660–664. Springer, 2010.

[HHWZ10b] Ernst Moritz Hahn, Holger Hermanns, Björn Wachter, and Lijun Zhang. PASS: Abstraction refinement for infinite probabilistic models. In *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 353–357. Springer, 2010.

[HKK14] Holger Hermanns, Jan Krcál, and Jan Kretínský. Probabilistic bisimulation: Naturally on distributions. *CoRR*, abs/1404.5084, 2014.

[HKQ11] Henri Hansen, Marta Z. Kwiatkowska, and Hongyang Qu. Partial order reduction for model checking Markov decision processes under unconditional fairness. In *QEST*, pages 203–212. IEEE Computer Society, 2011.

[HLMP04] Thomas Hérault, Richard Lassaigne, Frédéric Magniette, and Sylvain Peyronnet. Approximate probabilistic model checking. In *VMCAI*, volume 2937 of *Lecture Notes in Computer Science*, pages 73–84. Springer, 2004.

[HMKS99] H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi terminal binary decision diagrams to represent and analyse continuous time Markov chains. In *NSMC*, pages 188–207. Prensas Universitarias de Zaragoza, 1999.

[HMZ+12] David Henriques, João Martins, Paolo Zuliani, André Platzer, and Edmund M. Clarke. Statistical model checking for Markov decision processes. In *QEST*, pages 84–93. IEEE Computer Society, 2012.

[HNSY94] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Inf. Comput.*, 111(2):193–244, 1994.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hoe63] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, March 1963.

[Hof13] Markus Hoffmann. Implicit and explicit stochastic semantics for timed automata in UPPAAL and Modest. B.Sc. thesis, Universität des Saarlandes, 2013.

[HS87] Peter J. Haas and Gerald S. Shedler. Regenerative generalized semi-Markov processes. *Communications in Statistics. Stochastic Models*, 3(3):409–438, 1987.

[HS00] Peter G. Harrison and B. Strulo. Spades - a process algebra for discrete event simulation. *Journal of Logic and Computation*, 10(1):3–42, 2000.

[HS11] Hanno Hildmann and Fabrice Saffre. Influence of variable supply and load flexibility on demand-side management. In *EEM'11*, pages 63–68. IEEE Conference Publications, 2011.

[HSV93] Leen Helmink, M. P. A. Sellink, and Frits W. Vaandrager. Proof-checking a data link protocol. In *TYPES*, volume 806 of *Lecture Notes in Computer Science*, pages 127–165. Springer, 1993.

[HT13] Arnd Hartmanns and Mark Timmer. On-the-fly confluence detection for statistical model checking. In *NASA Formal Methods*, volume 7871 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 2013.

[HT14a] Henri Hansen and Mark Timmer. A comparison of confluence and ample sets in probabilistic and non-probabilistic branching time. *Theoretical Computer Science*, 538C:103–123, 2014.

[HT14b] Arnd Hartmanns and Mark Timmer. Sound statistical model checking for MDP using partial order and confluence reduction. *Software Tools for Technology Transfer*, 2014. To appear.

[HW09] Holger Hermanns and Holger Wiechmann. Future design challenges for electric energy supply. In *ETFA*. IEEE, 2009.

[HW12] Holger Hermanns and Holger Wiechmann. *Embedded Systems for Smart Appliances and Energy Managment*, volume 3 of *Embedded Systems*, chapter Demand-Response Managment for Dependable Power Grids. Springer Science+Business Media, New York, 2012.

[HWZ08] Holger Hermanns, Björn Wachter, and Lijun Zhang. Probabilistic CEGAR. In *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 162–175. Springer, 2008.

[HZ11] Holger Hermanns and Lijun Zhang. From concurrency models to numbers - performance and dependability. In *Software and Systems Safety - Specification and Verification*, volume 30 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 182–210. IOS Press, 2011.

[ISO11] ISO/IEC 9899:2011. Information technology – Programming languages – C, 2011.

[ISO12a] ISO/IEC 19505-1:2012. Information technology – Object Management Group Unified Modeling Language (OMG UML) – Part 1: Infrastructure, 2012.

[ISO12b] ISO/IEC 19505-2:2012. Information technology – Object Management Group Unified Modeling Language (OMG UML) – Part 2: Superstructure, 2012.

[Job96] Macarthur Job. *Air Disaster*, volume 2. Aerospace Publications, 1996.

[KMN02] Michael J. Kearns, Yishay Mansour, and Andrew Y. Ng. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. *Machine Learning*, 49(2–3):193–208, 2002.

[KNP09] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Stochastic games for verification of probabilistic timed automata. In *FORMATS*, volume 5813 of *Lecture Notes in Computer Science*, pages 212–227. Springer, 2009.

[KNP10] M. Kwiatkowska, G. Norman, and D. Parker. Advances and challenges of probabilistic model checking. In *2010 48th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 1691–1698, 2010.

[KNP11] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer, 2011.

[KNP12] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. The PRISM benchmark suite. In *QEST*, pages 203–204. IEEE Computer Society, 2012.

[KNPS06] Marta Z. Kwiatkowska, Gethin Norman, David Parker, and Jeremy Sproston. Performance analysis of probabilistic timed automata using digital clocks. *Formal Methods in System Design*, 29(1):33–78, 2006.

[KNS03] Marta Z. Kwiatkowska, Gethin Norman, and Jeremy Sproston. Probabilistic model checking of deadline properties in the IEEE 1394 Firewire root contention protocol. *Formal Asp. Comput.*, 14(3):295–318, 2003.

[KNSS00] Marta Z. Kwiatkowska, Gethin Norman, Roberto Segala, and Jeremy Sproston. Verifying quantitative properties of continuous probabilistic timed automata. In *CONCUR*, volume 1877 of *Lecture Notes in Computer Science*, pages 123–137. Springer, 2000.

[KNSS02] Marta Z. Kwiatkowska, Gethin Norman, Roberto Segala, and Jeremy Sproston. Automatic verification of real-time systems with discrete probability distributions. *Theor. Comput. Sci.*, 282(1):101–150, 2002.

[KNSW07] Marta Z. Kwiatkowska, Gethin Norman, Jeremy Sproston, and Fuzhi Wang. Symbolic model checking for probabilistic timed automata. *Inf. Comput.*, 205(7):1027–1077, 2007.

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *ECML*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2006.

[KZ09] Joost-Pieter Katoen and Ivan S. Zapreev. Simulation-based CTMC model checking: An empirical evaluation. In *QEST*, pages 31–40. IEEE Computer Society, 2009.

[LDB10] Axel Legay, Benoît Delahaye, and Saddek Bensalem. Statistical model checking: An overview. In *RV*, volume 6418 of *Lecture Notes in Computer Science*, pages 122–135. Springer, 2010.

[Leh12] Sebastian Lehnhoff, 2012. Private communication.

[LHK01] Gabriel G. Infante López, Holger Hermanns, and Joost-Pieter Katoen. Beyond memoryless distributions: Model checking semi-Markov chains. In *PAPM-PROBMIV*, volume 2165 of *Lecture Notes in Computer Science*, pages 57–70. Springer, 2001.

[LKR11] Sebastian Lehnhoff, Olav Krause, and Christian Rehtanz. Dezentrales autonomes Energiemanagement (distributed autonomous power management). *Automatisierungstechnik*, 59(3):167–179, 2011.

[LP12] Richard Lassaigne and Sylvain Peyronnet. Approximate planning and verification for large Markov decision processes. In *SAC*, pages 1314–1319. ACM, 2012.

[LSS10] Tom Lauricella, Kara Scannell, and Jenny Strasburg. How a trading algorithm went awry. The Wall Street Journal, October 2010. Available online at http://online.wsj.com/article/SB10001424052748704029304575526390131916792.html, last checked on 2014-07-12.

[LST14] Axel Legay, Sean Sedwards, and Louis-Marie Traonouez. Scalable verification of markov decision processes. In *FMDS*, volume 8938 of *Lecture Notes in Computer Science*, pages 350–362. Springer, 2014.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[Mil89] Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.

[Min99] Marius Minea. Partial order reduction for model checking of timed automata. In *CONCUR*, volume 1664 of *Lecture Notes in Computer Science*, pages 431–446. Springer, 1999.

[MPL11] João Martins, André Platzer, and João Leite. Statistical model checking for distributed probabilistic-control hybrid automata with smart grid applications. In *ICFEM*, volume 6991 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2011.

[MW12] Radu Mateescu and Anton Wijs. Sequential and distributed on-the-fly computation of weak tau-confluence. *Science of Computer Programming*, 77(10-11):1075–1094, 2012.

[Nim10] Vincent Nimal. Statistical approaches for probabilistic model checking. Master's thesis, Oxford University, 2010.

[NPS13] Gethin Norman, David Parker, and Jeremy Sproston. Model checking for probabilistic timed automata. *Formal Methods in System Design*, 43(2):164–190, 2013.

[Pel94] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *CAV*, volume 818 of *Lecture Notes in Computer Science*, pages 377–390. Springer, 1994.

[Pel96] Doron Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in System Design*, 8(1):39–64, 1996.

[PLM03] Gordon J. Pace, Frédéric Lang, and Radu Mateescu. Calculating-confluence compositionally. In *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 446–459. Springer, 2003.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.

[Put94] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Mathematical Statistics: Applied Probability and Statistics. John Wiley & Sons Inc., New York, 1994.

[Ros06] Sheldon M. Ross. *Simulation*. Elsevier Academic Press, fourth edition, 2006.

[SB98] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[SC13] U.S. Securities and Exchange Commission. In the Matter of Knight Capital Americas LLC, 2013. Adm. Proc. File No. 3-15570 (October 16, 2013).

[Seg95] Roberto Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, MIT, Cambridge, MA, USA, 1995.

[SGS+13] Songzheng Song, Lin Gui, Jun Sun, Yang Liu, and Jin Song Dong. Improved reachability analysis in DTMC via divide and conquer. In *IFM*, volume 7940 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 2013.

[Sha12] Arpit Sharma. Weighted probabilistic equivalence preserves ω-regular properties. In *MMB/DFT*, volume 7201 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2012.

[SL95] Roberto Segala and Nancy A. Lynch. Probabilistic simulations for probabilistic processes. *Nordic Journal of Computing*, 2(2):250–273, 1995.

[Spr09] Jeremy Sproston. Strict divergence for probabilistic timed automata. In *CONCUR*, volume 5710 of *Lecture Notes in Computer Science*, pages 620–636. Springer, 2009.

[SSBM11] Marten Sijtema, Mariëlle Stoelinga, Axel Belinfante, and Lawrence Marinelli. Experiences with formal engineering: Model-based specification, implementation and testing of a software bus at Neopost. In *FMICS*, volume 6959 of *Lecture Notes in Computer Science*, pages 117–133. Springer, 2011.

[Sto02] Mariëlle Stoelinga. *Alea jacta est: Verification of Probabilistic, Real-Time and Parametric Systems*. PhD thesis, Katholieke U. Nijmegen, The Netherlands, 2002.

[TA09] Martin Tröschel and Hans-Jürgen Appelrath. Towards reactive scheduling for large-scale virtual power plants. In *MATES*, volume 5774 of *Lecture Notes in Computer Science*, pages 141–152. Springer, 2009.

[Tim13] Mark Timmer. *Efficient Modelling, Generation and Analysis of Markov Automata*. PhD thesis, University of Twente, The Netherlands, 2013.

[Tre08] Jan Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008.

[TSvdP11] Mark Timmer, Mariëlle Stoelinga, and Jaco van de Pol. Confluence reduction for probabilistic systems. In *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 311–325. Springer, 2011.

[TvdPS13] Mark Timmer, Jaco van de Pol, and Mariëlle Stoelinga. Confluence reduction for Markov automata. In *FORMATS*, volume 8053 of *Lecture Notes in Computer Science*, pages 243–257. Springer, 2013.

[Val90] Antti Valmari. A stubborn attack on state explosion. In *CAV*, volume 531 of *Lecture Notes in Computer Science*, pages 156–165. Springer, 1990.

[Wal45] Abraham Wald. Sequential tests of statistical hypotheses. *The Annals of Mathematical Statistics*, 16(2):117–186, 1945.

[Wan06] Farn Wang. REDLIB for the formal verification of embedded systems. In *ISoLA*, pages 341–346. IEEE, 2006.

[Wie12] Holger Wiechmann, 2012. Private communication.

[WKHB08] Ralf Wimmer, Alexander Kortus, Marc Herbstritt, and Bernd Becker. Probabilistic model checking and reliability of results. In *DDECS*, pages 207–212. IEEE Computer Society, 2008.

[Wol12] Nicolás Wolovick. *Continuous Probability and Nondeterminism in Labeled Transition Systems*. PhD thesis, FaMAF - UNC, Córdoba, Argentina, 2012.

[YBKK11] Haidi Yue, Henrik C. Bohnenkamp, Malte Kampschulte, and Joost-Pieter Katoen. Analysing and improving energy efficiency of distributed slotted aloha. In *NEW2AN*, volume 6869 of *Lecture Notes in Computer Science*, pages 197–208. Springer, 2011.

[YKNP06] Håkan L. S. Younes, Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Numerical vs. statistical probabilistic model checking. *STTT*, 8(3):216–228, 2006.

[YS02] Håkan L. S. Younes and Reid G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 223–235. Springer, 2002.

[ZPC10] Paolo Zuliani, André Platzer, and Edmund M. Clarke. Bayesian statistical model checking with application to simulink/stateflow verification. In *HSCC*, pages 243–252. ACM, 2010.

[ZSR+10] Lijun Zhang, Zhikun She, Stefan Ratschan, Holger Hermanns, and Ernst Moritz Hahn. Safety verification for probabilistic hybrid systems. In *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 196–211. Springer, 2010.

[ZZ14] Hao Zheng and Yingying Zhang. Local state space analysis leads to better partial order reduction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(6):839–852, 2014.

List of Abbreviations

AIMD	additive-increase, multiplicative-decrease
ALAP	as late as possible
APMC	approximate probabilistic model checking
ASAP	as soon as possible
BDD	binary decision diagram
BEB	binary exponential backoff
BRP	bounded retransmission protocol
BRTDP	bounded real-time dynamic programming
cdf	cumulative distribution function
CEGAR	counterexample-guided abstraction refinement
CSMA/CD	carrier sense multiple access with collision detection
CTL	computation tree logic
CTMC	continuous-time Markov chain
DQL	delayed Q-learning
DTMC	discrete-time Markov chain
GSMP	generalised semi-Markov process
HA	hybrid automaton
IMC	Interactive Markov chain
LP	linear programming
LTL	linear temporal logic
LTS	labelled transition system
MA	Markov automaton
MDP	Markov decision process
NLMP	nondeterministic labelled Markov process
NLTS	network of labelled transition systems
PA	probabilistic automaton
PCTL	probabilistic computation tree logic
pdf	probability density function
PHA	probabilistic hybrid automaton
POR	partial order reduction
PRNG	pseudo-random number generator
PTA	probabilistic timed automaton
PTCTL	probabilistic timed computation tree logic
PV	photovoltaic
SHA	stochastic hybrid automaton

SHA	stochastic hybrid automaton
SMC	statistical model checking
SPRT	sequential probability ratio test
SSS	self-stabilising system
STA	stochastic timed automaton
TA	timed automaton
TCP	transmission control protocol
TCTL	timed computation tree logic
TDPTA	time-deterministic probabilistic timed automaton
TDSTA	time-deterministic stochastic timed automaton
TPTS	timed probabilistic transition system
TTS	timed transition system
VDTMC	discrete-time Markov chain with variables
VLTS	labelled transition system with variables
VMDP	Markov decision process with variables
VPTA	probabilistic timed automaton with variables
XML	Extensible Markup Language

Index